Copyrighted Material

WILEY

Mastering UNIX Shell Scripting

Randal K. Michael

SECOND EDITION

BASH, Bourne, and KORN Shell Scripting for Programmers, System Administrators, and Mittagarus

Mastering UNIX[®]Shell Scripting

Bash, Bourne, and Korn Shell Scripting for Programmers, System Administrators, and UNIX Gurus

Second Edition

Randal K. Michael



Wiley Publishing, Inc.

Mastering UNIX[®]Shell Scripting Second Edition

Mastering UNIX[®]Shell Scripting

Bash, Bourne, and Korn Shell Scripting for Programmers, System Administrators, and UNIX Gurus

Second Edition

Randal K. Michael



Wiley Publishing, Inc.

Mastering UNIX[®]Shell Scripting: Bash, Bourne, and Korn Shell Scripting for Programmers, System Administrators, and UNIX Gurus, Second Edition

Published by Wiley Publishing, Inc. 10475 Crosspoint Boulevard Indianapolis, IN 46256 www.wiley.com

Copyright © 2008 by Randal K. Michael

Published by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-470-18301-4

Manufactured in the United States of America

 $10\,9\,8\,7\,6\,5\,4\,3\,2\,1$

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4355, or online at http://www.wiley.com/go/permissions.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Website is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Website may provide or recommendations it may make. Further, readers should be aware that Internet Websites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services or to obtain technical support, please contact our Customer Care Department within the U.S. at (800) 762-2974, outside the U.S. at (317) 572-3993 or fax (317) 572-4002.

Library of Congress Cataloging-in-Publication Data is available from publisher.

Trademarks: Wiley, the Wiley logo, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. UNIX is a registered trademark of The Open Group. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

This book is dedicated to my wife Robin, the girls, Andrea and Ana, and the grandchildren, Gavin, Jocelyn, and Julia – my true inspiration.

About the Author

Randal K. Michael is a UNIX Systems Administrator working as a contract consultant. He teaches UNIX shell scripting in corporate settings, where he writes shell scripts to address a variety of complex problems and tasks, ranging from monitoring systems to replicating large databases. He has more than 30 years of experience in the industry and 15 years of experience as a UNIX Systems Administrator, working on AIX, HP-UX, Linux, OpenBSD, and Solaris.

Credits

Executive Editor Carol Long

Development Editor John Sleeva

Technical Editor John Kennedy

Production Editor Dassi Zeidel

Copy Editor Kim Cofer

Editorial Manager Mary Beth Wakefield **Production Manager** Tim Tate

Vice President and Executive Group Publisher Richard Swadley

Vice President and Executive Publisher Joseph B. Wikert

Project Coordinator, Cover Lynsey Stanford

Proofreader Candace English

Indexer Robert Swanson

Contents

Acknowled	gments	XXV
Introduction		xxvii
Part One	The Basics of Shell Scripting	
Chapter 1	Scripting Quick Start and Review	3
-	Case Sensitivity	3
	UNIX Special Characters	3
	Shells	4
	Shell Scripts	4
	Functions	4
	Running a Shell Script	5
	Declare the Shell in the Shell Script	6
	Comments and Style in Shell Scripts	6
	Control Structures	8
	if then statement	8
	if then else statement	8
	if then elif (else) statement	9
	for in statement	9
	while statement	9
	until statement	9
	case statement	10
	Using break, continue, exit, and return	10
	Here Document	11
	Shell Script Commands	12
	Symbol Commands	14
	Variables	15
	Command-Line Arguments	15
	shift Command	16
	Special Parameters \$* and \$@	17
	Special Parameter Definitions	17
	Double Quotes, Forward Tics, and Back Tics	18

Using awk on Solaris	19
Using the echo Command Correctly	19
Math in a Shell Script	20
Operators	20
Built-In Mathematical Functions	21
File Permissions, suid and sgid Programs	21
chmod Command Syntax for Each Purpose	22
To Make a Script Executable	22
To Set a Program to Always Execute as the Owner	23
To Set a Program to Always Execute as a Member of the	
File Owner's Group	23
To Set a Program to Always Execute as Both the File	
Owner and the File Owner's Group	23
Running Commands on a Remote Host	23
Setting Traps	25
User-Information Commands	25
who Command	26
w Command	26
last Command	26
ps Command	27
Communicating with Users	27
Uppercase or Lowercase Text for Easy Testing	28
Check the Return Code	29
Time-Based Script Execution	30
Cron Tables	30
Cron Table Entry Syntax	31
at Command	31
Output Control	32
Silent Running	32
Using getopts to Parse Command-Line Arguments	33
Making a Co-Process with Background Function	34
Catching a Delayed Command Output	36
Fastest Ways to Process a File Line-by-Line	37
Using Command Output in a Loop	40
Mail Notification Techniques	41
Using the mail and mailx Commands	41
Using the sendmail Command to Send Outbound Mail	41
Creating a Progress Indicator	43
A Series of Dots	43
A Rotating Line	43
Elapsed Time	44
Working with Record Files	45
Working with Strings	46
Creating a Pseudo-Random Number	47
Using /dev/random and /dev/urandom	48
Checking for Stale Disk Partitions in AIX	48

	Automated Host Pinging	49
	Highlighting Specific Text in a File	49
	Keeping the Printers Printing	50
	AIX "Classic" Printer Subsystem	50
	System V and CUPS Printing	50
	Automated FTP File Transfer	51
	Using rsync to Replicate Data	51
	Simple Generic rsync Shell Script	52
	Capturing a List of Files Larger than \$MEG	53
	Capturing a User's Keystrokes	53
	Using the bc Utility for Floating-Point Math	54
	Number Base Conversions	55
	Using the typeset Command	55
	Using the printf Command	55
	Create a Menu with the select Command	56
	Removing Repeated Lines in a File	58
	Removing Blank Lines from a File	58
	Testing for a Null Variable	58
	Directly Access the Value of the Last Positional Parameter, \$#	59
	Remove the Column Headings in a Command Output	59
	Arrays	60
	Loading an Array	60
	Testing a String	61
	Summary	65
Chapter 2	24 Ways to Process a File Line-by-Line	67
-	Command Syntax	67
	Using File Descriptors	68
	Creating a Large File to Use in the Timing Test	68
	Creating a Large File to Use in the Timing Test 24 Methods to Parse a File Line-by-Line	68 73
	24 Methods to Parse a File Line-by-Line	73
	24 Methods to Parse a File Line-by-Line Method 1: cat_while_read_LINE	73 74
	24 Methods to Parse a File Line-by-Line Method 1: cat_while_read_LINE Method 2: while_read_LINE_bottom	73 74 75
	24 Methods to Parse a File Line-by-Line Method 1: cat_while_read_LINE Method 2: while_read_LINE_bottom Method 3: cat_while_LINE_line	73 74 75 76
	24 Methods to Parse a File Line-by-Line Method 1: cat_while_read_LINE Method 2: while_read_LINE_bottom Method 3: cat_while_LINE_line Method 4: while_LINE_line_bottom	73 74 75 76 77
	24 Methods to Parse a File Line-by-Line Method 1: cat_while_read_LINE Method 2: while_read_LINE_bottom Method 3: cat_while_LINE_line Method 4: while_LINE_line_bottom Method 5: cat_while_LINE_line_cmdsub2	73 74 75 76 77 78
	24 Methods to Parse a File Line-by-Line Method 1: cat_while_read_LINE Method 2: while_read_LINE_bottom Method 3: cat_while_LINE_line Method 4: while_LINE_line_bottom Method 5: cat_while_LINE_line_cmdsub2 Method 6: while_LINE_line_bottom_cmdsub2	73 74 75 76 77 78 79
	24 Methods to Parse a File Line-by-Line Method 1: cat_while_read_LINE Method 2: while_read_LINE_bottom Method 3: cat_while_LINE_line Method 4: while_LINE_line_bottom Method 5: cat_while_LINE_line_cmdsub2 Method 6: while_LINE_line_bottom_cmdsub2 Method 7: for_LINE_cat_FILE	73 74 75 76 77 78 79 79
	24 Methods to Parse a File Line-by-Line Method 1: cat_while_read_LINE Method 2: while_read_LINE_bottom Method 3: cat_while_LINE_line Method 4: while_LINE_line_bottom Method 5: cat_while_LINE_line_cmdsub2 Method 6: while_LINE_line_bottom_cmdsub2 Method 7: for_LINE_cat_FILE Method 8: for_LINE_cat_FILE_cmdsub2	73 74 75 76 77 78 79 79 80
	24 Methods to Parse a File Line-by-Line Method 1: cat_while_read_LINE Method 2: while_read_LINE_bottom Method 3: cat_while_LINE_line Method 4: while_LINE_line_bottom Method 5: cat_while_LINE_line_cmdsub2 Method 6: while_LINE_line_bottom_cmdsub2 Method 7: for_LINE_cat_FILE Method 8: for_LINE_cat_FILE_cmdsub2 Method 9: while_line_outfile	73 74 75 76 77 78 79 79 80 81
	24 Methods to Parse a File Line-by-Line Method 1: cat_while_read_LINE Method 2: while_read_LINE_bottom Method 3: cat_while_LINE_line Method 4: while_LINE_line_bottom Method 5: cat_while_LINE_line_cmdsub2 Method 6: while_LINE_line_bottom_cmdsub2 Method 7: for_LINE_cat_FILE Method 8: for_LINE_cat_FILE_cmdsub2 Method 9: while_line_outfile Method 10: while_read_LINE_FD_IN Method 11: cat_while_read_LINE_FD_OUT Method 12: while_read_LINE_bottom_FD_OUT	73 74 75 76 77 78 79 79 80 81 81
	24 Methods to Parse a File Line-by-Line Method 1: cat_while_read_LINE Method 2: while_read_LINE_bottom Method 3: cat_while_LINE_line Method 4: while_LINE_line_bottom Method 5: cat_while_LINE_line_cmdsub2 Method 6: while_LINE_line_bottom_cmdsub2 Method 7: for_LINE_cat_FILE Method 8: for_LINE_cat_FILE_cmdsub2 Method 9: while_line_outfile Method 10: while_read_LINE_FD_IN Method 11: cat_while_read_LINE_FD_OUT Method 12: while_read_LINE_bottom_FD_OUT Method 13: while_LINE_line_bottom_FD_OUT	73 74 75 76 77 78 79 79 80 81 81 81
	24 Methods to Parse a File Line-by-Line Method 1: cat_while_read_LINE Method 2: while_read_LINE_bottom Method 3: cat_while_LINE_line Method 4: while_LINE_line_bottom Method 5: cat_while_LINE_line_cmdsub2 Method 6: while_LINE_line_bottom_cmdsub2 Method 7: for_LINE_cat_FILE Method 8: for_LINE_cat_FILE_cmdsub2 Method 9: while_line_outfile Method 10: while_read_LINE_FD_IN Method 11: cat_while_read_LINE_FD_OUT Method 12: while_read_LINE_bottom_FD_OUT Method 13: while_LINE_line_bottom_FD_OUT Method 14: while_LINE_line_bottom_cmdsub2_FD_OUT	73 74 75 76 77 78 79 79 80 81 81 83 85
	24 Methods to Parse a File Line-by-Line Method 1: cat_while_read_LINE Method 2: while_read_LINE_bottom Method 3: cat_while_LINE_line Method 4: while_LINE_line_bottom Method 5: cat_while_LINE_line_cmdsub2 Method 6: while_LINE_line_bottom_cmdsub2 Method 7: for_LINE_cat_FILE Method 8: for_LINE_cat_FILE_cmdsub2 Method 9: while_line_outfile Method 10: while_read_LINE_FD_IN Method 11: cat_while_read_LINE_FD_OUT Method 12: while_read_LINE_bottom_FD_OUT Method 13: while_LINE_line_bottom_FD_OUT	73 74 75 76 77 78 79 79 80 81 81 83 85 86
	24 Methods to Parse a File Line-by-Line Method 1: cat_while_read_LINE Method 2: while_read_LINE_bottom Method 3: cat_while_LINE_line Method 4: while_LINE_line_bottom Method 5: cat_while_LINE_line_cmdsub2 Method 6: while_LINE_line_bottom_cmdsub2 Method 7: for_LINE_cat_FILE Method 8: for_LINE_cat_FILE_cmdsub2 Method 9: while_line_outfile Method 10: while_read_LINE_FD_IN Method 11: cat_while_read_LINE_FD_OUT Method 12: while_read_LINE_bottom_FD_OUT Method 13: while_LINE_line_bottom_FD_OUT Method 14: while_LINE_line_bottom_cmdsub2_FD_OUT	73 74 75 76 77 78 79 79 80 81 81 81 83 85 86 87

	Method 18: while_line_outfile_FD_OUT	90
	Method 19: while_line_outfile_FD_IN_AND_OUT	91
	Method 20: while_LINE_line_FD_IN	92
	Method 21: while_LINE_line_cmdsub2_FD_IN	93
	Method 22: while_read_LINE_FD_IN_AND_OUT	94
	Method 23: while_LINE_line_FD_IN_AND_OUT	96
	Method 24: while_LINE_line_cmdsub2_FD_IN_AND_OUT	97
	Timing Each Method	98
	Timing Script	99
	Timing Data for Each Method	117
	Timing Command-Substitution Methods	127
	What about Using Command Input Instead of File Input?	128
	Summary	129
	Lab Assignments	129
Chapter 3	Automated Event Notification	131
chapter 5	Basics of Automating Event Notification	131
	Using the mail and mailx Commands	132
	Setting Up a sendmail Alias	134
	Problems with Outbound Mail	134
	Creating a "Bounce" Account with a .forward File	136
	Using the sendmail Command to Send Outbound Mail	137
	Dial-Out Modem Software	139
	SNMP Traps	139
	Summary	140
	Lab Assignments	141
Chapter 4	Progress Indicators Using a Series of Dots, a Rotating	
	Line, or Elapsed Time	143
	Indicating Progress with a Series of Dots	143
	Indicating Progress with a Rotating Line	145
	Indicating Progress with Elapsed Time	148
	Combining Feedback Methods	151
	Other Options to Consider	153
	Summary	153
	Lab Assignments	154
Part Two	Scripts for Programmers, Testers, and Analysts	
Chapter 5	Working with Record Files	157
chapter 5	What Is a Record File?	157
	Fixed-Length Record Files	158
	Variable-Length Record Files	159
	Processing the Record Files	160
	Tasks for Records and Record Files	164
	Tasks on Fixed-Length Record Files	161
	0	
	Tasks on Variable-Length Record Files	166
	Tasks on Variable-Length Record Files The Merge Process	166 169

	Working with Strings	171
	Putting It All Together	173
	Other Things to Consider	183
	Summary	184
	Lab Assignments	184
Chapter 6	Automated FTP Stuff	187
	Syntax	187
	Automating File Transfers and Remote Directory Listings	190
	Using FTP for Directory Listings on a Remote Machine	190
	Getting One or More Files from a Remote System	192
	Pre and Post Events	195
	Script in Action	196
	Uploading One or More Files to a Remote System	196
	Replacing Hard-Coded Passwords with Variables	199
	Example of Detecting Variables in a Script's Environment	200 203
	Modifying Our FTP Scripts to Use Password Variables What about Encryption?	203
	Creating Encryption Keys	209
	Setting Up No-Password Secure Shell Access	210
	Secure FTP and Secure Copy Syntax	211
	Automating FTP with autoexpect and expect Scripts	212
	Other Things to Consider	217
	Use Command-Line Switches to Control Execution	217
	Keep a Log of Activity	217
	Add a Debug Mode to the Scripts	217
	Reading a Password into a Shell Script	217
	Summary	218
	Lab Assignments	218
Chapter 7	Using rsync to Efficiently Replicate Data	219
	Syntax	219
	Generic rsync Shell Script	220
	Replicating Multiple Directories with rsync	222
	Replicating Multiple Filesystems with rsync	237
	Replicating an Oracle Database with rsync	251
	Filesystem Structures	252 254
	rsync Copy Shell Script	234 289
	Summary Lab Assignments	289
Chapter 8	Automating Interactive Programs with Expect and	
Simpler 0	Autoexpect	291
	Downloading and Installing Expect	291
	The Basics of Talking to an Interactive Script or Program	293
	Using autoexpect to Automatically Create an Expect Script	296
	Working with Variables	304
	What about Conditional Tests?	306

	Expect's Version of a case Statement	306
	Expect's Version of an ifthenelse Loop	313
	Expect's Version of a while Loop	314
	Expect's Version of a for Loop	315
	Expect's Version of a Function	317
	Using Expect Scripts with Sun Blade Chassis and JumpStart	318
	Summary	323
	Lab Assignments	324
Chapter 9	Finding Large Files and Files of a Specific Type	325
	Syntax	326
	Remember That File and Directory Permissions Thing	327
	Don't Be Shocked by the Size of the Files	327
	Creating the Script	327
	Narrowing Down the Search	333
	Other Options to Consider	333
	Summary	334
	Lab Assignments	334
Chapter 10	Process Monitoring and Enabling Pre-Processing, Startup,	
-	and Post-Processing Events	335
	Syntax	336
	Monitoring for a Process to Start	336
	Monitoring for a Process to End	338
	Monitor and Log as a Process Starts and Stops	342
	Timed Execution for Process Monitoring, Showing Each PID,	
	and Timestamp with Event and Timing Capability	347
	Other Options to Consider	367
	Common Uses	367
	Modifications to Consider	367
	Summary	367
	Lab Assignments	368
Chapter 11	Pseudo-Random Number and Data Generation	369
	What Makes a Random Number?	369
	The Methods	370
	Method 1: Creating a Pseudo-Random Number Utilizing the	
	PID and the RANDOM Shell Variable	371
	Method 2: Creating Numbers between 0 and 32,767	371
	Method 3: Creating Numbers between 1 and a User-Defined	
	Maximum	372
	Method 4: Creating Fixed-Length Numbers between 1 and a	
	User-Defined Maximum	373
	Why Pad the Number with Zeros the Hard Way?	375
	Method 5: Using the /dev/random and /dev/urandom	
	Character Special Files	376
	Shell Script to Create Pseudo-Random Numbers	379
	Creating Unique Filenames	384

	Creating a File Filled with Random Characters	392
	Other Things to Consider	399
	Summary	399
	Lab Assignments	400
Chapter 12	Creating Pseudo-Random Passwords	401
	Randomness	401
	Creating Pseudo-Random Passwords	402
	Syntax	403
	Arrays	403
	Loading an Array	403
	Building the Password-Creation Script	405
	Order of Appearance	405
	Define Functions	406
	Testing and Parsing Command-Line Arguments	414
	Beginning of Main	418
	Setting a Trap	418
	Checking for the Keyboard File	419
	Loading the KEYS Array	419
	Building a New Pseudo-Random Password	420
	Printing the Manager's Password Report for Safekeeping	421
	Other Options to Consider	431
	Password Reports?	432
	Which Password?	432
	Other Uses?	432
	Summary	432
	Lab Assignments	432
Chapter 13	Floating-Point Math and the bc Utility	433
	Syntax	433
	Creating Some Shell Scripts Using bc	434
	Creating the float_add.ksh Shell Script	434
	Testing for Integers and Floating-Point Numbers	440
	Building a Math Statement for the bc Command	441
	Using a Here Document	442
	Creating the float_subtract.ksh Shell Script	443
	Using getopts to Parse the Command Line	449
	Building a Math Statement String for bc	450
	Here Document and Presenting the Result	451
	Creating the float_multiply.ksh Shell Script	452
	Parsing the Command Line for Valid Numbers	458
	Creating the float_divide.ksh Shell Script	460
	Creating the float_average.ksh Shell Script	467
	Other Options to Consider	472
	Creating More Functions	472
	Summary	473
	Lab Assignments	473

Chapter 14	Number Base Conversions	475
-	Syntax	475
	Example 1: Converting from Base 10 to Base 16	476
	Example 2: Converting from Base 8 to Base 16	476
	Example 3: Converting Base 10 to Octal	477
	Example 4: Converting Base 10 to Hexadecimal	477
	Scripting the Solution	477
	Base 2 (Binary) to Base 16 (Hexadecimal) Shell Script	478
	Base 10 (Decimal) to Base 16 (Hexadecimal) Shell Script	481
	Script to Create a Software Key Based on the Hexadecimal	
	Representation of an IP Address	485
	Script to Translate between Any Number Base	490
	Using getopts to Parse the Command Line	495
	Example 5: Correct Usage of the equate_any_base.ksh	
	Shell Script	495
	Example 6: Incorrect Usage of the equate_any_base.ksh	
	Shell Script	495
	Continuing with the Script	497
	Beginning of Main	498
	An Easy, Interactive Script to Convert Between Bases	500
	Using the bc Utility for Number Base Conversions	506
	Other Options to Consider	512
	Software Key Shell Script	512
	Summary	512
	Lab Assignments	513
Chapter 15	hgrep: Highlighted grep Script	515
•	Reverse Video Control	516
	Building the hgrep.Bash Shell Script	517
	Other Options to Consider	524
	Other Options for the tput Command	524
	Summary	525
	Lab Assignments	525
Chapter 16	Monitoring Processes and Applications	527
chapter ro	Monitoring Local Processes	527
	Remote Monitoring with Secure Shell and Remote Shell	530
	Checking for Active Oracle Databases	536
	Using autoexpect to Create an expect Script	539
	Checking if the HTTP Server/Application Is Working	545
	What about Waiting for Something to Complete Executing?	546
	•	549
	Other Things to Consider Proper echo Usage Application APIs and SNMP Traps Summary Lab Assignments	547 548 548 548 549

Part Three	Scripts for Systems Administrators	
Chapter 17	Filesystem Monitoring Syntax Adding Exceptions Capability to Monitoring The Exceptions File Using the MB-of-Free-Space Method Using MB of Free Space with Exceptions Percentage Used — MB Free and Large Filesystems Running Filesystem Scripts on AIX, Linux, HP-UX, OpenBSD, and Solaris Command Syntax and Output Varies between Operating Systems Programming a Shell-Neutral Script Other Options to Consider Event Notification Automated Execution Modify the egrep Statement Summary Lab Assignments	553 559 559 565 568 573 583 583 585 590 600 600 600 600 601 601 601 602
Chapter 18	Monitoring Paging and Swap Space Syntax AIX lsps Command HP-UX swapinfo Command Linux free Command OpenBSD swapctl Command Solaris swap Command Creating the Shell Scripts AIX Paging Monitor HP-UX Swap-Space Monitor Linux Swap-Space Monitor OpenBSD Swap-Space Monitor Solaris Swap-Space Monitor All-in-One Paging- and Swap-Space Monitor Other Options to Consider Event Notification Log File Scheduled Monitoring Summary Lab Assignments	603 604 604 605 606 607 607 607 613 618 622 625 630 638
Chapter 19	Monitoring System Load Installing the System-Statistics Programs in Linux Syntax Syntax for uptime	641 642 644 644

	Linux	645
	What's the Common Denominator?	645
	Syntax for iostat	645
	AIX	646
	HP-UX	646
	Linux	647
	OpenBSD	647
	Solaris	647
	What Is the Common Denominator?	648
	Syntax for sar	649
	AIX	649
	HP-UX	649
	Linux	650
	Solaris	650
	What Is the Common Denominator?	650
	Syntax for vmstat	651
	AIX	652
	HP-UX	652
	Linux	652
	OpenBSD	652
	Solaris	653
	What Is the Common Denominator?	653
	Scripting the Solutions	654
	Using uptime to Measure the System Load	655
	Scripting with the uptime Command	655
	Using sar to Measure the System Load	659
	Scripting with the sar Command	660
	Using iostat to Measure the System Load	665
	Scripting with the iostat Command	665
	Using vmstat to Measure the System Load	670
	Scripting with the vmstat Command	670
	Other Options to Consider	674
	Try to Detect Any Possible Problems for the User	674
	Show the User the Top CPU Hogs	675
	Gathering a Large Amount of Data for Plotting	675
	Summary	675
	Lab Assignments	675
Chapter 20	Monitoring for Stale Disk Partitions (AIX-Specific)	677
•	AIX Logical Volume Manager (LVM)	677
	The Commands and Methods	678
	Disk Subsystem Commands	678
	Method 1: Monitoring for Stale PPs at the LV Level	679
	Method 2: Monitoring for Stale PPs at the PV Level	684
	Method 3: VG, LV, and PV Monitoring with a resync	687
	Other Options to Consider	694
	SSA Disks	694

	Log Files Automated Execution Event Notification Summary Lab Assignment	695 695 695 696 696
Chapter 21	Turning On/Off SSA Identification Lights Syntax Translating an hdisk to a pdisk Identifying an SSA Disk The Scripting Process Usage and User Feedback Functions Control Functions The Full Shell Script Other Things to Consider Error Log Cross-Reference Root Access and sudo Summary Lab Assignment	697 698 698 698 699 703 709 721 721 721 721 721 721 722
Chapter 22	Automated Hosts Pinging with Notification of Failure Syntax Creating the Shell Script Define the Variables Creating a Trap The Whole Shell Script Other Options to Consider \$PINGLIST Variable-Length-Limit Problem Ping the /etc/hosts File Instead of a List File Logging Notification of "Unknown Host" Notification Method Automated Execution Using a Cron Table Entry Summary Lab Assignments	723 725 725 728 736 736 736 737 737 737 737 738 738 738 739 739
Chapter 23	Creating a System-Configuration Snapshot Syntax Creating the Shell Script Other Options to Consider Summary Lab Assignment	741 742 744 774 774 775
Chapter 24	Compiling, Installing, Configuring, and Using sudo The Need for sudo Configuring sudo on Solaris Downloading and Compiling sudo	777 777 778 778

	Compiling sudo	779
	Configuring sudo	790
	Using sudo	797
	Using sudo in a Shell Script	798
	Logging to the syslog with sudo	801
	The sudo Log File	806
	Summary	806
	Lab Assignments	807
Chapter 25	Print-Queue Hell: Keeping the Printers Printing	809
	System V versus BSD versus CUPS Printer Systems	809
	AIX Print-Control Commands	810
	Classic AIX Printer Subsystem	810
	System V Printing on AIX	814
	More System V Printer Commands	818
	CUPS — Common UNIX Printing System	820
	HP-UX Print-Control Commands	823
	Linux Print-Control Commands	825
	Controlling Queuing and Printing Individually	831
	Solaris Print-Control Commands	833
	More System V Printer Commands	837
	Putting It All Together	839
	Other Options to Consider	849
	Logging	849
	Exceptions Capability	849
	Maintenance	849
	Scheduling	849
	Summary	850
	Lab Assignments	850
Chapter 26	Those Pesky Sarbanes-Oxley (SOX) Audits	851
	What to Expect	852
	How to Work with the Auditors	852
	What the Auditors Want to See	853
	Some Handy Commands	854
	Using the id Command	854
	Using the find Command	855 856
	Using the awk and cut Commands	856
	Using the sed Command	862
	Using the dirname and basename Commands	863 864
	Other Things to Consider	864
	Summary Lab Assignments	865
-		
Chapter 27	Using Dirvish with rsync to Create Snapshot-Type Backups	867
	How Does Dirvish Work?	868
	How Much Disk Storage Will I Need?	868
	Configuring Dirvish	868

	Installing Dirvish	869
	Modifying the master.conf Dirvish Configuration File	872
	Creating the default.conf File for Each Filesystem Backup	873
	Performing a Full System Backup	874
	Using Dirvish on the Command Line	875
	A Menu-Interface Shell Script to Control Dirvish	876
	Running All Backups	878
	Running a Particular Backup	879
	Locating and Restoring Images	880
	Expiring and Deleting Backup Images	881
	Using sed to Modify the summary File	883
	Adding a New Backup	884
	Removing a Backup	889
	Managing the Dirvish Backup Banks	890
	Adding a New Dirvish Backup Bank	891
	Deleting a Dirvish Backup Bank	892
	Putting It All Together	893
	Using the dirvish_ctrl Shell Script	918
	Running All Backups Defined in the Runall: Stanza	918
	Running One Particular Backup	919
	Locating and Restoring Files	919
	Deleting Expired Backups and Expiring Backups	921
	Adding a New Dirvish Backup Vault	925
	Removing a Dirvish Vault	930
	Managing Dirvish Backup Banks	930
	Adding a New Dirvish Backup Bank	931
	Removing a Dirvish Backup Bank	932
	Other Things to Consider	932
	Summary	933
	Lab Assignments	933
Chapter 28	Monitoring and Auditing User Keystrokes	935
	Syntax	936
	Scripting the Solution	937
	Logging User Activity	937
	Starting the Monitoring Session	939
	Where Is the Repository?	939
	The Scripts	940
	Logging root Activity	942
	Some sudo Stuff	946
	Monitoring Other Administration Users	948
	Other Options to Consider	951
	Emailing the Audit Logs	951
	Compression	952
	Need Better Security?	953
	Inform the Users	953
	Sudoers File	953

	Summary	953
	Lab Assignments	954
	A Closing Note from the Author	954
Appendix A	What's on the Web Site	955
	Shell Scripts	955
	Functions	966
Index		977

Acknowledgments

The information that I gathered together in this book is the result of working with some of the most talented UNIX professionals on the topic. I have enjoyed every minute of my association with these UNIX gurus and it has been my pleasure to have the opportunity to gain so much knowledge from the pros. I want to thank every one of these people for asking and answering questions over the past 20 years. If my brother Jim had not kept telling me, "you should write a book," after querying me for UNIX details on almost a weekly basis, I doubt the first edition of this book would have ever been written.

I especially want to thank Jack Renfro at Chrysler Corporation for giving me my first shell scripting project so long ago. I had to start with the man pages, but that is how I learned to dig deep to get the answer. Since then I have been on a mission to automate, through shell scripting, support tasks on every system I come in contact with. I certainly value the years I was able to work with Jack.

I must also thank the talented people at Wiley Publishing. As executive editor, Carol Long helped keep things going smoothly. Development editor John Sleeva kept me on schedule and made the edits that make my writing flow with ease. Dassi Zeidel, my production editor, helped with the final edits and prepared the book for layout. John Kennedy, my technical editor, kept me honest, gave me some tips, and ensured the code did not have any errors. It has been a valuable experience for me to work with such a fine group of professionals at Wiley Publishing. I also want to thank my agent, Carole McClendon, at Waterside Productions for all her support on this project. Carole is the best agent that anyone could ever ask for. She is a true professional with the highest ethics.

Of course, my family had a lot to do with my success on this and every project. I want to thank Mom, Pop, Gene, Jim, Marcia, Rusty, Mallory, Anica, and Chad. I want to thank my beautiful bride forever, Robin, for her understanding, patience, and support for the long hours required to complete this project. The girls, Andrea and Ana, always keep a smile on my face, and Steve is always on my mind. The grandchildren, Gavin, Jocelyn, and Julia, are an inspiration for long life, play time, learning, and adventure. I am truly living the dream.

I could not have written this book without the support of all these people and the many others that remain unnamed. It has been an honor!

Introduction

In UNIX there are many ways to accomplish the same task. Given a problem to solve, we may be able to get to a solution in any number of ways. Of course, some techniques will be more efficient, use fewer system resources, and may or may not give the user feedback on what is going on or give more accurate details and more precision to the result. In this book we are going to step through every detail of creating shell scripts to solve real-world UNIX problems and tasks. The shell scripts range from using a pseudo-random number generator to creating passwords using arrays to replicating data with rsync to working with record files. The scope of solutions is broad and detailed. The details required to write a good shell script include commenting each step for future reference. Other details include combining many commands together into a single command statement when desirable, separating commands on several lines of code when readability and understanding the concept may be diminished, and making a script readable and easy to maintain through the life cycle. We will see the benefits of variables and files to store data, show methods to strip out unneeded data from command output, and format data for a particular purpose. Additionally, we are going to show how to write and use functions in our shell scripts and demonstrate the benefits of functions over a shell script written without functions.

This book is intended for any flavor of UNIX, but it emphasizes the AIX, HP-UX, Linux, OpenBSD, and Solaris operating systems. Almost every script in the book is also included on the book's companion web site (www.wiley.com/go/michael2e). Many of the shell scripts are rewritten for various UNIX flavors, when it is necessary. Other shell scripts are not platform-dependent. These script rewrites are necessary because command syntax and output vary, sometimes in a major way, between UNIX flavors. The variations are sometimes as small as extracting data out of a different column or using a different command switch to get the same result, or they can be as major as putting several commands together to accomplish the same task and get a similar output or result on different flavors of UNIX.

In each chapter we start with the very basic concepts to accomplish a task, and then work our way up to some very complex and difficult concepts. The primary purpose of a shell script is to automate repetitive and complex tasks. This alleviates keystroke errors and allows for time-scheduled execution of the shell scripts. It is always better to have the system tell us that it has a problem than to find out too late to be proactive. This book will help us to be more proactive and efficient in our dealing with the system. At every level you will gain more knowledge to allow you to move on to ever

increasingly complex ideas with ease. You are going to see different ways to solve real-world example tasks. There is not just one way to solve a challenge, and we are going to look at the pros and cons of attacking a problem in various ways. Our goal is to be confident and flexible problem solvers. Given a task, we can solve it in any number of ways, and the solution will be intuitively obvious when you complete this book.

Overview of the Book and Technology

This book is intended as a learning tool and study guide to learn how to write shell scripts to solve a multitude of problems by starting with a clear goal. We will cover most shell scripting techniques about seven times, each time hitting the topic from a different angle, solving a different problem. I have found this technique to work extremely well for retention of the material.

Each chapter ends with Lab Assignments that let you either write a new script or modify a shell script covered in the chapter. There is not a "solutions" book. The solution is to make it work! I urge everyone to read this book from cover to cover to get the maximum benefit. The shells covered in this book include Bash, Bourne, and Korn. C shell is not covered. Advanced topics include using rsync to replicate data, creating snapshot-style backups utilizing Dirvish, working with record files to parse data, and many others.

This book goes from some trivial task solutions to some rather advanced concepts that everyone from high school and college students to Systems Administrators will benefit from, and a lot in between. There are several chapters at each level of complexity scattered throughout the book. The shell scripts presented in this book are complete shell scripts, which is one of the things that sets this book apart from other shell-scripting books on the market. The solutions are explained thoroughly, with each part of the shell scripts explained in minute detail down to the philosophy and mindset of the author.

How This Book Is Organized

Each chapter starts with a typical UNIX challenge that occurs every day in the computer world. With each challenge we define a specific goal and start the shell script by defining the correct command syntax to solve the problem. After we present the goal and command syntax, we start by building the shell script around the commands. The next step is to filter the commands' output to strip out the unneeded data, or we may decide to just extract the data we need from the output. If the syntax varies between UNIX flavors, we show the correct syntax to get the same or a similar result. When we get to this point we go further to build options into the shell script to give the end user more flexibility on the command line.

When a shell script has to be rewritten for each operating system, a combined shell script is shown at the end of the chapter that will run on all the UNIX flavors studied in this book, except where noted. To do this last step, we query the system for the UNIX flavor using the uname command. By knowing the flavor of the operating system, we are able to execute the proper commands for each UNIX flavor by using a simple

case statement. If this is new to you, don't worry; everything is explained in detail throughout the book.

Each chapter targets a different real-world problem. Some challenges are very complex, whereas others are just interesting to play around with. Some chapters hit the problem from several different angles in a single chapter, and others leave you the challenge to solve on your own — of course, with a few hints to get you started. Each chapter solves the challenge presented and can be read as a single unit without referencing other chapters in the book, except where noted. Some of the material, though, is explained in great detail in one chapter and lightly covered in other chapters. Because of this variation, I recommend that you start at the beginning of the book and read and study every chapter, and solve each of the Lab Assignments through to the end of the book, because this *is* a learning experience!

Who Should Read this Book

This book is intended for anyone who works with UNIX from the command line on a daily basis. The topics covered in this book are mainly for UNIX professionals computer science students, programmers, programmer-analysts, Systems Operators, application support personnel, Systems Administrators, and anyone who is interested in getting ahead in the support and development arenas. Beginners will get a lot out of this book, too, although some of the material may be a little high-level, so a basic UNIX book may be needed to answer some questions. Everyone should have a good working knowledge of common UNIX commands before starting this book; we do not explain basic UNIX commands in much detail.

I started my career in UNIX by learning on the job how to be a Systems Operator. I wish I had a book like this when I started. Having this history, I wanted others to get a jump-start on their careers. I wrote this book with the knowledge that I was in your shoes at one time, and I remember that I had to learn everything from the man pages, one command at a time. Use this book as a study guide, and you will have a jump-start to get ahead quickly in the UNIX world, which is getting bigger all the time.

Tools You Will Need

To get the most benefit from this book you need access to a UNIX machine, preferably with AIX, HP-UX, Linux, OpenBSD, or Solaris installed. You can run Linux, Solaris, and OpenBSD on standard PC hardware, and this is relatively inexpensive, if not free. Your default shell should be set to Bash or Korn shell. You can find your default shell by entering **echo \$SHELL** on the command line. None of the shell scripts in this book requires a graphical terminal, but it does not hurt to have Gnome, CDE, KDE, or X-Windows running. This way you can work in multiple windows at the same time and cut and paste code between windows.

You also need a text editor that you are comfortable using. UNIX operating systems come with the **vi** editor, and many include **emacs**. You can also use the text editor that comes with KDE, CDE, and Gnome. Remember that the editor must be a text editor that stores files in a standard ANSII format. You will also need some time, patience, and an open, creative mind that is ready to learn.

Another thing to note is that all of the variables used in the shell scripts and functions in this book are in uppercase characters. I did this because it is much easier to follow along with the shell script if you know quickly where the variables are located in the code. When you write your own shell scripts, please use lowercase for all shell script and function variables. The reason this is important is that the operating system, and applications, use *environment variables* that are uppercase. If you are not careful, you can overwrite a critical system or application variable with your own value and hose the system; however, this is dependent on the scope of where the variable is in the code. Just a word of warning; be careful with uppercase variables!

What's on the Web Site

On the book's companion web site, www.wiley.com/go/michael2e, all the shell scripts and most of the functions that are studied in the book can be found. The functions are easy to cut and paste directly into your own shell scripts to make the scripting process a little easier. Additionally, there is a shell script *stub* that you can copy to another filename. This script stub has everything to get started writing quickly. The only thing you need to do is fill in the fields for the following: Script Name, Author, Date, Version, Platform, and Rev. List, when revisions are made. There is a place to define variables and functions, and then you have the "BEGINNING OF MAIN" section to start the main body of the shell script.

Summary

This book is for learning how to be creative, proactive, and professional problem solvers. Given a task, the solution will be *intuitively obvious* to you on completion of this book. This book will help you attack problems logically and present you with a technique of building on what you know. With each challenge presented you will see how to take basic syntax and turn it into the basis for a shell scripting solution. We always start with the basics and build more and more logic into the solution before we add additional options the end user can use for more flexibility.

Speaking of end users, we must always keep our users informed about how processing is proceeding. Giving the user a blank screen to look at is the worst thing that you can do, so for this we can create progress indicators. You will learn how to be proactive by building tools that monitor for specific system events and situations that indicate the beginning stages of an upcoming problem. This is where knowing how to query the system puts you ahead of the game.

With the techniques presented in this book, you will learn. You will learn about problem resolution. You will learn about starting with what you know about a situation and building a solution effectively. You will learn how to make a single shell script work on other platforms without further modifications. You will learn how to be proactive. You will learn how to use plenty of comments in a shell script. You will learn how to write a shell script that is easy to read and follow through the logic. Basically, you will learn to be an effective problem solver, and the solution to any challenge will be *intuitively obvious*!

PART

The Basics of Shell Scripting

Chapter 1: Scripting Quick Start and Review

Chapter 2: 24 Ways to Process a File Line-by-Line

Chapter 3: Automated Event Notification

Chapter 4: Progress Indicators Using a Series of Dots, a Rotating Line, or Elapsed Time

CHAPTER 1

Scripting Quick Start and Review

We are going to start out by giving a targeted refresher course. The topics that follow are short explanations of techniques that we always have to search the book to find; here they are all together in one place. The explanations range from showing the fastest way to process a file line-by-line to the simple matter of case sensitivity of UNIX and shell scripts. This should not be considered a full and complete list of scripting topics, but it is a very good starting point and it does point out a sample of the topics covered in the book. For each topic listed in this chapter there is a very detailed explanation later in the book.

We urge everyone to study this entire book. Every chapter hits a different topic using a different approach. The book is written this way to emphasize that there is never only one technique to solve a challenge in UNIX. All the shell scripts in this book are real-world examples of how to solve a problem. Thumb through the chapters, and you can see that we tried to hit most of the common (and some uncommon!) tasks in UNIX. All the shell scripts have a good explanation of the thinking process, and we always start out with the correct command syntax for the shell script targeting a specific goal. I hope you enjoy this book as much as I enjoyed writing it. Let's get started!

Case Sensitivity

UNIX is case sensitive. Because UNIX is case sensitive, our shell scripts are also case sensitive.

UNIX Special Characters

All of the following characters have a special meaning or function. If they are used in a way that their special meaning is not needed, they must be *escaped*. To escape,

or remove its special function, the character must be immediately preceded with a backslash, $\$, or enclosed within ' ' forward tic marks (single quotes).

\ / ; , . ~ # \$? & * () [] ' ' " + - ! ^ = | < >

Shells

A *shell* is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of shells, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions. This book works with the Bourne, Bash, and Korn shells. Shells are located in either the /usr/bin/ directory or the /bin/ directory, depending on the UNIX flavor and specific version.

Table 1	-1
---------	----

SHELL	DIRECTORY
Bourne	/bin/sh Of /usr/bin/sh
Bash	/bin/Bash Of /usr/bin/Bash
Korn	/bin/ksh Of /usr/bin/ksh

Shell Scripts

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by a pound sign or hash mark, #, describing the steps. There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, variables to read and store data, and the script may include *functions*.

We are going to write a lot of scripts in the next several hundred pages, and we should always start with a *clear goal* in mind. With a clear goal, we have a specific purpose for the script, and we have a set of expected results. We will also hit on some tips, tricks, and, of course, the gotchas in solving a challenge one way as opposed to another to get the same result. All techniques are not created equal.

Shell scripts and functions are both *interpreted*. This means they are not compiled. Both shell scripts and functions are ASCII text that is read by the shell command interpreter. When we execute a shell script, or function, a command interpreter goes through the ASCII text line-by-line, loop-by-loop, test-by-test, and executes each statement as each line is reached from the top to the bottom.

Functions

A function is written in much the same way as a shell script but is different in that it is defined, or written, within a shell script most of the time, and is called within the

script. This way we can write a piece of code, which is used over and over, just once and use it without having to rewrite the code every time. We just call the function instead.

We can also define functions at the system level that is always available in our environment, but this is a topic for later discussion.

A function has the following form:

```
function function_name
{
     commands to execute
}
or
function_name ()
{
     commands to execute
}
```

When we write functions into our scripts we must remember to declare, or write, the function *before* we use it. The function must appear above the command statement calling the function. We can't use something that does not yet exist.

Running a Shell Script

A shell script can be executed in the following ways:

ksh shell_script_name

will create a Korn shell and execute the shell_script_name in the newly created Korn shell environment. The same is true for sh and Bash shells.

shell_script_name

will execute shell_script_name *if the execution bit is set on the file* (see the manual page on the chmod command, **man chmod**). The script will execute in the shell that is *declared* on the first line of the shell script. If no shell is declared on the first line of the shell script, it will execute in the default shell, which is the user's system-defined shell. Executing in an unintended shell may result in a failure and give unpredictable results.

COMMAND	DESCRIPTION
#!/bin/sh Of #!/usr/bin/sh	Declares a Bourne shell
#!/bin/ksh 0f #!/usr/bin/ksh	Declares a Korn shell
#!/bin/csh 0f #!/usr/bin/csh	Declares a C shell
#!/bin/Bash Of #!/usr/bin/Bash	Declares a Bourne-Again (Bash) shell

Table 1-2 Different Types of Shells to Declare

Declare the Shell in the Shell Script

Declare the shell! If we want to have complete control over how a shell script is going to run and in which shell it is to execute, we must *declare* the shell in *the first line of the script*. If no shell is declared, the script will execute in the default shell, defined by the system for the user executing the shell script. If the script was written, for example, to execute in Bash shell, Bash, and the default shell for the user executing the shell script is the C shell, csh, the script will most likely have a failure during execution. To declare a shell, one of the declaration statements in Table 1-2 must appear on the *first line* of the shell script.

Comments and Style in Shell Scripts

Making good comments in our scripts is stressed throughout this book. What is intuitively obvious to us may be total Greek to others who follow in our footsteps. We have to write code that is readable and has an easy flow. This involves writing a script that is easy to read and easy to maintain, which means that it must have plenty of comments describing the steps. For the most part, the person who writes the shell script is not the one who has to maintain it. There is nothing worse than having to hack through someone else's code that has no comments to find out what each step is supposed to do. It can be tough enough to modify the script in the first place, but having to figure out the mindset of the author of the script will sometimes make us think about rewriting the entire shell script from scratch. We can avoid this by writing a clearly readable script and inserting plenty of comments describing what our philosophy is and how we are using the input, output, variables, and files.

For good style in our command statements, we need it to be readable. For this reason it is sometimes better, for instance, to separate a command statement onto three separate lines instead of stringing, or *piping*, everything together on the same line of code; it may be just too difficult to follow the pipe and understand what the expected result should be for a new script writer. However, in some cases it is more desirable to create a long pipe. But, again, it should have comments describing our thinking step by step. This way someone later will look at our code and say, "Hey, now that's a groovy way to do that."

7

Command readability and step-by-step comments are just the very basics of a well-written script. Using a lot of comments will make our life much easier when we have to come back to the code after not looking at it for six months, and believe me; we will look at the code again. Comment everything! This includes, but is not limited to, describing what our variables and files are used for, describing what loops are doing, describing each test, maybe including expected results and how we are manipulating the data and the many data fields. A hash mark, #, precedes each line of a comment.

The script stub that follows is on this book's companion web site at www.wiley.com/ go/michael2e. The name is script.stub. It has all the comments ready to get started writing a shell script. The script.stub file can be copied to a new filename. Edit the new filename, and start writing code. The script.stub file is shown in Listing 1-1.

```
#!/bin/Bash
#
# SCRIPT: NAME_of_SCRIPT
# AUTHOR: AUTHORS NAME
# DATE: DATE of CREATION
# REV: 1.1.A (Valid are A, B, D, T and P)
             (For Alpha, Beta, Dev, Test and Production)
# PLATFORM: (SPECIFY: AIX, HP-UX, Linux, OpenBSD, Solaris
#
                   or Not platform dependent)
#
# PURPOSE: Give a clear, and if necessary, long, description of the
#
       purpose of the shell script. This will also help you stay
        focused on the task at hand.
#
#
# REV LIST:
#
       DATE: DATE_of_REVISION
       BY: AUTHOR of MODIFICATION
#
       MODIFICATION: Describe what was modified, new features, etc--
#
#
# set -n # Uncomment to check script syntax, without execution.
       # NOTE: Do not forget to put the comment back in or
#
        # the shell script will not execute!
#
# set -x # Uncomment to debug this shell script
#
DEFINE FILES AND VARIABLES HERE
***********
```

Listing 1-1 script.stub shell script starter listing

Listing 1-1 (continued)

The shell script starter shown in Listing 1-1 gives you the framework to start writing the shell script with sections to declare variables and files, create functions, and write the final section, BEGINNING OF MAIN, where the main body of the shell script is written.

Control Structures

The following control structures will be used extensively.

if ... then statement

```
if [ test_command ] then commands
```

if ... then ... else statement

```
if [ test_command ]
then
commands
else
commands
```

fi

if ... then ... elif ... (else) statement

fi

for ... in statement

```
for loop_variable in argument_list
do
```

commands

done

while statement

```
while test_condition_is_true
do
```

commands

done

until statement

```
until test_condition_is_true do
```

commands

done

```
case statement
       case $variable in
       match_1)
               commands_to_execute_for_1
               ;;
       match_2)
               commands_to_execute_for_2
               ;;
       match_3)
               commands_to_execute_for_3
               ;;
       *)
               (Optional - any other value)
               commands_to_execute_for_no_match
               ;;
  esac
  NOTE The last part of the case statement, shown here,
             *)
                 commands_to_execute_for_no_match
       ;;
```

is optional.

Using break, continue, exit, and return

It is sometimes necessary to *break* out of a for or while loop, *continue* in the next block of code, *exit* completely out of the script, or *return* a function's result back to the script that called the function.

- The break command is used to terminate the execution of the entire loop, after completing the execution of all the lines of code up to the break statement. It then steps down to the code following the end of the loop.
- The continue command is used to transfer control to the next set of code, but it continues execution of the loop.
- The exit command will do just what one would expect: it exits the entire script. An integer may be added to an exit command (for example, exit 0), which will be sent as the return code.
- The return command is used in a function to send data back, or return a result or return code, to the calling script.

Here Document

A *here document* is used to redirect input *into* an interactive shell script or program. We can run an interactive program within a shell script without user action by supplying the required input for the interactive program, or interactive shell script. This is why it is called a here document: the required input is here, as opposed to somewhere else.

This is the syntax for a here document:

```
program_name <<LABEL
Program_Input_1
Program_Input_2
Program_Input_3
Program_Input_#
LABEL
Example:
/usr/local/bin/My_program << EOF
Randy
Robin
Rusty
Jim
EOF</pre>
```

Notice in the here documents that there are *no spaces* in the program input lines, between the two EOF labels. If a space is added to the input, the here document may fail. The input that is supplied must be the *exact* data that the program is expecting, and many programs will fail if spaces are added to the input.

Shell Script Commands

The basis for the shell script is the automation of a series of commands. We can execute most any command in a shell script that we can execute from the command line. (One exception is trying to set an execution *suid* or *sgid*, *sticky bit*, within a shell script; it is not supported for security reasons.) For commands that are executed often, we reduce errors by putting the commands in a shell script. We will eliminate typos and missed device definitions, and we can do conditional tests that can ensure there are not any failures due to unexpected input or output. Commands and command structure will be covered extensively throughout this book.

Most of the commands shown in Table 1-3 are used at some point in this book, depending on the task we are working on in each chapter.

COMMAND	DESCRIPTION
passwd	Changes user password
pwd	Prints current directory
cd	Changes directory
ls	Lists files in a directory
wildcards	* matches any number of characters; ? matches a single character
file	Prints the type of file
cat	Displays the contents of a file
pr	Displays the contents of a file
pg 01 page	Displays the contents of a file one page at a time
more	Displays the contents of a file one page at a time
clear	Clears the screen
cp or copy	Copies a file
chown	Changes the owner of a file
chgrp	Changes the group of a file
chmod	Changes file modes, permissions
rm	Removes a file from the system
mv	Renames a file
mkdir	Creates a directory

Table 1-3 UNIX Commands Review

COMMAND	DESCRIPTION
rmdir	Removes a directory
grep	Pattern matching
egrep	grep command for extended regular expressions
find	Locates files and directories
>>	Appends to the end of a file
>	Redirects, creates, or overwrites a file
	Strings commands together, known as a pipe
	Logical OR — command1 command2 — execute command2 if command1 fails
&	Executes in background
δε δε	Logical AND — command1 && command2 — execute command2 if command1 succeeds
date	Displays the system date and time
echo	Writes strings to standard output
sleep	Halts execution for the specified number of seconds
WC	Counts the number of words, lines, and characters in a file
head	Views the top of a file
tail	Views the end of a file
diff	Compares two files
sdiff	Compares two files side by side (requires 132-character display)
spell	Spell checker
lp, lpr, enq, qprt	Prints a file
lpstat	Status of system print queues
enable	Enables, or starts, a print queue
disable	Disables, or stops, a print queue
cal	Displays a calendar
who	Displays information about users on the system
W	Extended who command

Table 1-3 (continued)

(continued)

COMMAND	DESCRIPTION
whoami	Displays <pre>\$LOGNAME or \$USER environment parameters</pre>
who am I	Displays login name, terminal, login date/time, and where logged in
f,finger	Displays information about logged-in users, including the users .plan and .project
talk	Enables two users to have a split-screen conversation
write	Displays a message on a user's screen
wall	Displays a message on all logged-in users' screens
rwall	Displays a message to all users on a remote host
rsh Oľ remsh	Executes a command, or login, on a remote host
df	Displays filesystem statistics
ps	Displays information on currently running processes
netstat	Shows network status
vmstat	Shows virtual memory status
iostat	Shows input/output status
uname	Shows name of the current operating system, as well as machine information
sar	Reports system activity
basename	Displays base filename of a string parameter
man	Displays the online reference manual
su	Switches to another user, also known as super-user
cut	Writes out selected characters
awk	Programming language to parse characters
sed	Programming language for character substitution
vi	Starts the vi editor
emacs	Starts the emacs editor

Table 1-3 (continued)

Symbol Commands

The symbols shown in Table 1-4 are actually commands, and are used extensively in this book.

COMMAND	DESCRIPTION
()	Runs the enclosed command in a sub-shell
(())	Evaluates and assigns value to a variable and does math in a shell
\$(())	Evaluates the enclosed expression
[]	Same as the test command
< >	Used for string comparison
\$()	Command substitution
'command'	Command substitution

Table 1-4 Sym	bol Commands
---------------	--------------

Variables

A *variable* is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data. A variable is nothing more than a pointer to the actual data. We are going to use variables so much in our scripts that it will be unusual for us not to use them. In this book we are always going to specify a variable in uppercase — for example, UPPERCASE. Using uppercase variable names is not recommended in the real world of shell programming, though, because these uppercase variables may step on system environment variables, which are also in uppercase. Uppercase variables are used in this book to emphasize the variables and to make them stand out in the code. When you write your own shell scripts or modify the scripts in this book, make the variables lowercase text. To assign a variable to point to data, we use UPPERCASE="value_to_assign" as the assignment syntax. To access the data that the variable, UPPERCASE, is pointing to, we must add a dollar sign, \$, as a prefix — for example, \$UPPERCASE. To view the data assigned to the variable, we use echo \$UPPERCASE, print \$UPPERCASE for variables, or cat \$UPPERCASE, if the variable is pointing to a file, as a command structure.

Command-Line Arguments

The command-line arguments \$1, \$2, \$3,...\$9 are positional parameters, with \$0 pointing to the actual command, program, shell script, or function and \$1, \$2, \$3, ...\$9 as the arguments to the command.

The positional parameters, \$0, \$2, and so on in a function are for the function's use and may not be in the environment of the shell script that is calling the function. Where a variable is known in a function or shell script is called the *scope* of the variable.

shift Command

The shift command is used to move positional parameters to the left; for example, shift causes \$2 to become \$1. We can also add a number to the shift command to move the positions more than one position; for example, shift 3 causes \$4 to move to the \$1 position.

Sometimes we encounter situations where we have an unknown or varying number of arguments passed to a shell script or function, \$1, \$2, \$3... (also known as positional parameters). Using the shift command is a good way of processing each positional parameter in the order they are listed.

To further explain the shift command, we will show how to process an unknown number of arguments passed to the shell script shown in Listing 1-2. Try to follow through this example shell script structure. This script is using the shift command to process an unknown number of command-line arguments, or positional parameters. In this script we will refer to these as *tokens*.

```
#!/usr/bin/sh
#
# SCRIPT: shifting.sh
#
# AUTHOR: Randy Michael
# DATE: 12/30/2007
#
# REV: 1.1.A
#
# PLATFORM: Not platform dependent
#
# PURPOSE: This script is used to process all of the tokens which
# are pointed to by the command-line arguments, $1, $2, $3,etc...
#
# REV. LIST:
# Initialize all variables
         # Initialize the TOTAL counter to zero
TOTAL=0
# Start a while loop
while true
do
    TOTAL=`expr $TOTAL + 1` # A little math in the
                              # shell script, a running
                              # total of tokens processed.
```

Listing 1-2 Example of using the shift command

Listing 1-2 (continued)

We will go through similar examples of the shift command in great detail later in the book.

Special Parameters \$* and \$@

There are special parameters that allow accessing *all* the command-line arguments at once. \$* and \$@ both will act the same unless they are enclosed in double quotes, " ".

Special Parameter Definitions

- The \$* special parameter specifies all command-line arguments.
- The \$@ special parameter also specifies all command-line arguments.
- The "\$*" special parameter takes the entire list as one argument with spaces between.
- The "\$@" special parameter takes the entire list and separates it into separate arguments.

We can rewrite the shell script shown in Listing 1-2 to process an unknown number of command-line arguments with either the \$* or \$@ special parameters, as shown in Listing 1-3.

```
#!/usr/bin/sh
#
# SCRIPT: shifting.sh
# AUTHOR: Randy Michael
# DATE: 12-31-2007
# REV: 1.1.A
# PLATFORM: Not platform dependent
#
```

Listing 1-3 Example using the special parameter \$*

Listing 1-3 (continued)

We could have also used the \$@ special parameter just as easily. As we see in the preceding code segment, the use of the \$@ or \$* is an alternative solution to the same problem, and it was less code to write. Either technique accomplishes the same task.

Double Quotes, Forward Tics, and Back Tics

How do we know which one of these to use in our scripts, functions, and command statements? This decision causes the most confusion in writing scripts. We are going to set this straight now.

Depending on what the task is and the output desired, it is very important to use the correct enclosure. Failure to use these correctly will give unpredictable results.

We use ", double quotes, in a statement where we want to allow character or command substitution. Double quotes are required when defining a variable with data that contains white space, as shown here.

```
NAME="Randal K. Michael"
```

If the double quotes are missing we get the following error.

NAME=Randal K. Michael -Bash: K.: command not found

We use ', forward tics (single quotes), in a statement where we do *not* want character or command substitution. Enclosing in ', forward tics, is intended to use the *literal text* in the variable or command statement, without any substitution. All special meanings and functions are removed. It is also used when you want a variable reread each time it is used; for example, '\$PWD' is used a lot in processing the PS1 command-line prompt. Additionally, preceding the same string with a backslash, \, also removes the special meaning of a character, or string.

We use `, back tics, in a statement where we want to execute a command, or script, and have its output substituted instead; this is *command substitution*. The ` key is located to the left of the **1** key, and below the Escape key, Esc, on most keyboards. Command substitution is also accomplished by using the \$(command) command syntax. We are going to see many different examples of these throughout this book.

Using awk on Solaris

We use awk a lot in this book to parse through lines of text. There is one special case where, on Solaris, we must to use **nawk** instead. If we need to specify a field separator other than a blank space, which is the default field delimiter, using awk -F :, for example, the awk statement will fail on a Solaris machine. To get around this problem, use nawk if we find the UNIX flavor is Solaris. Add the following code segment to the variable declaration section of all your shell scripts to eliminate the problem:

```
# Setup the correct awk usage. Solaris needs to
# use nawk instead of awk.
case $(uname) in
SunOS) alias awk=nawk
    ;;
esac
```

Using the echo Command Correctly

We use the **echo** command to display text. The echo command allows a lot of cursor control using backslash operators: \n for a new line, \c to continue on the same line, \b to backspace the cursor, \t for a tab, \r for a carriage return, and \v to move vertically one line. In Korn shell the echo command recognizes these command options by default. In Bash shell we must add the -e switch to the echo command, echo -e " \n " for one new line.

We can query the system for the executing shell by querying the \$SHELL shell variable in the script. Many Linux distributions will execute in a Bash shell even though we specify Korn shell on the very first line of the script. Because Bash shell requires the use of the echo -e switch to enable the backslash operators, we can use a case statement to *alias* the echo command to echo -e if the executing shell is */bin/Bash. Now when we need to use the echo command, we are assured it will display text correctly.

Add the following code segment to all your Korn shell scripts in the variable declaration section, and this little problem is resolved:

- # Set up the correct echo command usage. Many Linux
- # distributions will execute in Bash even if the
- # script specifies Korn shell. Bash shell requires
- # we use echo -e when we use n, c, etc.

```
case $SHELL in
*/bin/Bash) alias echo="echo -e"
;;
esac
```

Math in a Shell Script

We can do arithmetic in a shell script easily. The shell let command and the ((expr)) command expressions are the most commonly used methods to evaluate an integer expression. Later we will also cover the bc function to do floating-point arithmetic.

Operators

The shells use arithmetic operators from the C programming language (see Table 1-5), in decreasing order of precedence.

Table	1-5	Math	Operators
-------	-----	------	-----------

OPERATOR	DESCRIPTION
++ 	Auto-increment and auto-decrement, both prefix and postfix
+	Unary plus
_	Unary minus
! ~	Logical negation; binary inversion (one's complement)
* / %	Multiplication; division; modulus (remainder)
+ -	Addition; subtraction
<< >>	Bitwise left shift; bitwise right shift
<= >=	Less than or equal to; greater than or equal to
< >	Less than; greater than
== !=	Equality; inequality (both evaluated left to right)
óc	Bitwise AND
^	Bitwise exclusive OR
	Bitwise OR
&&	Logical AND
	Logical OR

A lot of these math operators are used in the book, but not all. In this book we try to keep things very straightforward and not confuse you with obscure expressions.

Built-In Mathematical Functions

The shells provide access to the standard set of mathematical functions. They are called using C function call syntax. Table 1-6 shows a list of shell functions.

NAME	FUNCTION
abs	Absolute value
log	Natural logarithm
acos	Arc cosine
sin	Sine
asin	Arc sine
sinh	Hyperbolic sine
cos	Cosine
sqrt	Square root
cosh	Hyperbolic cosine
tan	Tangent
exp	Exponential function
tanh	Hyperbolic tangent
int	Integer part of floating-point number

Table 1-6 Built-In Shell Functions

We do not have any shell scripts in this book that use any of these built-in shell functions except for the int function to extract the integer portion of a floating-point number.

File Permissions, suid and sgid Programs

After writing a shell script we must remember to set the file permissions to make it *executable*. We use the chmod command to change the file's mode of operation. In addition to making the script executable, it is also possible to change the mode of the file to always execute as a particular user (suid) or to always execute as a member of a particular system group (sgid). This is called setting the *sticky bit*. If you try to suid or sgid a shell script, it is ignored for security reasons.

Setting a program to always execute as a particular user, or member of a certain group, is often used to allow all users, or a set of users, to run a program in the proper environment. As an example, most system-check programs need to run as an administrative user, sometimes root. We do not want to pass out passwords, so we can just make the program always execute as root and it makes everyone's life easier. We can use the options shown in Table 1-7 in setting file permissions. Also, please review the chmod man page, man chmod.

By using combinations from the chmod command options, you can set the permissions on a file or directory to anything that you want. Remember that setting a shell script to suid or sgid is ignored by the system.

4000	Sets user ID on execution
2000	Sets group ID on execution
1000	Sets the link permission to directories or sets the save-text attribute for files
0400	Permits read by owner
0200	Permits write by owner
0100	Permits execute or search by owner
0040	Permits read by group
0020	Permits write by group
0010	Permits execute or search by group
0004	Permits read by others
0002	Permits write by others
0001	Permits execute or search by others

Table 1-7 chmod Permission Options

chmod Command Syntax for Each Purpose

The chmod command can be used with the octal file permission representation or by r, w, x notation. Both of these examples produce the same result.

To Make a Script Executable

chmod 754 my_script.sh

or

chmod u+rwx,g+rx,o+r my_script.ksh

The owner can read, write, and execute. The group can read and execute. The world can read.

To Set a Program to Always Execute as the Owner

chmod 4755 my_program

The program will always execute as the owner of the file if it is not a shell script. The owner can read, write, and execute. The group can read and execute. The world can read and execute. So, no matter who executes this file, it will always execute as if the owner actually executed the program.

To Set a Program to Always Execute as a Member of the File Owner's Group

```
chmod 2755 my_program
```

The program will always execute as a member of the file's group, as long as the file is not a shell script. The owner of the file can read, write, and execute. The group can read and execute. The world can read and execute. So, no matter who executes this program, it will always execute as a member of the file's group.

To Set a Program to Always Execute as Both the File Owner and the File Owner's Group

chmod 6755 my_program

The program will always execute as the file's owner and as a member of the file owner's group, as long as the program is not a shell script. The owner of the file can read, write, and execute. The group can read and execute. The world can read and execute. No matter who executes this program, it will always execute as the file owner and as a member of the file owner's group.

Running Commands on a Remote Host

We sometimes want to execute a command on a remote host and have the result displayed locally. An example would be getting filesystem statistics from a group of machines. We can do this with the **rsh** command. The syntax is rsh hostname command_to_execute. This is a handy little tool but two system files will need to be set up on all of the hosts before the rsh command will work. The files are .rhosts, which would be created in the user's home directory and have the file permissions of 600 (permission to read and write by the owner only), and the /etc/hosts.equiv file.

For security reasons the .rhosts and hosts.equiv files, by default, are not set up to allow the execution of a remote shell. *Be careful*! The systems' security could be threatened. Refer to each operating system's documentation for details on setting up these files.

Speaking of security, a better solution is to use Open Secure Shell (OpenSSH) instead of rsh. OpenSSH is a freeware encrypted replacement for rsh, telnet, and ftp, for the most part. To execute a command on another machine using OpenSSH, use the following syntax:

```
ssh user@hostname command_to_execute
```

or

```
ssh -1 user hostname command_to_execute
```

This command prompts you for a password if the encryption key pairs have not been set up. Setting up the key pair relationships manually usually takes a few minutes, or you can use one of the keyit scripts shown in Listings 1-4 and 1-5 to set up the keys for you. The details of the procedure are shown in the ssh manual page (**man ssh**). You can download the OpenSSH code from http://www.openssh.org.

The keyit.dsa script in Listing 1-4 will set up DSA keys, if the DSA keys exist.

```
#!/bin/Bash
#
# SCRIPT: keyit.dsa
# PURPOSE: This script is used to set up DSA SSH keys. This script must
# be executed by the user who needs the keys setup.
REM_HOST=$1
cat $HOME/.ssh/id_dsa.pub | ssh $REM_HOST "cat >> ~/.ssh/authorized_keys"
```

Listing 1-4 keyit.dsa script used to set up DSA SSH keys

The keyit.rsa script in Listing 1-5 will set up the RSA keys, if the RSA keys exist.

```
#!/bin/Bash
#
# SCRIPT: keyit.rsa
# PURPOSE: This script is used to set up RSA SSH keys.
This script must be executed by the user who needs the keys setup.
REM_HOST=$1
cat $HOME/.ssh/id_rsa.pub | ssh $REM_HOST "cat >> ~/.ssh/authorized_keys"
```

Listing 1-5 keyit.rsa script used to set up RSA SSH keys

If you need to set up the encryption keys for a new user, first su to that user ID, and then issue one of the following commands.

To set up DSA keys issue this command:

ssh-keygen -t dsa

To set up RSA keys issue this one:

ssh-keygen -t rsa

Read the ssh-keygen man page for more details: man ssh-keygen.

Setting Traps

When a program is terminated before it would normally end, we can catch an exit signal. This is called a *trap*. Table 1-8 lists some of the exit signals.

0	_	Normal termination, end of script
1	SIGHUP	Hang up, line disconnected
2	SIGINT	Terminal interrupt, usually Ctrl + C
3	SIGQUIT	Quit key, child processes to die before terminating
9	SIGKILL	kill -9 command, cannot trap this type of exit status
15	SIGTERM	kill command's default action
19	SIGSTOP	Stop, usually Ctrl + z

Table 1-8 Exit Signals

To see the entire list of supported signals for your operating system, enter the following command:

```
# kill -1 [That's kill -(ell)]
```

This is a really nice tool to use in our shell scripts. On catching a trapped signal we can execute some cleanup commands before we actually exit the shell script. Commands can be executed when a signal is trapped. If the following command statement is added in a shell script, it will print to the screen "EXITING on a TRAPPED SIGNAL" and then make a clean exit on the signals 1, 2, 3, and 15. We cannot trap a kill -9.

trap 'echo "\nEXITING on a TRAPPED SIGNAL"; exit' 1 2 3 15

We can add all sorts of commands that may be needed to clean up before exiting. As an example, we may need to delete a set of files that the shell script created before we exit.

User-Information Commands

Sometimes we need to query the system for some information about users on the system.

who Command

The **who** command gives this output for each logged-in user: *username*, *tty*, *login time*, and *where* the user *logged in from*:

rmichael	pts/0	Mar 13 10:24	192.168.1.104
root	pts/1	Mar 15 10:43	(yogi)

w Command

The **w** command is really an extended who. The output looks like the following:

12:29PM	up 27 da	ys, 21:53,2	users,	load average	1.03, 1.17, 1.09
User	tty	login@	idle	JCPU PCP	U what
rmichael	pts/0	Mon10AM 0	3:00	1	W
root	pts/1	10:42AM 37	5:12	5:12	tar

Notice that the top line of the preceding output is the same as the output of the **uptime** command. The w command gives a more detailed output than the who command by listing job process time and total user process time, but it does not reveal *where* the users have logged in *from*. We often are interested in this for security purposes. One nice thing about the w command's output is that it also lists *what* the users are doing at the instant the command w is entered. This can be very useful.

last Command

The **last** command shows the history of who has logged in to the system since the wtmp file was created. This is a good tool when you need to do a little investigation of who logged in to the system and when. The following is example output:

root	ftp	booboo	Aug	06	19:22	-	19:23	(00:01)
root	pts/3	mrranger	Aug	06	18:45		still	logged in.
root	pts/2	mrranger	Aug	06	18:45		still	logged in.
root	pts/1	mrranger	Aug	06	18:44		still	logged in.
root	pts/0	mrranger	Aug	06	18:44		still	logged in.
root	pts/0	mrranger	Aug	06	18:43	-	18:44	(00:01)
root	ftp	booboo	Aug	06	18:19	-	18:20	(00:00)
root	ftp	booboo	Aug	06	18:18	-	18:18	(00:00)
root	tty0		Aug	06	18:06		still	logged in.
root	tty0		Aug	02	12:24	-	17:59	(4+05:34)
reboot	~		Aug	02	12:00			
shutdown	tty0		Jul	31	23 : 23			
root	ftp	booboo	Jul	31	21:19	-	21:19	(00:00)
root	ftp	bambam	Jul	31	21:19	-	21:19	(00:00)
root	ftp	booboo	Jul	31	20:42	-	20:42	(00:00)
root	ftp	bambam	Jul	31	20:41	-	20:42	(00:00)

The output of the last command shows the username, the login port, where the user logged in from, the time of the login/logout, and the duration of the login session.

ps Command

The **ps** command will show information about current system processes. The ps command has many switches that will change what we look at. Table 1-9 lists some common command options.

ps	The user's currently running processes
ps -f	Full listing of the user's currently running processes
ps -ef	Full listing of all processes, except kernel processes
ps -A	All processes, including kernel processes
ps -Kf	Full listing of kernel processes
ps auxw	Wide listing sorted by percentage of CPU usage, CPU

Table 1-9 Common ps Command Options

Communicating with Users

Communicate with the system's users and let them know what is going on! All Systems Administrators have the *maintenance window* where we can finally get control and handle some offline tasks. This is just one example of a need to communicate with the system users, if any are still logged in.

The most common way to get information to the system users is to use the /etc/motd file. This file is displayed each time the user logs in. If users stay logged in for days at a time they will not see any new messages of the day. This is one reason why real-time communication is needed. The commands shown in Table 1-10 allow communication to, or between, users who are currently logged into the system.

wall	Writes a message on the screen of all logged-in users on the <i>local</i> host.
rwall	Writes a message on the screen of all logged-in users on a <i>remote</i> host.
write	Writes a message to an individual user. The user must currently be logged in.
talk	Starts an interactive program that allows two users to have a conversation. The screen is split in two, and both users can see what each person is typing.

Table 1-10 Commands for Real-Time User Communication	Table 1-10	Commands	for Real-Time	User	Communication
--	------------	----------	---------------	------	---------------

NOTE When using these commands, be aware that if a user is using a program – for example, an accounting software package – and has that program's screen on the terminal, the user might not get the message or the user's screen may become scrambled.

Uppercase or Lowercase Text for Easy Testing

We often need to test text strings like filenames, variables, file text, and so on, for comparison. It can sometimes vary so widely that it is easier to uppercase or lowercase the text for ease of comparison. The **tr** and **typeset** commands can be used to uppercase and lowercase text. This makes testing for things like variable input a breeze. Following are some examples of using the tr command:

Variable values:

Expected input	TRUE
Real input	TRUE
Possible input	true TRUE True True, and so on

Upcasing:

UPCASEVAR=\$(echo \$VARIABLE | tr '[a-z]' '[A-Z]')

Downcasing:

DOWNCASEVAR=\$(echo \$VARIABLE | tr '[A-Z]' '[a-z]')

In the preceding example of the tr command, we echo the string and use a pipe (|) to send the output of the echo statement to the tr command. As the preceding examples show, uppercasing uses '[a-z]' + [A-z]'.

NOTE The single quotes are required around the square brackets.

```
'[a-z]' '[A-Z]' Used for lower to uppercase
'[A-Z]' '[a-z]' Used for upper to lowercase
```

No matter what the user input is, we will always have the *stable* input of TRUE, if uppercased, and true, if lowercased. This reduces our code testing and also helps the readability of the script.

We can also use typeset to control the attributes of a variable in the shell. In the previous example we are using the variable VARIABLE. We can set the attribute to always translate all of the characters to uppercase or lowercase. To set the case attribute of the variable VARIABLE to always translate characters assigned to it to uppercase, we use

typeset -u VARIABLE

The -u switch to the typeset command is used for uppercase. After we set the attribute of the variable VARIABLE, using the typeset command, anytime we assign text characters to VARIABLE they are automatically translated to uppercase characters. Example:

```
typeset -u VARIABLE
VARIABLE="True"
echo $VARIABLE
```

TRUE

To set the case attribute of the variable VARIABLE to always translate characters to lowercase, we use

```
typeset -1 VARIABLE
Example:
typeset -1 VARIABLE
VARIABLE="True"
echo $VARIABLE
true
```

Check the Return Code

Whenever we run a command there is a response back from the system about the last command that was executed, known as the *return code*. If the command was successful the return code will be 0, zero. If it was *not* successful the return will be something other than 0, zero. To check the return code we look at the value of the \$? shell variable.

As an example, we want to check if the /usr/local/bin directory exists. Each of these blocks of code accomplishes the exact same thing:

```
test -d /usr/local/bin
if [ "$?" -eq 0 ] # Check the return code
then  # The return code is zero
    echo '/usr/local/bin does exist'
else  # The return code is NOT zero
    echo '/usr/local/bin does NOT exist'
fi
```

or

```
if test -d /usr/local/bin
  then
                # The return code is zero
       echo '/usr/local/bin does exist'
                # The return code is NOT zero
  else
       echo '/usr/local/bin does NOT exist'
  fi
or
  if [ -d /usr/local/bin ]
  then
                # The return code is zero
       echo '/usr/local/bin does exist'
  else
                # The return code is NOT zero
       echo '/usr/local/bin does NOT exist'
  fi
```

Notice that we checked the return code using \$? once. The other examples use the control structure's built-in test. The built-in tests do the same thing of processing the return code, but the built-in tests hide this step in the process. All three of the previous examples give the exact same result. This is just a matter of personal choice and readability.

Time-Based Script Execution

We write a lot of shell scripts that we want to execute on a timed interval or run once at a specific time. This section addresses these needs with several examples.

Cron Tables

A *cron table* is a system file that is read every minute by the system and will execute any entry that is scheduled to execute in that minute. By default, any user can create a cron table with the crontab -e command, but the Systems Administrator can control which users are allowed to create and edit cron tables with the cron.allow and cron.deny files. When a user creates his or her own cron table, the commands, programs, or scripts will execute in that user's environment. It is the same thing as running the user's \$HOME/.profile before executing the command.

The crontab -e command starts the default text editor, vi or emacs, on the user's cron table.

NOTE When using the crontab command, the current user ID is the crontable that is acted on. To list the contents of the current user's crontable, issue the crontab -1 command.

Cron Table Entry Syntax

It is important to know what each field in a cron table entry is used for. The following cron table entry executes the script, /usr/local/bin/somescript.ksh, at 3:15 a.m., January 8, on any day of the week that January 8 falls on. Notice that we used a *wildcard* for the weekday field.

				Minute of the Hour (0-59)
_				Hour of the Day (0-23)
	_			Day of the Month (1-31)
		_		Month of the Year (1-12)
				Day of the Week (0 - 6 for Sunday through Saturday)
				Command to Execute (Full path is required)
15 3	8 8	1	*	/usr/local/bin/somescript.Bash 2>&1 >/dev/null

The following cron table entry is another example:

0 0 1 1 * /usr/bin/banner "Happy New Year" > /dev/console

At midnight on New Year's Eve, 00:00 hours on January 1, on any weekday, this cron table entry writes to the system's console (/dev/console) **Happy New Year** in large banner letters. Wildcard characters are defined in this table.

Wildcards:

*	Match any number of characters
?	Match a single character

at Command

Like a cron table, the **at** command executes commands based on time. Using the at command, we can schedule a job to run *once*, at a specific time. When the job is executed, the at command will send an email of the standard output and standard error to the user who scheduled the job to run, unless the output is redirected. As Systems Administrators we can control which users are allowed to schedule jobs with the at.allow and at.deny files. Refer to each operating system's man pages before modifying these files, and refer to them to learn the many ways to use the at command for timed controlled command execution. To execute a command in 10 minutes, use the following syntax:

echo '/usr/local/bin/somescript.Bash' | at now + 10 minutes

You can also use the at command interactively by typing the following:

at now + 10 minutes Enter

Then type the full pathname of the command to execute, press Enter, and then press Ctrl+D. Here is an example:

```
at now + 10 minutes
/usr/local/bin/somescript.Bash
Ctrl+D
```

For more at command options, see the at command manual page, man at.

Output Control

How is the script going to run? Where will the output go? These questions come under output control.

Silent Running

To execute a script in *silent mode* we can use the following syntax:

```
/PATH/script_name 2>&1 > /dev/null
```

In this command statement the script_name shell script will execute without any output to the screen. The reason for this is that the command is terminated with the following:

 $2 \ge 1 > /dev/null$

By terminating a command like this it redirects standard error (stderr), specified by file descriptor 2, to standard output (stdout), specified by file descriptor 1. Then we have another redirection to /dev/null, which sends all of the output to the bit bucket.

We can call this *silent running*. This means that there is absolutely no output from the script going to our screen. Inside the script there may be some output directed to files or devices, a particular terminal, or even the system's console, /dev/console, but none to the user's screen. This is especially useful when executing a script from one of the system's cron tables.

In the following example cron table entry, we want to execute a script named /usr/local/bin/systemcheck.ksh, which needs to run as the root user, every 15 minutes, 24 hours a day, 7 days a week, and not have any output to the screen. There will not be any screen output because we are going to end the cron table entry with the following:

 $2 \ge 1 > /dev/null$

Inside the script it may do some kind of notification such as paging staff or sending output to the system's console, writing to a file or a tape device, but output such as echo "Hello World" would go to the *bit bucket*. But echo "Hello World" > /dev/console *would* go to the system's defined console if this command statement was within the shell script.

This cron table entry would need to be placed in the root cron table (must be logged in as the root user) with the following syntax:

5,20,35,50 * * * * /usr/local/bin/systemcheck.ksh 2>&1 >/dev/null

NOTE Most system check-type scripts need to be in the root cron table. Of course, a user must be logged in as root to edit root's cron table.

The preceding cron table entry would execute the /usr/local/bin/systemcheck .ksh every 15 minutes, at 5, 20, 35, and 50 minutes, each hour, 24 hours a day, 7 days a week. It would not produce any output to the screen due to the final 2>&1 > /dev/null. Of course, the minutes selected to execute can be any. We sometimes want to spread out execution times in the cron tables so that we don't have a lot of CPU-intensive scripts and programs starting execution at the same time.

Using getopts to Parse Command-Line Arguments

The getopts command is built into the shell. It retrieves valid command-line options specified by a single character preceded by a – (minus sign) or + (plus sign). To specify that a command switch requires an argument to the switch, it is followed by a : (colon). If the switch does not require any argument, the : should be omitted. All of the options put together are called the OptionString, and this is followed by some variable name. The argument for each switch is stored in a variable called OPTARG. If the entire OptionString is preceded by a : (colon), any unmatched switch option causes a ? to be loaded into the VARIABLE. The form of the command follows:

```
getopts OptionString VARIABLE [ Argument ... ]
```

The easiest way to explain this is with an example. For a script we need seconds, minutes, hours, days, and a process to monitor. For each one of these we want to supply an argument — that is, -s 5 -m10 -p my_backup. In this we are specifying 5 seconds, 10 minutes, and the process is my_backup. Notice that there does not have to be a space between the switch and the argument, and they can be entered in any order. This is what makes getopts so great! The command line to set up our example looks like this:

```
SECS=0 # Initialize all to zero
MINUTES=0
HOURS=0
DAYS=0
PROCESS= # Initialize to null
```

```
while getopts :s:m:h:d:p: TIMED 2>/dev/null
do
    case $TIMED in
       s) SECS=$OPTARG
          ;;
       m) (( MINUTES = $OPTARG * 60 ))
          ;;
       h) (( HOURS = $OPTARG * 3600 ))
          . .
       d) (( DAYS = $OPTARG * 86400 ))
          ;;
       p) PROCESS=$OPTARG
          ;;
       \?) usage
          exit 1
           ;;
    esac
done
(( TOTAL_SECONDS = SECONDS + MINUTES + HOURS + DAYS ))
```

There are a few things to note in the getopts command. The getopts command needs to be part of a while loop with a case statement within the loop for this example. On each option we specified, s, m, h, d, and p, we added a : (colon) after each switch. This tells getopts that an argument is required. The : (colon) before the OptionString list tells getopts that if an unspecified option is given, to set the TIMED variable to the ? character. This allows us to call the usage function and exit with a return code of 1. The first thing to be careful of is that getopts does not care what arguments it receives, so we have to take action if we want to exit. The last thing to note is that the first line of the while loop has output redirection of standard error (file descriptor 2) to the bit bucket. Anytime an unexpected argument is encountered, getopts sends a message to standard error (file descriptor 2). Because we expect this to happen, we can just ignore the messages and discard them to /dev/null. We will study getopts a lot in this book.

Making a Co-Process with Background Function

We also need to cover setting up a co-process. A co-process is a communications link between a foreground and a background process. The most common question is *why* is this needed? In one of the scripts we are going to call a function that will handle all of the process monitoring for us while we do the timing control in the main script. The problem arises because *we need to run this function in the background and it has an infinite loop*. Within this background process-monitoring function there is an infinite loop. Without the ability to tell the loop to break out, it will continue to execute on its own after the main script, and function, is interrupted. We know what this causes — *one or more defunct processes*! From the main script we need a way to communicate with this loop, thus background function, to tell it to break out of the loop and exit the function cleanly when the countdown is complete and if the script is interrupted, Ctrl+C. To

solve this little problem we kick off our proc_watch function as a *co-process* in the background. How do we do this, you ask? "*Pipe it to the background*" is the simplest way to put it, and that is what it looks like, too. Look at the next example code block in Listing 1-6.

```
*********************
function trap_exit
{
    # Tell the co-process to break out of the loop
    BREAK OUT='Y'
    print -p $BREAK_OUT # Use "print -p" to talk to the co-process
}
function proc_watch
# This function is started as a co-process!!!
   while :
             # Loop forever
   do
        Some Code Here
         read $BREAK_OUT # Do NOT need a "-p" to read!
         if [[ $BREAK_OUT = 'Y' ]]
         then
             return 0
         fi
   done
}
*****
##### Start of Main ########
****
### Set a Trap ###
trap 'trap_exit; exit 2' 1 2 3 15
TOTAL_SECONDS=300
BREAK_OUT='N'
proc_watch &
                # Start proc_watch as a co-process!!!!
PW PID=$1
                 # Process ID of the last background job
until (( TOTAL_SECONDS == 0 ))
do
    (( TOTAL_SECONDs = TOTAL_SECONDS - 1 ))
```

Listing 1-6 Example code using a co-process

Listing 1-6 (continued)

In the code segment in Listing 1-6, we defined two functions. The trap_exit function will execute on exit signals 1, 2, 3, and 15. The other function is the proc_watch function, which is the function that we want to start as a background process. As you can see in proc_watch, it has an infinite loop. If the main script is interrupted without a means to exit the loop, within the function the loop *alone* will continue to execute! To solve this we start the proc_watch as a co-process by "piping it to the background" using *pipe ampersand*, |&, as a suffix. Then when we want to communicate to this co-process background function we use print -p \$VARIABLE_NAME. Inside the co-process function we just use the standard read \$VARIABLE_NAME. This is the mechanism that we are going to use to break out of the loop if the main script is interrupted on a trapped signal; of course, we *cannot* catch a kill -9 with a trap.

Try setting up the scenario described previously with a background function that has an infinite loop. Then press the Ctrl+C key sequence to kill the main script, and do a ps -ef | more. You will see that the background *loop* is still executing! Get the PID, and do a kill -9 on that PID to kill it. Of course, if the loop's exit criteria is ever met, the loop will exit on its own.

Catching a Delayed Command Output

Have you ever had a hard time trying to catch the output of a command that has a delayed output? This can cause a lot of frustration when you *just miss it*! There is a little technique that allows you to catch these delayed responses. The trick is to use an until loop. Look at the code shown here:

```
OUTFILE="/tmp/outfile.out" # Define the output file
cat /dev/null > $OUTFILE # Create a zero size output file
# Start an until loop to catch the delayed response
```

This code segment first defines an output file to store the delayed output data. We start with a zero-sized file and then enter an until loop that will continue until the \$OUTFILE is no longer a zero-sized file, and the until loop exits. The last step is to show the user the data that was captured from the delayed output.

Fastest Ways to Process a File Line-by-Line

Most shell scripts work with files, and some use a file for data input. This next section shows the fastest methods studied in Chapter 2, "24 Ways to Process a File Line-by-Line." The two fastest techniques for processing a file line-by-line are shown in Listings 1-7 and 1-8.

```
function while_read_LINE_bottom_FD_OUT
{
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while read LINE
do
       echo "$LINE"
       :
done < $INFILE
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
```

Listing 1-7 Method 12 tied for first place

The method in Listing 1-7 is tied for first place. This method uses my favorite input redirection for files by redirecting input at the bottom of the loop, after the done loop terminator. This method does use a file descriptor for redirecting standard output, stdout, to file descriptor 1.

The input redirection using done < \$INFILE greatly speeds up any loop that requires file input. The nice thing about this method of input redirection is that it is intuitive to use for beginners to shell scripting. I was actually surprised that this method tied, actually won by 10 mS, using file descriptors for both input and output files, as shown in Listing 1-8.

```
function while_read_LINE_FD_IN_AND_OUT
{
# Method 22
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while read LINE
do
       echo "$LINE"
       :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
```

Listing 1-8 Method 22 tied for first place

I tend not to use the method in Listing 1-8 when I write shell scripts, because it can be difficult to maintain through the code life cycle. If a user is not familiar with using file descriptors, a script using this method is extremely hard to understand. The method in Listing 1-7 produces the same timing results, and it is much easier to understand. Listing 1-9 shows the second-place loop method.

```
for_LINE_cat_FILE_cmdsub2_FD_OUT ()
{
# Method 16
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
for LINE in $(cat $INFILE)
do
   echo "$LINE"
   :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
```

Listing 1-9 Method 16 made second place in timing tests

The method shown in Listing 1-10 is another surprise: a for loop using command substitution with file descriptor file output redirection.

```
for_LINE_cat_FILE_FD_OUT ()
{
    # Method 15

    # Zero out the $OUTFILE
    >$OUTFILE

    # Associate standard output with file descriptor 4
    # and redirect standard output to $OUTFILE
```

Listing 1-10 Method 15 made third place in timing tests

```
exec 4<&1
exec 1> $OUTFILE
for LINE in `cat $INFILE`
do
     echo "$LINE"
     :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
```

Listing 1-10 (continued)

The method in Listing 1-11 is the fastest method to process a file line-by-line that does not use file descriptors.

Listing 1-11 The fastest method not using file descriptors

I use this technique in almost every shell script that does file parsing, simply because of the ease of maintaining the shell script throughout the life cycle.

Using Command Output in a Loop

The technique shown here is a nice little trick to execute a command and use the command's output in a loop without using a pipe:

```
while read LINE
do
    echo "$LINE"
done < <(command)</pre>
```

I know this looks a bit odd. I got this trick from one of my co-workers, Brian Beers. What we are doing here is input redirection from the bottom of the loop, after the done loop terminator, specified by the done < notation. The < (command) notation executes the command and points the command's output into the bottom of the loop.

NOTE The space between < < in < <(command) is required!

Mail Notification Techniques

In a lot of the shell scripts in this book it is a good idea to send notifications to users when errors occur, when a task is finished, and for many other reasons. Some of the email techniques are shown in this section.

Using the mail and mailx Commands

The most common notification method uses the **mail** and **mailx** commands. The basic syntax of both these commands is shown in Listing 1-12.

```
mail -s "This is the subject" $MAILOUT_LIST < $MAIL_FILE
or
cat $MAIL_FILE | mail -s "This is the subject" $MAILOUT_LIST
or
mailx -s "This is the subject" $MAILOUT_LIST < $MAIL_FILE
or
cat $MAIL_FILE | mailx -s "This is the subject" $MAILOUT_LIST</pre>
```

Listing 1-12 Examples using the mail and mailx commands

Not all systems support the mailx command, but the systems that do have support use the same syntax as the mail command. To be safe when dealing with multiple UNIX platforms, always use the mail command.

Using the sendmail Command to Send Outbound Mail

In one shop I worked at I could not send outbound mail from the user named root. The *from* field had to be a valid email address that was recognized by the mail server,

and root is not valid. To get around this little problem I changed the command that I used from mail to sendmail. The sendmail command allows us to add the -f switch to indicate a valid internal email address for the *from* field. The sendmail command is in /usr/sbin/sendmail on AIX, HP-UX, Linux, and OpenBSD, but on SunOS the location changed to /usr/lib/sendmail. Look at the send_notification function in Listing 1-13.

Listing 1-13 send_notification function

The mail and mailx commands also support specifying a from field by using the -r switch, as shown in Listing 1-14.

```
mail -r randy@$THISHOST -s "This is the subject" \
$MAILOUT_LIST < $MAIL_FILE
or
cat $MAIL_FILE | mail -r randy@$THISHOST -s
"This is the subject" $MAILOUT_LIST
or
mailx -r randy@$THISHOST -s "This is the subject"
$MAILOUT_LIST < $MAIL_FILE
or
cat $MAIL_FILE | mailx -r randy@$THISHOST -s
"This is the subject" $MAILOUT_LIST</pre>
```

Listing 1-14 Specifying a from field with mail and mailx

Both techniques should allow you to get the message out quickly.

Creating a Progress Indicator

Anytime that a user is forced to wait as a long process runs, it is an excellent idea to give the user some feedback. This section deals with progress indicators.

A Series of Dots

The echo command prints a single dot on the screen, and the backslash c, \c, specifies a continuation on the same line without a new line or carriage return. To make a series of dots we will put this single command in a loop, with some sleep time between each dot. We will use a while loop that loops forever with a 10-second sleep between printing each dot on the screen:

```
while true
do
echo ".\c"
sleep 10
done
```

A Rotating Line

The rotate_line function, shown in Listing 1-15, shows what appears to be a rotating line as the process runs.

```
function rotate_line
{
INTERVAL=1 # Sleep time between "twirls"
TCOUNT="0" # For each TCOUNT the line twirls one increment
while :
          # Loop forever...until this function is killed
do
    TCOUNT=`expr $TCOUNT + 1` # Increment the TCOUNT
    case $TCOUNT in
          "1") echo '-'"\b\c"
              sleep $INTERVAL
              ;;
          "2") echo '\\'"\b\c"
              sleep $INTERVAL
              ;;
          "3") echo "|\b\c"
              sleep $INTERVAL
               ;;
```

Listing 1-15 rotate_line function

```
"4") echo "/\b\c"
    sleep $INTERVAL
    ;;
    *) TCOUNT="0" ;; # Reset the TCOUNT to "0", zero.
    esac
done
}
```

Listing 1-15 (continued)

To use rotate_line in a shell script, use the technique shown in Listing 1-16 to start and stop the rotation.

Listing 1-16 Using rotate_line in a shell script

Elapsed Time

The elapsed_time function requires as input the number of seconds, and produces as output the time represented in hours, minutes, and seconds. Listing 1-17 shows the elapsed_time function.

```
elapsed_time ()
{
SEC=$1
(( SEC < 60 )) && echo "[Elapsed time: $SEC seconds]\c"</pre>
```

Listing 1-17 elapsed_time function

```
(( SEC >= 60 && SEC < 3600 )) & echo "[Elapsed time: $(( SEC / 60 ))
min $(( SEC % 60 )) sec]\c"
(( SEC > 3600 )) & echo "[Elapsed time: $(( SEC / 3600 )) hr \
$(( (SEC % 3600) / 60 )) min $(( (SEC % 3600) % 60 )) sec]\c"
}
```

Listing 1-17 (continued)

An example using the elapsed_time function follows:

elapsed_time 6465 [Elapsed time: 1 hr 47 min 45 sec]

Note that the elapsed_time function continues on the same line of text without a new line. To add a new line, change the ending \c to \n on each of the three echo commands.

Working with Record Files

We often want to add a filename to each record in a record file. The code shown in Listing 1-18 shows a merge process where we loop through a list of record filenames and, as we append each record file to build the big batch-processing file, we append the record filename to each record. Check out Listing 1-18, and we will cover the details at the end.

```
MERGERECORDFILE=/data/mergerecord.$(date +%m%d%y)
RECORDFILELIST=/data/branch_records.lst
while read RECORDFILENAME
do
    sed s/$/$(basename $RECORDFILENAME)/g \
$RECORDFILENAME >> $MERGERECORDFILE
done < $RECORDFILELIST</pre>
```

Listing 1-18 Code for a merge process for fixed-length record files

Listing 1-18 is a merge script for fixed-length record files. We first define a couple of files. The MERGERECORDFILE variable definition specifies the name of the resulting merged record data file. The RECORDFILELIST variable defines the file that contains a list of record files that must be merged. Notice the use of the basename command to strip the directory part of the \$RECORDFILENAME on each loop iteration. As we feed the while loop from the bottom, after done, to process the \$RECORDFILELIST

file line-by-line, we assign a new record file to the RECORDFILENAME variable. We use this new value in our sed statement that appends the record filename to the end of each record in the file. As the last step, this record file is appended to the end of the \$MERGERECORDFILE file. Listing 1-19 shows the same technique working with variable-length record files.

```
MERGERECORDFILE=/data/mergerecord.$(date +%m%d%y)
RECORDFILELIST=/data/branch_records.lst
FD=:
while read RECORDFILENAME
do
sed s/$/${FD}$(basename $RECORDFILENAME)/g \
$RECORDFILENAME >> $MERGERECORDFILE
done < $RECORDFILELIST</pre>
```

Listing 1-19 Code for a merge process for variable-length record files

Listing 1-19 is a merge script for variable-length record files. The only difference between this script and Listing 1-18 is that we added the field delimiter variable, FD. We define the field delimiter and then add the field delimiter between the end of the record and the record filename.

Working with Strings

A useful thing to know when working with fixed-length record files is the string length. The length of a string assigned to a variable can be found using the following syntax:

```
echo ${#VAR}
```

For example, if I assign the VAR variable the string 1234567890, the string length should be 10:

```
VAR=1234567890
echo ${#VAR}
10
```

We sometimes need the string length for fixed-length records to pad the extra space in a data field that the data does not fill up. As an example, the TOTAL field is defined as characters 46–70, which is 25 characters long. It is unlikely that anyone would ever owe that kind of cash, but that is the data field we have to work with. This field is right-justified and filled with leading zeros, as shown here:

```
000000000000000000009825
```

If we change a record field and the data is not as long as the data field, we need to pad the rest of the data field with something — in this case, zeros. The hard way to pad the data with leading zeros is to check the original data for the string length, and then get the string length of the new data and start looping to add the correct number of leading zeros followed by the new data. It is so much easier to do this task with the typeset command. The -Z switch specifies right justification with leading zeros. Well, that's just what we need here. Our TOTAL variable is 25 characters long. The following typeset command makes this definition for us:

typeset -Z25 TOTAL

Now we can change the TOTAL data field to a different value and not worry about how many leading zeros to add. As an example, let's change the total to 0 — we paid this bill off!

The typeset command can do a lot of the work for us. Table 1-11 shows more typeset options you might find handy.

SWITCH	RESULTING TYPESET
-L	Left-justify and remove blank spaces.
-R	Right-justify and fill with leading blanks.
-Zn	Right-justify to <i>n</i> length and fill with leading zeros if the first non-blank character is a digit.
-i	Variable is an integer.
-1	Convert uppercase characters to lowercase characters.
-u	Convert lowercase characters to uppercase characters.
-x	Automatically export this variable.

Table 1-11 Options for the typeset Command

NOTE Using + instead of - turns off the typeset definition.

Creating a Pseudo-Random Number

There is a built-in shell variable that will create a pseudo-random number called RANDOM. The following code segment creates a pseudo-random number between 1 and an upper limit defined by the user:

```
RANDOM=$$ # Initialize the seed to the PID of the script
UPPER_LIMIT=$1
RANDOM_NUMBER=$(($RANDOM % $UPPER_LIMIT + 1))
echo "$RANDOM_NUMBER"
```

If the user specified the UPPER_LIMIT to be 100, the result would be a pseudo-random number between 1 and 100.

Using /dev/random and /dev/urandom

We can also use the /dev/random and /dev/urandom character special files to produce pseudo-random numbers. Trying to read the /dev/random and /dev/urandom character special files directly with dd returns non-printable binary data. To get some usable random numbers, we need to pipe the dd command output to the od, octal dump, command. Specifically, we use od to dump the data to an unsigned integer. The code shown in Listing 1-20 assigns an unsigned random integer to the RN variable.

Listing 1-20 Using /dev/random to return a random number

Notice in Listing 1-20 that the dd command uses /dev/random as the input file. We set the count equal to 1 to return one byte of data. Then, *and this is important*, we send all the standard error output, specified by file descriptor 2, to the bit bucket. If we omit the 2 > /dev/null redirection, we get unwanted data. The remaining standard output is piped to the od command to convert the binary data to an unsigned integer, specified by the -t u4 command switch. By changing the value assigned to u, we change the length of the random number returned. To create a 64-bit, not 64-character, random number, we just change the -t u4 to -t u8. More examples are in Chapter 11, "Pseudo-Random Number and Data Generation."

Checking for Stale Disk Partitions in AIX

Ideally, we want the stale disk partition value to be zero, 0. If the value is greater than zero we have a problem. Specifically, the mirrored disks in this Logical Volume are not in sync, which translates to a worthless mirror. Take a look at the following command statement:

```
LV=apps_lv
NUM_STALE_PP=$(lslv -L $LV | grep "STALE PP" | awk '{print $3}'
```

This statement saves the number of stale PPs into the <code>NUM_STALE_PP</code> variable. We accomplish this feat by command substitution, specified by the <code>VARIABLE=\$(commands)</code> notation.

Automated Host Pinging

Depending on the operating system that you are running, the ping command varies if you want to send three pings to each host to see if the machines are up. The ping_host function shown in Listing 1-21 can ping from AIX, HP-UX, Linux, OpenBSD, and SunOS machines.

```
function ping_host
{
HOST=$1 # Grab the host to ping from ARG1.
PING_COUNT=3
PACKET_SIZE=54
# This next case statement executes the correct ping
# command based on the Unix flavor
case $(uname) in
AIX OpenBSD Linux)
           ping -c${PING_COUNT} $HOST 2>/dev/null
           ;;
HP-UX)
           ping $HOST $PACKET_SIZE $PING_COUNT 2>/dev/null
           ;;
SunOS)
           ping -s $HOST $PACKET_SIZE $PING_COUNT 2>/dev/null
           ;;
*)
           echo "\nERROR: Unsupported Operating System - $(uname)"
           echo "\n\t...EXITING...\n"
           exit 1
esac
}
```

Listing 1-21 ping_host function

The main body of the shell script must supply the hostname to ping. This is usually done with a while loop.

Highlighting Specific Text in a File

The technique shown here highlights specific text in a file with reverse video while displaying the entire file. To add in the reverse video piece, we have to do some

command substitution within the sed statement using the tput commands. Where we specify the new_string, we will add in the control for reverse video using command substitution, one to turn highlighting on and one to turn it back off. When the command substitution is added, our sed statement will look like the following:

```
sed s/string/$(tput smso)string$(tput rmso)/g
```

We also want the string to be assigned to a variable, as in the next command:

sed s/"\$STRING"/\$(tput smso)"\$STRING"\$(tput rmso)/g

Notice the double quotes around the string variable, "\$STRING". Do not forget to add the double quotes around variables!

As an experiment using command substitution, try this next command statement to highlight the machine's hostname in the /etc/hosts file on any UNIX machine:

cat /etc/hosts | sed s/'hostname'/\$(tput smso)'hostname'\$(tput rmso)/g

Keeping the Printers Printing

Keeping the printers enabled in a large shop can sometimes be overwhelming. There are two techniques to keep the printers printing. One technique is for the AIX "classic" printer subsystem, and the other is for System V and CUPS printing.

AIX "Classic" Printer Subsystem

To keep AIX "classic" printer subsystem print queues running, use either of the following commands:

enable \$(enq -AW | tail +3 | grep DOWN | awk '{print \$1}') 2>/dev/null

or

enable \$(lpstat -W | tail +3 | grep DOWN | awk '{print \$1}') 2>/dev/null

System V and CUPS Printing

To keep System V and CUPS printers printing any of the following commands:

```
lpc enable $(lpstat -a | grep 'not accepting' | awk '{print $1}')
lpc start $( lpstat -p | grep disabled | awk '{print $2}')
lpc up all  # Enable all printing and queuing
```

It is a good idea to use the root cron table to execute the appropriate command every 15 minutes or so.

Automated FTP File Transfer

You can use a here document to script an FTP file transfer. The basic idea is shown here:

```
ftp -i -v -n wilma <<END_FTP
user randy mypassword
binary
lcd /scripts/download
cd /scripts
get auto_ftp_xfer.ksh
bye</pre>
```

END_FTP

Using rsync to Replicate Data

We use rsync much the same way that we use rcp and scp. All three methods require a source and destination file or directory, and there are command-line switches that allow us to save file permissions, links, ownership, and so on, as well as to copy directory structures recursively. A few examples are the best way to learn how to use rsync. Let's start with this simple rsync statement:

rsync myscript.Bash yogi:/scripts/

This would transfer the file myscript.Bash from the current directory on the local machine to the /scripts directory on the yogi server. Now, if the myscript.Bash file already exists on the server yogi, in the /scripts directory, then the *rsync remote-update protocol* is used to update the file by sending only the differences between the source and destination files:

rsync -avz yogi:/scripts/data /scripts/tmp

This rsync statement will recursively transfer all the files and directories in the /scripts/data directory on the yogi machine to the /scripts/tmp directory on the local machine. The -a rsync switch specifies *archive mode*, which preserves the permissions, ownerships, symbolic links, attributes, and so on, and specifies a recursive copy in the transfer. The -z rsync switch specifies compression is to be used in the transfer to reduce the amount of data in the transfer. Note that this example will create a new directory on the local machine /scripts/tmp/data. The -v rsync switch specifies verbose mode.

```
rsync -avz yogi:/scripts/data/ /scripts/tmp
```

Notice the trailing slash on the source: yogi:/scripts/data/

This trailing slash changes the behavior of rsync to eliminate creating the additional directory on the destination, as the previous example produced: /scripts/tmp/data. The trailing slash on the *source* side tells rsync to *copy the directory contents*, as opposed to *copy the directory name and all of its contents*.

```
rsync -avz /scripts/data /scripts/tmp
rsync -avz /scripts/data/ /scripts/tmp
```

As you can see by these two examples, we can copy files locally as well as remotely. Notice that local file copying does not have a hostname specified by the hostname: notation.

Simple Generic rsync Shell Script

A simple generic shell script for rsync consists only of defining some variables to point to the file/directory we want to copy, and a one-line rsync statement. Check out Listing 1-22 and we will cover the details at the end.

```
#!/bin/Bash
#
# SCRIPT: generic_rsync.Bash
# AUTHOR: Randy Michael
# DATE: 11/18/2007
# REV: 1.0
# PURPOSE: This is a generic shell script to copy files
#
       using rsync.
# set -n # Uncomment to check script syntax without execution
# set -x # Uncomment to debug this script
# REV LIST:
#
*****
# DEFINE FILES AND VARIABLES HERE
*****
# Define the source and destination files/directories
SOURCE_FL="/scripts/"
DESTIN FL="booboo:/scripts"
*****
# BEGINNING OF MAIN
****
# Start the rsync copy
```

Listing 1-22 generic_rsync.Bash script

```
rsync -avz "$SOURCE_FL" "$DESTIN_FL"
# End of generic_rsync.Bash
```

Listing 1-22 (continued)

As you can see, there is not much to this shell script. We define the source and destination files/directories to the SOURCE_FL and DESTIN_FL variables, respectively. Next we use these variables in our rsync statement:

```
rsync -avz "$SOURCE_FL" "$DESTIN_FL"
```

This rsync command will recursively transfer all the files and subdirectories in the local /scripts/ directory (notice the trailing slash on the source) to the /scripts directory on the booboo server using compression to reduce the amount of data transferred. Notice that the trailing slash avoided creating a second scripts directory on the destination: /scripts/.

Capturing a List of Files Larger than \$MEG

Who filled up that filesystem? If you want to look quickly for large files, use the following syntax:

```
# Search for files > $MEG_BYTES starting at the $SEARCH_PATH
#
HOLD_FILE=/tmp/largefiles.list
MEG_BYTES=$1
SEARCH_PATH=$(pwd) # Use the current directory
find $SEARCH_PATH -type f -size +${MEG_BYTES}000000c -print > $HOLDFILE
```

Note that in the find command after the -size parameter there is a plus sign (+) preceding the file size, and there is a c added as a suffix. This combination specifies files larger than \$MEG_BYTES measured in bytes, as opposed to blocks.

Capturing a User's Keystrokes

In most large shops there is a need, at least occasionally, to monitor a user's actions. You may even want to audit the keystrokes of anyone with root access to the system or other administration-type accounts, such as oracle. Contractors onsite can pose a particular security risk. Typically when a new application comes into the environment, one or two contractors are onsite for a period of time for installation, troubleshooting, and training personnel on the product.

The code shown in Listing 1-23 uses the script command to capture all the keystrokes.

```
TS=$(date +%m%d%y%H%M%S)
                            # File time stamp
THISHOST=$(hostname|cut -f1-2 -d.) # Host name of this machine
LOGFILE=${THISHOST}.${LOGNAME}.$TS # Creates the name of the log file
touch $LOGDIR/$LOGFILE
                           # Creates the actual file
# Set the command prompt
export PS1="[$LOGNAME:$THISHOST]@"'$PWD> '
chown $LOGNAME ${LOGDIR}/${LOGFILE} # Let the user own the log file
                             # while the script executes
chmod 600 ${LOGDIR}/${LOGFILE} # Change permission to RW for the owner
script ${LOGDIR}/${LOGFILE}
                             # Start the script monitoring session
chown root ${LOGDIR}/${LOGFILE}
                             # Change the ownership to root
chmod 400 ${LOGDIR}/${LOGFILE}
                             # Set permission to read-only by root
```

Listing 1-23 Capturing a user's keystrokes

Using the bc Utility for Floating-Point Math

On UNIX machines there is a utility called bc that is an interpreter for arbitraryprecision arithmetic language. The bc command is an interactive program that provides arbitrary-precision arithmetic. You can start an interactive bc session by typing bc on the command line. Once in the session you can enter most complex arithmetic expressions as you would in a calculator.

The code segment shown in Listing 1-24 creates the mathematical expression for the bc utility and then uses a here document to load the expression into bc.

Listing 1-24 Example of using bc in a shell script

```
# input to the bc command. The sum of the numbers is
# assigned to the SUM variable.
SUM=$(bc <<EOF
scale=$SCALE
(${ADD})
EOF)</pre>
```

Listing 1-24 (continued)

This is about as simple as bc gets. This is just a taste. Look for more later in the book.

Number Base Conversions

There are a lot of occasions when we need to convert numbers between bases. The code that follows shows some examples of how to change the base.

Using the typeset Command

Using the typeset command is valid up to base 36.

```
Convert a base 10 number to base 16
# typeset -i16 BASE_16_NUM
# BASE_16_NUM=47295
# echo $BASE_16_NUM
16#b8bf
Convert a base 8 number to base 16
[root@yogi:/scripts]> typeset -i16 BASE_16_NUM
[root@yogi:/scripts]> BASE_16_NUM=8#472521
[root@yogi:/scripts]> echo $BASE_16_NUM
16#735c9
```

Using the printf Command

We can use the printf command to convert base-10 numbers to octal and hexadecimal notation, as shown here:

```
Convert a base 10 number to base 8
# printf %o 20398
47656
```

Convert a base 10 number to base 16 # printf %x 20398 4fae

We can display a number in exponential notation with the printf command with the following syntax:

```
# printf %e 20398
2.039800e+04
```

Create a Menu with the select Command

There are many times when you just need to provide a menu for the end user to select from, and this is where a select statement comes in. The menu prompt is assigned to the PS3 system variable, and the select statement is used a lot like a for loop. A case statement is used to specify the action to take on each selection.

```
#!/bin/Bash
#
# SCRIPT: select_system_info_menu.Bash
# AUTHOR: Randy Michael
# DATE: 1/17/2008
# REV: 1.0
#
# PURPOSE: This shell script uses the shell's select
# command to create a menu to show system information
# Clear the screen
clear
# Display the menu title header
echo -e "\n\tsystem information menu\n"
# Define the menu prompt
PS3="Select an option and press Enter: "
# The select command defines what the menu
# will look like
select i in OS Host Filesystems Date Users Quit
do
   case $i in
```

```
0S)
          echo
          uname
          ;;
   Host)
          echo
          hostname
          ;;
   Filesystems)
          echo
          df -k | more
          ;;
   Date)
          echo
          date
          ;;
   Users) echo
          who
          ;;
   Quit) break
          ;;
esac
```

Setting the select command's REPLY variable
to NULL causes the menu to be redisplayed

REPLY=

Pause before redisplaying the menu

```
echo -e "\nPress Enter to Continue...\c" read
```

Ready to redisplay the menu again

clear the screen

clear

Display the menu title header

echo -e " $\n\tSYSTEM$ INFORMATION MENU \n "

done

Clear the screen before exiting

clear

Notice in this code segment the use of the select statement. This looks just like a for loop with a list of possible values. Next is an embedded case statement that allows us to specify the action to take when each selection is made.

The output of this simple menu is shown here:

```
SYSTEM INFORMATION MENU

1) OS 3) Filesystems 5) Users

2) Host 4) Date 6) Quit

Select an option and press Enter: 1

Linux

Press Enter to Continue...
```

Removing Repeated Lines in a File

The uniq command is used to report and remove repeated lines in a file. This is a valuable tool for a lot of scripting and testing.

If you have a file that has repeated lines named my_list and you want to save the list without the repeated lines in a file called my_list_no_repeats, use the following command:

uniq my_list my_list_no_repeats

If you want to see a file's output without repeated lines, use the following command:

cat repeat_file | uniq

Removing Blank Lines from a File

The easiest way to remove blank lines from a file is to use a sed statement. The following syntax removes the blank lines:

```
# cat my_file | sed /^$/d
or
```

```
# sed /^$/d my_file
```

Testing for a Null Variable

Variables that have nothing assigned to them are sometimes hard to deal with. The following test will ensure that a variable is either Null or has a value assigned to it. The double quotes are very important and must be used!

```
VAL= # Creates a NULL variable

if [[ -z "$VAL" && "$VAL" = '' ]]

then

    echo "The VAL variable is NULL"

fi

or

VAL=25

if [[ ! -z "$VAL" && "$VAL" != '' ]]

then

    echo "The VAL variable is NOT NULL"

fi
```

Directly Access the Value of the Last Positional Parameter, \$#

To access the value of the \$# positional parameter directly, use the following regular expression:

```
eval '$'$#
Or
eval \$$#
```

There are a lot of uses for this technique, as you will see later in this book.

Remove the Column Headings in a Command Output

There are many instances when we want to get rid of the column headings in a command's output. A lot of people try to use grep -v to pattern-match on something unique in the heading. A much easier and more reliable method is to use the tail command. An example is shown here with the df command output:

[root:yogi]@/scripts# df -k							
Filesystem	1024-blocks	Free	%Used	Iused	%Iused	Mounted	on
/dev/hd4	32768	15796	52%	1927	12%	/	
/dev/hd2	1466368	62568	96%	44801	13%	/usr	
/dev/hd9var	53248	8112	85%	1027	88	/var	
/dev/hd3	106496	68996	36%	245	1%	/tmp	
/dev/hd1	4096	3892	5%	55	6%	/home	

/proc	-	-	-	-	- /proc
/dev/hd10opt	655360	16420	98%	16261	10% /opt
/dev/scripts_lv	102400	24012	77%	1137	5% /scripts
/dev/lv_temp	409600	147452	65%	29	1% /tmpfs

Now look at the same output with the column headings removed:

[root:yogi]@/scripts# df -k tail +2						
/dev/hd4	32768	15796	52%	1927	12% /	
/dev/hd2	1466368	62568	96%	44801	13% /usr	
/dev/hd9var	53248	8112	85%	1027	8% /var	
/dev/hd3	106496	68996	36%	245	1% /tmp	
/dev/hd1	4096	3892	5%	55	6% /home	
/proc	-	-	-	-	- /proc	
/dev/hd10opt	655360	16420	98%	16261	10% /opt	
/dev/scripts_lv	102400	24012	77%	1137	5% /scripts	
/dev/lv_temp	409600	147452	65%	29	1% /tmpfs	

Just remember to add one to the total number of lines that you want to remove.

Arrays

The shell supports one-dimensional arrays. The maximum number of array elements is 1,024. When an array is defined, it is automatically dimensioned to 1,024 elements. A one-dimensional array contains a sequence of *array elements*, which are like the boxcars connected together on a train track. An array element can be just about anything, except for another array. I know; you're thinking that you can use an array to access an array to create two- and three-dimensional arrays. This can be done, but it is beyond the scope of this book.

Loading an Array

An array can be loaded in two ways. You can define and load the array in one step with the set -A command, or you can load the array one element at a time. Both techniques are shown here:

The first array element is referenced by 0, not 1. To access array elements use the following syntax:

```
echo ${MY_ARRAY[2] # Show the third array element
gamma
echo ${MY_ARRAY[*] # Show all array elements
alpha beta gamma
echo ${MY_ARRAY[@] # Show all array elements
alpha beta gamma
echo ${#MY_ARRAY[*]} # Show the total number of array elements
3
echo ${MY_ARRAY[@]} # Show the total number of array elements
3
echo ${MY_ARRAY[@]} # Show the total number of array elements
3
echo ${MY_ARRAY[@]} # Show array element 0 (the first element)
alpha
```

We will use arrays in shell scripts in several chapters in this book.

Testing a String

One of the hardest things to do in a shell script is to test the user's input from the command line. The shell script shown in Listing 1-25 will do the trick using regular expressions to define the string composition.

```
#!/bin/ksh
#
# SCRIPT: test_string.ksh
#
  AUTHOR: Randy Michael
#
 REV: 1.0.D - Used for developement
  DATE: 10/15/2007
#
#
  PLATFORM: Not Platform Dependent
#
#
  PURPOSE: This script is used to test a character
#
           string, or variable, for its composition.
#
           Examples include numeric, lowercase or uppercase
#
           characters, alpha-numeric characters, and IP address.
#
  REV LIST:
#
#
#
# set -x # Uncomment to debug this script
```

Listing 1-25 test_string.ksh shell script

```
# set -n # Uncomment to verify syntax without any execution.
        # REMEMBER: Put the comment back or the script will
#
#
        # NOT EXECUTE!
*****
*****
test_string ()
{
# This function tests a character string
# Must have one argument ($1)
if (( $# != 1 ))
then
    # This error would be a programming error
    print "ERROR: $ (basename $0) requires one argument"
    return 1
fi
# Assign arg1 to the variable --> STRING
STRING=$1
# This is where the string test begins
case $STRING in
+([0-9]).+([0-9]).+([0-9]).+([0-9]))
        # Testing for an IP address - valid and invalid
       INVALID=FALSE
        # Separate the integer portions of the "IP" address
        # and test to ensure that nothing is greater than 255
        # or it is an invalid IP address.
        for i in $(echo $STRING | awk -F . '{print $1, $2, $3, $4}')
       do
           if (( i > 255 ))
           then
               INVALID=TRUE
           fi
       done
       case $INVALID in
       TRUE) print 'INVALID_IP_ADDRESS'
             ;;
```

Listing 1-25 (continued)

```
FALSE) print 'VALID_IP_ADDRESS'
               ;;
         esac
         ;;
+([0-1])) # Testing for 0-1 only
         print 'BINARY_OR_POSITIVE_INTEGER'
         ;;
+([0-7])) # Testing for 0-7 only
         print 'OCTAL_OR_POSITIVE_INTEGER'
         ;;
+([0-9])) # Check for an integer
         print 'INTEGER'
         ;;
+([-0-9])) # Check for a negative whole number
          print 'NEGATIVE_WHOLE_NUMBER'
          ;;
+([0-9] |[.][0-9]))
          # Check for a positive floating point number
          print 'POSITIVE_FLOATING_POINT'
          ;;
+(+[0-9][.][0-9]))
          # Check for a positive floating point number
          # with a + prefix
          print 'POSITIVE_FLOATING_POINT'
          ;;
+(-[0-9][.][0-9]))
          # Check for a negative floating point number
          print 'NEGATIVE_FLOATING_POINT'
          ;;
+([-.0-9]))
          # Check for a negative floating point number
          print 'NEGATIVE_FLOATING_POINT'
          ;;
+([+.0-9]))
          # Check for a positive floating point number
          print 'POSITIVE_FLOATING_POINT'
          ;;
+([a-f])) # Test for hexidecimal or all lowercase characters
         print 'HEXIDECIMAL_OR_ALL_LOWERCASE'
         ;;
+([a-f] | [0-9])) # Test for hexidecimal or all lowercase characters
         print 'HEXIDECIMAL_OR_ALL_LOWERCASE_ALPHANUMERIC'
         ;;
+([A-F])) # Test for hexidecimal or all uppercase characters
         print 'HEXIDECIMAL_OR_ALL_UPPERCASE'
         ;;
+([A-F] | [0-9])) # Test for hexidecimal or all uppercase characters
         print 'HEXIDECIMAL_OR_ALL_UPPERCASE_ALPHANUMERIC'
```

Listing 1-25 (continued)

```
;;
+([a-f]|[A-F]))
       # Testing for hexidecimal or mixed-case characters
       print 'HEXIDECIMAL_OR_MIXED_CASE'
       ;;
+([a-f] | [A-F] | [0-9]))
       # Testing for hexidecimal/alpha-numeric strings only
       print 'HEXIDECIMAL OR MIXED CASE ALPHANUMERIC'
       ::
+([a-z] | [A-Z] | [0-9]))
       # Testing for any alpha-numeric string only
       print 'ALPHA-NUMERIC'
       ;;
+([a-z])) # Testing for all lowercase characters only
       print 'ALL_LOWERCASE'
       ;;
+([A-Z])) # Testing for all uppercase numbers only
       print 'ALL_UPPERCASE'
       ;;
+([a-z]|[A-Z]))
       # Testing for mixed case alpha strings only
       print 'MIXED_CASE'
       ;;
       *) # None of the tests matched the string composition
          print 'INVALID_STRING_COMPOSITION'
       ;;
esac
}
*****
usage ()
{
echo "\nERROR: Please supply one character string or variable\n"
echo "USAGE: $THIS_SCRIPT {character string or variable}\n"
}
*****
*****
# Query the system for the name of this shell script.
# This is used for the "usage" function.
THIS_SCRIPT=$(basename $0)
# Check for exactly one command-line argument
```

Listing 1-25 (continued)

```
if (( $# != 1 ))
then
    usage
    exit 1
fi

# Everything looks okay if we got here. Assign the
# single command-line argument to the variable "STRING"
STRING=$1
# Call the "test_string" function to test the composition
# of the character string stored in the $STRING variable.
test_string $STRING
# End of script
```

Listing 1-25 (continued)

This is a good start, but this shell script does not cover everything. Play around with it to see if you can make some improvements.

NOTE Bash shell does not support the +([0-9])-type regular expressions.

Summary

This chapter is just a primer to get you started with a quick review and some little tricks and tips. In the next 27 chapters, we are going to write a lot of shell scripts to solve some real-world problems. Sit back and get ready to take on the UNIX world!

The first thing that we are going to study is 24 ways to process a file line-by-line. I have seen a lot of good and bad techniques for processing a file line-by-line over the past 15 years, and some have been rather inventive. The next chapter presents the 24 techniques that I have seen the most; at the end of the chapter there is a shell script that times each technique to find the fastest. Read on, and find out which one wins the race. See you in the next chapter!

CHAPTER 2

24 Ways to Process a File Line-by-Line

Have you ever created a really slick shell script to process file data and found that you have to wait until after lunch to get the results? The script may be running so slowly because of how you are processing the file. Over my many years of UNIX administration I have seen some rather, well, let's call them *creative* ways to process a file line-by-line. I have listed 24 of these methods in this chapter. Some techniques are very fast, and some make you wait for half a day. Then there are a few that, on first look, should work but do not. The techniques used in this chapter are measurable, and I created a shell script that will time each method so that you can see which method suits your needs.

When processing an ASCII text/data file, we are normally inside a loop of some kind. Then, as we go through the file from the top to the bottom, we process each line of text. A UNIX shell script is really not meant to work on a text file character-by-character (Pearl is better for this), but you can accomplish the task using various techniques. The task for this chapter is to show the line-by-line parsing techniques by parsing each entire line at once, not character by character. We are also going to look at using file descriptors as a method of file input and output redirection for file processing.

Command Syntax

First, as always, we need to go over the command syntax that we are going to use. The commands that we want to concentrate on in this chapter have to deal with while and for loops. When parsing a file in a loop, we need a method to read in the entire line of data to a variable. The most prevalent command is **read**. The read command is flexible in that you can extract individual strings as well as the entire line. Speaking of lines, the **line** command is another alternative to grab a full line of text. Some operating

systems do not support the line command. For example, the line command is not available on OpenBSD and some versions of Linux and Solaris.

In addition to the read and line commands, we need to look at the different ways to use the while and for loops, which is the major cause of fast or slow execution times. A loop can be used as a standalone loop in a predefined configuration; it can be used in a command pipe or with file descriptors. Each method has its own set of rules. The use of the loop is critical to get the quickest execution times.

Using File Descriptors

Under the covers of the UNIX operating system, files are referenced, copied, and moved by unique numbers known as *file descriptors*. You already know about three of these file descriptors:

```
0 - stdin
1 - stdout
```

2 - stderr

We redirect output using the stdout (standard output) and stderr (standard error) a lot in this book, using several different techniques. This is the first time we are going to use the stdin (standard input) file descriptor. For a short definition of each of these we can talk about the devices on the computer. Standard input usually comes into the computer from the keyboard or mouse. Standard output usually has output to the screen or to a file. Standard error is where commands, programs, and scripts route error messages. We have used stderr before to send the error messages to the bit bucket, or /dev/null, and (more commonly) to combine the stdout and stderr outputs together. You should remember a command like the following one:

some_command 2>&1

The preceding command sends all the error messages to the same output device that standard output goes to, which is normally the terminal. We can also use other file descriptors. Valid descriptor values range from 0 to 19 on most operating systems. You have to do a lot of testing when you use the upper values to ensure that they are not reserved by the system for some reason. You will see more on using file descriptors in some of the following code listings.

Creating a Large File to Use in the Timing Test

Before I get into each method of parsing the file, I want to show you a little script you can use to create a file of random alphanumeric characters that is exactly a user-specified number of MB in size. The number of characters per line is 80, including the new-line character. The method to create a file of random alphanumeric characters is to load an array with upper- and lowercase alpha characters as well as the numbers 0–9. We then use a random number, in a loop, to point to array elements and append each character as we build each line. A while loop is used to create 12,800 lines per

MB the user specifies. To create a 1 MB random character file, you need only add the number of MB as a parameter to the shell script name. Using the shell script in Listing 2-1, you create a 1 MB, 12,800-line file, with the following syntax:

```
# random_file.Bash 1
```

The full shell script is shown in Listing 2-1.

```
#!/bin/Bash
#
# SCRIPT: random_file.Bash
# AUTHOR: Randy Michael
# DATE: 8/3/2007
# REV: 1.0
# PLATFORM: Not Platform Dependent
#
# PURPOSE: This script is used to create a
#
    specific size file of random characters.
#
     The methods used in this script include
#
     loading an array with alphanumeric
#
     characters, then using the /dev/random
#
     character special file to seed the RANDOM
#
     shell variable, which in turn is used to
      extract a random set of characters from the
#
#
     KEYS array to build the OUTFILE of a
     specified MB size.
#
# set -x # Uncomment to debug this script
#
# set -n # Uncomment to check script syntax
#
        # without any execution. Do not forget
#
        # to put the comment back in or the
        # script will never execute.
#
*****
# DEFINE FILES AND VARIABLES HERE
*****
typeset -i MB_SIZE=$1
typeset -i RN
typeset -i i=1
typeset -i X=0
OUTFILE=largefile.random.txt
>$OUTFILE
THIS_SCRIPT=$(basename $0)
CHAR_FILE=${WORKDIR}/char_file.txt
```

```
Listing 2-1 random_file.Bash shell script
```

```
*****
# DEFINE FUNCTIONS HERE
*****
build_random_line ()
{
# This function extracts random characters
# from the KEYS array by using the RANDOM
# shell variable
C=1
LINE=
until (( C > 79 ))
do
  LINE="${LINE}${KEYS[$(($RANDOM % X + 1))]}"
  ((C = C + 1))
done
# Return the line of random characters
echo "$LINE"
}
*****
elasped_time ()
{
SEC=$1
((SEC < 60)) & echo -e "[Elasped time: \
$SEC seconds]\c"
(( SEC >= 60 && SEC < 3600 )) && echo -e \
"[Elasped time: $(( SEC / 60 )) min $(( SEC % 60 )) sec]\c"
(( SEC > 3600 )) && echo -e "[Elasped time: \
$(( SEC / 3600 )) hr $(( (SEC % 3600) / 60 )) min \
$(( (SEC % 3600) % 60 )) sec]\c"
3
*****
load_default_keyboard ()
{
# Loop through each character in the following list and
# append each character to the $CHAR_FILE file. This
# produces a file with one character on each line.
for CHAR in 1 2 3 4 5 6 7 8 9 0 q w e r t y u i o \
```

Listing 2-1 (continued)

```
pasdfghjkl zxcvbnm \
          QWERTYUIOPASDFGHJKL \
           Z X C V B N M 0 1 2 3 4 5 6 7 8 9
do
   echo "$CHAR" >> $CHAR_FILE
done
}
*****
usage ()
{
echo -e "\nUSAGE: $THIS_SCRIPT Mb_size"
echo -e "Where Mb_size is the size of the file to build\n"
}
*****
# BEGINNING OF MAIN
*****
if (( $# != 1 ))
then
   usage
   exit 1
fi
# Test for an integer value
case $MB_SIZE in
[0-9]) : # Do nothing
        ;;
     *) usage
        ;;
esac
# Test for the $CHAR_FILE
if [ ! -s "$CHAR_FILE" ]
then
   echo -e "\nNOTE: $CHAR FILE does not esist"
   echo "Loading default keyboard data."
    echo -e "Creating $CHAR_FILE...\c"
   load_default_keyboard
    echo "Done"
fi
# Load Character Array
echo -e "\nLoading array with alphanumeric character elements"
```

Listing 2-1 (continued)

```
while read ARRAY_ELEMENT
do
   ((X = X + 1))
    KEYS[$X]=$ARRAY_ELEMENT
done < $CHAR_FILE
echo "Total Array Character Elements: $X"
# Use /dev/random to seed the shell variable RANDOM
echo "Querying the kernel random number generator for a random seed"
RN=$(dd if=/dev/random count=1 2>/dev/null \
   | od -t u4 | awk '{print $2}'| head -n 1)
# The shell variable RANDOM is limited to 32767
echo "Reducing the random seed value to between 1 and 32767"
RN=$(( RN % 32767 + 1 ))
# Initialize RANDOM with a new seed
echo "Assigning a new seed to the RANDOM shell variable"
RANDOM=$RN
echo "Building a $MB_SIZE MB random character file ==> $OUTFILE"
echo "Please be patient; this may take some time to complete..."
echo -e "Executing: .\c"
# Reset the shell SECONDS variable to zero seconds.
SECONDS=0
TOT_LINES=$(( MB_SIZE * 12800 ))
until (( i > TOT_LINES ))
do
    build_random_line >> $OUTFILE
     (($((i % 100)) == 0)) && echo -e ".\c"
     ((i = i + 1))
done
# Capture the total seconds
```

Listing 2-1 (continued)

Listing 2-1 (continued)

Each line produced by the random_file.Bash script is the same length and consists of random alphanumeric characters. The user specifies size in MB — thus, the total number of lines to create as a parameter to the shell script. The first few lines of the resulting file from the random_file.Bash shell script in Listing 2-1 are shown in Listing 2-2. Your file's data will vary from this example, of course.

```
\label{eq:stable} 1 fv765J85HBPbPym059qseLx2n5MH4NPKFhgtuFCeU2ep6B19chh4RY72mpXIvXrS7348y0NdwiYT61 \\ 1 RkW75vBjGiNZH4Y9HxXfQ2VShKS70znTLx1RPfn3I7zvNZW3m3zQ1NzG62Gj1xrdPD7M2rdE2AcOr3 \\ Pud2ij43br4K3729gbG4n19Yqx5NGI0212eHN154RuC4MtS4qmRphb209FJgzK8IcFW0sTn71niwLyi \\ JOqBQmA5KtbjV34vp31VBKCZp0PVJ4Zcy7fd5R1Fziseux4ncio32loIne1a7MPVqyIuJ8yv5IJ6s5P \\ 485YQX0117hUgqepiz9ejIupjZb1003B7NboGJMga2R1lu19JC0pn40mrnxfN025RMU6Qkv54v2fqfg \\ UmtbXV2mb4IuoBo113IgUg0bh8n2bhZ768Iiw2WMaemgGR6XcQWi0T6Fvg0MkiYELW2ia1oCO83sK06 \\ 2X05sU4Lv9XeV7BaOtC8Y5W7vqqxu69uwsFALripdZS7C8zX1WF6XvFGn4iFF1e5K560nooInX514jb \\ 0S16B1m771vqoDA73u1ZjbY7SsnS07eLxp96GrHDD75731bJXJa4Uz3t0LW2dCWNy6H3YmojVXQVYA1 \\ v3TPxyeJD071S20SBh4xoCCRH4PhqAWBijM9oXyhdZ6MM0t2JWegRo1iNJN5p01hZDmLttr1SCHBvP1 \\ kM3Hbgp0j1QLU8B0JjkY8q1c9NLSbGynKTbf9Meh95QU8rIAB4mDH80zUIEG2qadxQ0191686FHn9Pi \\ \end{tabular}
```

Listing 2-2 Example output from the random_file.Bash script

24 Methods to Parse a File Line-by-Line

The following sections describe 24 of the parsing techniques I have seen over the years, most of them common but also a few very uncommon methods. I have put

them all together in one shell script separated as functions. After the functions are defined, I execute each method, or function, while timing the execution using the shell's **time** command. To get accurate timing results, I use a 1 MB file that has 12,800 lines, where each line is the same 80-character length. (We built this file using the random_file.Bash shell script.) A 12,800-line file is an extremely large file to be parsing line-by-line in a shell script, at 1 MB, but my Linux and AIX machines are so fast that I needed a large file to get the timing data greater than zero!

Now it is time to look at the 24 methods to parse a file line-by-line. Each method uses a while or for statement to create a loop. The only command(s) within the loop are to read an input file and to send output to an output file, specified by \$INFILE and \$OUTFILE, respectively. The thing that makes each method different is how the while and for loops are used. You should also notice that each function contains a *no-op*, specified by the : (colon) character. A no-op does not do anything, and always has a 0 (zero) return code. I use the no-op here as a placeholder in case you pull one of the functions out to use; you can remove the guts of the loop, and the function will not bomb because nothing is within the loop.

Method 1: cat_while_read_LINE

Let's start with the most common method that I see, which is using the **cat** command to list the entire file and piping the output to a while read loop. On each loop iteration a single line of text is read into a variable named LINE. This continuous loop will run until all the lines in the file have been processed one at a time.

The pipe is the key to the popularity of this method. It is intuitively obvious that the output from the previous command in the pipe is used as input to the next command in the pipe. As an example, if I execute the **df** command to list filesystem statistics and it scrolls across the screen out of view, I can use a pipe to send the output to the **more** command, as in the following command:

```
df | more
```

When the df command is executed, the pipe stores the output in a temporary system file. Then this temporary system file is used as input to the more command, allowing us to view the df command output one page/line at a time. Our use of piping output to a while loop works the same way; the output of the cat command is used as input to the while loop and is read into the LINE variable on each loop iteration. Look at the complete function in Listing 2-3.

```
function cat_while_read_LINE
{
    # Method 1
# Zero out the $OUTFILE
>$OUTFILE
```

Listing 2-3 cat_while_read_LINE function

```
cat $INFILE | while read LINE
do
     echo "$LINE" >> $OUTFILE
     :
     done
}
```

Listing 2-3 (continued)

Each of these test loops is created as a function so that we can time each method using the shell script. You could also use the () GNU function definition if you wanted, as shown in Listing 2-4.

```
cat_while_read_LINE ()
{
  # Method 1

  # Zero out the $OUTFILE
 >$OUTFILE
 cat $INFILE | while read LINE
 do
        echo "$LINE" >> $OUTFILE
    :
    done
 }
```

Listing 2-4 Using the () declaration method function

Whether you use the function or the () technique, you get the same result. I tend to use the function method more often so that when someone edits the script they will know the block of code is a function. For beginners, the word "function" helps a lot in understanding the whole shell script. The *\$INFILE* and *\$OUTFILE* variables are set in the main body of the shell script.

Method 2: while_read_LINE_bottom

You are now entering one of my favorite methods of parsing through a file. We still use the while read LINE syntax, but this time we *feed the loop from the bottom* instead of using a pipe. You will find that this is one of the fastest ways to process each line of a file, and it is still intuitive to understand. The first time you see this method it looks a little unusual, but it does work very well.

Look at the code in Listing 2-5, and we will go over the function at the end.

```
function while_read_LINE_bottom
{
    # Method 2

    # Zero out the $OUTFILE
    >$OUTFILE
    while read LINE
    do
        echo "$LINE" >> $OUTFILE
        :
    done < $INFILE
    }
</pre>
```

Listing 2-5 while_read_LINE_bottom function

We made a few modifications to the function from Listing 2-3. The cat \$FILENAME to the pipe was removed. Then we use *input redirection* to let us read the file from the bottom of the loop. By using the <\$FILENAME notation after the done loop terminator we feed the while loop from the bottom, which greatly increases the input throughput to the loop. When we time each technique, this method will stand out close to the top of the list.

Method 3: cat_while_LINE_line

Now we are getting into some of the "creative" methods that I have seen in some shell scripts. As previously stated, not all UNIX operating systems support the line command. You will not find it in OpenBSD, nor in some versions of Linux and Solaris. The line command is available in AIX and HP-UX, and some releases of Linux and Solaris.

Using this loop strategy replaces the read command from Listings 2-3, 2-4, and 2-5 with the line command in a slightly different command structure. Look at the function in Listing 2-6, and you will see how it works at the end.

```
function cat_while_LINE_line
{
    # Method 3

# Zero out the $OUTFILE
>$OUTFILE
```

Listing 2-6 cat_while_LINE_line function

```
cat $INFILE | while LINE=`line`
do
        echo "$LINE" >> $OUTFILE
        :
        done
}
```

The function in Listing 2-6 is interesting. Because we are not using the read command to assign the line of text to a variable, we need some other technique. If your machine supports the line command, then this is an option. To see if your UNIX machine has the line command, enter the following command:

```
which line
```

The response should be something like /usr/bin/line. Otherwise, you will see the \$PATH list that was searched, followed by "line" not found.

The line command is used to grab one whole line of text at a time. The read command does the same thing if you use only one variable with the read statement; otherwise the line of text will be broken up between the different variables used in the read statement.

On each loop iteration the LINE variable is assigned a whole line of text using command substitution. This is done using the LINE=`line` command syntax. The line command is executed, and the result is assigned to the LINE variable. Of course, I could have used any variable name, such as this:

```
MY_LINE=`line`
TEXT=`line`
```

Please notice that the single tic marks are really *back tics* (`command`), which are located in the top-left corner of most keyboards below the ESC-key. Executing a command and assigning the output to a variable is called *command substitution*. Look for the timing data for this technique when you run the timing script. This extra variable assignment may have quite an effect on the timing result.

Method 4: while_LINE_line_bottom

Again, this is one of the more obscure techniques that I have seen in any shell script. This time we are going to feed our while loop from the bottom, but we'll use the line command instead of the read statement to assign the text to the LINE variable. This method is similar to the last technique, but we removed the cat \$FILENAME to the pipe and instead we redirect input into the loop from the bottom, after the done loop terminator.

Look at the function in Listing 2-7, and you will see how it works at the end.

```
function while_LINE_line_bottom
{
    # Method 4

    # Zero out the $OUTFILE
    >$OUTFILE
    while LINE=`line`
    do
        echo "$LINE" >> $OUTFILE
        :
    done < $INFILE
    }
</pre>
```

Listing 2-7 while_LINE_line_bottom function

We use command substitution to assign the line of file text to the LINE variable as we did in the previous method. The only difference is that we are feeding the while loop from the bottom using input redirection of the \$FILENAME file. You should be getting the hang of what we are doing by now. As you can see, there are many ways to parse through a file, but you are going to see that not all of these techniques are very good choices. This method is one of the poorer choices.

Next we are going to look at the other method of command substitution. The last two methods used the line command using the syntax LINE=`line`. We can also use the LINE=\$(line) technique. Is there a speed difference?

Method 5: cat_while_LINE_line_cmdsub2

Look familiar? This is the same method as Method 3 except for the way we use command substitution. As I stated in the beginning, we need a rather large file to parse through to get accurate timing results. When we do our timing tests, we might see a difference between the two command-substitution techniques. Just remember that the line command is not available on all UNIX operating systems.

Study the function in Listing 2-8, and we will cover the function at the end.

```
function cat_while_LINE_line_cmdsub2
{
    # Method 5
# Zero out the $OUTFILE
>$OUTFILE
```

Listing 2-8 cat_while_LINE_line_cmdsub2 function

```
cat $INFILE | while LINE=$(line)
do
        echo "$LINE" >> $OUTFILE
        :
        done
}
```

The only thing we are looking for in the function in Listing 2-8 is a timing difference between the two command-substitution techniques. As each line of file text enters the loop, the line command assigns the text to the LINE variable. Let's see how Methods 3 and 5 show up in the loop timing tests, because the only difference is the command-substitution assignment method.

Method 6: while_LINE_line_bottom_cmdsub2

This method is the same technique used in Listing 2-7 except for the command substitution. In this function we are going to use the LINE=\$ (line) technique. We are again feeding the while loop input from the bottom, after the done loop terminator. Please review the function in Listing 2-9.

```
function while_LINE_line_bottom_cmdsub2
{
    # Method 6

    # Zero out the $OUTFILE
    >$OUTFILE
    while LINE=$(line)
    do
        echo "$LINE" >> $OUTFILE
        :
    done < $INFILE
    }
</pre>
```

Listing 2-9 while_LINE_line_bottom_cmdsub2 function

By the look of the loop structure, you might assume that this while loop is very fast executing, but you will be surprised at how slow it is. The main reason is the variable assignment, but the line command has a large effect, too.

Method 7: for_LINE_cat_FILE

This is the first method to use a for loop. I think you will be surprised at the speed of this technique. Check out the code in Listing 2-10 and we will cover the details at the end.

```
for_LINE_cat_FILE ()
{
  # Method 7

  # Zero out the $OUTFILE
  >$OUTFILE
  for LINE in `cat $INFILE`
  do
     echo "$LINE" >> $OUTFILE
   :
  done
}
```

Listing 2-10 for_LINE_cat_FILE function

In Listing 2-10 we use a for loop in concert with command substitution to cat the file. On each loop iteration the LINE variable is assigned a full line of text. Look for the timing results at the end of this chapter.

Method 8: for_LINE_cat_FILE_cmdsub2

This method again changes our command substitution from the back tics `command` to \$(command). The timing results for Methods 7 and 8 should be interesting. As previously stated, the line command is available on AIX, HP-UX, and some versions of Linux and Solaris, but not on OpenBSD. Again we are using a for loop. Check out Listing 2-11 and we will cover the details at the end.

```
for_LINE_cat_FILE_cmdsub2 ()
{
    # Method 8

# Zero out the $OUTFILE
>$OUTFILE
for LINE in $(cat $INFILE)
do
    echo "$LINE" >> $OUTFILE
    :
done
}
```

Listing 2-11 for_LINE_cat_FILE_cmdsub2 function

Again, on each loop iteration the LINE variable is assigned a full line of text data, and we send this data to the *\$OUTFILE* by appending each line, specified by echo "*\$LINE*">>*\$OUTFILE*, as we process the file from top to bottom.

Method 9: while_line_outfile

I put this method in, but notice there is no variable assignment for each line of data in the input file. All we are doing here is using input redirection after the done loop terminator, and then using the line command to append each line of data to the \$OUTFILE. Look at the code in Listing 2-12.

```
while_line_outfile ()
{
    # Method 9

# Zero out the $OUTFILE
>$OUTFILE
# This function processes every other
# line of the $INFILE, so do not use
# this method
while read
do
    line >>$OUTFILE
    :
done < $INFILE
}</pre>
```

Listing 2-12 while_line_outfile function

The function in Listing 2-12 has a big problem: it only processes every other line of the input file. Do *not* use this method! Without any variable assignment it is not very useful anyway.

Method 10: while_read_LINE_FD_IN

So far we have been doing some very straightforward kinds of loops. Have you ever used *file descriptors* to parse through a file? The use of file descriptors is sometimes a little hard to understand. I'm going to do my best to make this easy! Under the covers of the UNIX operating system, files are referenced by file descriptors. You should already know three file descriptors right off the bat. The three that I am talking about are stdin, stdout, and stderr. Standard input, or stdin, is specified as file descriptor 0. This is usually the keyboard or mouse. Standard output, or stdout, is specified as file descriptor 1. Standard output can be your terminal screen or some kind of a file.

Standard error, or stderr, is specified as file descriptor 2. Standard error is how the system and programs and scripts are able to send out or suppress error messages.

You can use these file descriptors in combination with one another. I'm sure that you have seen a shell script send all output to the bit bucket, or /dev/null. Look at the following command:

```
my_shell_script.Bash >/dev/null 2>&1
```

The result of the preceding command is to run completely silent. In other words, there is not any external output produced. Internally the script may be reading and writing to and from files and may be sending output to a specific terminal, such as /dev/console. You may want to use this technique when you run a shell script as a cron table entry or when you just are not interested in seeing any output.

In the previous example we used two file descriptors. We can also use other file descriptors to handle file input and storage. Throughout the remaining sections of this chapter, we are going to associate file descriptor 3 with stdin, file descriptor 0, and then redirect input through file descriptor 0. And for output redirection we will associate file descriptor 4 with stdout, file descriptor 1, then redirect output to our \$OUTFILE. On most UNIX systems valid file descriptors range from 0 to 19. In our case we are going to use file descriptor 3, but we could have just as easily used file descriptor 5. Later we will use this same file descriptor technique for output redirection using file descriptor 4.

There are two steps to use a file descriptor for input redirection. The first step is to associate file descriptor 3 with stdin, file descriptor 0. We use the following syntax for this step:

exec 3<&0

Now all the keyboard and mouse input is going to our new file descriptor 3. The second step is to send our input file, specified by the variable *\$INFILE*, into file descriptor 0 (zero), which is standard input. This second step is done using the following syntax:

```
exec 0< $INFILE
```

At this point any command requiring input will receive the input from the *\$INFILE* file. Now is a good time for an example. Look at the function in Listing 2-13.

```
function while_read_LINE_FD_IN
{
    # Method 10

# Zero out the $OUTFILE
>$OUTFILE
```

Listing 2-13 while_read_LINE_FD_IN function

Within the function in Listing 2-13 we have our familiar while loop to read one line of text at a time. But the beginning of this function does a little file descriptor redirection. The first exec command redirects stdin to file descriptor 3. The second exec command redirects the \$INFILE file into stdin, which is file descriptor 0. Now the while loop can just execute without our having to worry about how we assign a line of text to the LINE variable. When the while loop exits we redirect the previously reassigned stdin, which was sent to file descriptor 3, back to its original file descriptor 0, and then close file descriptor 3.

```
exec 0<&3
exec 3>&-
```

In other words, we set it back to the system's default value and close the file descriptor we used for input redirection.

Pay close attention to this method in the timing tests later in this chapter. We have many more examples using file descriptors that utilize some of our previous while loops. Please make sure you compare all of the timing results at the end of the chapter to see how these methods fare.

Method 11: cat_while_read_LINE_FD_OUT

Again, we are going to use file descriptors, but this time for standard output. For this method we are going to associate file descriptor 4 with stdout, file descriptor 1. Then we redirect standard output to our *SOUTFILE*. Study Listing 2-14 and we will cover the details at the end.

```
function cat_while_read_LINE_FD_OUT
{
# Method 11
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
cat $INFILE | while read LINE
do
       echo "$LINE"
       :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
```

Listing 2-14 cat_while_read_LINE_FD_OUT function

In Listing 2-14 we associate standard output with file descriptor 4 with the following syntax:

exec 4<&1

The next step is to redirect standard output to our *SOUTFILE*, as shown here:

exec 1> \$OUTFILE

Now that the file descriptors are set up for output redirection, we execute our loop to process the file. We cat the input file and pipe the output to our while read loop. After our loop completes, we need to put everything back to the default configuration. To do this, we restore standard output, and close file descriptor 4. The next two lines of code put us back to the default:

exec 1<&4 exec 4>&-

The notation exec 4>&- closes file descriptor 4. Be sure to check out the timing data at the end of this chapter.

Method 12: while_read_LINE_bottom_FD_OUT

As with all our functions using file descriptors, we first set up our redirection so that the *\$OUTFILE* file remains open for writing. This function uses file descriptors for standard output, but uses normal input redirection as we have seen in many other methods. Study Listing 2-15 and we will cover the function at the end.

```
function while_read_LINE_bottom_FD_OUT
{
# Method 12
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while read LINE
do
      echo "$LINE"
       :
done < $INFILE
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
```

Listing 2-15 while_read_LINE_bottom_FD_OUT function

Notice in Listing 2-15 that we do not need to append output on each loop iteration because we took care of the output redirection with file descriptors.

We first associate file descriptor 4 with standard output, file descriptor 1. We then redirect standard output to out *SOUTFILE*. Once this is set up, we execute our while loop, feeding the loop from the bottom. When the loop has completed, we restore standard output and close file descriptor 4 with the following syntax:

exec 1<&4 exec 4>&-

This is a fast method of processing a file, but is it the winner? Keep reading to find the absolute fastest method.

Method 13: while_LINE_line_bottom_FD_OUT

This section is similar to Method 12 but we replace while read LINE with while LINE='line'. Study the function in Listing 2-16 and we will cover the details at the end.

```
function while_LINE_line_bottom_FD_OUT
{
# Method 13
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4 < \& 1
exec 1> $OUTFILE
while LINE=`line`
do
       echo "$LINE"
       :
done < $INFILE
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
```

Listing 2-16 while_LINE_line_bottom_FD_OUT function

I hope you are getting the hang of file descriptors! We use exec 4<&1 to associate file descriptor 4 to standard output, file descriptor 1. Next we use exec 1>\$OUTFILE to redirect all output to our \$OUTFILE. Now we can execute our while loop to process the file data one line at a time. Notice that we feed the while loop from the bottom using input redirection after the done loop terminator, and, on each loop iteration, assign the entire line of file data to the LINE variable. The **echo** command sends each line of data to standard output, which we redirected to our \$OUTFILE using file descriptor 4. When our loop terminates execution, we restore standard output and close file descriptor 4.

Method 14: while_LINE_line_bottom_cmdsub2_FD_OUT

This section is the same as Method 13 with the exception of our commandsubstitution method. Here we replace the `command` with the \$(command) notation. Study Listing 2-17 and check out the timing data at the end of this chapter.

```
function while_LINE_line_bottom_cmdsub2_FD_OUT
{
# Method 14
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while LINE=$(line)
do
      echo "$LINE"
       :
done < $INFILE
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
```

Listing 2-17 while_LINE_line_bottom_cmdsub2_FD_OUT function

We are only interested in the difference in timing between Method 13 and Method 14 for this section. Be sure to see the execution time difference at the end of this chapter. Pay special attention to the *user* and *sys* times.

Method 15: for_LINE_cat_FILE_FD_OUT

Now we are back to our for loop method of parsing a file line-by-line. This time we are using file descriptors for output redirection to our *\$OUTFILE*. This is a surprisingly fast method to process a file. It is not the fastest method, but it ranks close to the top. Study Listing 2-18 and we will cover the details at the end.

```
for_LINE_cat_FILE_FD_OUT ()
{
# Method 15
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
for LINE in `cat $INFILE`
do
    echo "$LINE"
    :
done
# Restore standard output and close file
# descriptor 4
exec 1<\&4
exec 4>&-
}
```

Listing 2-18 for_LINE_cat_FILE_FD_OUT function

Are you getting the file descriptor thing yet? In Listing 2-18 we associate standard output with file descriptor 4, then redirect standard output to our *SOUTFILE*, as we have been doing in the last several sections.

Our for loop assigns one line of file text data to the LINE variable as we cat the \$INFILE using the back tic (`command`) command-substitution method. Our echo command does not require any redirection to the \$OUTFILE because we took care of the output with file descriptor 4. When the for loop completes execution, we restore standard output and close file descriptor 4. Please be sure to check the timing data at the end of this chapter.

Method 16: for_LINE_cat_FILE_cmdsub2_FD_OUT

We are again interested in the difference in execution timing between the `command` and \$ (command) command-substitution techniques. Other than the command substitution, Listing 2-19 is the same as Listing 2-18.

```
for_LINE_cat_FILE_cmdsub2_FD_OUT ()
{
# Method 16
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
for LINE in $(cat $INFILE)
do
    echo "$LINE"
    :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
```

Listing 2-19 for_LINE_cat_FILE_cmdsub2_FD_OUT function

Be sure to compare the timing data for Methods 15 and 16 at the end of this chapter.

Method 17: while_line_outfile_FD_IN

Now we are back to using a file descriptor for input redirection. This is one of those methods that, at first look, should work. However, this function picks up only every other line of the input file data. Do *not* use this method! But *do* study it.

In Method 17 we are attempting to use a file descriptor for file input and the line command to redirect output to the \$OUTFILE. Check out Listing 2-20.

```
while_line_outfile_FD_IN ()
{
    # Method 17

# Zero out the $OUTFILE
>$OUTFILE
```

Listing 2-20 while_line_outfile_FD_IN function

```
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
# This function processes every other
# line of the $INFILE, so do not use
# this method
while read
do
   line >> $OUTFILE
    :
done
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
```

Listing 2-20 (continued)

The timing data at the end of this chapter shows the difference in the file sizes between the input and output files. As you will see, the input file is 1 MB and the output file is 512 KB, one-half the size of the input file. Not good!

Method 18: while_line_outfile_FD_OUT

This method is the same as Method 17 but here we are redirecting standard output using file descriptor 4. Again, do not use this method, because it picks up only every other line in input file data.

In Method 18 we are attempting to use file descriptor 4 for file output using the line command and feeding the while loop from the bottom, after the done loop terminator. You can see this in Listing 2-21.

```
while_line_outfile_FD_OUT ()
{
    # Method 18

# Zero out the $OUTFILE
>$OUTFILE
```

Listing 2-21 while_line_outfile_FD_OUT function

```
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
# This function processes every other
# line of the $INFILE, so do not use
# this method
while read
do
   line
    :
done < $INFILE
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
```

Listing 2-21 (continued)

For this I just want you to compare timing data for using file descriptors for input as opposed to using a file descriptor for output. Also notice that the output file is one-half the size of the input file.

Method 19: while_line_outfile_FD_IN_AND_OUT

Okay, here we go again. This is the same method that skips every other line of input data, but this time we use file descriptors for both input and output redirection. In our timing shell script, we have the function verify_files that verifies that the file sizes match. Because this technique skips every other line, it gives us the following error:

ERROR: largefile.random.txt and writefile.out do not match -rw-r--r- 1 root system 1024000 Aug 14 17:41 largefile.random.txt -rw-r--r- 1 root system 512000 Aug 20 15:05 writefile.out

Again, we are interested only in the timing data because it does not work as desired. This is a combination of Methods 17 and 18, but this time we use file descriptors for both input and output redirection. Listing 2-22 shows how file descriptors are set up, used, and finally closed when we are finished using them.

```
while_line_outfile_FD_IN_AND_OUT ()
{
# Method 19
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while read
do
   line
   :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
```

Listing 2-22 while_line_outfile_FD_IN_AND_OUT function

See how using file descriptors for input and output speeds things up? Or does it?

Method 20: while_LINE_line_FD_IN

Here we go again with the line command. In this function the line command replaces the read command; however, we are still going to use file descriptors to gain

access to the \$INFILE file as input to our while loop. Please study the function in Listing 2-23.

```
function while_LINE_line_FD_IN
{
# Method 20
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
while LINE=`line`
do
      echo "$LINE" >> $OUTFILE
       •
done
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
```

Listing 2-23 while_LINE_line_FD_IN function

The nice thing about using file descriptors is that standard input is implied. Standard input is there; we do not have to cat the file or use a pipe for data input. We just send the file's data directly into file descriptor 0, stdin. Just don't forget to reset the file descriptor when you are finished using it.

The first exec command redirects input of file descriptor 0 into file descriptor 3. The second exec command redirects our *\$INFILE* file into *stdin*, file descriptor 0. We process the file using a while loop and then reset the file descriptor 0 back to its default and close file descriptor 3. File descriptors are really not too hard to use after scripting with them a few times. Even though we are using file descriptors to try to speed up the processing, the line command variable assignment will produce slower results than anticipated.

Method 21: while_LINE_line_cmdsub2_FD_IN

This method is just like Method 20 except for the command-substitution technique. We are going to use a large file for our timing tests and hope that we can detect a difference

between the `command` and \$ (command) command-substitution techniques in overall run time. Please study the function in Listing 2-24.

```
function while_LINE_line_cmdsub2_FD_IN
{
# Method 21
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
while LINE=$(line)
do
       echo "$LINE" >> $OUTFILE
       :
done
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
```

Listing 2-24 while_LINE_line_cmdsub2_FD_IN function

The function in Listing 2-24 first redirects stdin, file descriptor 0, to file descriptor 3; however, I could have used any valid file descriptor, such as file descriptor 5. The second step is redirecting the \$FILENAME file into stdin, which is file descriptor 0. After the file descriptor redirection we execute the while loop, and on completion file descriptor 3 is redirected back to stdin. The end result is file descriptor 0, which again references stdin. The variable assignment produced by the command substitution has a negative impact on the timing results.

Method 22: while_read_LINE_FD_IN_AND_OUT

Now take a very close look at Listing 2-25. This could be the winner! This method processes the entire file in less than one second. Some of the techniques detailed in the chapter take longer than *eight minutes*! No, really! Study this method and learn how to use it. We will cover the details at the end.

```
function while_read_LINE_FD_IN_AND_OUT
{
# Method 22
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while read LINE
đo
       echo "$LINE"
        :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
```

Listing 2-25 while_read_LINE_FD_IN_AND_OUT function

Notice that I highlighted this entire function in bold text. Listing 2-25 combines the best of the best techniques into a single function. Let's start at the top.

The first step is to zero out the \$OUTFILE using the syntax >\$OUTFILE. This is equivalent to using cat /dev/null > \$OUTFILE. I'm just saving a few key-strokes here.

Next we set up the file descriptors to handle all the file input and output. We first associate file descriptor 3 with stdin, standard input, and then we redirect input from our *\$INFILE* to file descriptor 0, stdin. With standard input set up, we move to standard output, stdout. This step involves associating file descriptor 4 with stdout, file descriptor 1. With this association made, we now redirect standard output to our *\$OUTFILE*.

Now we can process the *\$INFILE* in a simple while loop, as shown here:

```
while read LINE
do
echo "$LINE"
:
```

done

It is important to note that if the double quotes are omitted in the echo "\$LINE" statement, the new lines will not align and the output will look garbled. Again we see our no-op after the echo statement, specified by the : (colon) character. This is just a placeholder; if you remove the echo statement, the loop will still execute without failure, but it will not do anything.

After our loop completes processing the file, we need to restore standard input and standard output, and close file descriptors 3 and 4. This is done with the exec statements at the end of the function in Listing 2-25.

Make sure you see how blistering-fast this method of processing a file is at the end of this chapter.

Method 23: while_LINE_line_FD_IN_AND_OUT

How about the line command with file descriptors on both the input and output? I think you may be surprised how slow this method of processing a file can be.

Check out the function in Listing 2-26 and we will cover the details at the end.

```
function while_LINE_line_FD_IN_AND_OUT
{
    # Method 23

# Zero out the $OUTFILE
>$OUTFILE
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
# Associate standard output with file descriptor 4</pre>
```

Listing 2-26 while_LINE_line_FD_IN_AND_OUT function

```
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while LINE='line'
do
      echo "$LINE"
       •
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
```

First, notice that we are using the back tic `command` command-substitution technique in Listing 2-26. The next section is the same function with the exception of the \$(command) command-substitution technique. This is another timing test result that you should study at the end of this chapter.

In this function we associate file descriptor 3 with stdin, file descriptor 0. With this association made, we redirect our *\$INFILE* to stdin, file descriptor 0. Next we associate file descriptor 4 with stdout, file descriptor 1. With this new association made, we redirect stdout, file descriptor 1, to our *\$OUTFILE*.

With the file input and output taken care of with file descriptors, we can now process the *\$INFILE*, and send output to the *\$OUTFILE* in the while *LINE=`line`* loop. When the file processing completes, we need to restore stdin and stdout to their defaults, then close file descriptors 3 and 4.

This method is very slow, so do not use it.

Method 24: while_LINE_line_cmdsub2_FD_IN_AND_OUT

The file-processing method in Listing 2-27 is identical to Method 23, shown in Listing 2-26, except for the \$(command) command-substitution technique used here. Study Listing 2-27 and look for the execution difference, if any, in the command-substitution techniques between Methods 23 and 24 at the end of this chapter.

```
function while_LINE_line_cmdsub2_FD_IN_AND_OUT
{
# Method 24
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while LINE=$(line)
do
       echo "$LINE"
        :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
```

Listing 2-27 while_LINE_line_cmdsub2_FD_IN_AND_OUT function

That does it for our 24 functions to process a file line-by-line. Now let's time each method and analyze the data.

Timing Each Method

We have created each of the functions for the 24 different methods to parse a file line-by-line. Now we can set up a shell script to time the execution of each function to

see which one is the fastest to process a file. Earlier we wrote the random_file.Bash script that creates a user-specified MB file of random alphanumeric characters, with 80 characters per line. This file is called largefile.random.txt, which is referenced by the <code>\$OUTFILE</code> variable.

The file used for our timing test is a 12,800-line, 1 MB file. We needed this large a file to get accurate timing results for each of the 24 methods. Before we start the timing let's look at the timing shell script.

Timing Script

The shell script to time each file is not too difficult to understand. The timing mechanism is the shell's time command. The time command is followed by the name of the function, shell script, or program whose execution you want to time. The timing data is broken down to the following fields:

```
real 1m30.34s
user 0m35.50s
sys 0m52.13s
```

In the preceding output we have three measurements: real, user, and sys. The real time is the total time of execution. The user time is the time spent processing at the user/application process level. The sys time is the time spent by the system at the system/kernel level. Different UNIX flavors produce slightly different output fields, but the concepts are identical.

The one thing that users get confused about using the time command is where the timing data output goes. All of the timing data goes to stderr, or standard error, which is file descriptor 2. So the shell script or program will execute with the normal stdin and stdout, and the timing data will go the stderr. Study the shell script in Listing 2-28, and we will go through the script at the end. Then we are going show some timing data for each method.

```
#!/usr/bin/ksh
# SCRIPT: 24_ways_to_parse.ksh.ksh
# AUTHOR: Randy Michael
# DATE: 08/15/2007
# REV: 2.5.Q
#
# PURPOSE: This script shows the different ways of reading
      a file line-by-line. Again there is not just one way
#
#
      to read a file line-by-line and some are faster than
      others and some are more intuitive than others.
#
#
# REV LIST:
#
#
       03/15/2007 - Randy Michael
```

Listing 2-28 24_ways_to_parse.ksh shell script

```
Set each of the while loops up as functions and the timing
       of each function to see which one is the fastest.
#
#
#################
#
#
      08/10/2007 - Randy Michael
      Modified this script to include a total of 24 functions
#
      to parse a file line-by-line.
#
***********
#
#
      NOTE: To output the timing to a file use the following syntax:
#
#
         24_ways_to_parse.ksh file_to_process > output_file_name 2>&1
#
#
      The actual timing data is sent to standard error, file
#
      descriptor (2), and the function name header is sent
#
       to standard output, file descriptor (1).
***********
# set -n # Uncomment to check command syntax without any execution
# set -x # Uncomment to debug this script
INFILE="$1"
OUTFILE=writefile.out
TIMEFILE="/tmp/loopfile.out"
>$TIMEFILE
THIS_SCRIPT=$(basename $0)
*****
function usage
{
echo "\nUSAGE: $THIS_SCRIPT file_to_process\n"
echo "OR - To send the output to a file use: "
echo "\n$THIS_SCRIPT file_to_process > output_file_name 2>&1 \n"
exit 1
3
******
function verify_files
diff $INFILE $OUTFILE >/dev/null 2>&1
if (( $? != 0 ))
then
   echo "ERROR: $INFILE and $OUTFILE do not match"
   ls -l $INFILE $OUTFILE
fi
```

```
}
*****
function cat_while_read_LINE
{
# Method 1
# Zero out the $OUTFILE
>$OUTFILE
cat $INFILE | while read LINE
do
     echo "$LINE" >> $OUTFILE
     :
done
}
****
function while_read_LINE_bottom
{
# Method 2
# Zero out the $OUTFILE
>$OUTFILE
while read LINE
do
     echo "$LINE" >> $OUTFILE
     :
done < $INFILE
}
*****
function cat_while_LINE_line
{
# Method 3
# Zero out the $OUTFILE
>$OUTFILE
cat $INFILE | while LINE=`line`
do
     echo "$LINE" >> $OUTFILE
     :
```

```
done
}
*******
function while_LINE_line_bottom
{
# Method 4
# Zero out the $OUTFILE
>$OUTFILE
while LINE=`line`
do
      echo "$LINE" >> $OUTFILE
      :
done < $INFILE
}
*****
function cat_while_LINE_line_cmdsub2
{
# Method 5
# Zero out the $OUTFILE
>$OUTFILE
cat $INFILE | while LINE=$(line)
do
      echo "$LINE" >> $OUTFILE
      :
done
}
*****
function while_LINE_line_bottom_cmdsub2
{
# Method 6
# Zero out the $OUTFILE
>$OUTFILE
while LINE=$(line)
do
      echo "$LINE" >> $OUTFILE
```

```
:
done < $INFILE
}
****
for_LINE_cat_FILE ()
{
# Method 7
# Zero out the $OUTFILE
>$OUTFILE
for LINE in `cat $INFILE`
do
  echo "$LINE" >> $OUTFILE
   :
done
}
****
for_LINE_cat_FILE_cmdsub2 ()
{
# Method 8
# Zero out the $OUTFILE
>$OUTFILE
for LINE in $(cat $INFILE)
do
  echo "$LINE" >> $OUTFILE
   :
done
}
****
while_line_outfile ()
{
# Method 9
# Zero out the $OUTFILE
>$OUTFILE
# This function processes every other
# line of the $INFILE, so do not use
```

Listing 2-28 (continued)

```
# this method
while read
do
   line >>$OUTFILE
   :
done < $INFILE
}
****
function while_read_LINE_FD_IN
{
# Method 10
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
while read LINE
do
      echo "$LINE" >> $OUTFILE
      :
done
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
*****
function cat_while_read_LINE_FD_OUT
{
# Method 11
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
```

Listing 2-28 (continued)

```
exec 4<&1
exec 1> $OUTFILE
cat $INFILE | while read LINE
do
      echo "$LINE"
      :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
****
function while_read_LINE_bottom_FD_OUT
{
# Method 12
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while read LINE
do
      echo "$LINE"
      :
done < $INFILE
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
****
function while_LINE_line_bottom_FD_OUT
{
# Method 13
```

```
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while LINE=`line`
do
       echo "$LINE"
       :
done < $INFILE
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
*****
function while_LINE_line_bottom_cmdsub2_FD_OUT
{
# Method 14
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while LINE=$(line)
do
       echo "$LINE"
        :
done < $INFILE
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
```

```
}
*****
for_LINE_cat_FILE_FD_OUT ()
{
# Method 15
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
for LINE in `cat $INFILE`
do
   echo "$LINE"
    :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
****
for_LINE_cat_FILE_cmdsub2_FD_OUT ()
{
# Method 16
# Zero out the $OUTFILE
>$OUTFILE
\ensuremath{\texttt{\#}} Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
for LINE in $(cat $INFILE)
do
```

```
echo "$LINE"
    :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
******
while_line_outfile_FD_IN ()
{
# Method 17
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
# This function processes every other
# line of the $INFILE, so do not use
# this method
while read
do
   line >> $OUTFILE
   :
done
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
****
while_line_outfile_FD_OUT ()
{
# Method 18
```

```
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
# This function processes every other
# line of the $INFILE, so do not use
# this method
while read
do
   line
   :
done < $INFILE
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
****
while_line_outfile_FD_IN_AND_OUT ()
# Method 19
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
```

```
while read
do
   line
   :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
**********
function while_LINE_line_FD_IN
{
# Method 20
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
while LINE=`line`
do
      echo "$LINE" >> $OUTFILE
      :
done
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
*****
```

```
function while_LINE_line_cmdsub2_FD_IN
{
# Method 21
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
while LINE=$(line)
do
       echo "$LINE" >> $OUTFILE
        :
done
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
function while_read_LINE_FD_IN_AND_OUT
{
# Method 22
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while read LINE
```

```
do
      echo "$LINE"
      :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
*****
function while_LINE_line_FD_IN_AND_OUT
{
# Method 23
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while LINE=`line`
do
       echo "$LINE"
       :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
```

Listing 2-28 (continued)

```
exec 4>&-
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
****
function while_LINE_line_cmdsub2_FD_IN_AND_OUT
{
# Method 24
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while LINE=$(line)
do
      echo "$LINE"
       :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
```

Listing 2-28 (continued)

```
*****
# Test the Input
# Looking for exactly one parameter
(( $# == 1 )) || usage
# Does the file exist as a regular file?
[[ -f $1 ]] || usage
echo "\nStarting File Processing of each Method\n"
echo "Method 1:"
echo "function cat_while_read_LINE"
time cat_while_read_LINE
verify_files
sleep 1
echo "\nMethod 2:"
echo "function while_read_LINE_bottom"
time while_read_LINE_bottom
verify_files
sleep 1
echo "\nMethod 3:"
echo "function cat_while_LINE_line"
time cat_while_LINE_line
verify_files
sleep 1
echo "\nMethod 4:"
echo "function while_LINE_line_bottom"
time while_LINE_line_bottom
verify_files
sleep 1
echo "\nMethod 5:"
echo "function cat_while_LINE_line_cmdsub2"
time cat_while_LINE_line_cmdsub2
verify_files
sleep 1
echo "\nMethod 6:"
echo "function while_LINE_line_botton_cmdsub2"
time while_LINE_line_bottom_cmdsub2
verify_files
sleep 1
echo "\nMethod 7:"
echo "function for_LINE_cat_FILE"
time for_LINE_cat_FILE
verify_files
```

Listing 2-28 (continued)

sleep 1 echo "\nMethod 8:" echo "function for_LINE_cat_FILE_cmdsub2" time for_LINE_cat_FILE_cmdsub2 verify_files sleep 1 echo "\nMethod 9:" echo "function while line outfile" time while_line_outfile verify_files sleep 1 echo "\nMethod 10:" echo "function while_read_LINE_FD_IN" time while_read_LINE_FD_IN verify_files sleep 1 echo "\nMethod 11:" echo "function cat_while_read_LINE_FD_OUT" time cat_while_read_LINE_FD_OUT verify_files sleep 1 echo "\nMethod 12:" echo "function while read LINE bottom FD OUT" time while_read_LINE_bottom_FD_OUT verify_files sleep 1 echo "\nMethod 13:" echo "function while_LINE_line_bottom_FD_OUT" time while_LINE_line_bottom_FD_OUT verify_files sleep 1 echo "\nMethod 14:" echo "function while_LINE_line_bottom_cmdsub2_FD_OUT" time while_LINE_line_bottom_cmdsub2_FD_OUT verify_files sleep 1 echo "\nMethod 15:" echo "function for_LINE_cat_FILE_FD_OUT" time for_LINE_cat_FILE_FD_OUT verify_files sleep 1 echo "\nMethod 16:" echo "function for_LINE_cat_FILE_cmdsub2_FD_OUT" time for_LINE_cat_FILE_cmdsub2_FD_OUT verify_files sleep 1 echo "\nMethod 17:" echo "function while_line_outfile_FD_IN"

Listing 2-28 (continued)

```
time while_line_outfile_FD_IN
verify_files
sleep 1
echo "\nMethod 18:"
echo "function while_line_outfile_FD_OUT"
time while_line_outfile_FD_OUT
verify_files
sleep 1
echo "\nMethod 19:"
echo "function while_line_outfile_FD_IN_AND_OUT"
time while_line_outfile_FD_IN_AND_OUT
verify_files
sleep 1
echo "\nMethod 20:"
echo "function while_LINE_line_FD_IN"
time while_LINE_line_FD_IN
verify_files
sleep 1
echo "\nMethod 21:"
echo "function while_LINE_line_cmdsub2_FD_IN"
time while_LINE_line_cmdsub2_FD_IN
verify_files
sleep 1
echo "\nMethod 22:"
echo "function while_read_LINE_FD_IN_AND_OUT"
time while_read_LINE_FD_IN_AND_OUT
verify_files
sleep 1
echo "\nMethod 23:"
echo "function while_LINE_line_FD_IN_AND_OUT"
time while_LINE_line_FD_IN_AND_OUT
verify_files
sleep 1
echo "\nMethod 24:"
echo "function while_LINE_line_cmdsub2_FD_IN_AND_OUT"
time while_LINE_line_cmdsub2_FD_IN_AND_OUT
verify_files
```

Listing 2-28 (continued)

The shell script in Listing 2-28 first defines all the functions that we previously covered in the Method sections. After the functions are defined, we do a little testing of the input. We are expecting exactly one command parameter, and it should be a regular file. Look at the code block in Listing 2-29 to see the file testing.

```
# Test the Input
# Looking for exactly one parameter
(( $# == 1 )) || usage
```

Listing 2-29 Code to test command input

```
# Does the file exist as a regular file?
[[ -f $1 ]] || usage
```

Listing 2-29 (continued)

The first test checks to ensure that the number of command parameters, specified by the \$# operator, is exactly one. Notice that we used the *double parentheses* mathematical test, specified as ((*math test*)). Additionally, we used a logical OR, specified by ||, to execute the usage function if the number of parameters is not equal to one.

We use the same type of test for the file to ensure that the file exists and the file is a regular file, as opposed to a character or block special file. When we do the test, notice that we use the *double bracket* test for character data, specified by [[*character test*]]. This is an important distinction to note. We again use the logical OR to execute the usage function if the return code from the test is nonzero.

Now we start the actual timing tests. In doing these tests we execute the Method functions one at a time. The function's internal loop does the file processing, but we redirect each function's output to our *SOUTFILE* file so that we always perform the exact same task of reading the *SINFILE* line-by-line, and writing to the *SOUTFILE*. This allows us to get measurable system activity. As I stated before, the timing measurements produced by the time commands go to *stderr*, or file descriptor 2, which will just go to the screen by default. When this shell script executes, there are three things that go to the screen, as you will see in Listing 2-30. You can also send all this output to a file by using the following command syntax:

24_ways_to_parse.ksh largefile.random.txt > 24_ways_timing.out 2>&1

The preceding command starts with the script name, followed by the file to parse through. The output is redirected to the file 24_ways_timing.out with stderr (file descriptor 2) redirected to stdout (file descriptor 1), specified by 2>&1. Do not forget the ampersand, &, before the 1. If the & is omitted, a file with the name 1 will be created. This is a common mistake when working with file descriptors. The placement of the stderr to stdout is important in this case. If the 2>&1 is at the end of the command, you will not get the desired result, which is all of the timing data going to a data file. In some cases the placement of the 2>&1 redirection does not matter, but it does matter here.

Timing Data for Each Method

Now all the hard stuff has been done. We have a 12,800-line 1MB file of random alphanumeric characters, 80 per line, largefile.random.txt, and we have our shell script written, so let's find out which function is the fastest in Listing 2-30. Pay particular attention to the bold text in the timing results in Listing 2-30.

```
Starting File Processing of each Method
Method 1:
function cat_while_read_LINE
real 0m8.31s
```

Listing 2-30 Timing data for each loop method

```
0m3.18s
user
       0m8.86s
sys
Method 2:
function while_read_LINE_bottom
       0m0.61s
real
       0m0.41s
user
       0m0.20s
sys
Method 3:
function cat_while_LINE_line
real
       0m30.87s
user 0m13.74s
     0m20.18s
sys
Method 4:
function while_LINE_line_bottom
real 0m31.24s
user 0m13.57s
sys 0m19.70s
Method 5:
function cat_while_LINE_line_cmdsub2
real
      0m30.86s
user
      0m13.84s
sys 0m20.20s
Method 6:
function while_LINE_line_botton_cmdsub2
      0m31.20s
real
      0m13.69s
user
    0m19.70s
sys
Method 7:
function for_LINE_cat_FILE
real
       0m0.74s
user
       0m0.54s
sys
       0m0.21s
Method 8:
function for_LINE_cat_FILE_cmdsub2
real
        0m0.75s
```

Listing 2-30 (continued)

```
user
       0m0.50s
      0m0.24s
sys
Method 9:
function while_line_outfile
real
       4m21.14s
       0m9.44s
user
sys
      4m8.86s
ERROR: largefile.random.txt and writefile.out do not match
            1 root system
                                 1024000 Aug 14 17:41
-rw-r--r--
largefile.random.txt
-rw-r--r-- 1 root system 512000 Aug 20 14:43 writefile.out
Method 10:
function while_read_LINE_FD_IN
real
       0m0.62s
       0m0.42s
user
      0m0.19s
sys
Method 11:
function cat_while_read_LINE_FD_OUT
real
     0m7.95s
     0m3.10s
user
sys Om8.89s
Method 12:
function while_read_LINE_bottom_FD_OUT
       0m0.30s
real
       0m0.29s
user
sys 0m0.00s
Method 13:
function while_LINE_line_bottom_FD_OUT
real
     8m33.44s
user 0m14.17s
    8m15.83s
sys
Method 14:
function while_LINE_line_bottom_cmdsub2_FD_OUT
     8m28.52s
real
     0m13.97s
user
sys 8m11.70s
```

Listing 2-30 (continued)

```
Method 15:
function for_LINE_cat_FILE_FD_OUT
real
       0m0.47s
user
       0m0.38s
sys
      0m0.06s
Method 16:
function for_LINE_cat_FILE_cmdsub2_FD_OUT
real
       0m0.45s
       0m0.38s
user
sys
      0m0.05s
Method 17:
function while_line_outfile_FD_IN
real
       4m21.52s
       0m9.54s
user
svs
      4m9.39s
ERROR: largefile.random.txt and writefile.out do not match
-rw-r--r--
            1 root
                     system
                                 1024000 Aug 14 17:41
largefile.random.txt
-rw-r--r-- 1 root
                       system
                                 512000 Aug 20 15:05 writefile.out
Method 18:
function while_line_outfile_FD_OUT
real 4m16.12s
      0m9.33s
user
      4m4.43s
sys
ERROR: largefile.random.txt and writefile.out do not match
-rw-r--r-- 1 root system 1024000 Aug 14 17:41 largefile.random.txt
-rw-r--r-- 1 root system
                                   512000 Aug 20 15:09 writefile.out
Method 19:
function while_line_outfile_FD_IN_AND_OUT
      4m16.67s
real
user 0m9.34s
      4m4.93s
sys
ERROR: largefile.random.txt and writefile.out do not match
-rw-r--r-- 1 root system 1024000 Aug 14 17:41 largefile.random.txt
-rw-r--r--
           1 root
                     system
                                   512000 Aug 20 15:13 writefile.out
Method 20:
function while_LINE_line_FD_IN
```

Listing 2-30 (continued)

```
real
     8m33.41s
user 0m14.41s
sys 8m16.23s
Method 21:
function while_LINE_line_cmdsub2_FD_IN
real 8m35.62s
user 0m14.46s
sys 8m17.71s
Method 22:
function while_read_LINE_FD_IN_AND_OUT
       0m0.31s
real
       0m0.29s
user
sys 0m0.00s
Method 23:
function while_LINE_line_FD_IN_AND_OUT
real
     8m27.93s
user 0m14.03s
sys 8m11.04s
Method 24:
function while_LINE_line_cmdsub2_FD_IN_AND_OUT
real 8m27.55s
user 0m14.04s
sys 8m10.56s
```

Listing 2-30 (continued)

As you can see, all file-processing loops are not created equal. Two of the methods are tied for first place. Methods 12 and 22 produce the same real execution times at 0.30 and 0.31 seconds to process a 12,800-line 1 MB file. Method 16 came in second at 0.45 seconds. The remaining methods range from almost less than 1 second to more than 8 minutes. The sorted timing output for the real time is shown in Listing 2-31.

```
        real
        0m0.30s
        Method 12:

        real
        0m0.31s
        Method 22:

        real
        0m0.45s
        Method 16:

        real
        0m0.45s
        Method 15:
```

Listing 2-31 Sorted timing data by method

```
Om0.61s Method 2:
real
real 0m0.62s Method 10:
real 0m0.74s Method 7:
real 0m0.75s Method 8:
real 0m30.86s Method 5:
real 0m30.87s Method 3:
real 0m31.20s Method 6:
real 0m31.24s Method 4:
real 0m7.95s Method 11:
real 0m8.31s Method 1:
real 4m16.12s Method 18: *FAILS*
real 4m16.67s Method 19: *FAILS*
real 4m21.14s Method 9: *FAILS*
real 4m21.52s Method 17: *FAILS*
real 8m27.55s Method 24:
real 8m27.93s Method 23:
real 8m28.52s Method 14:
real 8m33.41s Method 20:
real 8m33.44s Method 13:
real 8m35.62s Method 21:
```

Listing 2-31 (continued)

Notice that the first eight methods all process this 1 MB text file in *less than one second*. Then notice that the last six methods all take longer than *eight minutes*! Now I hope you see why it is very important *how* we process files. The percentage difference between the fastest method and the slowest method is a whopping 177,773.333 percent!

Let's take a look at the code for the top four techniques. The order of appearance is Methods 12, 22, 16, 15, and 2. For the sake of argument, let's just say that Methods 12 and 22 are tied for first place. Listing 2-32 shows Method 12.

```
function while_read_LINE_bottom_FD_OUT
{
    # Method 12

    # Zero out the $OUTFILE
    >$OUTFILE

    # Associate standard output with file descriptor 4
    # and redirect standard output to $OUTFILE
    exec 4<&1
    exec 1> $OUTFILE
    while read LINE
```

Listing 2-32 Method 12 – tied for first place

```
do
        echo "$LINE"
        :
        done < $INFILE
    # Restore standard output and close file
    # descriptor 4
    exec 1<&4
    exec 4>&-
    }
```

Listing 2-32 (continued)

The method in Listing 2-32 is tied for first place. This method uses my favorite input redirection for files by redirecting input at the bottom of the loop, after the done loop terminator. This method does use a file descriptor for redirecting standard output, stdout, to file descriptor 1.

The input redirection using done < \$INFILE greatly speeds up any loop that requires file input. The nice thing about this method of input redirection is that it is intuitive to use for beginners to shell scripting. I was actually surprised that this method tied, actually won by 10 ms, using file descriptors for both input and output files, as shown in Listing 2-33.

```
function while_read_LINE_FD_IN_AND_OUT
{
    # Method 22

# Zero out the $OUTFILE
>$OUTFILE
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while read LINE
```

Listing 2-33 Method 22 – tied for first place

```
do
        echo "$LINE"
        :
        done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
```

Listing 2-33 (continued)

I tend not to use this method when I write shell scripts because it can be difficult to maintain through the code life cycle. If a user is not familiar with using file descriptors, then a script using this method is extremely hard to understand. The method in Listing 2-32 produces the same timing results, and it is much easier to understand. Listing 2-34 shows the second-place loop method.

```
for_LINE_cat_FILE_cmdsub2_FD_OUT ()
{
    # Method 16

# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
for LINE in $(cat $INFILE)
do
    echo "$LINE"
:
```

Listing 2-34 Method 16 - second place in timing tests

```
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
```

Listing 2-34 (continued)

The method shown in Listing 2-34 is another surprise: a for loop using command substitution with file descriptor file output redirection. We try this same type of for loop in Listing 2-35, only changing the command-substitution method.

```
for_LINE_cat_FILE_FD_OUT ()
{
# Method 15
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
for LINE in `cat $INFILE`
do
   echo "$LINE"
   :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
```

Listing 2-35 Method 15 – third place in timing tests

As shown in Listing 2-36, the only difference between Methods 15 and 16 is the technique used for command substitution.

```
Method 15:

function for_LINE_cat_FILE_FD_OUT

real 0m0.47s

user 0m0.38s

sys 0m0.06s

Method 16:

function for_LINE_cat_FILE_cmdsub2_FD_OUT

real 0m0.45s

user 0m0.38s

sys 0m0.05s
```

Listing 2-36 Timing difference between Methods 15 and 16

The timing results indicate 10 ms in system time, or system/kernel time between the command-substitution methods, for a total of 20 ms in real execution time. The method in Listing 2-37 is the fastest method to process a file line-by-line that does not use file descriptors.

```
function while_read_LINE_bottom
{
    # Method 2

    # Zero out the $OUTFILE
    >$OUTFILE
    while read LINE
    do
        echo "$LINE" >> $OUTFILE
        :
    done < $INFILE
    }
</pre>
```

Listing 2-37 The fastest method not using file descriptors

I use this technique in almost every shell script that does file parsing simply because of the ease of maintaining the shell script throughout the life cycle. Remember, it is not *if* someone will come behind you and need to edit the script sometime in the future, but *when* someone comes behind you to edit your *most excellent script*. They must be able to understand what you did without having to either hack through your code, or just give up and resort to rewriting the entire script.

Timing Command-Substitution Methods

We also want to take a look at the difference in timing when we used the two different methods of command substitution using `command` versus \$(command). See Listing 2-38, in which Method 4 uses the back tic command-substitution method and Method 5 uses the dollar parentheses method of command substitution.

```
function while_LINE_line_bottom
{
# Method 4
# Zero out the $OUTFILE
>$OUTFILE
while LINE=`line`
do
       echo "$LINE" >> $OUTFILE
       :
done < $INFILE
}
****
function cat_while_LINE_line_cmdsub2
{
# Method 5
# Zero out the $OUTFILE
>$OUTFILE
cat $INFILE | while LINE=$(line)
do
       echo "$LINE" >> $OUTFILE
       :
done
}
```

Listing 2-38 Command-substitution methods

The timing data for these two methods is shown in Listing 2-39.

```
Method 4:
function while_LINE_line_bottom
real 0m31.24s
```

Listing 2-39 Command-substitution timing difference

```
user 0m13.57s
sys 0m19.70s
Method 5:
function cat_while_LINE_line_cmdsub2
real 0m30.86s
user 0m13.84s
sys 0m20.20s
```

Listing 2-39 (continued)

In Method 4 the command-substitution technique uses back tics, `command`, which are located in the top-left corner of a standard keyboard. The command-substitution technique used in Method 5 is the dollar parentheses technique, \$(command). Both command-substitution methods give the same end result, but one method is slightly faster than the other in *real* time. From the timing of each method in Listing 2-39, the \$(command) method won the *real time* race by only 0.38 seconds. But, look at both the *user* and *sys* times! The back tic method uses fewer system resources. However, this difference in timing is so small that it is really not an issue.

What about Using Command Input Instead of File Input?

I get this a lot. We do not always want to redirect command output to a file, and then process the file. That extra step takes up both system CPU cycles and disk I/O time. Of course, we can run the command and pipe the output to a while read loop. The problem with a pipe is the 2,048-character limit of "system temporary files" that the pipe uses.

The technique shown in Listing 2-40 is a nice little trick to save both CPU cycles and disk I/Os, and to get around the pipe's 2,048-character limit.

```
while read LINE
do
    echo "$LINE"
done < <(command)</pre>
```

Listing 2-40 Using command output in a loop

Now, I know this looks a bit odd. I got this trick from one of my co-workers, Brian Beers. What we are doing here is input redirection from the bottom of the loop, after the done loop terminator, specified by the done < notation. The < (command) notation executes the command and points the command's output into the bottom of the loop.

NOTE The space between the two < < in < < (command) is required!

Summary

This chapter covered the various techniques for parsing a file line-by-line that I have seen over the years. You may have seen even more oddball ways to process a file. The two points that I wanted to make in this chapter are these: first, there are many ways to handle any task on a UNIX platform, and second, some techniques that are used to process a file waste a lot of CPU time. Most of the wasted time is spent in unnecessary variable assignments and continuously opening and closing the same file over and over. Using a pipe also has a negative impact on the loop timing, and adds the limitation of the 2,048-character length that the pipe has.

On a small file this is not a big deal, but for large parsing jobs a large delta in timing can have a huge impact both on the machine resources and on the time involved.

Lab Assignments

1. Evaluate the following function:

```
while_true_read_echo_line ()
{
while true
do
    read LINE
    echo "$LINE" > $OUTFILE
    :
    done < $INFILE
}</pre>
```

Does it work?

If it works, how long does it take to process a 1 MB file?

If it works, by what percentage can you improve the timing by using file descriptors for

- Input file?
- Output file?
- Both input and output files?
- 2. Write a shell script that will display every line in the /etc/hosts file that is not commented out with a # (hash mark), using command, not file, input redirection.

CHAPTER 3

Automated Event Notification

To solve problems proactively, early warning is essential. In this chapter we are going to look at some techniques of getting the word out by automating the notification when a system event occurs. When we write monitoring shell scripts and there is a failure, success, or request, we need a method of getting a message to the right people. There are really three main strategies of notification in shell scripts. The first is to send an email directly to the user. We can also send an alphanumeric page by email to the user for immediate notification to a cell phone or pager. The third is to send a text page by dialing a modem to the service provider. We are mainly going to look at the first two methods, but we will also list some good software products that will send text pages by dialing the modem and transferring the message to the cellular/ pager provider.

In some shops email is so restricted that you have to use a little trick or two to get around some of the restrictions. We will cover some of these situations, too.

Basics of Automating Event Notification

In a shell script there are times when you want to send an automated notification. As an example, if you are monitoring filesystems and your script finds that one of the filesystems has exceeded the maximum threshold, then most likely you want to be informed of this situation. I always like an email notification when the backups complete every night — not just when there is a backup error, but when the backup is successful, too. This way I always know the status of last night's backup every morning just by checking my email. I also know that a major backup problem occurred if no email was sent at all. There are a few ways to do the notification, but the most common is through email to a cell phone, a text pager, or an email account. In the next few sections we are going to look at the techniques to get the message out, even if only one server has mail access.

Using the mail and mailx Commands

The most common notification methods use the **mail** and **mailx** commands. To add a subject to the email, we can add the -s switch followed by the subject enclosed in quotes. The basic syntax of both email commands is shown in the following code:

```
mail -s "This is the subject" $MAILOUT_LIST < $MAIL_FILE
or
cat $MAIL_FILE | mail -s "This is the subject" $MAILOUT_LIST
or
mailx -s "This is the subject" $MAILOUT_LIST < $MAIL_FILE
or
cat $MAIL_FILE | mailx -s "This is the subject" $MAILOUT_LIST</pre>
```

Notice the use of the MAILOUT_LIST and MAIL_FILE variables. The MAILOUT_LIST variable contains a list of email addresses, or email aliases, to send the message to.

The following examples use a *sendmail alias*, specified by mail_alias. The MAIL_FILE variable points to a filename that holds the message text to be sent.

mail -s "This is the subject" mail_alias > \$MAIL_FILE

or

cat \$MAIL_FILE | mail -s "This is the subject" mail_alias

or

mailx -s "This is the subject" mail_alias > \$MAIL_FILE

or

cat \$MAIL_FILE | mailx -s "This is the subject" mail_alias

NOTE Not all systems support the mailx command, but the systems that do have support use the same syntax as the mail command. To be safe when dealing with multiple UNIX platforms, always use the mail command.

The MAIL_FILE variable points to a filename that holds the message text to be sent. Let's look at each of these individually.

Suppose we are monitoring the filesystems on a machine and the /var filesystem has reached 98 percent utilization, which is over the 85-percent threshold for a filesystem to be considered full. The Systems Administrator needs to get a page about this situation

quickly, or we may have a machine crash when /var fills up; the same is true for the root filesystem, /. In the monitoring shell script there is a MAIL_FILE variable defined to point to the filename /tmp/mailfile.out,MAILFILE=/tmp/mailfile.out.Then we create a zero-sized mail-out file using cat /dev/null > \$MAIL_FILE. When an error is found, which in our case is when /var has reached 98 percent, a message is appended to the \$MAIL_FILE for later mailing. If more errors are found, they are also appended to the file as the shell script processes each task. At the end of the shell script we can test the size of the \$MAIL_FILE. If the \$MAILFILE has any data in it, the file will have a size greater than 0 bytes. If the file has data, we mail the file. If the file is empty with a 0-byte file size, we do nothing.

To illustrate this idea, let's study the code segment in Listing 3-1.

```
MAIL_FILE=/tmp/mailfile.out
cat /dev/null > $MAIL_FILE
MAIL_LIST="randy@my.domain.com 1234567890@mypage_somebody.net"
check_filesystems  # This function checks the filesystems percentage
if [ -s $MAIL_FILE ]
then
            mail -s "Filesystem Full" $MAIL_LIST > $MAIL_FILE
fi
```

Listing 3-1 Typical mail code segment

In Listing 3-1 we see a code segment that defines the MAIL_FILE and MAIL_LIST variables that we use in the mail command. After the definitions this code segment executes the function that looks for filesystems that are over the threshold. If the threshold is exceeded, a message is appended to the \$MAIL_FILE file, as shown in the following code segment:

```
FS=/var
PERCENT=98
THISHOST=$(uname -n)
echo "$THISHOST: $FS is $PERCENT" | tee -a $MAIL_FILE
```

This code segment is from the checkfilesystems function. For my machine, this echo command statement would both display the following message to the screen and append it to the \$MAILFILE file:

yogi: /var is 98%

The hostname is yogi, the filesystem is /var, and the percentage of used space is 98. Notice the **tee** command after the pipe (|) from the echo statement. In this case we want to display the results on the screen and send an email with the same data.

The tee -a command does this double duty when you pipe the output to | tee -a \$FILENAME.

After the checkfilesystems function finishes, we test the size of the \$MAILFILE. If it is greater than 0 bytes in size, we send a mail message using the mail command. The following message is sent to the randy@my.domain.com and 1234567890@mypage_ somebody.net email addresses:

yogi: /var is 98%

Setting Up a sendmail Alias

It is far easier to set up and maintain a sendmail alias than to add and delete email addresses by editing a shell script. The first step is to find the aliases file. Depending on the UNIX flavor, you will find the aliases file in /etc, /etc/mail, or /etc/sendmail. Next, edit the aliases file using your favorite text editor. As an example, let's add a new alias called sasupport. This new alias will consist of the randy@my.domain.com and 1234567890@mypagesomebody.net email addresses:

sa_support: randy@my.domain.com, 1234567890@mypagesomebody.net

For the new alias to be recognized, we must run the **newaliases** command. The newaliases command forces sendmail to reread the aliases file. This command also does error checking for you.

This sendmail alias entry for sasupport allows us to send an email to everyone in the list by using the sasupport sendmail alias, as follows:

mailx -s "This is the subject" sasupport < \$MAIL_FILE</pre>

Remember to put a comma (,) between each email address in the aliases file.

Problems with Outbound Mail

Before we hard-code the mail command into your shell script we need to do a little test to see if we can get the email to the destination without error. To test the functionality, add the -v switch to the mail or mailx command, as shown in Listing 3-2.

Listing 3-2 Testing the mail service using mail -v

With the -v switch added to the mail command, all the details of the delivery are displayed on the user's terminal. From the delivery details we can see any errors that happen until the file is considered "sent" by the local host. If the message is not delivered to the target email address, then further investigation is needed. Listing 3-3 shows an example of using the -v, verbose, mail switch.

```
randy@yogi... Connecting to [127.0.0.1] via relay ...
220 booboo ESMTP Sendmail 8.14.1/8.14.1; Wed, 5 Dec 2007 08:27:05 -0500
>>> EHLO booboo
250-booboo Hello localhost.localdomain [127.0.0.1], pleased to meet you
250-ENHANCEDSTATUSCODES
250-PIPELINING
250-8BITMIME
250-SIZE
250-DSN
250-ETRN
250-AUTH DIGEST-MD5 CRAM-MD5
250-DELIVERBY
250 HELP
>>> MAIL From:>root@booboo> SIZE=1692 AUTH=root@booboo
250 2.1.0 >root@booboo>... Sender ok
>>> RCPT To:>randy@yogi>
>>> DATA
250 2.1.5 >randy@yogi>... Recipient ok
354 Enter mail, end with "." on a line by itself
>>> .
250 2.0.0 1B5DR5av009324 Message accepted for delivery
randy@yogi... Sent (1B5DR5av009324 Message accepted for delivery)
Closing connection to [127.0.0.1]
>>> OUIT
221 2.0.0 booboo closing connection
```

Listing 3-3 Example of verbose mail mode using the -v switch

As you can see, using mail-v gives a lot of detail for troubleshooting a failed connection to the mail server. A few troubleshooting tips are included here. The first step is to ensure sendmail is running by executing the following command:

```
[yogi]@/etc/mail# ps ax | grep sendmail
5168 - A 2:37 sendmail: accepting connections
14810 pts/0 A 0:00 grep sendmail
[root:yogi]@/etc/mail#
```

If you do not see sendmail running, it needs to be started. See the Systems Administrator, if that person is not you.

If sendmail is running, you may need to add your location's mail gateway if you have a centralized email server. To configure this, locate the sendmail.cf file in the

/etc, /etc/mail, or /etc/sendmail directory, depending on your UNIX flavor. The sendmail.cf file is a very complex file full of rules. We just want to search for the DS field, which specifies the smart relay host. If the DS field is not set, add your mail gateway server name. For example, we have one called mailgw defined in our /etc/hosts file, so I add the following entry to the sendmail.cf file:

DSmailgw

or

DSmailgw.my_company_domain.com

Notice we may need the fully qualified name of the mail server. Do not add a space between the DS and mailgw. You can also use an IP address.

Now we need to have sendmail reread the configuration file sendmail.cf. To do this we need to find the sendmail process ID and run a kill-HUP process_id. Don't worry; this command just causes a reread of the sendmail.cf file. In AIX, if sendmail is under control of the Systems Resource Controller (SRC), you can use refresh-s sendmail to have sendmail reread the configuration file. You can check by running lssrc-s sendmail.

```
[root:yogi]@/etc/mail# ps ax | grep sendmail
5168 - A 2:37 sendmail: accepting connections
14810 pts/0 A 0:00 grep sendmail
[root:yogi]@/etc/mail# kill -HUP 5168
[root:yogi]@/etc/mail#
```

After completing this step, try sending another email. Remember to add the -v switch to the mail command.

If the email still does not get to the destination, we can try one more thing. Let's ensure that the hosts have a fully qualified hostname in the /etc/hosts file, unless you are using DNS. A fully qualified hostname includes the domain part of the network name. Here is an example:

yogi.mycompany_domain.com

If you use a /etc/hosts file, add an alias using the fully qualified hostname. Be sure to check the mail gateway, or smart relay server, as well as the machines where the script is executing.

If you still cannot send an email, see the Systems Administrator and Network Administrator. The next two sections look at some alternative techniques.

Creating a "Bounce" Account with a .forward File

I worked at one shop where only one UNIX machine in the network, other than the mail server, was allowed to send email outside of the LAN. This presented a problem for all of the other machines to get the message out when a script detected an error.

The solution we used was to create a user account on the UNIX machine that could send email outbound. Then we locked down this user account so no one could log in remotely. Let's say we create a user account called bounce. In the /home/bounce directory we create a file called /home/bounce/.forward. Then in the .forward file we add the email address to which we want to forward all mail. You can add as many email addresses to this file as you want, but be aware that every single email will be forwarded to *each* address listed in the .forward file.

On this single machine that has outside LAN mailing capability we added the user bounce to the system. Then in the /home/bounce directory we created a file called . forward that has the following entries:

```
randy@my.domain.com
1234567890@mypage_somebody.net
```

This .forward file will forward all mail received by the bounce user to the randy@my.domain.com and 1234567890@mypagesomebody.net email addresses. This way I have an email to my desktop, and I am also notified on my cell phone by text page. On all the other machines we have two options. The first option is to edit all the shell scripts that send email notification and change the \$MAIL_LIST variable to

MAIL_LIST="bounce@dino."

This entry assumes that the dino host is in the same domain, specified by the period that follows the hostname **dino** (dino.).

As we saw earlier, an easier way is to create some entries in the aliases file for sendmail. The aliases file is usually located in /etc/aliases, but you may find it in /etc/mail/aliases or /etc/sendmail/aliases on some operating systems. The format of defining an alias is a name, username, or tag, followed by one or more email addresses. The following is an example of an aliases file:

admin: bounce@dino., randy, brad, cindy, jon, pepe

This aliases file entry creates a new alias called admin that automatically sends email to the bounce account on dino and also to randy, brad, cindy, jon, and pepe.

Before these changes will take effect, we need to run the newaliases command. The sendmail -bi command works, too.

Using the sendmail Command to Send Outbound Mail

In another shop where I worked, I could not send outbound mail from any user named root. The *from* field had to be a valid email address that is recognized by the mail server, and root is not valid. To get around this little problem, I changed the command that I used from mail to sendmail or mailx. The **sendmail** command enables us to add the -f switch to indicate a valid internal email address for the *from* field. The mailx command supports the same functionality using the -r switch. The sendmail command is in /usr/sbin/sendmail on AIX, HP-UX, Linux, and OpenBSD, but on SunOS the location changed to /usr/lib/sendmail. Look at the function in Listing 3-4, and we will cover the details at the end.

Listing 3-4 send_notification function

Notice in Listing 3-4 that we added another variable, MAILOUT. This variable is used to turn on/off the email notifications. If the \$MAILOUT variable points to TRUE, and the \$MAIL_FILE file is not empty, then the email is sent. If the \$MAILOUT variable does not equal the string TRUE, the email is disabled. This is just another way to control the email notifications.

In the case statement we use the output of the **uname** command to set the correct command path for the sendmail command on the UNIX platform. For AIX, HP-UX, Linux, and OpenBSD the sendmail command path is /usr/sbin. On SunOS the sendmail path is /usr/lib. We assign the correct path to the SENDMAIL variable, and we use this variable as the command to send the mail. Once the command is defined, we issue the command, as shown here:

```
$SENDMAIL -f randy@$THISHOST $MAIL_LIST > $MAIL_FILE
```

We issue the sendmail command using the -f switch and follow the switch by a valid email account name, which is randy@\$THISHOST. Remember that we defined the THISHOST variable to the local machine's hostname. The *from* address is followed by the list of email addresses, and the message file is used by redirecting input into the sendmail command. We can also use the following syntax:

cat \$MAIL_FILE | \$SENDMAIL -f randy@\$THISHOST \$MAIL_LIST

Either sendmail statement will send the mail, assuming the mail server and firewall allow outgoing mail.

We can do the exact same thing using mailx-r randy@\$THISHOST command syntax:

cat \$MAIL_FILE | mailx -r randy@\$THISHOST \$MAIL_LIST

The mailx command is located in /usr/bin. If you want a subject heading, add the -s switch followed by the subject text enclosed in double quotes.

Dial-Out Modem Software

Many good products are on the market, both freeware and commercial, that handle large amounts of paging better than any shell script could ever do. They also have the ability to dial the modem and send the message to the provider. A list of such products is shown in Table 3-1.

Table 3-1 shows only a sample of the products available for paging in a UNIX environment. There are others for Windows systems. The nice thing about these products is the ability to dial out on a modem. At some level in every shop there is a need to use a phone line for communications instead of the network. This gives you the ability to get the message out even if the network is having a problem.

PRODUCT	DESCRIPTION
Freeware and Shareware Products	
QuickPage	Client/server software used to send messages to alphanumeric pagers.
SMS Client	Command-line utility for UNIX that allows you to send SMS messages to cell phones and pagers.
HylaFAX+	Faxing product for UNIX that allows dial-in, dial-out, fax-in, fax-out, and pager notifications.
Commercial Products	
EtherPage	Enterprise-wide alphanumeric pager software made by iTechTool.
TelAlert	Pager notification and interactive voice response software made by CalAmp.
FirstPAGE	Supports all national paging networks using IXO/TAP; made by Segent.

Table 3-1 Products That Handle High-Volume Paging and Modem Dialing

SNMP Traps

Most large shops use an enterprise-monitoring tool to monitor all of the systems from a central management console. The server software is installed on a single machine called the management station. All of the managed/monitored machines have the client software installed. This client software is an SNMP agent and uses a local MIB to define the *managed objects*, or *management variables*. These managed objects define things such as the filesystems to monitor and the trigger threshold for detecting a full filesystem. When the managed object, which in this case is a full filesystem, exceeds the set threshold, a local SNMP *trap* is generated and the management station captures the trap and performs the predefined action, which may be to send a text page

to the Systems Administrator. To understand what an SNMP trap is, let's review a short explanation of each of the pieces:

- SNMP (Simple Network Management Protocol) SNMP is a protocol used for agent communications. The most common use for the SNMP protocol is client/server system management software.
- MIB (Management Information Base) Each managed machine, or agent, in an SNMP-managed network maintains a local database of information (MIB) defined to the network managed machine. An SNMP-compliant MIB contains information about the property definitions of each of the managed resources.
- SNMP trap When a threshold is exceeded there is an event notification to the management server from an agent-generated event, called a trap. The server management station receives and sets objects in the MIB, and the local machine, or agent, notifies the management station of client-generated events, or traps. All of the communication between the network management server and its agents, or management clients, takes place using SNMP.

The nice thing about using an enterprise management tool is that it utilizes SNMP. With most products you can write your own shell scripts using SNMP traps. The details vary for the specific syntax for each product, but with the software installed you can have your shell scripts perform the same notifications that the enterprise management software produces. Using Tivoli NetView, EcoTools, or BMC Patrol (just to name a few), you have the ability to incorporate SNMP traps into your own shell scripts for event notifications. Please refer to the product documentation for details on creating and using SNMP traps.

Summary

This chapter is intended to give a brief overview of some techniques of getting critical information out to the system-management community. This chapter mainly focused on email and some different techniques for using the mail commands.

The topics discussed here form the basics for notification of system problems. You should be able to extend the list of notification techniques without much effort. If you have an enterprise management solution installed at your shop, study the vendor documentation on using and creating SNMP traps. There are books based entirely on SNMP, and the information is just too long to cover in this book, but it is an important notification method that you need to be familiar with. If you have trouble getting the email solution to work, talk with the Network Manager to find a solution.

In the next chapter we move on to look at creating progress indicators to give our users feedback on long-running processes. The topics include a series of dots as the processing continues, a line that appears to rotate as processing continues, and elapsed time.

Lab Assignments

1. Send the following email message to yourself from your UNIX server:

This is a test from *hostname*

where hostname is the UNIX hostname from which the email originates. Use each method, including mail, mailx, and sendmail commands in a shell script.

2. Send the same message using mailx and sendmail commands, but this time specify a return email address that is not your own. When you receive the email, who is it from?

CHAPTER 4

Progress Indicators Using a Series of Dots, a Rotating Line, or Elapsed Time

Giving your end users feedback that a script or program is not hung is vital on long processing jobs. We sometimes write shell scripts that take a long time to completely execute — for example, system-backup scripts and replicating data using rsync. A good way to keep everyone content is to have some kind of progress indicator. Just about anything can be a progress indicator as long as the end user gets the idea that job processing is continuing. In this chapter we are going to examine the following three progress indicators, which are fairly common:

- A series of dots
- A rotating line
- Elapsed time

We see the series-of-dots, rotating-line, and elapsed-time methods for user feedback in many installation programs, backup routines, and other long-running processes. Each of these methods can be started as a separate script, as a function, putting the code in a loop directly, or as a background process. We will cover the most common practices here.

Indicating Progress with a Series of Dots

The simplest form of progress indicator is to print a period to the screen every five seconds to five minutes, depending on the expected processing time. It is simple, clean, and very easy to do. As with every script, we start out with the command syntax. All we want to do is echo a dot to the screen while continuing on the same line.

For Korn shell, use

```
echo ".\c"
```

For Bourne and Bash shell, use

echo -e ".\c"

For the sake of simplicity, we are going to treat both options as **echo**. This echo command prints a single dot on the screen, and the backslash c (\c) specifies a continuation on the same line without a new line or carriage return. To make a series of dots, we will put this single command in a loop with some sleep time between each dot. For example, we will use a while loop that loops forever with a 10-second sleep between printing each dot on the screen:

```
while true
do
echo ".\c"
sleep 10
done
```

The placement of the sleep statement can be important; it just depends on your application. If, for instance, we want to allow time for a backup script to start executing before we monitor the process, we want to put the sleep statement at the top of the while loop. Then again, if we are trying to monitor a currently running process, we want to verify that the process is running first. In this case, we want the sleep statement at the end of the while loop. We could also put this while loop in the background and save the last background process ID (PID) This way, we can kill the progress indicator when the background, or we can create a function with this loop and run the function in the background. Both methods are shown in Listings 4-1 and 4-2, respectively.

```
while true
do
        sleep 60
        echo ".\c"
done &
BG_PID=$!
/usr/local/bin/my_backup.ksh
kill $BG_PID
echo
```

Listing 4-1 Looping in the background

To accomplish the background loop, notice that we just put an ampersand, &, after the end of the while loop, after done. The next line uses the \$! operator, which saves the PID of the last background process, BG_PID=\$!. The background loop sleeps at the beginning of the loop, allowing time for the backup script,

/usr/local/bin/my_backup.ksh, to start executing in the foreground before the dots start ticking. When the backup script is complete, we use the **kill** command to stop the dots by killing the background job, specified by kill \$BG_PID. Notice that we add one more echo at the end. This echo takes care of the new line/carriage return because we have been continuing on the same line with the dots.

We can accomplish the same task with a function, as shown in Listing 4-2.

```
function dots
{
SEC=$1 # How many seconds between dots?
while true
do
   sleep $SEC
   echo ".\c"
done
}
****
   Beginning of Main
****
dots 60 &
BG_PID=$!
/usr/local/bin/my_backup.ksh
kill $BG_PID
echo
```

Listing 4-2 Using a background function

The script and function in Listing 4-2 accomplish the same task but use a background function instead of just putting the while loop itself in the background. We still capture the PID of the last background process of the dots function, specified by \$!, so we can kill the function when the backup script has completed, as we did in the previous example. We could also put the loop in a separate shell script and run the external script in the background, but this would be overkill for three lines of code.

We are going to come back to the series-of-dots method again in this chapter.

Indicating Progress with a Rotating Line

If a series of dots is too boring, we could use a rotating line as a progress indicator. To rotate the line, we will again use the echo command, but this time we need a little more cursor control. This method requires that we display, in a series, the forward slash

(/), and then a hyphen (-), followed by a backslash (\), and then a pipe (|), and then repeat the process. For this character series to appear seamless, we need to backspace over the last character and erase it, or overwrite it with the new character that makes the line appear to rotate. We will use a case statement inside a while loop, as shown in Listing 4-3.

```
function rotate
{
# PURPOSE: This function is used to give the end user
# some feedback that "something" is running. It gives
# a line rotating in a circle. This function is started
# as a background process. Assign its PID to a variable
# using
#
       rotate &
                    # To start
#
       ROTATE_PID=$! # Get the PID of the last
                      # background job
#
#
#
      At the end of execution just break out by
#
       killing the $ROTATE_PID process. We also need
#
       to do a quick "cleanup" of the leftover line of
#
       rotate output.
#
#
      FROM THE SCRIPT:
#
             kill -9 $ROTATE_PID
#
             echo "\b\b "
INTERVAL=1 # Sleep time between rotation intervals
RCOUNT="0" # For each RCOUNT the line rotates 1/8
            # cycle
while :
               # Loop forever...until this function is killed
do
     (( RCOUNT = RCOUNT + 1 )) # Increment the RCOUNT
     case $RCOUNT in
           1) echo '-'"\b\c"
                sleep $INTERVAL
                ;;
           2) echo '\\'"\b\c"
                sleep $INTERVAL
                ;;
           3) echo "|\b\c"
                sleep $INTERVAL
                ;;
           4) echo "/\b\c"
                sleep $INTERVAL
                ;;
```

Listing 4-3 Rotate function

```
*) RCOUNT="0" # Reset the RCOUNT to "0", zero.
;;
esac
done
} # End of Function - rotate
```

Listing 4-3 (continued)

NOTE If you are executing this code in Bash shell, you will need to enable the backslash echo operators by adding the -e switch, as shown here:

```
echo -e "-""\b\c"
echo -e '\\'"\b\c"
```

In the function in Listing 4-3, we first define an interval to sleep between updates. If we do not have some sleep time, the load on the system will be noticeable just doing I/O to our tty terminal device doing the echo statement updates. We just want to give the end user some feedback, not load down the system. We will use a one-second interval between screen updates. Next we start an infinite while loop and use the RCOUNT variable to control which part of the rotating line is displayed during the interval. Notice that each time we echo a piece of the rotating line, we also back up the cursor with \b and continue on the same line with \c; both are needed. This way the next loop iteration will overwrite the previous character with a new character, and then we again back up the cursor and continue on the same line. This series of characters gives the appearance of a rotating line.

We use this function just like the previous example using the dots function in Listing 4-4. We start the function in the background, save the PID of the background function using the \$! operator, then start our time-consuming task. When complete, we kill the background rotate function. We could also just put the while loop in the background without using a function. In either case, when the rotating line is killed, we need to clean up the last characters on the screen. To do the cleanup we just back up the cursor and overwrite the last character with a blank space (see Listing 4-4).

Listing 4-4 Example of rotate function in a shell script

```
kill -9 $ROTATE_PID
# Cleanup...
echo "\b\b "
# End of Example
```

Listing 4-4 (continued)

These scripts work well and execute cleanly, but do not forget to give some sleep time on each loop iteration. Now we have shown the series-of-dots and rotating-line methods. Another method that is very useful is elapsed time.

Indicating Progress with Elapsed Time

For this method, we need to do a little math. This is not difficult math, so if math is not your top subject, don't worry. We will use the shell variable SECONDS to let the system keep up with the time. We just need to calculate the hours, minutes, and seconds, and then display the data in a readable manner.

In this chapter, we will use the ECHO variable where we need to use the echo command. We define the correct usage for the echo command to enable the backslash operators based on the executing shell. If the executing shell is Bash, we add the -e switch to the echo command.

The math we need is simple division and remainder calculations. To do math, we use our math operator \$((math statement)). To do simple division, in an echo statement, use the following syntax:

```
$ECHO "Elapsed time: $(( $SECONDS / 60 )) minutes"
```

This statement will divide the value of the SECONDS shell variable by 60 (calculating minutes) and display the result followed by the word "minutes." The problem here is that we can lose up to 59 seconds in the timing. To calculate the seconds left over, we use the remainder operator % just like we use the forward slash (/) for division. The next statement will calculate both the minutes and seconds:

```
$ECHO "Elapsed time: $(( $SECONDS / 60 )) minutes\
$(( $SECONDS % 60 )) seconds"
```

The backslash (\) continues the previous command statement on the next line. To calculate hours, we just take this procedure one more level. The elapsed_time function will show the entire elapsed time calculation process (see Listing 4-5).

```
elapsed_time ()
{
  SEC=$1
  (( SEC < 60 )) && $ECHO "[Elapsed time: \
  $SEC seconds]\c"
  (( SEC >= 60 && SEC < 3600 )) && $ECHO \
  "[Elapsed time: $(( SEC / 60 )) min $(( SEC % 60 )) \
  sec]\c"
  (( SEC > 3600 )) && $ECHO "[Elapsed time: \
  $(( SEC / 3600 )) hr $(( (SEC % 3600) / 60 )) min \
  $(( (SEC % 3600) % 60 )) sec]\c"
}
```

Listing 4-5 elapsed_time function

Notice in Listing 4-5 that we need to test the value of \$SEC, which is passed to the function as argument \$1. The three tests required include less than 60 seconds, between 60 and 3600 seconds (1 minute to 1 hour), and more than 3600 seconds (over 1 hour). These tests are to ensure we display the correct number of hours, minutes, and seconds. The specific tests are shown here:

```
(( SEC < 60 )) && $ECHO "[Elapsed
(( SEC >= 60 && SEC < 3600 )) && $ECHO "[Elapsed
(( SEC > 3600 )) && $ECHO "[Elapsed
```

The calculation for SECONDS greater than 3600 goes like this: To calculate hours, we divide the \$SECONDS value by 3600 (the number of seconds in one hour). Then we use the remainder calculation to get the minutes. Then we use the remainder of the minutes to calculate the seconds. The use of the elapsed_time function is shown in the timing_test_function.Bash script shown in Listing 4-6.

```
#!/bin/Bash
# SCRIPT: timing_test_function.Bash
# AUTHOR: Randy Michael
#
#
```

Listing 4-6 Shell script listing for timing_test_function.Bash

```
# Ready script for ksh or Bash
ECHO=echo
[[ $(basename $SHELL) = Bash ]] && ECHO="echo -e"
elapsed_time ()
{
SEC=$1
(( SEC < 60 )) && $ECHO "[Elapsed time: $SEC seconds]\c"
(( SEC >= 60 && SEC < 3600 )) && $ECHO "[Elapsed time:
$(( SEC / 60 )) min $(( SEC % 60 )) sec]\c"
(( SEC > 3600 )) && $ECHO "[Elapsed time: \
$(( SEC / 3600 )) hr $(( (SEC % 3600) / 60 )) min\ $((
(SEC % 3600) % 60 )) sec]\c"
}
###### BEGINNING OF MAIN
SECONDS=15
elapsed_time $SECONDS
$ECHO
SECONDS=60
elapsed_time $SECONDS
$ECHO
SECONDS=3844
elapsed_time $SECONDS
$ECHO
```

Listing 4-6 (continued)

This quick little script in Listing 4-6 shows how you can arbitrarily set the SECONDS shell variable to anything you want, and it keeps on ticking from that new value. The script is shown in action in Listing 4-7.

```
[Elapsed time: 15 seconds]
[Elapsed time: 1 min 0 sec]
[Elapsed time: 1 hr 4 min 4 sec]
```

Listing 4-7 Elapsed time shell script in action

Now let's put together some of the pieces.

Combining Feedback Methods

One of my favorite methods of user feedback for long-running process and log files is to combine the series-of-dots and elapsed-time methods. Now that we have the elapsed_time function, this task is pretty simple. The method in Listing 4-8 is from the rsync_daily_copy.ksh script in Chapter 7, "Using rsync to Efficiently Replicate Data." Let's look at some example code in Listing 4-8.

```
# While the rsync processes are executing this
# loop will place a . (dot) every 60 seconds
# as feedback to the end user that the background
# rsync copy processes are still executing. When
# the remaining rsync sessions are less than the
# total number of sessions (normally 36) this loop
# will give feedback as to the number of remaining
# rsync session, as well as the elapsed time of
# the procesing, every 5 minutes.
# Set the default echo command usage
ECHO="echo"
# Test the $SHELL for Bash and set Bash echo usage
[[ $(basename $SHELL) = Bash ]] && ECHO="echo -e"
SECONDS=10
MIN_COUNTER=0
PATTERN="oradata${DAY}_[0-1][0-9]"
RSYNC_CMD="rsync -avz"
TOTAL_SESSIONS=$(ps -ef | grep "$RSYNC_CMD" \
               grep "$PATTERN" \
                | grep -v grep \
               | awk '{print $2}' | wc -1)
# Loop until all rsync processing completes.
until (( REM_SESSIONS == 0 ))
do
   sleep 60
   $ECHO ".\c" # Type a dot every minute
```

Listing 4-8 Combining feedback methods

```
# Query the system for remaining rsync sessions
    REM_SESSIONS=$(ps -ef | grep "$RSYNC_CMD" \
                   | grep "$PATTERN" \
                   | grep -v grep \
                   | awk '{print $2}' | wc -1)
   # If the remaining sessions is less than the
   # $TOTAL sessions then give the elapsed time
   # every 5 minutes with an update of the
   # remaining sessions.
   if (( REM SESSIONS < TOTAL SESSIONS ))
   then
      # Increment the minute counter
      (( MIN_COUNTER = MIN_COUNTER + 1 ))
      if (( MIN_COUNTER >= 5 ))
      then
         # Hit the 5 minute mark!
         MIN_COUNTER=0
         $ECHO "\n$REM_SESSIONS of $TOTAL_SESSIONS rsync
sessions remaining $(elapsed_time $SECONDS)\c"
       if (( REM_SESSIONS <= $(( TOTAL / 4 )) ))
          then
             $ECHO "\nRemaining rsync sessions include:\n"
             ps -ef | grep "rsync -avz"
                                          \
                    | grep "$PATTERN" \
                    | grep -v grep
              $ECHO
           fi
        fi
    fi
done
```

Listing 4-8 (continued)

This is a code snippet of the feedback process. What we do not see in this code from Chapter 7 are all of the rsync sessions that were started in the background. Additionally, the variable assignments are added for clarification. Listing 4-9 shows the rsync copy script in action.

```
12 of 36 rsync sessions remaining [Elapsed time: 1 hr 4 min 4 sec].....
10 of 36 rsync sessions remaining [Elapsed time: 1 hr 9 min 14 sec].....
10 of 36 rsync sessions remaining [Elapsed time: 1 hr 14 min 4 sec].....
4 of 36 rsync sessions remaining [Elapsed time: 1 hr 19 min 4 sec].....
```

Listing 4-9 rsync shell script output showing end-user feedback

Remaining rsync sessions include the following:

root 22899 2786 0 Jul24 tty4 00:2:18 rsync -avz /oradata2_21 root 28872 2786 0 Jul24 tty4 00:2:18 rsync -avz /oradata2_03 root 22765 2786 0 Jul24 tty4 00:2:18 rsync -avz /oradata2_02 root 22733 2786 0 Jul24 tty4 00:2:18 rsync -avz /oradata2_10

As you can see from the output in Listing 4-9, when the remaining sessions drops below the total sessions, we give the user an update on the remaining sessions and the elapsed time. Then we print a dot every minute until we count to five minutes, when we update the elapsed time and the number of remaining rsync sessions still executing. When the remaining sessions reaches one-fourth the \$TOTAL_SESSIONS, we display the remaining processes to the end user. This is very useful for long-running processes. End users always wonder "what's taking so long?!" Well, let them know.

Other Options to Consider

As with any script, we may be able to improve on the techniques. The series-of-dots method is so simple that I cannot think of any real improvements. The rotating line is a fun little script to play with, and I have accomplished the same result in several different ways. Each method I used produced a noticeable load on the system if the sleep 1 statement was removed, so that the line rotated as fast as possible. Try to see if you can find a technique that will not produce a noticeable load and does not require a one-second sleep, and using a shell script!

For the elapsed_time function we may want to add the ability to expand to days. This task is not very difficult if we expand the three seconds, minutes, and hours calculations with a new \$SEC test for seconds greater than 86,400 (seconds in 24 hours) and add a days calculation.

Play around with each of these techniques, and always strive to keep your end users informed. A blank or "frozen" screen makes people uncomfortable.

Summary

In this chapter we presented three techniques to help keep our script users content. Each technique has its place, and they are all easy to implement within any shell script or function. We covered how to save the PID of the last background job and how to put an entire loop in the background. The background looping can make a script a little easier to follow if you are not yet proficient at creating and using functions. The elapsed time combined with a series of dots and a process listing is really going the extra mile.

Remember, informed users are happy users!

In the next chapter, we are going to work with record files. I have had many requests for this chapter from my application-support staff. I hope you stick around for some interesting techniques.

Lab Assignments

I really hope you paid attention to this chapter, because you will encounter many of these techniques throughout this book. I have a few script tasks that you may be interested in:

- Write a shell script to start six sleep sessions, with each session 5 seconds longer than the previous session. The first sleep interval is 10 seconds. When the sleep sessions are less than or equal to half the total, display the remaining sleep processes, update the elapsed time, and continue printing dots every second. Repeat the process until all the sleep sessions have completed execution.
- 2. White a shell script to count down from two hours to zero seconds using the hour, minute, second display format utilizing the shell's SECOND variable.

Extra credit: Repeat task 2, but overwrite the same line of text with the updated values, as opposed to writing a new line for every second.

3. Expand the elapsed_time function to calculate days, hours, minutes, and seconds.

PART

Scripts for Programmers, Testers, and Analysts

Chapter 5: Working with Record Files
Chapter 6: Automated FTP Stuff
Chapter 7: Using rsync to Efficiently Replicate Data
Chapter 8: Automating Interactive Programs with Expect and Autoexpect
Chapter 9: Finding Large Files and Files of a Specific Type
Chapter 10: Process Monitoring and Enabling Pre-Processing, Startup, and Post-Processing Events
Chapter 11: Pseudo-Random Number and Data Generation
Chapter 12: Creating Pseudo-Random Passwords
Chapter 13: Floating-Point Math and the bc Utility
Chapter 14: Number Base Conversions
Chapter 15: hgrep: Highlighted grep Script
Chapter 16: Monitoring Processes and Applications

CHAPTER 5

Working with Record Files

Every large business that processes its own data works with record files. Record files contain lines of data that are either field delimited, called a *variable-length record file*, or each line of data is the same length and each data field is defined by string length and position, called a *fixed-length record file*. We will look at examples of each record file format type and methods of processing the files quickly and manipulating the data. I added this chapter to this second edition at the request of a co-worker at Coca-Cola Enterprises, Mark Sellers. Mark asked me one day for a quick way to add the name of the record file to the end of every record, or line, in the file. Mark's problem was that when he merged a bunch of record files together in a single file for batch processing, and the batch file had an error with a particular record, he had no idea which record file the record came from. We will look at methods to solve this problem and many others.

What Is a Record File?

A *record file* is an alphanumeric character flat file (text file) that contains one or more lines of data. Each separate line of data is called a *record*. A record is made up of different *fields* that contain the actual data. There are two types of record files:

- **Fixed-length record files** are files with fixed column width with each data field residing within a specific character length and position within the record.
- Variable-length record files are files where the individual records may vary in length with the fields delimited (separated) by a character, such as a : (colon), a , (comma), or a | (pipe).

Let's look at an example of both types of record files.

Fixed-Length Record Files

The best way to understand a fixed-length record file is to look at an example. An example record might consist of a Company Branch, Account Number, Name, Total, and Due Date. For this fixed-length record file example we will assume that each record, or line of data, consists of the fields shown in Table 5-1.

CHARACTERS	DATA FIELD
1-6	Branch
7–25	Account Number
26-45	Name
46-70	Total
71–78	Due Date

Table 5-1 Fixed-Length Record File Definition

Using the data in Table 5-1 as a guide, a sample fixed-length record might look like the following:

```
US72271234567890123456789Randal.K.Michael.\
###00000000000000000000982512312007
```

Now, this looks like a bunch of random data, but it does specify very specific data, as shown in Table 5-2.

This data breaks down to Randal.K.Michael with account number 1234567890123456 789 owes the US7227 company branch \$98.25 due on 12/31/2007. Notice that several of the data fields are padded with fill data because the data-string length is shorter than the field-string length. The Name field is left-justified with hash marks (#) padding the extra field data and a period (.) delimiting the different parts of the Name. So, fixed-length record files may need additional parsing for some data fields. Now I just made this example up out of thin air, but I hope you get the idea of a fixed-length record file. Let's move on and look at variable-length record files.

Table 5-2 Fixed-Length Record File Data Fields

FIELD	DATA
Branch	US7227
Account Number	1234567890123456789
Name	Randal.K.Michael.###
Due Date	12312007

Variable-Length Record Files

In a variable-length record file, the individual records, or lines of data, vary in length and are separated by a field delimiter. For our example, we are going to assume that the : (colon) character is the field delimiter. Again, the best way to learn about variable-length record files is to look at an example.

Instead of worrying about character positions within each record, we only need to know which field the data resides in, and which character we are using as a data field delimiter. This is what we have been doing with awk in our scripts when we specify awk '{print \$2}' to select the second field of input/output. An example variable-length record file definition is shown in Table 5-3.

1 Branch 2 Account Number
2 Account Number
3 Name
4 Total
5 Due Date

 Table 5-3
 Variable-Length Record File Definition

Using the data in Table 5-3 as a guide, a sample variable-length record might look like the following:

US7227:1234567890123456789:Randal.K.Michael:9825:12312007

As you can see this record is not as long as the fixed-length record. Take a close look at this record and you can see the :, colon, field delimiters that separate each data field. Now let's look at the specific data as shown in Table 5-4.

FIELD	DATA
Branch	US7227
Account Number	1234567890123456789
Name	Randal.K.Michael
Due Date	12312007

Table 5-4 Variable-Length Record File Data Fields

Again, this data breaks down to Randal.K.Michael with account number 123456789 0123456789 owes the US7227 company branch \$98.25 due on 12/31/2007. Notice that the data fields are not padded with extra characters. Depending on your data, a

variable-length record file is easier to work with, and in most cases faster to process. This data is the same data representation as the fixed-length record file example.

Processing the Record Files

The most important aspect of processing a record file of any size is to select a method of parsing the file line-by-line. Recall from Chapter 2, "24 Ways to Process a File Line-by-Line," that not all file-processing methods are created equal. Depending on the file-processing method, it could take 0.5 seconds or 10 minutes to process the same file. The fastest methods to process a file line by line are shown in Listings 5-1, 5-2, 5-3, and 5-4, and are the methods discussed in detail in Chapter 2.

```
function while_read_LINE_bottom_FD_OUT
{
# Method 12
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while read LINE
do
       echo "$LINE"
        :
done < $INFILE</pre>
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
```

Listing 5-1 Method 12 tied for first place

The method in Listing 5-1 tied for first place in our Chapter 2 timing tests. This method uses my favorite input redirection for files by redirecting input at the bottom of the loop, after the done loop terminator. This method does use a file descriptor for redirecting standard output, stdout, to file descriptor 4. If you are not up to speed on using file descriptors, please go back and review Chapter 2 for more details. The input redirection using done < \$INFILE greatly speeds up any loop that requires file

input. The nice thing about this method of input redirection is that it's intuitive for beginners to shell scripting. I was actually surprised that this method tied, actually beat by 10 mS, using file descriptors for both input and output files, as shown in Listing 5-2.

```
function while_read_LINE_FD_IN_AND_OUT
{
# Method 22
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard input with file descriptor 3
# and redirect standard input to $INFILE
exec 3<&0
exec 0< $INFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while read LINE
do
      echo "$LINE"
       :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
# Restore standard input and close file
# descriptor 3
exec 0<&3
exec 3>&-
}
```

Listing 5-2 Method 22 tied for first place

I tend not to use this method when I write shell scripts because it can be difficult to maintain through the code life cycle. If a user is not familiar with using file descriptors, a script using this method is extremely hard to understand. The method

in Listing 5-1 produces the same timing results, and it is much easier to understand. Listing 5-3 shows the second-place loop method.

```
for_LINE_cat_FILE_cmdsub2_FD_OUT ()
{
# Method 16
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
for LINE in $(cat $INFILE)
do
   echo "$LINE"
   :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
```

Listing 5-3 Method 16 made second place in timing tests

The method shown in Listing 5-3 is another surprise — a for loop using command substitution with file descriptor for output redirection. Listing 5-4 shows the third place method in our timing tests.

```
for_LINE_cat_FILE_FD_OUT ()
{
    # Method 15

# Zero out the $OUTFILE
>$OUTFILE
```

Listing 5-4 Method 15 made third place in timing tests

```
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
for LINE in `cat $INFILE`
do
    echo "$LINE"
    :
done
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
```

The only difference between Methods 15 and 16 is the technique used for command substitution, the back tics method `command` versus the dollar parentheses method \$ (command). Listing 5-5 shows this timing data.

```
Method 15:

function for_LINE_cat_FILE_FD_OUT

real 0m0.47s

user 0m0.38s

sys 0m0.06s

Method 16:

function for_LINE_cat_FILE_cmdsub2_FD_OUT

real 0m0.45s

user 0m0.38s

sys 0m0.05s
```

Listing 5-5 Timing difference between Methods 15 and 16

The timing results indicate a single mS in system time, or system/kernel time between the command substitution methods, and a total of 2 mS in real execution time. Now let's look at the fastest method to process a file *without* using file descriptors, as shown in Listing 5-6.

```
function while_read_LINE_bottom
{
    # Method 2

    # Zero out the $OUTFILE
    >$OUTFILE
    while read LINE
    do
        echo "$LINE" >> $OUTFILE
        :
    done < $INFILE
    }
</pre>
```

Listing 5-6 The fastest method not using file descriptors

The method in Listing 5-6 is the fastest method to process a file line by line that does *not* use file descriptors. I use this technique in almost every shell script that does file parsing simply because of the ease of maintaining the shell script throughout the life cycle. Remember, it is not *if* someone will come behind you and need to edit the script some time in the future, but *when* someone comes behind you to edit your *most excellent script*, they must be able to understand what you did without having to either hack through your code, or just give up and resort to rewriting the entire script. This is one important reason to put plenty of comments in your shell scripts!

If your goal is absolute speed, use the technique in Listing 5-1. Now let's take a look at some of the things we want to do with record files and individual records.

Tasks for Records and Record Files

Now that we have an idea of the ways to process the record file line by line, we need to look at tasks we can perform on individual records. The most obvious task is to extract each data field from each record and perform some action on that data. The method we use to extract the data depends on the type of record file we are working with. For fixed-length record files, we need to know the placement of each data field within the record. Because we need to select specific characters in a string, the **cut** command is the best option. For variable-length record fields, we need to know the field delimiter and which data field represents each value. For this type of record file, we can use **awk** or cut to select the correct data field.

Tasks on Fixed-Length Record Files

For our examples of parsing record files, we are going to use the method shown in Listing 5-1. The only difference between fixed-length and variable-length records is how we extract the data fields from each record as we process the file line by line. For

the fixed-length record files, we are going to use the cut command to select specific groups of characters residing in specific positions and assign these data to variables on each loop iteration.

For our example, we need to refer back to Table 5-1 to determine where each data field resides within the record. Our example specifies characters 1–6 as the Branch, characters 7–25 as the Account Number, characters 26–45 as the Name, characters 46–70 as the Total, and characters 71–78 as the Due Date field. The data fields are shown again in Table 5-5.

CHARACTERS	DATA FIELD
1-6	Branch
7–25	Account Number
26-45	Name
46-70	Total
71-78	Due Date

With these definitions from Table 5-5, we know how to extract each data field. Check out the code in Listing 5-7, and we will cover the details at the end.

```
function parse_fixed_length_records
{
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while read RECORD
do
    # On each loop iteration extract the data fields
    # from the record as we process the record file
    # line by line
    BRANCH=$(echo "$RECORD" | cut -c1-6)
    ACCOUNT=$(echo "$RECORD" | cut -c7-25)
    NAME=$(echo "$RECORD" | cut -c26-45)
    TOTAL=$(echo "$RECORD" | cut -c46-70)
    DUEDATE=$(echo "$RECORD" | cut -c71-78)
```

Listing 5-7 Parsing a fixed-length record file

This is the fastest method to process a file line by line. Notice that we feed the file data into the loop after the done loop terminator, done < \$INFILE, and the output is redirected to file descriptor 4.

On each loop iteration, we assign the characters residing in positions 1–6 to the BRANCH variable, positions 7–25 are assigned to the ACCOUNT variable, positions 26–45 are assigned to the NAME variable, the TOTAL variable is assigned the characters residing in positions 46–70, and characters 71–78 are assigned to the DUEDATE variable. After the variable assignments, we execute some process_data function (to be discussed later), passing each set of variable assignments on each loop iteration. This is just an example of how you can extract the data fields from a fixed-length record file and perform some type of processing on that data. Now let's look at variable-length record files.

NOTE This method works well with record files that contain fields with spaces and comments as long as the variables are double quoted when used.

Example: echo "\$NAME"

Tasks on Variable-Length Record Files

The main difference between processing variable-length and fixed-length record files is how we extract the data field from each record in the record file. For fixed-length record files, we had to know which character positions in a record represented each data field. In the variable-length record files, we need to know how many fields of data we have and what data it represents and what the field delimiter character is that separates each data field in the record.

For our example, we have five data fields that are field separated by a : (colon), as follows:

US7227:1234567890123456789:Randal.K.Michael:9825:12312007

To process this record, we can use awk or cut to select each of the fields and assign them to variables. I tend to use awk for this process. Check out Listing 5-8, and we will cover the details at the end.

```
function parse_variable_length_records
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while read RECORD
do
   # On each loop iteration extract the data fields
   # from the record as we process the record file
   # line by line
   echo $RECORD | awk -F : '{print $1, $2, $3, $4, $5}' \
                | while read BRANCH ACCOUNT NAME TOTAL DATEDUE
   đo
       # Perform some action on the data
       process_data $BRANCH $ACCOUNT $NAME $TOTAL $DATEDUE
       if (( $? != 0 ))
       then
             # Note that $LOGFILE is a global variable
             echo "Record Error: $RECORD" | tee -a $LOGFILE
       fi
   done
done < $INFILE
# Restore standard output and close file
```

Listing 5-8 Parsing a variable-length record file

```
# descriptor 4
exec 1<&4
exec 4>&-
}
```

This is the fastest method to process a file line by line. Notice that we feed the file data into the loop after done, done < \$INFILE, and the output is redirected to file descriptor 4.

On each loop iteration, we echo each \$RECORD and pipe this line of data to an awk statement where we define the field delimiter as the : (colon) character, specified by awk -F :. With the field delimiter defined, we just select the first five fields and pipe these values to a while read loop where the variable assignments are made.

NOTE The Solaris version of awk does not support defining a field delimiter using the -F : switch option. For Solaris we must use nawk (new-awk). Adding the following alias statement in the variable definition section of the shell script will correct this problem:

```
case $(uname) in
SunOS) alias awk=nawk
;;
esac
```

The full statement is shown here:

echo \$RECORD | awk -F : '{print \$1, \$2, \$3, \$4, \$5}' \ | while read BRANCH ACCOUNT NAME TOTAL DATEDUE

Notice that we continued the command statement on a second line by ending the first line with a backslash, $\$. The first line is the awk statement, and the second line contains the variable assignments.

We can accomplish the same task using the cut command, but it requires multiple echo statements, as shown here:

BRANCH=\$(echo \$RECORD | cut -d : -f1) ACCOUNT=\$(echo \$RECORD | cut -d : -f2) NAME=\$(echo \$RECORD | cut -d : -f3) TOTAL=\$(echo \$RECORD | cut -d : -f4) DATEDUE=\$(echo \$RECORD | cut -d : -f5)

The cut command is not a direct replacement for awk and requires multiple system calls compared to awk. We can extract more than one field in a variable-length record

file using the cut command, but cut will also display the field delimiter character if more than one field is extracted. If the field delimiter is a blank space this is not a problem, but the :, colon, would require more parsing. See Lab Assignment 3 at the end of this chapter for more details.

After the variable assignments, we execute some process_data function (to be discussed later) from within the while loop, passing each variable assignment on each loop iteration. This is just an example of how you can extract the data fields from a variable-length record file and perform some type of processing on that data.

NOTE This method of a loop assigning all of the fields' data to variables at the same time on each loop iteration *does not work* if any of the fields being extracted contain white space. For record files that contain data fields with spaces or comments, the individual field(s) containing white space must be extracted as a single variable assignment one at a time. When using variables with white space the variables must be double quoted.

Example: echo "\$NAME"

The Merge Process

If you work with data files for very long, you realize that troubleshooting *batch-processing* errors can be a daunting task. For example, let's say that we have record files coming in every day from each Branch of the company. When we get the record files, we have a process where we append all the record files from all the various Branches into a single file. Then we process the big file every day to do all of the processing in one big batch job. This process of appending the record files into a single file is called a *merge process*. After the merge process takes place, we have all the branch records in a single file. At first look, the only way to tell where an individual record came from is to reference the Branch field. However, this can be time consuming when you are trying to troubleshoot a batch-processing problem. One option is to append the name of the record file to the end of every record in the file. Then after the merge process you can look at the merged record file and see the filename associated with each record in the batch-processing file. This helps to ease the troubleshooting headache to find where the error came from.

So, how do we add the record filename to the end of a record? When my co-worker Mark Sellers came to me with this problem, he had a very slow process where the files were processed line by line, and then an echo statement listed each line of record data followed by the filename and appended this data to a "new" record, and then to the end of a new record file. This process took many hours to complete. I looked at this code and thought **sed** should be able to do this with character substitution, and at the file level as opposed to the record level. I played around with sed for a half hour or so and came up with the following sed statement to add a filename to the end of each record in a file:

```
sed s/$/$(basename $RecordFileName)/g $RecordFileName > ${RecordFileName}
.preprocess
```

This is a simple character substitution sed statement with a twist. The "substitution" is to replace the End-of-Line (EOL) with the record filename. The \$ (dollar sign) is the regular expression defining EOL. (The ^, caret, is the regular expression defining the Beginning-of-Line.) An important point in this sed statement is where we use \$ (basename \$RecordFileName) in the substitution. If we had omitted the basename part and the filename has a forward slash (/), the sed statement would produce an error. Notice that we redirected output to a new filename specified by the .preprocess filename suffix, but you may just want to do this step during the merge process as you build the batch-processing file. Just remember that substituting EOL with a filename results in the filename being appended to the end of each record in the record file, as shown here.

Fixed-length record:

US72271234567890123456789Randal.K.Michael.###0000000000000000000825 \ 12312007record.dat

Variable-length record:

US7227:1234567890123456789:Randal.K.Michael:9825:12312007:record.dat

In this example, our record filename is record.dat, and this name is appended to the end of our fixed-length and variable-length records. Listing 5-9 shows a merge process where we loop through a list of record filenames and, as we append each record file to build the big batch-processing file, we append the record filename to each record. Check out Listing 5-9, and we will cover the details at the end.

```
MERGERECORDFILE=/data/mergerecord.$(date +%m%d%y)
RECORDFILELIST=/data/branch_records.lst
while read RECORDFILENAME junk
do
    sed s/$/$(basename $RECORDFILENAME)/g $RECORDFILENAME >>\
$MERGERECORDFILE
done < $RECORDFILELIST</pre>
```

Listing 5-9 Code for a merge process for fixed-length record files

Listing 5-9 is a merge script for fixed-length record files. We first define a couple of files. The MERGERECORDFILE variable definition specifies the name of the resulting merged record data file. The RECORDFILELIST variable defines the file that contains

a list of record files that must be merged. Notice the use of the basename command to strip the directory part of the \$RECORDFILENAME on each loop iteration. As we feed the while loop from the bottom, after done, to process the \$RECORDFILELIST file line by line, we assign a new record file to the RECORDFILENAME variable. We use this new value in our sed statement that appends the record filename to the end of each record in the file. As the last step, this record file is appended to the end of the \$MERGERECORDFILE file.

Listing 5-10 is a merge script for variable-length record files. The only difference between this script and Listing 5-9 is that we added the field-delimiter variable, FD. We define the field delimiter and then add the field delimiter between the end of the record and the record filename. These modifications are highlighted in bold in Listing 5-10. As with Listing 5-9, we first define a couple of files. The MERGERECORDFILE variable definition specifies the name of the resulting merged record data file. The RECORDFILELIST variable defines the file that contains a list of record files that must be merged. As we feed the while loop from the bottom, after done, we execute a sed statement that substitutes EOL with the name of the record file, \$RECORDFILENAME. Again, notice the use of the basename command to strip out the directory part of the filename, if it exists. This new merged record file, with the name of each source record file appended to the end of each record, is appended to the \$MERGERECORDFILE.

```
MERGERECORDFILE=/data/mergerecord.$(date +%m%d%y)
RECORDFILELIST=/data/branch_records.lst
FD=:
while read RECORDFILENAME
do
sed s/$/${FD}$(basename $RECORDFILENAME)/g $RECORDFILENAME >>\
$MERGERECORDFILE
done < $RECORDFILELIST</pre>
```

Listing 5-10 Code for a merge process for variable-length record files

Working with Strings

A useful thing to know when working with fixed-length record files is the string length. The length of a string assigned to a variable can be found using the following syntax:

```
echo ${#VAR}
```

For example, if we assign the VAR variable the string 1234567890, the string length should be 10:

```
VAR=1234567890
echo ${#VAR}
10
```

Sometimes we need the string length for fixed-length records to pad the extra space in a data field that the data does not fill up. As an example, the TOTAL field is defined as characters 46–70, which is 25 characters long. It is unlikely that anyone would ever owe that kind of cash, but that is the data field we have to work with. This field is right-justified and filled with leading zeros as shown here:

```
000000000000000000009825
```

When we change a record field and the data is not as long as the data field, we need to pad the rest of the data field with something — in this case, zeros. The hard way to pad the data with leading zeros is to check the original data for the string length, then get the string length of the new data and start looping to add the correct number of leading zeros followed by the new data. It is much easier to do this task with the **typeset** command. The -Z switch specifies right justification with leading zeros, which is just what we need here. Our TOTAL variable is 25 characters long. The following typeset command makes this definition for us:

typeset -Z25 TOTAL

Now we can change the TOTAL data field to a different value and not worry about how many leading zeros to add. For example, let's change the total to 0. (We paid off this bill!)

The typeset command can do a lot of the work for us. Table 5-6 lists additional typeset options you might find handy.

NOTE Using + instead of – turns off the typeset definition.

Table 5-6	Options for th	e typeset Command
-----------	----------------	-------------------

SWITCH	RESULTING TYPESET
-L	Left justify and remove blank spaces
-R	Right justify and to fill with leading blanks
-Zn	Right justify to <i>n</i> length and fill with leading zeros if the first non-blank character is a digit
-i	Variable is an integer
-1	Convert uppercase characters to lowercase characters
-u	Convert lowercase characters to uppercase characters
-x	Automatically export this variable

Putting It All Together

Now that we have covered all the pieces in detail, it is time put them together into one shell script. Our task is to merge and process records based on record-format type. For this selection, we are expecting a single command-line argument, -f for fixed-length record files or -v for variable-length record files.

We have two data record files for each type of record format shown here:

branch01_fixed.dat branch02_fixed.dat branch01_variable.dat branch02_variable.dat

The record files are shown in Listings 5-11, 5-12, 5-13, and 5-14, respectively.

```
US72271234567890123456789Randal.K.Michael.###00000000000000000009825
12312007
US72271239922890123456789James.M.Jones.######00000000000000000001426
12312007
US72271528857890123456789William.C.Rogers.###000000000000000000427732 \
12312007
US72271222945530123456789Robin.M.Kim.#########00000000000000002263320
12312007
US72271234066820123456789Ana.Marie.Larson.###00000000000000000004741
12312007
US72271247783490123456789Stacy.M.Spry.#######00000000000000000008847
12312007
US72271998233490123456789Marcus.S.Moore.#####000000000000000000095248
12312007
US72271235583290123456789Paul.W.Bryant.######0000000000000000000886799 \
12312007
```

Listing 5-11 File listing for branch01_fixed.dat

```
US82281299277300088258239Andy.C.James.#######00000000000000000882991 \
12312007
US82281200299938882456789Shakar.C.Divini.####000000000000000002211426 \
12312007
US82282990388872334456789Joe.M.Baker.########00000000000000000088234 \
12312007
US82281222945530092276611George.Williams.###00000000000000000881091 \
2312007
US82281234066820199822389Larry.C.Guy.#########000000000000000000000882 \
12312007
```

Listing 5-12 File listing for branch02_fixed.dat

```
US82281247783490123436021Lacy.C.Cocaran.#####00000000000000009921556 \
12312007
US82281998233490123426401Herman.G.Munster.###000000000000000000055324 \
12312007
US82281235583290123402173LaQuita.H.Dabney.###0000000000000000882371 \
12312007
```

```
US7227:1234567890123456789:Randal.K.Michael:9825:12312007
US7227:1239922890123456789:James.M.Jones:1426:12312007
US7227:1528857890123456789:William.C.Rogers:427732:12312007
US7227:1222945530123456789:Robin.M.Kim:2263320:12312007
US7227:1234066820123456789:Ana.Marie.Larson:4741:12312007
US7227:1247783490123456789:Stacy.M.Spry:8847:12312007
US7227:1998233490123456789:Marcus.S.Moore:95248:12312007
US7227:1235583290123456789:Paul.W.Bryant:886799:12312007
```

Listing 5-13 File listing for branch01_variable.dat

```
US8228:1299277300088258239:Andy.C.James:882991:12312007
US8228:1200299938882456789:Shakar.C.Divini:2211426:12312007
US8228:2990388872334456789:Joe.M.Baker:88234:12312007
US8228:1222945530092276611:George.Williams:88109:12312007
US8228:1234066820199822389:Larry.C.Guy:882:12312007
US8228:1247783490123436021:Lacy.C.Cocaran:9921556:12312007
US8228:1998233490123426401:Herman.G.Munster:55324:12312007
US8228:1235583290123402173:LaQuita.H.Dabney:882371:12312007
```

Listing 5-14 File listing for branch02_variable.dat

An important thing to notice here is that Listings 5-11 and 5-13 represent the same data and Listings 5-12 and 5-14 represent the same data. The only difference is the record format we represent the data in.

In our shell script we are first going to merge the record files together, based on record-file format, and then process the merged data file to produce a post-processing output file. Our task is to change the Due Date from 12/31/2007 to 1/31/2008. Check out the script in Listing 5-15, and we will cover it in more detail at the end.

NOTE For the parse_records_files.Bash script in Listing 5-15 to work, as written, you must create a directory and some files. Specifically, you need to create the directory /data.Next, depending on your record-file data type, you need to create the /data/branch_records_variable.lst or /data/branch_records_fixed.lst file that contains a list of record files to process. Example record files to process are shown in Listings 5-11,

5-12, 5-13, and 5-14. Otherwise, edit the shell script to reflect your particular environment. Each of these files is on the web site accompanying this book.

```
#!/bin/Bash
#
# SCRIPT: parse_record_files.Bash
# AUTHOR: Randy Michael
# DATE: 12/7/2007
# REV: 1.0
# PURPOSE: This script is used to parse both
# fixed-length and variable-length record files.
# Before we parse the records we first merge the
# files into a single file for batch processing.
#
# set -n # Uncomment to check script syntax
      # without any execution
#
# set -x # Uncomment to debug
#
# REV LIST:
#
# Revised by:
# Revision date:
# Revision:
#
#
#
*****
# VERIFY INPUT
*****
if (( $# != 1 ))
then
   echo -e "\nUSAGE: $(basename $0) -f|-v"
   echo -e "\nWhere -f = fixed-length records"
   echo -e "and -v = variable-length records\n"
   exit 1
else
   case $1 in
   -f) RECORD_TYPE=fixed
       ;;
   -v) RECORD_TYPE=variable
       ;;
    *) echo -e "\nUSAGE: $(basename $0) -f|-v"
       echo -e "\nWhere -f = fixed-length records"
       echo -e "and -v = variable-length records \n"
```

Listing 5-15 Shell script listing for parse_record_files.Bash

```
exit 1
       ;;
   esac
fi
*****
# DEFINE FILES AND VARIABLES HERE
*****
DATADIR=/data # This variable defines the directory to use for data
if [ $RECORD_TYPE = fixed ]
then
  MERGERECORDFILE=${DATADIR}/mergedrecords_fixed.$(date +%m%d%y)
  >$MERGERECORDFILE # Zero out the file to start
   RECORDFILELIST=${DATADIR}/branch_records_fixed.lst
  OUTFILE=${DATADIR}/post_processing_fixed_records.dat
   >$OUTFILE # Zero out the file to start
else
  MERGERECORDFILE=${DATADIR}/mergedrecords_variable.$(date +%m%d%y)
  >$MERGERECORDFILE # Zero out the file to start
  RECORDFILELIST=${DATADIR}/branch_records_variable.lst
  OUTFILE=${DATADIR}/post_processing_variable_records.dat
   >$OUTFILE # Zero out the file to start
fi
# Test for Solaris to alias awk to nawk
case $(uname) in
SunOS) alias awk=nawk
      ;;
esac
FD=: # This variable defines the field delimiter for fixed-length
records
NEW_DATEDUE=01312008
*****
function process_fixedlength_data_new_duedate
{
# set -x
# Local positional variables
branch=$1
account=$2
name=$3
total=$4
```

```
datedue=$5
recfile=$6
new_datedue=$7
echo "${branch}${account}${name}${total}${new_datedue}${recfile}" \
    >> $OUTFILE
}
****
function process_variablelength_data_new_duedate
{
# set -x
# Local positional variables
branch=$1
account=$2
name=$3
total=$4
datedue=$5
recfile=$6
new_datedue=$7
echo "${branch}${FD}${account}${FD}${name}${FD}${total}\
${FD}${new_datedue}${FD}${recfile}" >> $OUTFILE
}
*****
function merge_fixed_length_records
{
# set -x
while read RECORDFILENAME
do
   sed s/$/$(basename $RECORDFILENAME 2>/dev/null)/g $RECORDFILENAME\
>> $MERGERECORDFILE
done < $RECORDFILELIST
}
function merge_variable_length_records
{
# set -x
while read RECORDFILENAME
do
```

```
sed s/$/${FD}$(basename $RECORDFILENAME 2>/dev/null)/g $RECORDFILENAME\
>> $MERGERECORDFILE
done < $RECORDFILELIST
}
****
function parse_fixed_length_records
{
# set -x
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while read RECORD
do
    # On each loop iteration extract the data fields
    # from the record as we process the record file
    # line by line
    BRANCH=$(echo "$RECORD" | cut -c1-6)
    ACCOUNT=$(echo "$RECORD" | cut -c7-25)
    NAME=$(echo "$RECORD" | cut -c26-45)
    TOTAL=$(echo "$RECORD" | cut -c46-70)
    DUEDATE=$(echo "$RECORD" | cut -c71-78)
    RECFILE=$(echo "$RECORD" | cut -c79-)
    # Perform some action on the data
    process_fixedlength_data_new_duedate $BRANCH $ACCOUNT $NAME \
          $TOTAL $DUEDATE $RECFILE $NEW DATEDUE
    if (($? != 0))
      then
          # Note that $LOGFILE is a global variable
          echo "Record Error: $RECORD" | tee -a $LOGFILE
    fi
done < $MERGERECORDFILE</pre>
# Restore standard output and close file
# descriptor 4
```

```
exec 1<&4
exec 4>&-
}
*****
function parse_variable_length_records
{
# set -x
# Zero out the $OUTFILE
>$OUTFILE
# Associate standard output with file descriptor 4
# and redirect standard output to $OUTFILE
exec 4<&1
exec 1> $OUTFILE
while read RECORD
do
   # On each loop iteration extract the data fields
   # from the record as we process the record file
   # line by line
   echo $RECORD | awk -F : '{print $1, $2, $3, $4, $5, $6}' \
                | while read BRANCH ACCOUNT NAME TOTAL DATEDUE RECFILE
  do
       # Perform some action on the data
      process_variablelength_data_new_duedate $BRANCH $ACCOUNT $NAME \
              $TOTAL $DATEDUE $RECFILE $NEW_DATEDUE
      if (( $? != 0 ))
      then
           # Note that $LOGFILE is a global variable
          echo "Record Error: $RECORD" | tee -a $LOGFILE
      fi
   done
done < $MERGERECORDFILE</pre>
# Restore standard output and close file
# descriptor 4
exec 1<&4
exec 4>&-
}
```

The first test is to ensure that we have exactly one command-line argument presented to the shell script. In the second test, we use a case statement to ensure that the argument presented is either -f or -v; any other value is a usage error. If we had more command-line options for this shell script, we could use getopts to parse the arguments.

Once the input is verified, we define some files and variables. Notice that we use the data we gathered in the last test to set up the file definitions, as shown here:

```
if [ $RECORD_TYPE = fixed ]
then

MERGERECORDFILE=${DATADIR}/mergedrecords_fixed.$(date +%m%d%y)
>$MERGERECORDFILE # Zero out the file to start
RECORDFILELIST=${DATADIR}/branch_records_fixed.lst
OUTFILE=${DATADIR}/post_processing_fixed_records.dat
>$OUTFILE # Zero out the file to start
else

MERGERECORDFILE=${DATADIR}/mergedrecords_variable.$(date +%m%d%y)
>$MERGERECORDFILE # Zero out the file to start
RECORDFILELIST=${DATADIR}/branch_records_variable.lst
OUTFILE=${DATADIR}/post_processing_variable_records.dat
>$OUTFILE=${DATADIR}/post_processing_variable_records.dat
>$OUTFILE=${DATADIR}/post_processing_variable_records.dat
}fi
```

Notice how we "zero out," or initialize, the files to zero size, by redirecting NULL data to the file > \$FILENAME. This is the same as cat /dev/null > \$FILENAME. We also define the field delimiter and assign the value to the FD variable, FD=: for this example. If we are working with a variable-length, we will need this information.

Next we define six functions, three for each record format. We have a merge function, a record-file parsing function, and a record-processing function for both types of record formats.

The record files that we need to process, from Listings 5-11, 5-12, 5-13, and 5-14, are defined in the *QDATADIR*/branch_records_fixed.lst file for fixed-length records, and *DATADIR*/branch_records_variable.lst file for variable-length records. Examples of these list files are shown here.

BRANCH_RECORDS_FIXED.LST

```
/data/branch01_fixed.dat
/data/branch02_fixed.dat
```

BRANCH_RECORDS_VARIABLE.LST

/data/branch01_variable.dat /data/branch02_variable.dat

With all these definitions we are now ready to start processing record files. At BEGINNING OF MAIN we use the RECORD_TYPE, which was defined on the command line, to decide which set of functions to execute. We take care of this decision-making with the following short case statement:

```
case $RECORD_TYPE in
fixed) merge_fixed_length_records
    parse_fixed_length_records
    ;;
variable) merge_variable_length_records
    parse_variable_length_records
    ;;
esac
```

Let's look at some results. For the first example, we merge and process the fixed-length record files using the following syntax:

```
# parse_record_files.Bash -f
```

The resulting file, post_processing_fixed_records.dat, is shown in Listing 5-16.

Listing 5-16 Data file listing for post_processing_fixed_records.dat

```
US72271247783490123456789Stacy.M.Spry.#######0000000000000000008847
01312008branch01_fixed.dat
US72271998233490123456789Marcus.S.Moore.#####000000000000000000095248 \
01312008branch01_fixed.dat
US72271235583290123456789Paul.W.Bryant.######000000000000000000886799 \
01312008branch01_fixed.dat
US82281299277300088258239Andy.C.James.#######00000000000000000882991 \
01312008branch02_fixed.dat
US82281200299938882456789Shakar.C.Divini.####000000000000000002211426
01312008branch02_fixed.dat
US82282990388872334456789Joe.M.Baker.########00000000000000000088234
01312008branch02_fixed.dat
US82281222945530092276611George.Williams.###000000000000000000881090
1312008branch02_fixed.dat
US82281234066820199822389Larry.C.Guy.#########00000000000000000000882 \
01312008branch02_fixed.dat
US82281247783490123436021Lacy.C.Cocaran.######000000000000000009921556
01312008branch02_fixed.dat
US82281998233490123426401Herman.G.Munster.###000000000000000000055324
01312008branch02_fixed.dat
US82281235583290123402173LaQuita.H.Dabney.###000000000000000000882371
01312008branch02_fixed.dat
```

To merge and process the variable-length record files, we use the following syntax:

```
# parse_record_files.Bash -v
```

The resulting file, post_processing_variable_records.dat, is shown in Listing 5-17.

```
US7227:1234567890123456789:Randal.K.Michael:9825:01312008:branch01_ \
variable.dat
US7227:1239922890123456789:James.M.Jones:1426:01312008: \
branch01_variable.dat
US7227:1528857890123456789:William.C.Rogers:427732:01312008: \
branch01_variable.dat
US7227:122945530123456789:Robin.M.Kim:2263320:01312008: \
branch01_variable.dat
US7227:1234066820123456789:Ana.Marie.Larson:4741:01312008: \
branch01_variable.dat
US7227:1247783490123456789:Stacy.M.Spry:8847:01312008: \
branch01_variable.dat
US7227:1998233490123456789:Marcus.S.Moore:95248:01312008: \
branch01_variable.dat
```

Listing 5-17 Data file listing for post_processing_variable_records.dat

```
US7227:1235583290123456789:Paul.W.Bryant:886799:01312008: \
branch01_variable.dat
US8228:1299277300088258239:Andy.C.James:882991:01312008: \
branch02_variable.dat
US8228:1200299938882456789:Shakar.C.Divini:2211426:01312008: \
branch02_variable.dat
US8228:2990388872334456789:Joe.M.Baker:88234:01312008: \
branch02_variable.dat
US8228:1222945530092276611:George.Williams:88109:01312008: \
branch02_variable.dat
US8228:1234066820199822389:Larry.C.Guy:882:01312008: \
branch02_variable.dat
US8228:1247783490123436021:Lacy.C.Cocaran:9921556:01312008: \
branch02_variable.dat
US8228:1998233490123426401:Herman.G.Munster:55324:01312008: \
branch02_variable.dat
US8228:1235583290123402173:LaQuita.H.Dabney:882371:01312008: \
branch02_variable.dat
```

By studying the resulting post-processing files in Listings 5-16 and 5-17, you can see that we accomplished all the tasks we set out to accomplish. We merged the files based on record format and added the record filename to the end of each record in the file, and then appended the file to the merged file ready for processing. We then processed the merged file, based on record format, to change the Date Due from 12/31/2007 to 1/31/2008. Study the parse_record_files.Bash script in Listing 5-15 in detail. You can pick up a few tips and tricks.

Other Things to Consider

As with any chapter there is not enough room to get to every single point in great detail. If you have record data files you are processing, you may need to do a lot of testing of the data integrity before using the data. So, you may need to add in tests for string length, integers, alphanumeric, date format, currency format, and many others.

You need to really know your data so that when you define variables you can specify the data type with the typeset command. For example, to define the variable VAR to be an integer, use the following syntax:

```
typeset -i VAR
```

We did not even touch on setting traps, logging errors, methods to cleanup after a failure, and all the other things that go with daily processing of data. If you know your data, you have a much better chance of success in writing your scripts.

As you process the data, you need to ensure you are checking return codes at each and every step. Missing just one return code can make troubleshooting an error a

two-week job, especially if is an intermittent error. Then there is the log file. If you are processing data, you should keep at least a week's worth of log files so that you can go back and look at previous processing runs. It could be that the error you are troubleshooting has been around for a while. You will not know if you are not keeping some of the previous log files. One week of log files is a good starting point. You may be required, by business requirements, to keep logs for 30 days or more.

Summary

Working with record files can be a challenge. To write a good and fast shell script to process record files, we need a good understanding of the data we are processing and the format that the data is in. For fixed-length records, we need to know where in the record each data field resides. For variable-length record files, we need to know how many fields of data we have and what they represent, and which character is used as the field delimiter. With this understanding we can build a comprehensive shell script that will process the data fast and efficiently. We can set up pre-processing and post-processing events, log errors, and send notifications of any issues that arise.

I hope you gained a good understanding of the challenges to processing record files and some techniques to help you succeed. In the next chapter we are going to look at automating FTP file transfers and related tasks.

Lab Assignments

- Using the parse_record_files.Bash script as a guide, write a shell script that will automatically detect the type of record file and process the record file correctly. The key is to check the string-length of the first record in the file. If the record is exactly 83 characters, the file is a fixed-length record file and has the following record definitions:
 - a. Characters 1-18 = Account number
 - b. Characters 19-29 = Phone number
 - c. Characters 30-55 = Name
 - d. Characters 56-76 = Address
 - e. Characters 77-83 = Balance
- 2. If the record file has variable-length records, the field delimiter is a : (colon), and there are five fields of data.

 - a. Field 1 =Account number
 - b. Field 2 = Phone number
 - c. Field 3 = Name

- d. Field 4 =Address
- e. Field 5 = Balance
- 3. You will have to create some longer record files for this exercise.
- 4. Rewrite the shell script in Lab Assignment 1 to add functionality to first merge a group of like record files (all fixed-length or all variable-length records). During the merge process your new script will add the record filename to the end of each record before appending the record file to the correct merge file. There are two merge files, one for fixed-length records and one for variable-length records. The last step is to process the merged batch-processing files. Again, the script must recognize the difference between fixed-length and variable-length record files and merge the record files into the correct merge file before processing each file.
- 5. Evaluate the following code by writing a shell script to incorporate this while loop:

echo \$RECORD | cut -d : -f1, -f2, -f3, -f4, -f5 \ | while read BRANCH ACCOUNT NAME TOTAL DATEDUE

to evaluate this variable-length record:

US7227:1234567890123456789:Randal.K.Michael:9825:12312007

When you echo each variable, is the data assigned correctly? Explain your findings.

6. Evaluate the following code by writing a shell script to incorporate this while loop:

```
while read BRANCH ACCOUNT NAME TOTAL DATEDUE
do
  echo $BRANCH
  echo $ACCOUNT
  echo $NAME
  echo $TOTAL
  echo $DATEDUE
  done < <(echo "$RECORD" | sed s/':'/''/g)</pre>
```

to evaluate this variable-length record:

US7227:1234567890123456789:Randal.K.Michael:9825:12312007

When you echo each variable, is the data assigned correctly? Explain your findings.

GHAPTER 6

Automated FTP Stuff

In many shops the business relies on nightly, or even hourly, file transfers of data that is to be processed. Due to the importance of this data, the data movement must be automated. The extent of automation in the FTP (and SFTP) world is threefold. We want the ability to move outbound files to another site, move inbound files from a remote location to our local machine, and check a remote site on a regular basis for files that are ready to download. In this chapter we are going to create some shell scripts to handle each of these scenarios.

Most businesses that rely on this type of data movement also require some pre-FTP and post-FTP processing to ready the system for the files before the transfer takes place and to verify the data integrity or file permissions after the transfer. For this pre and post processing we need to build into the shell script the ability to either hard-code the pre- and post-processing events or point to a file that performs these tasks. Now we are up to five pieces of code that we need to create.

Before you settle on a technique to move the data, be sure to also look at remote copy (rcp), Secure Copy (scp), and rsync, which are covered in Chapter 7, "Using rsync to Efficiently Replicate Data." If you need to move a large amount of data, rsync is your best option because of its built-in compression and the ability to update changes only, for certain file types. A database file cannot be updated with changes only, of course, but the compression makes rsync faster than FTP, remote copy, and Secure Copy.

Okay, the topic of this chapter is FTP and SFTP. Before we go any further let's look at the syntax for the FTP connections.

Syntax

Normally when we ftp a file, the remote machine's hostname is included as an argument to the ftp command. We are prompted for the password and, if it is entered correctly, we are logged in to the remote machine. We then can then transfer files

between the systems by moving to the local system directory containing files using the lcd (local change directory) command, and then move to the remote directory using the cd (change directory) command. In either case, we are working with an interactive program. A typical FTP session looks like the output shown in Listing 6-1.

```
[root:yogi]@/scripts# ftp wilma
Connected to wilma.
220 wilma FTP server (SunOS 5.8) ready.
Name (wilma:root): randy
331 Password required for randy.
Password:
230 User randy logged in.
ftp> lcd /scripts/download
250 CWD command successful.
ftp> cd /scripts
250 CWD command successful.
ftp> get auto ftp xfer.ksh
200 PORT command successful.
150 ASCII data connection for auto_ftp_xfer.ksh (10.10.10.1,32787)
  (227 bytes).
226 ASCII Transfer complete.
246 bytes received in 0.0229 seconds (10.49 Kbytes/s)
local: auto_ftp_xfer.ksh remote: auto_ftp_xfer.ksh
ftp> bye
221 Goodbye.
[root:yogi]@/scripts/download#
```

Listing 6-1 Typical FTP file download

As you can see in Listing 6-1, the ftp command requires interaction with the user to make the transfer of the file from the remote machine to the local machine. How do we automate this interactive process? One option is to use a *here document*. A here document is a coding technique that allows us to place all the required interactive command input between two *labels*. However, a here document will not work for a Secure Shell (ssh, scp, and sftp) login. We will cover this topic later in this chapter. Let's look at an example of coding a simple FTP transfer using this automation technique in Listing 6-2.

```
#!/bin/ksh
#
# SCRIPT: tst_ftp.ksh
# AUTHOR: Randy Michael
# DATE: 6/12/2007
# REV: 1.1.A
# PLATOFRM: Not platform dependent
#
```

Listing 6-2 Simple here document for FTP transfer in a script

```
# PURPOSE: This shell script is a simple demonstration of
# using a here document in a shell script to automate
# an FTP file transfer.
#
# Connect to the remote machine and begin a here document.
ftp -i -v -n wilma <<END_FTP
user randy mypassword
binary
lcd /scripts/download
cd /scripts
get auto_ftp_xfer.ksh
bye
END_FTP
```

Listing 6-2 (continued)

Notice in Listing 6-2 where the beginning and ending labels are located. The first label, <<END_FTP, begins the here document and is located just after the interactive command that requires input, which is the ftp command in our case. Next comes all of the input that a user would have to supply to the interactive command. In this example we log in to the remote machine, wilma, using the user randy mypassword syntax. This ftp command specifies that the user is randy and the password is mypassword. Once the user is logged in, we set up the environment for the transfer by setting the transfer mode to binary, locally changing directory to /scripts/download, then changing the directory on the remote machine to /scripts. The last step is to get the auto_ftp_xfer.ksh file. To exit the FTP session, we use bye; quit also works. The last label, END_FTP, ends the here document, and the script exits.

Also notice the ftp command switches used in Listing 6-2. The -i command switch turns off interactive prompting during multiple file transfers so there is no prompt for the username and password. See the FTP man pages for prompt, mget, mput, and mdelete subcommands for descriptions of prompting during multiple file transfers. The -n switch prevents an automatic login on the initial connection. Otherwise, the ftp command searches for a \$HOME/.netrc entry that describes the login and initialization process for the remote host. See the user subcommand in the man page for ftp. The -v switch was added to the ftp command to set *verbose* mode, which allows us to see the commands as the FTP sessions execute. The tst_ftp.ksh shell script from Listing 6-2 is shown in action in Listing 6-3.

```
[root:yogi]@/scripts# ./tst_ftp.ksh
Connected to wilma.
220 wilma FTP server (SunOS 5.8) ready.
```

Listing 6-3 Simple automated FTP file transfer using a script

```
331 Password required for randy.
230 User randy logged in.
200 Type set to I.
Local directory now /scripts/download
250 CWD command successful.
200 PORT command successful.
150 Binary data connection for auto_ftp_xfer.ksh
(10.10.10.1,32793) (227 bytes).
226 Binary Transfer complete.
227 bytes received in 0.001092 seconds (203 Kbytes/s)
local: auto_ftp_xfer.ksh remote:
auto_ftp_xfer.ksh
221 Goodbye.
[root:yogi]@/scripts#
```

Listing 6-3 (continued)

Using these techniques, we are going to create shell scripts to tackle some of the common needs of a business that depends on either receiving data from or transferring data to a remote host.

Automating File Transfers and Remote Directory Listings

We have the basic idea of automating an FTP file transfer, but what do we want to accomplish? We really want to be able to do three things: download one or more files with the get or mget FTP subcommands, upload one or more files with the FTP put or mput subcommands, and show a directory listing from a remote host. The first two items are standard uses for any FTP script, but getting a remote directory listing has not been explained in any of the documentation of a scripting technique that I have seen.

Additionally, we need to add the ability of pre-event and post-event processing. For example, a pre-FTP event may be getting a directory listing from a remote host. A post-ftp event may be changing the ownership and file permissions on a newly downloaded file(s). This last example brings up another point. When you FTP a file that has the execute bit set, the file will be received with the execute bit *un*set. Any time you ftp a file, the execution bit is stripped out of the file permissions. However, this is not the case if you use sftp (secure FTP).

Let's look at these topics one at a time.

Using FTP for Directory Listings on a Remote Machine

To save a remote directory listing from a remote system to a local file, we use the FTP subcommand nlist. The nlist subcommand has the following form:

```
nlist [RemoteDirectory][LocalFile]
```

The nlist subcommand writes a listing of the contents of the specified remote directory (RemoteDirectory) to the specified local file (LocalFile). If the Remote-Directory parameter is not specified, the nlist subcommand lists the contents of the current remote directory. If the LocalFile parameter is not specified or is a - (hyphen), the nlist subcommand displays the listing on the local terminal.

Let's create a little shell script to test this idea. We can use most of the shell script contents shown in Listing 6-2, but we remove the get command and replace it with the nlist subcommand. Take a look at Listing 6-4.

```
#!/bin/ksh
#
# SCRIPT: get_remote_dir_listing.ksh
# AUTHOR: Randy Michael
# DATE: July 15, 2007
# REV: 1.1.P
# PLATFORM: Not Platform Dependent
#
# PURPOSE: This shell script uses FTP to get a remote directory listing
      and save this list in a local file.
±
# set -n # Uncomment to check the script syntax without any execution
# set -x # Uncomment to debug this shell script
RNODE="wilma"
USER="randy"
UPASSWD="mypassword"
LOCALDIR="/scripts/download"
REMOTEDIR="/scripts"
DIRLISTFILE="${LOCALDIR}/${RNODE}.$(basename ${REMOTEDIR}).dirlist.out"
cat /dev/null > $DIRLISTFILE
*********
ftp -i -v -n $RNODE <<END_FTP
user $USER $UPASSWD
nlist $REMOTEDIR $DIRLISTFILE
bye
END_FTP
```

Listing 6-4 get_remote_dir_listing.ksh shell script

There are several things to point out in Listing 6-4. We start out with a variable definition section. In this section we define the remote node, the username and password for the remote node, a local directory, a remote directory, and finally the local file that is to hold the remote directory listing. Notice that we had to create this file. If the local file does not already exist, the remote listing to the local file will fail. To create the file you can use either of the following techniques:

cat /dev/null > \$DIRLISTFILE
>\$DIRLISTFILE
touch \$DIRLISTFILE

The first two examples create an empty file or will make an existing file empty. The touch command will update the time stamp for the file modification for an existing file and will create the file if it does not exist.

At the BEGINNING OF MAIN we have our five lines of code that obtain the directory listing from the remote node. We use the same technique as we did in Listing 6-2 except that we use variables for the remote node name, username, and password. Variables are also used for the directory name on the remote machine and for the local filename that holds the directory listing from the remote machine using the FTP subcommand nlist.

Notice that the password is hard-coded into this shell script. This is a security nightmare! The file permissions on this shell script should be set to read and execute by the owner only. In a later section in this chapter we will cover a technique for replacing hard-coded passwords with hidden password variables.

Getting One or More Files from a Remote System

Now we get to some file transfers. Basically we are going to combine the shell scripts in Listings 6-3 and 6-4. We are also going to add the functionality to add pre-FTP and post-FTP events. Let's start by looking at the shell script in Listing 6-5, get_ftp_files.ksh.

```
#!/bin/ksh
#
#
# SCRIPT: get_ftp_files.ksh
# AUTHOR: Randy Michael
# DATE: July 15, 2007
# REV: 1.1.P
#
#
# PLATFORM: Not Platform Dependent
#
# PURPOSE: This shell script uses FTP to get a list of one or more
# remote files from a remote machine.
#
# set -n # Uncomment to check the script syntax without any execution
# set -x # Uncomment to debug this shell script
#
```

Listing 6-5 get_ftp_files.ksh shell script

```
*********
REMOTEFILES=$1
THISSCRIPT=$(basename $0)
RNODE="wilma"
USER="randy"
UPASSWD="mypassword"
LOCALDIR="/scripts/download"
REMOTEDIR="/scripts"
# Set up the correct echo command usage. Many Linux
# distributions will execute in Bash even if the
# script specifies Korn shell. Bash shell requires
# we use echo -e when we use n, c, etc.
case $SHELL in
*/bin/Bash) alias echo="echo -e"
        ;;
esac
*********
pre event ()
{
# Add anything that you want to execute in this function. You can
# hard-code the tasks in this function or create an external shell
# script and execute the external function here.
: # no-op: The colon (:) is a no-op character. It does nothing and
  # always produces a 0, zero, return code.
}
post_event ()
# Add anything that you want to execute in this function. You can
# hard-code the tasks in this function or create an external shell
# script and execute the external function here.
 # no-op: The colon (:) is a no-op character. It does nothing and
:
  # always produces a 0, zero, return code.
}
```

Listing 6-5 (continued)

```
****
usage ()
{
echo "\nUSAGE: $THISSCRIPT \"One or More Filenames to Download\" \n"
exit 1
}
usage_error ()
{
echo "\nERROR: This shell script requires a list of one or more
    files to download from the remote site.\n"
usage
}
# Test to ensure that the file(s) is/are specified in the $1
# command-line argument.
(($# != 1)) && usage_error
pre event
# Connect to the remote site and begin the here document.
ftp -i -v -n $RNODE <<END_FTP
user $USER $UPASSWD
binary
lcd $LOCALDIR
cd $REMOTEDIR
mget $REMOTEFILES
bye
END_FTP
post_event
```

Listing 6-5 (continued)

We made a few changes in Listing 6-5. The two major changes are that we obtain the list of files to download from the \$1 command-line argument, and we set up the correct echo command usage. Even though we declare the shell as Korn on the very first line

of the script, many Linux distributions will still execute in Bash shell. I have found this to be true with my Linux Fedora 7 release. If the executing shell is */bin/Bash, the echo command requires a -e to enable the backslash operators \n, \c, and so on. We set up the proper echo command usage with the following code segment:

```
# Set up the correct echo command usage. Many Linux
# distributions will execute in Bash even if the
# script specifies Korn shell. Bash shell requires
# we use echo -e when we use \n, \c, etc.
case $SHELL in
*/bin/Bash) alias echo="echo -e"
;;
esac
```

Moving on, if more than one file is listed on the command line, they must be enclosed in quotes, "file1 file2 file3 filen", so they are interpreted as a single argument in the shell script. A blank space is assumed when separating the filenames in the list.

Notice that the local and remote directories are hard-coded into the shell script. If you want, you can modify this shell script and use getopts to parse through some command-line switches. This sounds like a great Lab Assignment!

Because we are now requiring a single argument on the command line, we also need to add a usage function to this shell script. We are looking for exactly one command-line argument. If this is not the case, we execute the usage_error function, which in turn executes the usage function. We could just as easily combine these functions into a single function.

Because we may have more than one filename specified on the command line, we need to use the FTP subcommand mget, as opposed to get. We have already turned off interactive prompting by adding the -i switch to the ftp command so there will not be any prompting when using mget.

Pre and Post Events

Notice in Listing 6-5 that we added two new functions, pre_event and post_event. By default, both of these functions contain only the no-op character, : (colon). A : does nothing but always has a return code of 0, zero. We are using this as a placeholder to have something in the function.

If you have a desire to perform a task before or after the FTP activity, enter the tasks in the pre_event and/or the post_event functions. It is a good idea to enter only a filename of an external shell script rather than editing this shell script and trying to debug a function in an already working shell script. An external shell script filename that is executable is all that is needed to execute the pre and post events.

In the external shell script enter everything that needs to be done to set up the environment for the FTP file transfers. Some things that you may want to do include removing the old files from a directory before downloading new files or getting a directory listing of a remote host to see if there is anything to even download. You can make the code as long or as short as needed to accomplish the task at hand.

Script in Action

To see the shell script in Listing 6-5 in action, look at Listing 6-6, where we are transferring the shell script to another host in the network.

```
[root:yogi]@/scripts# ./get_ftp_files.ksh get_ftp_files.ksh
Connected to wilma.
220 wilma FTP server (SunOS 5.8) ready.
331 Password required for randy.
230 User randy logged in.
200 Type set to I.
Local directory now /scripts/download
250 CWD command successful.
200 PORT command successful.
150 Binary data connection for get_ftp_files.ksh (10.10.10.1,32808)
(1567 bytes)
226 Binary Transfer complete.
1567 bytes received in 0.001116 seconds (1371 Kbytes/s)
local: get_ftp_files.ksh remote: get_ftp_files.ksh
221 Goodbye.
[root:yogi]@/scripts#
```

Listing 6-6 get_ftp_files.ksh shell script in action

In this example the transfer is taking place between a local AIX machine called yogi and the remote SunOS machine called wilma.

Uploading One or More Files to a Remote System

Uploading files to another machine is the same as downloading the files except we now use the put and mput commands. Let's slightly modify the shell script in Listing 6-5 to make it into an upload script. This script modification is shown in Listing 6-7.

```
#!/bin/ksh
#
# SCRIPT: put_ftp_files.ksh
# AUTHOR: Randy Michael
# DATE: July 15, 2007
# REV: 1.1.P
#
# PLATFORM: Not Platform Dependent
#
# PURPOSE: This shell script uses FTP to put a list of one or more
# local files to a remote machine.
```

Listing 6-7 put_ftp_files.ksh shell script

```
# set -n # Uncomment to check the script syntax without any execution
# set -x # Uncomment to debug this shell script
LOCALFILES=$1
THISSCRIPT=$(basename $0)
RNODE="wilma"
USER="randy"
UPASSWD="mypassword"
LOCALDIR="/scripts"
REMOTEDIR="/scripts/download"
# Set up the correct echo command usage. Many Linux
# distributions will execute in Bash even if the
# script specifies Korn shell. Bash shell requires
# we use echo -e when we use \n, \c, etc.
case $SHELL in
*/bin/Bash) alias echo="echo -e"
        ;;
esac
*********
pre_event ()
# Add anything that you want to execute in this function. You can
# hard-code the tasks in this function or create an external shell
# script and execute the external function here.
: # no-op: The colon (:) is a no-op character. It does nothing and
  # always produces a 0, zero, return code.
}
post event ()
# Add anything that you want to execute in this function. You can
# hard-code the tasks in this function or create an external shell
# script and execute the external function here.
```

Listing 6-7 (continued)

```
: # no-op: The colon (:) is a no-op character. It does nothing and
  # always produces a 0, zero, return code.
}
*************
usage ()
{
echo "\nUSAGE: $THISSCRIPT \"One or More Filenames to Download\" \n"
exit 1
}
usage_error ()
{
echo "\nERROR: This shell script requires a list of one or more
    files to download from the remote site.\n"
usage
}
# Test to ensure that the file(s) is/are specified in the $1
# command-line argument.
(($# != 1)) && usage_error
pre_event
# Connect to the remote site and begin the here document.
ftp -i -v -n $RNODE <<END_FTP
user $USER $UPASSWD
binary
lcd $LOCALDIR
cd $REMOTEDIR
mput $LOCALFILES
bye
END_FTP
post_event
```

Listing 6-7 (continued)

The script in Listing 6-7 uses the same techniques as the get_ftp_files.ksh shell script in Listing 6-5. We have changed the \$1 variable assignment to LOCALFILES instead of REMOTEFILES and changed the FTP transfer mode to mput to upload the files to a remote machine. Other than these two changes the scripts are identical.

In all of the shell scripts in this chapter we have a security nightmare with hard-coded passwords. In the next section is a technique that allows us to remove these hard-coded passwords and replace them with hidden password variables. Following the next section we will use this technique to modify each of our shell scripts to utilize hidden password variables.

Replacing Hard-Coded Passwords with Variables

Traditionally, when a password is required in a shell script, it is hard-coded into the script. Using this hard-coded technique presents us with a lot of challenges, ranging from a security nightmare to the inability to change key passwords on a regular basis. The variable technique presented in this section is very easy to implement with only minor changes to *each* shell script.

NOTE It's important to note that each shell script must be changed to properly implement the technique throughout the infrastructure.

The variable-replacement technique is the same thing as sourcing the environment and consists of a single file that contains unique variable assignments for each password required for shell scripts on the system. A sample password file looks like the following:

```
DBORAPW=abc123
DBADMPW=def456
BACKUPW=ghi789
RANDY=mypassword
```

Some of the considerations of implementing this variable-replacement technique include the following:

- The scope of where the variable containing the password can be seen
- The file permission of the password variable file that contains the hard-coded passwords

To limit the scope of the variable, it is extremely important that the variable *not* be exported in the password variable file. If the variable is exported, you will be able to see the password *in plain text* in the process environment of the shell script that is using the password variable. Additionally, with all of these passwords in a single file, the file must be locked down to *read-only* by root, ideally.

The best illustration of this technique is a real example of how it works. In the following code sections, shown in Listings 6-8, 6-9, and 6-10, there is a password file that contains the password variable assignments that has the name setpwenv.ksh (notice this file is a shell script!). In the first file, the password variable *is* exported. In the second file, the password variable is *not* exported. Following these two files is a shell

script, mypwdtest.ksh, that executes the password environment file, setpwenv.ksh, and tests to see if the password is visible in the environment.

NOTE Test results of using each technique are detailed in the next section of this chapter.

Example of Detecting Variables in a Script's Environment

We start the examples with a setpwenv.ksh file that exports the password variable in Listing 6-8.

```
#!/bin/ksh
#
# SCRIPT: setpwenv.ksh
#
# PURPOSE: This shell script is executed by other shell
# scripts that need a password contained in
# a variable
#
#
# This password is NOT exported
# MYPWDTST=bonehead
# This password IS exported
MYPWDTST=bonehead
export MYPWDTST
```

Listing 6-8 Password file with the password variable exported

Notice in Listing 6-8 that the password *is* exported. As you will see, this export of the password variable will cause the password to be visible in plain text in the calling shell script's environment.

The password file in Listing 6-9 shows an example of *not* exporting the password variable.

```
#!/bin/ksh
#
# SCRIPT: setpwenv.ksh
#
# PURPOSE: This shell script is executed by other shell
# scripts that need a password contained in
# a variable
#
```

Listing 6-9 Password file showing the variable not exported

```
#
# This password is NOT exported
MYPWDTST=bonehead
# This password IS exported
# MYPWDTST=bonehead
# export MYPWDTST
```

Listing 6-9 (continued)

Notice in Listing 6-9 that the password is *not* exported. As you will see, this variable assignment without exporting the password variable will cause the password to *not* be visible in the calling shell script's environment, which is the result that we are looking for.

The shell script shown in Listing 6-10 performs the test of the visibility of the password assigned for each of the password environment files.

```
#!/usr/bin/ksh
#
# SCRIPT: mypwdtest.ksh
#
# PURPOSE: This shell script is used to demonstrate the
#
          use of passwords hidden in variables.
# set -x # Uncomment to debug this script
# set -n # Uncomment to check syntax without any execution
# Set up the correct echo command usage. Many Linux
# distributions will execute in Bash even if the
# script specifies Korn shell. Bash shell requires
# we use echo -e when we use \n, \c, etc.
case $SHELL in
*/bin/Bash) alias echo="echo -e"
            ;;
esac
# Set the BIN directory
BINDIR=/usr/local/bin
# Execute the shell script that contains the password variable
# assignments.
. ${BINDIR}/setpwenv.ksh
```

Listing 6-10 Shell script to demonstrate the scope of a variable

Listing 6-10 (continued)

The shell script shown in Listing 6-10, mypwdtest.ksh, tests the environment using each of the setpwenv.ksh shell scripts, one with the password environment exported and the second file that does not do the export. You can see the results of the tests here.

Example with the Password Variable Exported

PASS is bonehead Searching for the password in the environment... MYPWDTST=bonehead ERROR: Password was found in the environment

Notice in the previous example that the password is visible in the shell script's environment. When the password is visible you can run the **env** command while the mypwdtest.ksh shell script is executing and see the password in plain text. Therefore, anyone could conceivably get the passwords very easily. Notice in the next output that the password is hidden.

Example with the Password Variable Not Exported

PASS is bonehead Searching for the password in the environment... SUCCESS: Password was NOT found in the environment

NOTE As you can see, it is extremely important never to export a password variable.

To implement this variable-password substitution into your shell scripts you only need to add the password to the password environment file using a unique variable name. Then inside the shell script that requires the password you execute the password file, which is setpwenv.ksh in our case. After the password file is executed the password variable(s) is/are ready to use.

NOTE The preceding content uses this technique for passwords; however, this practice can also be utilized for usernames, hostnames, and application variables. The main purpose of this exercise is to have a central point of changing passwords on a regular basis and to eliminate hard-coded passwords in shell scripts.

Modifying Our FTP Scripts to Use Password Variables

As you saw in the previous section, it is an easy task to modify a shell script to take advantage of hidden password variables. Here we make the two lines of modifications to our nlist, get, and put shell scripts.

The first thing that we need to do is create a password environment file. Let's use a name that is a little more obscure than setpwenv.ksh. How about setlink.ksh? Also, let's hide the setlink.ksh shell script in /usr/sbin for a little more security. Next let's set the file permissions to 400, *read-only* by the owner (root). Now you may be asking how we can execute a shell script that is read-only. All we need to do is to *dot* the filename, or source the filename. An example of dotting the file is shown here:

. /usr/sbin/setlink.ksh

The dot just says to execute the filename that follows. Now let's set up the password environment file. This example assumes that the user is randy:

echo "RANDY=mypassword" >> /usr/sbin/setlink.ksh
chown 400 /usr/sbin/setlink.ksh

Now in each of our shell scripts we need to *dot* the /usr/sbin/setlink.ksh file and replace the hard-coded password with the password variable defined in the external file, /usr/sbin/setlink.ksh, which is \$RANDY in our case.

Listings 6-11, 6-12, and 6-13 show the modified shell scripts with the hard-coded passwords removed.

```
#!/bin/ksh
#
#
SCRIPT: get_remote_dir_listing_pw_var.ksh
# AUTHOR: Randy Michael
# DATE: July 15, 2007
# REV: 1.1.P
#
#
# PLATFORM: Not Platform Dependent
#
# PURPOSE: This shell script uses FTP to get a remote directory listing
# and save this list in a local file. This shell script uses
# remotely defined passwords.
#
```

Listing 6-11 get_remote_dir_listing_pw_var.ksh script

```
# set -n # Uncomment to check the script syntax without any execution
# set -x # Uncomment to debug this shell script
RNODE="wilma"
USER="randy"
LOCALDIR="/scripts/download"
REMOTEDIR="/scripts"
DIRLISTFILE="${LOCALDIR}/${RNODE}.$(basename ${REMOTEDIR}).dirlist.out"
cat /dev/null > $DIRLISTFILE
# Get a password
. /usr/sbin/setlink.ksh
ftp -i -v -n $RNODE <<END_FTP
user $USER $RANDY
nlist $REMOTEDIR $DIRLISTFILE
bye
END FTP
```

Listing 6-11 (continued)

In Listing 6-11 the only modifications that we made to the original shell script include a script name change, the removal of hard-coded passwords, adding the execution of the /usr/sbin/setlink.ksh shell script, and adding the \$RANDY remotely defined password variable.

```
#!/bin/ksh
#
# SCRIPT: get_ftp_files_pw_var.ksh
# AUTHOR: Randy Michael
# DATE: July 15, 2007
# REV: 1.1.P
#
# PLATFORM: Not Platform Dependent
#
```

Listing 6-12 get_ftp_files_pw_var.ksh shell script

```
# PURPOSE: This shell script uses FTP to get one or more remote
       files from a remote machine. This shell script uses a
#
       remotely defined password variable.
# set -n # Uncomment to check the script syntax without any execution
# set -x # Uncomment to debug this shell script
*********
REMOTEFILES=$1
THISSCRIPT=$(basename $0)
RNODE="wilma"
USER="randy"
LOCALDIR="/scripts/download"
REMOTEDIR="/scripts"
# Set up the correct echo command usage. Many Linux
# distributions will execute in Bash even if the
# script specifies Korn shell. Bash shell requires
# we use echo -e when we use \n, \c, etc.
case $SHELL in
*/bin/Bash) alias echo="echo -e"
        ;;
esac
*****
pre_event ()
# Add anything that you want to execute in this function. You can
# hard-code the tasks in this function or create an external shell
# script and execute the external function here.
: # no-op: The colon (:) is a no-op character. It does nothing and
  # always produces a 0, zero, return code.
3
post_event ()
{
# Add anything that you want to execute in this function. You can
```

Listing 6-12 (continued)

```
# hard-code the tasks in this function or create an external shell
# script and execute the external function here.
: # no-op: The colon (:) is a no-op character. It does nothing and
  # always produces a 0, zero, return code.
}
***********
usage ()
{
echo "\nUSAGE: $THISSCRIPT \"One or More Filenames to Download\" \n"
exit 1
}
***********
usage_error ()
{
echo "\nERROR: This shell script requires a list of one or more
     files to download from the remote site.\n"
usage
}
# Test to ensure that the file(s) is/are specified in the $1
# command-line argument.
(($# != 1)) && usage_error
# Get a password
. /usr/sbin/setlink.ksh
pre_event
ftp -i -v -n $RNODE <<END_FTP
user $USER $RANDY
binary
lcd $LOCALDIR
cd $REMOTEDIR
mget $REMOTEFILES
bye
```

Listing 6-12 (continued)

END_FTP post_event

Listing 6-12 (continued)

In Listing 6-12 the only modifications that we made to the original shell script include a script name change, the removal of hard-coded passwords, adding the execution of the /usr/sbin/setlink.ksh shell script, and adding the \$RANDY remotely defined password variable.

```
#!/bin/ksh
#
# SCRIPT: put_ftp_files_pw_var.ksh
# AUTHOR: Randy Michael
# DATE: July 15, 2007
# REV: 1.1.P
#
# PLATFORM: Not Platform Dependent
#
# PURPOSE: This shell script uses FTP to put a list of one or more
        local files to a remote machine. This shell script uses
#
#
        remotely defined password variables
# set -n # Uncomment to check the script syntax without any execution
# set -x # Uncomment to debug this shell script
**********
*****
LOCALFILES=$1
THISSCRIPT=$(basename $0)
RNODE="wilma"
USER="randy"
LOCALDIR="/scripts"
REMOTEDIR="/scripts/download"
# Set up the correct echo command usage. Many Linux
# distributions will execute in Bash even if the
# script specifies Korn shell. Bash shell requires
# we use echo -e when we use n, c, etc.
case $SHELL in
```

Listing 6-13 put_ftp_files_pw_var.ksh shell script

```
*/bin/Bash) alias echo="echo -e"
        ;;
esac
***********
pre_event ()
{
# Add anything that you want to execute in this function. You can
# hard-code the tasks in this function or create an external shell
# script and execute the external function here.
: # no-op: The colon (:) is a no-op character. It does nothing and
  # always produces a 0, zero, return code.
}
post_event ()
{
# Add anything that you want to execute in this function. You can
# hard-code the tasks in this function or create an external shell
# script and execute the external function here.
: # no-op: The colon (:) is a no-op character. It does nothing and
  # always produces a 0, zero, return code.
}
usage ()
{
echo "\nUSAGE: $THISSCRIPT \"One or More Filenames to Download\" \n"
exit 1
}
usage_error ()
{
echo "\nERROR: This shell script requires a list of one or more
    files to download from the remote site.\n"
usage
}
```

Listing 6-13 (continued)

```
# Test to ensure that the file(s) is/are specified in the $1
# command-line argument.
(($# != 1)) && usage_error
# Get a password
. /usr/sbin/setlink.ksh
pre_event
# Connect to the remote site and begin the here document.
ftp -i -v -n $RNODE <<END_FTP
user $USER $RANDY
binary
lcd $LOCALDIR
cd SREMOTEDIR
mput $LOCALFILES
bye
END_FTP
post_event
```

Listing 6-13 (continued)

In Listings 6-11, 6-12, and 6-13, the only modifications that we made to the original shell scripts include a script name change, the removal of hard-coded passwords, adding the execution of the /usr/sbin/setlink.ksh shell script, and adding the \$RANDY remotely defined password variable.

What about Encryption?

I'm glad you asked! Most shops are now disabling FTP in favor of secure FTP. If you work for a publicly traded U.S. corporation, your company must comply with the Sarbanes-Oxley (SOX) Act of 2002. To comply with SOX requirements, the CEO and CFO of the corporation must sign off on the validity of the financial statement sent to the Securities and Exchange Commission (SEC) on a yearly basis. This effort also includes making sure there are controls in place to ensure the data integrity. Most non-encrypted communications must be disabled in favor of the Secure Shell alternatives.

For encrypted file transfers, we can use OpenSSH's secure FTP (sftp) or Secure Copy (scp). Secure FTP does not support the use of here documents, so we have two options. First, set up the encryption key pairs so that we are not required to use a password. Second, create an expect script to handle the interaction with sftp or scp. By far the simplest solution is to create the Secure Shell encryption keys so that a password is not required.

OpenSSH supports RSA and DSA encryption. You can set up one or both sets of keys for a user. Usually an organization uses one or the other, but not both as a company standard.

Creating Encryption Keys

If you want to transfer files between systems, you first must decide on the username who will own the files, and thus the file transfer tasks. The encryption keys are created in the user's \$HOME/.ssh directory by executing the following command:

ssh-keygen -t rsa|dsa

where rsa or dsa represents the encryption key type you want to create. For our example, we are going to use RSA encryption keys. To create the encryption keys, first log in as the user you want to set up the keys for. Then change the directory to this user's \$HOME directory. Just type cd and press Enter. Next cd to the .ssh subdirectory. If the \$HOME/.ssh directory exists, the keys may have already been set up. Look for files named id_rsa.pub or id_dsa.pub. These files contain the public encryption keys for the particular encryption protocol. If you do not see either of these files, you need to set up the keys:

ssh-keygen -t rsa

Take the defaults for all the prompts and just press Enter when prompted to enter a passphrase. It takes a few seconds to create the encryption keys, and then you are returned to the command prompt. Now you should have a new set of encryption keys in the \$HOME/.ssh directory.

Setting Up No-Password Secure Shell Access

To set up the no-password Secure Shell access, the user needs an account on both servers as well as a \$HOME directory. To log in remotely without a password, the user's local public encryption key, located in the \$HOME/.ssh/id_rsa.pub file, needs to be appended to the end of the remote system's \$HOME/.ssh/authorized_keys file. The following command will take care of the task for us. This is an example of setting up the ftpadmin user for no-password access to the remote machine booboo:

```
cat $HOME/.ssh/id_rsa.pub | ssh ftpadmin@booboo \
"cat >> ~ftpuser/.ssh/authorized_keys"
```

If it is the first time this user has used Secure Shell to access the server, the user will be prompted to add a local key; just type **yes**. Next the user is prompted for a password and, if it's correct, the key is appended to the remote \$HOME/.ssh/authorized_keys file. The next time you log in, you should not need a password.

Now that we have the encryption keys set up for our file transfer "owner" created, we only need to get the syntax for Secure Shell file transfers.

Secure FTP and Secure Copy Syntax

The syntax for Secure Copy (scp) and secure FTP (sftp) is almost identical for transferring files. Study the syntax of the sftp versus the scp syntax:

```
sftp ${COPYMGR}@${REM_MACHINE}:$REM_FILE $LOC_FILE
```

```
scp ${COPYMGR}@${REM_MACHINE}:$REM_FILE $LOC_FILE
```

The only difference is the command we use. Check out Listing 6-14 and we will cover the details at the end.

```
#!/bin/Bash
# SCRIPT: sftp-scp-getfile.Bash
# AUTHOR: Randy Michael
# DATE: 11/15/2007
# REV: 1.0
# PURPOSE: This is a simple script to show how
# Secure FTP and Secure Copy work with and
# without passwords. This is done by removing
# and adding the remote host's key in the local
# $HOME/.ssh/known_hosts file.
#
*****
# DEFINE FILES AND VARIABLES HERE
*****
COPYMGR=ftpadmin
REM MACHINE=booboo
REM_FILE=/home/ftpadmin/getall.sh
LOC_FILE=$HOME # Use the same filename but
            # store the file in $COPYMGR
            # $HOME directory.
*****
# BEGINNING OF MAIN
*****
```

Listing 6-14 sftp-scp-getfile.Bash shell script

```
echo -e "\nSecure FTP Method\n"
sftp ${COPYMGR}@${REM_MACHINE}:$REM_FILE $LOC_FILE
echo -e "\nSecure Copy Method\n"
scp ${COPYMGR}@${REM_MACHINE}:$REM_FILE $LOC_FILE
echo -n
```

Listing 6-14 (continued)

In Listing 6-14, we define the COPYMGR as the ftpadmin user. The remote machine is defined as booboo, and the remote file to download is /home/ftpadmin/getall.sh. We are just going to store the files in the ftpadmin user's \$HOME directory. Both file transfers are shown in Listing 6-15.

```
[sysadmin@yogi scripts]$ ./sftp-scp-getfile.Bash
Secure FTP Method
Connecting to booboo...
Fetching /home/sysadmin/getall.sh to /home/sysadmin/getall.sh
                        0% 0 0.0KB/s --:-- ETA
/home/sysadmin/getall.sh
/home/sysadmin/getall.sh
                          100%
  666 0.7KB/s 00:00
Secure Copy Method
getall.sh 0 0.0KB/s --:--
                                   ETA
                          100%
getall.sh
 666 0.7KB/s 00:00
[ftpadmin@yogi scripts]$
```

Listing 6-15 sftp-scp-getfile.Bash shell script in action

I really have no preference between these two methods; they both get the job done. Now let's change gears a bit and look at a way to automatically create a script to do the work for us.

Automating FTP with autoexpect and expect Scripts

Expect is a scripting language to automate an interactive program, such as FTP. The idea for an expect script is to send data to an interactive program, then *expect* data back,

and send more data back to the interactive application. The result is fully automated interaction with an interactive program.

Another nice thing about expect is the ability to create a complete script automatically, just by interacting with the program manually once, and, of course, not making a mistake. This magic is possible because of the autoexpect program. The autoexpect program works like the shell's script program works: by recording each keystroke you type as well as the reply from the application or shell. Then you press Ctrl + D to save the new expect script in the current directory. The default filename is script.exp, and this file is overwritten each time you execute autoexpect from this same directory. So, it's a good idea to rename the script.exp file to something a bit more meaningful and descriptive. You can also specify the resulting expect script filename with the -f filename switch.

autoexpect

saves the resulting expect script in the current directory with the filename script.exp.

autoexpect -f ftp-ls-getfile.exp

saves the resulting expect script in the current directory with the filename ftp-ls-getfile.exp.

```
# autoexpect -f /scripts/ftp-ls-getfile.exp
```

saves the resulting expect script in the /scripts directory with the filename ftp-ls-getfile.exp.

Now let's create a little expect script to log in to a remote server, list the contents of the login directory, and download the getall.ksh file. We are going to use autoexpect to create this script. The only thing to remember is that you cannot make a mistake. You cannot even press the Backspace or arrow keys. Every keystroke is recorded, and when we execute the expect script, it will "replay" the actions we recorded during the autoexpect session. Check out ftp-ls-getfile.exp in Listing 6-16 and we will cover the details at the end.

```
#!/usr/local/bin/expect -f
#
# This Expect script was generated by autoexpect on Fri Nov 16
    12:29:38 2007
# Expect and autoexpect were both written by Don Libes, NIST.
#
# Note that autoexpect does not guarantee a working script. It
# necessarily has to guess about certain things. Two reasons a
# script might fail are
#
# 1) timing - A surprising number of programs (rn, ksh, zsh, telnet,
```

Listing 6-16 expect script listing ftp-ls-getfile.exp

```
# etc.) and devices discard or ignore keystrokes that arrive "too
# quickly" after prompts. If you find your new script hanging up at
# one spot, try adding a short sleep just before the previous send.
# Setting "force_conservative" to 1 (see below) makes Expect do this
# automatically - pausing briefly before sending each character. This
# pacifies every program I know of. The -c flag makes the script do
# this in the first place. The -C flag allows you to define a
# character to toggle this mode off and on.
set force_conservative 0 ;# set to 1 to force conservative mode even if
                     ;# script wasn't run conservatively originally
if {$force_conservative} {
     set send_slow {1 .1}
     proc send {ignore arg} {
             sleep .1
             exp_send -s -- $arg
     }
}
# 2) differing output - Some programs produce different output each time
# they run. The "date" command is an obvious example. Another is
# ftp, if it produces throughput statistics at the end of a file
# transfer. If this causes a problem, delete these patterns or replace
# them with wildcards. An alternative is to use the -p flag (for
# "prompt") which makes Expect only look for the last line of output
# (i.e., the prompt). The -P flag allows you to define a character to
# toggle this mode off and on.
# Read the man page for more info.
# -Don
set timeout -1
spawn $env(SHELL)
match_max 100000
expect -exact "\]0;root@yogi:/scripts\[root@yogi scripts\]# "
send -- "ftp booboo\r"
expect -exact "ftp booboo\r
Connected to booboo.\r
220 (vsFTPd 1.2.1)\r
530 Please login with USER and PASS.\r
530 Please login with USER and PASS.\r
KERBEROS_V4 rejected as an authentication type\r
Name (booboo:root): "
send -- "ftpadmin\r"
expect -exact "ftpadmin\r
331 Please specify the password.\r
```

```
Listing 6-16 (continued)
```

```
Password: "
send -- "abc1234\r"
expect -exact "\r
230 Login successful.\r
Remote system type is UNIX.\r
Using binary mode to transfer files.\r
ftp> "
send -- "ls\r"
expect -exact "ls\r
227 Entering Passive Mode (192,168,37,43,182,234)\r
150 Here comes the directory listing.\r
-rw-r--r-- 1 4465 4465 29187 Sep 06 21:52
   Linux_booboo_cronlistings.txt\r
-rwxr-xr-x 1 4465 4465 1238 Sep 06 19:51 cron_capture\r
-rwxr-xr-x 1 4465 4465
                                       1230 Sep 06 20:23 cron_capture-LINUX\r

      -rw-r--r-
      1
      0
      1250
      Sep 00
      20.25
      Clon_capture HN0x(1

      -rw-rw-r--
      1
      0
      84
      Apr 03
      2007
      drinfo.sh\r

      -rw-rw-rw-r--
      1
      4465
      4465
      68535
      Sep 07
      20:40
      get_AIX_sysinfo.log\r

      -rw-rw-r--
      1
      4465
      4465
      205
      Sep 07
      15:48
      get_dr_info.ksh\r

-rw-rw-r-- 1 4465 4465 10933 Sep 07 20:35 get_sysinfo2.log\r
-rw-r--r-- 1 0 0
                                        666 Apr 03 2007 getall.sh\r
                                       256 Apr 03 2007 imp.sh\r
323 Apr 03 2007 pb_getnet.sh\r
167 Apr 03 2007 pb_getvg.sh\r
618 Apr 03 2007 putall.sh\r
-rw-r--r-- 1 0 0
                          0
-rw-r--r-- 1 0
-rw-r--r-- 1 0
                          0
                        0
-rw-r--r-- 1 0
                          0
                                    10240 Sep 07 15:50 scripts.tar\r
-rw-r--r-- 1 0
-rw-r--r-- 1 0 0
                                          32 Apr 03 2007 um.sh\r
226 Directory send OK.\r
ftp> "
send -- "get getall.sh\r"
expect -exact "get getall.sh\r
local: getall.sh remote: getall.sh\r
227 Entering Passive Mode (192,168,37,43,52,152)\r
150 Opening BINARY mode data connection for getall.sh (666 bytes).\r
226 File send OK.\r
666 bytes received in 0.064 seconds (10 Kbytes/s)\r
ftp> "
send -- "bye\r"
expect -exact "bye\r
221 Goodbye.\r
\]root@yogi:/scripts\[root@yogi scripts\]# "
send -- ""
expect eof
```

Listing 6-16 (continued)

This is the resulting expect script that the autoexpect program created. As you can see, autoexpect adds a lot of extra stuff. One of the key things I've learned writing expect scripts, without using autoexpect, is to *not expect too much*. If you look closely at Listing 6-16, you will see expect -exact "something"\r. The -exact part of

that statement is what will get you sometimes. If a space is in the wrong place, you can have a problem. That's why I say don't expect too much. Most of the time, you want to leave off -exact when specifying expected output.

Listing 6-17 shows the shell script in action.

```
[root@yogi scripts\]# ftp-ls-getfile.exp
ftp booboo
Connected to booboo.
220 (vsFTPd 1.2.1)
530 Please login with USER and PASS.
530 Please login with USER and PASS.
KERBEROS_V4 rejected as an authentication type
Name (booboo:root): ftpadmin
ftpadmin
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls
227 Entering Passive Mode (192,168,37,43,182,234)
150 Here comes the directory listing.
-rw-r--r-- 1 0 0 666 Apr 03 2007 getall.sh

      -rw-r--r-
      1 0
      0
      0000 Apr 03
      2007 getallish

      -rw-r--r-
      1 0
      0
      256 Apr 03
      2007 imp.sh

      -rw-r--r-
      1 0
      0
      323 Apr 03
      2007 pb_getnet.sh

      -rw-r--r-
      1 0
      0
      167 Apr 03
      2007 pb_getvg.sh

      -rw-r--r-
      1 0
      0
      618 Apr 03
      2007 putall.sh

      -rw-r--r-
      1 0
      0
      322 Apr 03
      2007 um.sh

                                              256 Apr 03 2007 imp.sh
226 Directory send OK.
ftp> get getall.sh
get getall.sh
local: getall.sh remote: getall.sh
227 Entering Passive Mode (192,168,37,43,52,152)
150 Opening BINARY mode data connection for getall.sh (666 bytes).
226 File send OK.
666 bytes received in 0.064 seconds (10 Kbytes/s)
ftp> bye
bye
221 Goodbye.
[root@yogi scripts\]#
```

Listing 6-17 expect script ftp-ls-getfile.exp in action

Expect is a programming language that you just have to play around with to learn. Chapter 8, Automating Interactive Programs with Expect and Autoexpect," goes into expect in more detail, but for now, study the expect man page (**man expect**).

Other Things to Consider

This set of shell scripts is very useful to a lot of businesses, but you will need to tailor the shell scripts to fit your environment. Some options that you may want to consider are listed in the following sections.

Use Command-Line Switches to Control Execution

By using getopts to parse command-line options you can modify these shell scripts to have all of the variables assigned on the command line with switches and switch arguments. This modification can allow you to specify the target host, the local and remote working directories, and the file(s) to act on. We have used getopts a lot in this book, so look at some of the other chapters that use getopts to parse the command-line switches and try making some modifications as an exercise.

Keep a Log of Activity

It is a very good idea to keep a log of each connection and check the return codes. If you use the ftp switch -v you will have a detailed account of the connection activity of each transaction. Remember to add a date stamp to each log entry, and also remember to trim the log file periodically so the filesystem does not fill up.

Add a Debug Mode to the Scripts

If a connection fails you could put the script into debug mode by adding a function called debug. In this function the first thing to do is to ping the remote machine to see if it is reachable. If the machine is not reachable by pinging, attempting to ftp to the remote node is useless.

You can also issue the ftp command with the debug option turned on, specified by the -d switch. For more information on FTP debug mode, see the man pages for the ftp command (man ftp).

Reading a Password into a Shell Script

There may come a time when you write a shell script where you want a user to enter a password. I am not talking about a UNIX login or an FTP session; I am talking about a password that the shell script requires.

Any time you read in a password, you want to suppress echoing the password the user types to the screen. This is a little trick my technical editor, John Kennedy, gave me. The idea is to save the current tty definition, turn off the echo response to the screen, read in the user's password, and finally restore the user's tty device:

```
oldSTTY=$( stty --save )
stty -echo
read -p "Password: " PASSWD; echo
stty $oldSTTY
```

The password the user entered is stored in the variable PASSWD.

Summary

This chapter is meant to form a basis for creating larger shell scripts that require the transfer of files between machines. The set of shell scripts presented in this chapter can be modified or made into functions to suit your needs. There are too many variables for the use of FTP to follow each path to its logical end, so in this case you get the building blocks. If you have trouble with a shell script, always try to do the same thing the shell script is doing, except do it on the command line. Usually you can find where the problem is very quickly. If you cannot reach a remote node, try to ping the machine. If you cannot ping the machine, network connectivity or name resolution is the problem.

In the next chapter we are moving on to replicating data between machines using rsync. I hope you found this chapter useful, and I'll see you in the next chapter!

Lab Assignments

- 1. Use autoexpect to create an expect script to ftp a file to a remote machine. Ensure the new expect script works, and then edit the script and remove everything that is not required. This expect script should be no more than 10 lines of code.
- 2. Modify the put_ftp_files_pw_var.ksh shell script in Listing 6-13 to use getopts to parse command-line switches -1, local file, and -r, remote file. Both -1 and -r switches require a string value.

The script syntax will be

put_ftp_files_pw_var.ksh -l LocalFile -r RemoteFile

CHAPTER 7

Using rsync to Efficiently Replicate Data

We use *rsync* as a faster option for copy (cp), remote copy (rcp), secure copy (scp), file transfer protocol (ftp), and secure FTP (sftp). We can use rsync to copy local files as well as to copy files between the local machine and a remote machine. However, rsync does *not* support copying files between two remote machines. The reason that rsync is faster than cp, rcp, and scp is that it uses the rsync *remote-update protocol*, which greatly speeds up file transfers when the file already exists on the destination server, for certain file types, and offers the option of compression. The rsync remote-update protocol uses an efficient checksum search algorithm that allows rsync to transfer only the differences between two sets of files, whether locally or remotely, across a network connection. Adding the -z switch to rsync compresses the files, which offers even faster file transfers.

NOTE rsync must be installed on both machines to transfer files remotely.

In this chapter we are going to look at the syntax to transfer files from a local machine to a remote machine, transferring files from a remote machine to the local machine, copying files locally, and using compression. Then we will study a simple, generic shell script to transfer files in archive-compressed mode. The final script is a pretty complicated shell script that I use to move Oracle data from a master database server to two remote Oracle OLTP (online transaction processing) machines that alternate between day 1 data and day 2 data. This will become clear when we study this technique. Let's start with the rsync command syntax.

Syntax

We use rsync in much the same way that we use rcp and scp. All three methods require a source and destination file or directory, and there are command-line switches

that allow us to save file permissions, links, ownership, and so on, as well as to copy directory structures recursively. A few examples are the best way to learn how to use rsync. Let's start with this simple rsync statement:

```
rsync myscript.Bash yogi:/scripts/
```

This would transfer the file myscript.Bash from the current directory on the local machine to the /scripts directory on the yogi server. Now, if the myscript.Bash file already exists on the server yogi, in the /scripts directory, the *rsync remote-update protocol* is used to update the file by sending only the differences between the source and destination files:

```
rsync -avz yogi:/scripts/data /scripts/tmp
```

This rsync statement will recursively transfer all the files and directories in the /scripts/data directory on the yogi machine to the /scripts/tmp directory on the local machine. The -a rsync switch specifies *archive mode*, which preserves the permissions, ownerships, symbolic links, attributes, and so on, and specifies a recursive copy in the transfer. The -z rsync switch specifies compression is to be used in the transfer to reduce the amount of data in the transfer. Note that this example will create a new directory, /scripts/tmp/data, on the local machine. The -v rsync switch specifies verbose mode. Now check out the next rsync example.

```
rsync -avz yogi:/scripts/data/ /scripts/tmp
```

Notice the trailing slash on the source: yogi:/scripts/data/.

This trailing slash changes the behavior of rsync to eliminate creating the additional directory on the destination, as the previous example produced: /scripts/tmp/data. The trailing slash on the *source* side tells rsync to *copy the directory contents*, as opposed to *copy the directory name and all its contents*.

```
rsync -avz /scripts/data /scripts/tmp
rsync -avz /scripts/data/ /scripts/tmp
```

As you can see by these two examples, we can copy files locally as well as remotely. Notice that local file copying does not have a hostname specified by the *hostname*: notation.

Generic rsync Shell Script

A simple, generic shell script for rsync consists of only defining some variables to point to the file and/or directory we want to copy, and a one-line rsync statement. Check out Listing 7-1 and we will cover the details at the end.

```
#!/bin/Bash
#
# SCRIPT: generic_rsync.Bash
# AUTHOR: Randy Michael
# DATE: 11/18/2007
# REV: 1.0
# PURPOSE: This is a generic shell script to copy files
#
        using rsync.
# set -n # Uncomment to check script syntax without execution
# set -x # Uncomment to debug this script
# REV LIST:
#
#
*****
# DEFINE FILES AND VARIABLES HERE
*****
# Define the source and destination files/directories
SOURCE_FL="/scripts/"
DESTIN_FL="booboo:/scripts"
*****
# BEGINNING OF MAIN
****
# Start the rsync copy
rsync -avz "$SOURCE_FL" "$DESTIN_FL"
# End of generic_rsync.Bash
```

Listing 7-1 generic_rsync.Bash script

As you can see, there is not much to this shell script. We define the source and destination files/directories to the SOURCE_FL and DESTIN_FL variables, respectively. Next we use these variables in our rsync statement, as follows:

rsync -avz "\$SOURCE_FL" "\$DESTIN_FL"

This rsync command will recursively transfer all the files and subdirectories in the local /scripts/ directory (notice the trailing slash on the source) to the /scripts directory on the booboo server using compression to reduce the amount of data transferred. Notice the trailing slash avoided creating a second scripts directory on the destination: /scripts/scripts/.

Now let's look at some example shell scripts to replicate data at the directory structure level and filesystem level, respectively.

Replicating Multiple Directories with rsync

Ever need a technique to replicate one or more directory structures to one or more remote machines? If so, this section is your miracle! Using rsync to replicate data is both efficient and reliable to an unbelievable degree when you see how fast the rsync remote-update protocol is when files already exist on the target(s). We have already looked at using rsync to replicate a single directory structure. The only thing we are adding is a loop to rsync multiple directory structures in parallel by starting each rsync session in the background, specified by rsync_command &. As the rsync process progresses, we are also going to monitor each rsync session at the process table level as we give the user very detailed feedback. This feedback is crucial to any long-running process or procedure that has user interaction.

For this script we will need a list file to hold the names of each top-level directory structure we want to replicate. We can name this list file rsync_directory_list.lst for this example. Because we are going to be looping through this list file, the script will expect one directory listed on each line. An example file, rsync_directory_list.lst, is shown in Listing 7-2.

```
# This file is used by the generic_DIR_rsync_copy.Bash
# shell script to define top-level directory structures
# to replicate with rsync to one or more remote servers.
# The script is expecting full pathnames (must begin with
# a forward slash, EX: /directory/path. The may end in a
# forward slash but not required.
#
# Example list:
#
# /data/path01
# /home/myhome
# /scripts
#
/data01
/data02
/data03/
/data04/
/data05
```

Listing 7-2 Example rsync_directory_list.lst file

As we previously studied, to prevent rsync from creating an extra directory on the target, we must specify the directory path with a trailing forward slash on the source — for example, /directory/path/. Notice in Listing 7-2 that only two entries have the trailing forward slash, /data03/ and /data04/. Not to worry. If the trailing forward slash is missing, we will add it in the script.

For this example shell script there are plenty of comments to follow along, so I think it is best if you study the generic_DIR_rsync_copy.Bash shell script in Listing 7-3 in detail and we will study the details at the end.

```
#!/bin/Bash
#
# SCRIPT: generic_DIR_rsync_copy.Bash
# AUTHOR: Randy Michael
# DATE: 1/14/2007
# REV: 1.0
# PLATFORM: Not platform dependent
#
# REQUIRED INSTALLATION: rsync must be
        installed on the source and
#
#
        all target systems
#
# REQUIRED FILE FOR OPERATION:
        The file pointed to by the
#
#
        $DIR_LIST_FILE variable refers
        to the required file listing the
#
#
        top-level directory structures
        that rsync is to replicate. The
#
        DIR_LIST_FILE is defined in the
#
        variables declarations section
#
         of this shell script
#
#
# PURPOSE: This shell script is used to replicate
#
   directory structures to one or more remote
#
     machines using rsync in archive and
#
    compressed mode. This script requires
#
    a file referenced by the variable
#
    DIR_LIST_FILE that lists the top level
#
    of each of the local directory structures
#
    that we want to replicate to one or more
#
    remote machines
#
# set -x # Uncomment to debug this script
#
# set -n # Uncomment to check script syntax
#
       # without execution. Remember to put
       # the comment back into the script or it
        # will never execute.
# DEFINE FILES AND VARIABLES HERE
*******
```

Listing 7-3 generic_DIR_rsync_copy.Bash shell script

Define the target machines to copy data to. # To specify more than one host enclose the # hostnames in double quotes and put at least # one space between each hostname # EXAMPLE: MACHINE_LIST="fred yogi booboo" MACHINE_LIST="fred" # The DIR_LIST_FILE variable define the shell script # required file listing the directory structures # to replicate with rsync. DIR_LIST_FILE="rsync_directory_list.lst" # Define the directory containing all of the # output files this shell script will produce WORK_DIR=/usr/local/bin # Define the rsync script log file LOGFILE="generic_DIR_rsync_copy.log" # Query the system for the hostname THIS_HOST=\$(hostname) # Query the system for the UNIX flavor OS=\$(uname) **** # DEFINE FUNCTIONS HERE elapsed_time () { SEC=\$1 ((SEC < 60)) && echo -e "[Elapsed time: \$SEC seconds]\c" ((SEC >= 60 && SEC < 3600)) && echo -e "[Elapsed time: $\$ \$((SEC / 60)) min \$((SEC % 60)) sec]\c" ((SEC > 3600)) && echo -e "[Elapsed time: \$((SEC / 3600)) \ hr \$(((SEC % 3600) / 60)) min \$(((SEC % 3600) % 60)) \ sec]\c" }

```
verify_copy ()
{
# set -x
MYLOGFILE=${WORK_DIR}/verify_rsync_copy.log
>$MYLOGFILE
ERROR=0
# Enclose this loop so we can redirect output to the log file
# with one assignment at the bottom of the function
# Put a header for the verification log file
echo -e "\nRsync copy verification starting between $THIS_HOST \
and machine(s) $MACHINE_LIST\n\n" >$MYLOGFILE
for M in $MACHINE_LIST
do
   for DS in $(cat $DIR_LIST_FILE)
   do
      LS_FILES=$(find $DS -type f)
      for FL in $LS_FILES
       do
          LOC_FS=$(ls -1 $FL | awk '{print $5}' 2>&1)
          REM_FS=$(ssh $M ls -1 $FL | awk '{print $5}' 2>&1)
          echo "Checking File: $FL"
          echo -e "Local $THIS_HOST size:\t$LOC_FS"
          echo -e "Remote $M size:\t$REM_FS"
          if [ "$LOC_FS" -ne "$REM_FS" ]
          then
             echo "ERROR: File size mismatch between $THIS_HOST and $M"
             echo "File is: $FL"
             ERROR=1
          fi
      done
   done
done
if (( ERROR != 0 ))
then
   # Record the failure in the log file
   echo -e "\n\nRSYNC ERROR: $THIS_HOST Rsync copy failed...file\
   size mismatch...\n\n" | tee -a $MYLOGFILE
    echo -e "\nERROR: Rsync copy Failed!"
```

Listing 7-3 (continued)

```
echo -e "\n\nCheck log file: $MYLOGFILE\n\n"
   echo -e "\n...Exiting...\n"
    # Send email notification with file size log
    mailx -r from_someone@somewhere.??? -s "Rsync copy verification \
#failed $THIS_HOST -- File size mismatch" \
    somebody@fsf.com < $MYLOGFILE
#
   sleep 2 # Give a couple of seconds to send the email
   exit 3
else
   echo -e "\nSUCCESS: Rsync copy completed successfully..."
   echo -e "\nAll file sizes match...\n"
fi
} | tee -a $MYLOGFILE
}
*****
# BEGINNING OF MAIN
*****
# We enclose the entire MAIN in curly braces
# so that all output of the script is logged
# in the $LOGFILE with a single output
# redirection
{
# Save the last logfile with a .yesterday filename extension
cp -f $LOGFILE ${LOGFILE}.yesterday 2>/dev/null
# Zero out the $LOGFILE to start a new file
>$LOGFILE
# Start all of the rsync copy sessions at once by looping
# through each of the mount points and issuing an rsync
# command in the background, saving the PID of the
# background process, and incrementing a counter.
echo -e "\nStarting a bunch of rsync sessions...$(date)\n"
# Initialize the rsync session counter, TOTAL
# to zero
```

```
TOTAL=0
# Loop through each machine in the $MACHINE_LIST
for M in $MACHINE LIST
do
   # Loop through all of the directory structures
   # listed in the $DIR_LIST_FILE and start an rsync
   # session in the background for each directory
   for DS in $(cat $DIR_LIST_FILE)
   do
      # Ensure each directory structure has a trailing
      # forward slash to ensure an extra directory is
      # not created on the target
      if ! $(echo "$DS" | grep -q '/$')
      then
           # If the directory structure does not
           # have a trailing forward slash, add it
           DS="${DS}/"
      fi
      # Start a timed rsync session in the background
      time rsync -avz ${DS} ${M}:${DS} 2>&1 &
      # Keep a running total of the number of rsync
      # sessions started
      ((TOTAL = TOTAL + 1))
   done
done
# Sleep a 10 seconds before monitoring processes
sleep 10
# Query the process table for the number of rsync sessions
# that continue executing and store that value in the
# REM_SESSIONS variablei
# Create a list of directory structures for an egrep list
EGREP_LIST=
               # Initialize to null
while read DS
do
```

```
if [ -z $EGREP_LIST ]
   then
       EGREP_LIST="$DS"
   else
       EGREP_LIST=" | $DS"
   fi
done < $DIR_LIST_FILE
REM_SESSIONS=$(ps x | grep "rsync -avz" | egrep "$EGREP_LIST" \
               | grep -v grep | awk '{print $1}' | wc -1)
# Give some feedback to the end user.
if (( REM_SESSIONS > 0 ))
then
   echo -e "\n$REM_SESSIONS of $TOTAL rsync copy sessions require \
   further updating ... "
   echo -e "\nProcessing rsync copies from THIS_HOST to both \
   $MACHINE_LIST machines"
   echo -e "\nPlease be patient, this process may a very long time...\n"
   echo -e "Rsync is running [Start time: $(date)]\c"
else
   echo -e "\nAll files appear to be in sync...verifying file sizes...\
   Please wait...\n"
fi
# While the rsync processes are executing this loop
# will place a . (dot) every 60 seconds as feedback
# to the end user that the background rsync copy
# processes are still executing. When the remaining
# rsync sessions are less than the total number of
# sessions this loop will give feedback as to the
# number of remaining rsync session, as well as
# the elapsed time of the processing, every 5 minutes
# Set the shell variable SECONDS to 10 to make up for the
# time we slept while the rsync sessions started
SECONDS=10
# Initialize the minute counter to zero minutes
MIN_COUNTER=0
# Loop until all of the rsync sessions have completed locally
until (( REM_SESSIONS == 0 ))
do
```

```
sleep 60
    echo -e ".\c"
    REM_SESSIONS=$(ps x | grep "rsync -avz" | egrep "$EGREP_LIST" \
                   | grep -v grep | awk '{print $1}' | wc -1)
    if (( REM_SESSIONS < TOTAL ))
    then
        (( MIN_COUNTER = MIN_COUNTER + 1 ))
        if ((MIN COUNTER >= ((TOTAL / 2)))
        then
          MIN_COUNTER=0
           echo -e "\n$REM_SESSIONS of $TOTAL rsync sessions \
           remaining $(elapsed_time $SECONDS)\c"
           if (( REM_SESSIONS <= $(( TOTAL / 4 )) ))
           then
              echo -e "\nRemaining rsync sessions include:\n"
              ps x | grep "rsync -avz" | egrep "$EGREP_LIST" \
                     grep -v grep
              echo
           fi
        fi
    fi
done
echo -e "\n...Local rsync processing completed on ${THIS_HOST}...$(date)"
echo -e "\n...Checking remote target machine(s): ${MACHINE_LIST}..."
# Just because the local rsync processes have completed does
# not mean that the remote rsync processes have completed
# execution. This loop verifies that all of the remote
# rsync processes completed execution. The end user will
# receive feedback if any of the remote processes are running.
for M in $MACHINE_LIST
do
   for DS in $(cat $DIR_LIST_FILE)
   do
       RPID=$(ssh $M ps x | grep rsync | grep $DS | grep -v grep \
       | awk'{print $1}')
       until [ -z "$RPID" ]
       do
           echo "rsync is processing ${MP} on ${M}...
   sleeping one minute..."
           sleep 60
           RPID=$(ssh $M ps x | grep rsync | grep $DS | grep -v grep \
                | awk '{print $1}')
       done
```

```
Listing 7-3 (continued)
```

We begin the generic_DIR_rsync_copy.Bash shell script in Listing 7-3 by declaring some files and variables. The MACHINE_LIST contains one or more remote machines to replicate data to. If more than one target machine is listed, the list must be enclosed in quotes and the hostnames must be separated by at least one blank space — *no commas*! The WORK_DIR is the directory where the files this shell script creates will be located. Next we define our directory list file with the variable DIR_LIST_FILE. This variable points to our directory list file rsync_directory_list.lst. This script assumes this directory list file is located in the same directory as the shell script. The last assignments define the name of the log file, hostname, and the UNIX flavor.

We define two functions in the generic_DIR_rsync_copy.Bash shell script in Listing 7-3. The elapsed_time function converts seconds into hours, minutes, and seconds, and the verify_copy function verifies the rsync copy by ensuring the file sizes match locally and remotely. We will come back to the verification process; for now, though, let's start at the BEGINNING OF MAIN and look closer at the details.

The first thing we do is to open a curly brace, {. There is a closing curly brace at the end of the shell script. We enclosed the entire main body of the shell script so we can log all the activity as the shell script executes with a single output redirection to pipe the output to tee -a \$LOGILE. An example is shown here:

} 2>&1 | tee -a \$LOGFILE

Now any output sent to standard output (stdout) or standard error (stderr) will be displayed on the screen and logged to the \$LOGFILE at the same time. This technique saves a lot of >>\$LOGFILE output redirections and makes the script a lot cleaner and easier to read.

The next thing to do is to save a copy of the last log file. Ideally, at least a week's worth of log files should be kept to troubleshoot any rsync problems that come up. I will show a set of cron table entries that can be used to keep seven days of log files. This script makes a copy of the log file with a .yesterday filename extension, cp \$LOGFILE \${LOGFILE}.yesterday.

Now we are ready to kick off a bunch of rsync sessions in the background. We start by looping through the \$MACHINE_LIST, and then start a second nested loop that starts looping through the \$DIR_LIST_FILE, as follows:

```
# Loop through each machine in the $MACHINE_LIST
for M in $MACHINE_LIST
do
    # Loop through all of the directory structures
    # listed in the $DIR_LIST_FILE and start an rsync
    # session in the background for each directory
    for DS in $(cat $DIR_LIST_FILE)
    do
```

As we loop through all the directory structures, we next test for a trailing forward slash. If the forward slash is missing, we add it, as shown here:

To test for a trailing forward slash, we use the regular expression /\$. The dollar sign (\$) specifies *ends with*, so this regular expression means *ends with* /. We echo the \$DS variable and pipe the output to grep -q '/\$'. The -q switch specifies quiet mode, so we are looking for the return code in the if statement. The if is negated with an exclamation point, !, specifying that if the forward slash does not exist, we create it with the DS='\${DS}/' variable assignment.

Now we start an rsync session for the directory structure we are currently processing in the loop with the following rsync command:

```
time rsync -avz ${DS} ${M}:${DS} 2>&1 &
```

Notice that we start this rsync session with a time command. This is not the operating system's /usr/bin/time command, but the shell's built-in time command. The shell version of time has a more granular detail in the timing results.

The rsync command is specified with the -a, -v, and -z switches:

- The -a switch copies in archive mode, preserving all file and directory characteristics.
- The -v switch is verbose mode for a more detailed output.
- The -z switch specifies using compression during the copy process.

We use the variable \${DS} to point to the directory structure to replicate, and the \${M} points to the remote target machine. We end the rsync command with 2>&1 &. This sends standard error to standard output, and the final & executes the rsync session in the background. As we start each rsync session, we increment the TOTAL counter so that we can keep track of how many sessions we started.

After all the rsync sessions have started execution in the background, we sleep for 10 seconds to allow time for rsync to make an initial check to see if anything needs to be updated using rsync's remote-update protocol. After 10 seconds, we check the process table for any remaining rsync sessions. To run the ps command, we need a pattern to grep on. We first grep on the rsync command statement, rsync -avz. We also have a list of directory structures we need to grep on to find each rsync session. To do this step, we need to build an egrep command list of the directory structures. The following loop builds the EGREP_LIST variable:

```
EGREP_LIST= # Initialize to null
while read DS
do
    if [ -z $EGREP_LIST ]
    then
       EGREP_LIST="$DS"
    else
       EGREP_LIST="|$DS"
    fi
done < $DIR_LIST_FILE
```

The EGREP_LIST variable is first initialized to NULL. Next we feed the while loop from the bottom with input redirection after the done loop terminator. The if statement begins by checking if the EGREP_LIST variable is zero length, or NULL. If so, this is the first assignment and we assign the current \$DS to EGREP_LIST. For all subsequent assignments, a pipe is added in front of the \$DS assignment, |\$DS. The result is a pipe-delimited list, as shown here:

```
/data01 | /data02 | /data03 | /data04 | /data05
```

Now we can assemble the ps command statement to query the system for remaining rsync sessions. The full ps statement is shown here:

```
REM_SESSIONS=$(ps x | grep "rsync -avz" | egrep "$EGREP_LIST" \
| grep -v grep | awk '{print $1}' | wc -1)
```

If we find any rsync sessions running, we enter a monitoring loop until all the local rsync sessions have completed. During the execution we continuously give the user feedback. While the \$REM_SESSIONS variable is equal to the \$TOTAL sessions variable, we display a dot on the screen every 60 seconds. When the \$REM_SESSIONS value drops below the \$TOTAL value, we start updating the user every 5 minutes, 300 seconds, with the number of remaining sessions and the elapsed time. When the REM_SESSIONS reaches one-fourth of the TOTAL sessions, we start listing the remaining session residing in the process table. Users are always wondering "What's taking so long?" Well, this script answers that question.

The generic_DIR_rsync_copy. Bash shell script in Listing 7-3 is shown in action in Listings 7-4 and 7-5. The initial copy can take quite a while to replicate because the files do not yet exist on the target(s). However, after the files are copied the first time, the rsync remote-update protocol allows for much faster replication. Listing 7-4 shows a first-time rsync copy session.

```
Starting a bunch of rsync sessions...Sun Jan 13 01:45:16 EST 2008
building file list ... done
building file list ... done
./
./
random_file.out0
random_file.out2
building file list ... done
./
building file list ... done
random_file.out4
./
random_file.out6
building file list ... done
./
random_file.out7
5 of 5 rsync copy sessions require further updating ...
Processing rsync copies from booboo to both fred machines
Please be patient, this process may a very long time ...
Rsync is running [Start time: Sun Jan 13 01:45:26 EST 2008]...
sent 1244739 bytes received 48 bytes 6334.79 bytes/sec
total size is 261428695 speedup is 210.02
```

Listing 7-4 generic_DIR_rsync_copy.Bash script when files do not exist on target

```
real 3m17.007s
user 0m5.817s
sys 0m0.682s
.random_file.out5
random_file.out3
random_file.out1
4 of 5 rsync sessions remaining [Elapsed time: 5 min 10 sec]..
4 of 5 rsync sessions remaining [Elapsed time: 7 min 10 sec]
sent 3153396 bytes received 70 bytes 7274.43 bytes/sec
total size is 662550995 speedup is 210.10
real 7m13.186s
user 0m14.756s
sys 0m1.658s
sent 3153396 bytes received 70 bytes 7241.02 bytes/sec
total size is 662550995 speedup is 210.10
real 7m15.980s
user 0m14.737s
sys 0m1.693s
sent 3145092 bytes received 48 bytes 7028.25 bytes/sec
total size is 661235625 speedup is 210.24
real 7m26.847s
user 0m14.768s
sys 0m1.684s
sent 4316924 bytes received 70 bytes 9117.20 bytes/sec
total size is 907336545 speedup is 210.18
real 7m53.409s
user 0m20.179s
sys 0m2.270s
...Local rsync processing completed on booboo...Sun Jan 13 01:53:26
  EST 2008
... Checking remote target machine(s): fred...
...Remote rsync processing completed Sun Jan 13 01:53:31 EST 2008
Checking File: /data01/random_file.out1
Local booboo size:
                   446004385
Remote fred size: 446004385
Checking File: /data01/random_file.out0
```

```
Local booboo size: 461332160
Remote fred size: 461332160
Checking File: /data02/random_file.out2
Local booboo size: 401122300
Remote fred size: 401122300
Checking File: /data02/random_file.out3
Local booboo size: 261428695
Remote fred size: 261428695
Checking File: /data03/random_file.out5
Local booboo size: 261428695
Remote fred size: 261428695
Checking File: /data03/random_file.out4
Local booboo size: 401122300
Remote fred size: 401122300
Checking File: /data04/random_file.out6
Local booboo size: 261428695
Remote fred size: 261428695
Checking File: /data05/random_file.out7
Local booboo size: 661235625
Remote fred size: 661235625
SUCCESS: Rsync copy completed successfully...
All file sizes match...
Rsync copy completed Sun Jan 13 01:53:38 EST 2008
Elapsed time: [Elapsed time: 8 min 22 sec]
```

As we can see, it took more than eight minutes to complete the initial rsync copy. Listing 7-5 shows an rsync replication process after the files exist on the target(s). I know this is a hard log file to read and understand. Because we are starting a bunch of rsync sessions in the background and because all the output, both stderr and stdout, is going to the screen and being logged, we get a lot of output that varies and is fluid as the rsync sessions are executing. When the rsync copy is running, we see dot files like .random_file.out5. These dot files are the rsync temporary files used by the rsync remote-update protocol and go away when the replication completes. Listing 7-5 shows the same rsync process after the files exist on the target(s).

```
Starting a bunch of rsync sessions...Sun Jan 13 02:08:31 EST 2008
building file list ... done
building file list ... done
```

Listing 7-5 generic_DIR_rsync_copy.Bash script with files existing at target

```
sent 74 bytes received 20 bytes 37.60 bytes/sec
total size is 261428695 speedup is 2781156.33
real 0m2.148s
user 0m0.014s
sys 0m0.006s
building file list ... done
sent 86 bytes received 20 bytes 42.40 bytes/sec
total size is 907336545 speedup is 8559778.73
real 0m2.197s
user 0m0.012s
sys 0m0.007s
sent 74 bytes received 20 bytes 37.60 bytes/sec
total size is 661235625 speedup is 7034421.54
real 0m2.193s
user 0m0.013s
sys 0m0.008s
building file list ... done
sent 86 bytes received 20 bytes 30.29 bytes/sec
total size is 662550995 speedup is 6250481.08
real 0m2.527s
user 0m0.014s
sys 0m0.008s
building file list ... done
sent 86 bytes received 20 bytes 30.29 bytes/sec
total size is 923979690 speedup is 8716789.53
real 0m2.568s
user 0m0.013s
sys 0m0.009s
All files appear to be in sync...verifying file sizes...Please wait...
...Local rsync processing completed on booboo...Sun Jan 13 02:08:41
  EST 2008
... Checking remote target machine(s): fred...
... Remote rsync processing completed Sun Jan 13 02:08:45 EST 2008
```

Listing 7-5 (continued)

```
Checking File: /data01/random_file.out1
Local booboo size: 446004385
Remote fred size: 446004385
Checking File: /data01/random_file.out0
Local booboo size: 461332160
Remote fred size: 461332160
Checking File: /data02/random_file.out2
Local booboo size: 401122300
Remote fred size: 401122300
Checking File: /data02/random_file.out3
Local booboo size: 261428695
Remote fred size: 261428695
Checking File: /data03/random_file.out5
Local booboo size: 261428695
Remote fred size: 261428695
Checking File: /data03/random_file.out4
Local booboo size: 662550995
Remote fred size: 662550995
Checking File: /data04/random_file.out6
Local booboo size: 261428695
Remote fred size: 261428695
Checking File: /data05/random_file.out7
Local booboo size: 661235625
Remote fred size: 661235625
SUCCESS: Rsync copy completed successfully ...
All file sizes match...
Rsync copy completed Sun Jan 13 02:08:52 EST 2008
Elapsed time: [Elapsed time: 21 seconds]
```

Wow, what a difference! The rsync remote-update protocol found the targets to be in sync with the source files and completed the update in 21 seconds. As you can see, rsync is a very nice tool for replicating data. Now let's look at replicating data at the filesystems level.

Replicating Multiple Filesystems with rsync

In this section we are going to create a generic shell script that will replicate one or more filesystems to one or more remote servers. This time we are going to match the filesystems to replicate with a regular expression from the df command output. Because we are querying the system for the filesystems to replicate, we do not have to worry about a list file to specify what to replicate.

For this example we are again going to replicate the /data01 through /data05 directory structures, but this time they are filesystems, not just directories. The regular expression to match these filesystems is /data0[1-5]. For ease of updating, this regular expression is assigned to the FS_PATTERN variable. We use this regular expression in the following df command statement:

```
# df | grep "$FS_PATTERN" | awk '{print $6}'
/data01
/data02
/data03
/data04
/data05
```

With the filesystems known, we loop through each one and start a bunch of rsync sessions in the background, the same way we did in Listing 7-3. Just as the previous shell had a lot of comments describing the steps, the generic_FS_rsync_copy.Bash shell script is easy to follow through. Study Listing 7-6 in detail and we will cover it further at the end.

```
#!/bin/Bash
# SCRIPT: generic_FS_rsync_copy.Bash
# AUTHOR: Randy Michael
# DATE: 1/14/2008
# REV: 1.2.1
# PURPOSE: This script is used to replicate
#
#
# EXIT CODES:
#
            0 ==> Normal execution.
            2 ==> Remote host is not pingable.
#
#
            3 ==> Copy verification failed. One or
#
                 more local files are not the same
                  size as the remote files.
#
            4 ==> No machines are defined to copy
#
#
                  data to.
#
            5 ==> Exited on a trapped signal.
#
# set -x # Uncomment to debug this script
#
# set -n # Uncomment to check the script's syntax
        # without any execution. Do not forget to
       # recomment this line!
*****
# REVISION LIST:
```

Listing 7-6 generic_FS_rsync_copy.Bash script

```
*****
# IF YOU MODIFY THIS SCRIPT, DOCUMENT THE CHANGE(S) !!!
*****
# Revised by:
# Revision Date:
# Revision:
****
# DEFINE FILES AND GLOBAL VARIABLES HERE
*****
# Define the target machines to copy data to.
# To specify more than one host enclose the
# hostnames in double quotes and put at least
# one space between each hostname
# EXAMPLE: MACHINE_LIST="fred yogi booboo"
MACHINE_LIST="fred"
# The FS_PATTERN variable defines the regular expression
# matching the filesystems we want to replicate with rsync.
FS_PATTERN="/data0[1-5]"
# Add /usr/local/bin to the PATH
export PATH=$PATH:/usr/local/bin
# Define the directory containing all of the
# output files this shell script will produce
WORK_DIR=/usr/local/bin
# Define the rsync script log file
LOGFILE=${WORK_DIR}/generic_FS_rsync_copy.log
# Capture the shell script file name
THIS_SCRIPT=$(basename $0)
# Initialize the background process ID list
# variable to NULL
```

Listing 7-6 (continued)

```
BG_PID_LIST=
# Define the file containing a list of files to verify
# the sizes match locally and remotely
LS_FILES=ls_output.dat
>$LS_FILES
# Initialize the TOTAL variable to 0
TOTAL=0
# Query the system for the hostname
THIS_HOST=$(hostname)
# Query the system for the UNIX flavor
THIS_OS=$(uname)
*****
# DEFINE FUNCTIONS HERE
*****
verify_copy ()
{
# set -x
MYLOGFILE=${WORK_DIR}/verify_rsync_copy_day.log
>$MYLOGFILE
ERROR=0
# Enclose this loop so we can redirect output to the log file
# with one assignment at the bottom of the function
{
# Put a header for the verification log file
echo -e "\nRsync copy verification between THIS_HOST and machine(s) \
$MACHINE_LIST\n\n" >> $MYLOGFILE
# Loop through each machine in the $MACHINE_LIST
for M in $MACHINE_LIST
do
   # Loop through each filesystem that matches the regular
   # regular expression point to by "$FS_PATTERN". It is
```

Listing 7-6 (continued)

```
# important that this variable is enclosed within double
   # quotes. Also note the column for the mount point is 6;
   # in AIX this should be changed to 7.
   for MP in $(df | grep "$FS_PATTERN" | awk '{print $6}')
   do
       # For each filesystem mount point, $MP, execute a
       # find command to find all files in the filesystem.
       # and store them in the $LS_FILES file. We will use this
       # file list to verify the file sizes on the remote
       # machines match the file sizes on the local machine.
       find $MP -type f > $LS_FILES
       # Loop through the file list
       for FL in $(cat $LS_FILES)
       do
           # Find the local file size
           LOC_FS=$(1s -1 $FL | awk '{print $5}' 2>&1)
           # Find the remote file size using a remote shell command
           REM_FS=$(rsh $M ls -1 $FL | awk '{print $5}' 2>&1)
           echo "Checking File: $FL"
           echo -e "Local $THIS_HOST size:\t$LOC_FS"
           echo -e "Remote $M size:\t$REM_FS"
           # Ensure the file sizes match
           if [ "$LOC_FS" -ne "$REM_FS" ]
           then
              echo "ERROR: File size mismatch between $THIS_HOST and $M"
              echo "File is: $FL"
              ERROR=1
           fi
       done
   done
done
if (( ERROR != 0 ))
then
    # Record the failure in the log file
    echo -e "\n\nRSYNC ERROR: $THIS_HOST Rsync copy failed...file
    size mismatch...\n\n" | tee -a $MYLOGFILE
    echo -e "\nERROR: Rsync copy Failed!"
    echo -e "\n\nCheck log file: $MYLOGFILE\n\n"
    echo -e "\n...Exiting...\n"
    return 3
```

```
exit 3
else
   echo -e "\nSUCCESS: Rsync copy completed successfully..."
   echo -e "\nAll file sizes match...\n"
fi
} | 2>&1 tee -a $MYLOGFILE
}
*****
elapsed_time ()
{
SEC=$1
(( SEC < 60 )) && echo -e "[Elapsed time: $SEC seconds]\c"
(( SEC >= 60 && SEC < 3600 )) && echo -e "[Elapsed time:
  $(( SEC / 60 )) min $(( SEC % 60 )) sec]\c"
(( SEC > 3600 )) && echo -e "[Elapsed time: $(( SEC / 3600 ))
  hr $(( (SEC % 3600) / 60 )) min $(( (SEC % 3600) % 60 )) sec]\c"
}
*****
# BEGINNING OF MAIN
*****
# Enclose the entire main part of the script in
# curly braces so we can redirect all output of
# the shell script with a single redirection
# at the bottom of the script to the $LOGFILE
{
# Save a copy of the last log file
cp -f $LOGFILE ${LOGFILE}.yesterday
echo -e "\n[[ $THIS_SCRIPT started execution $(date) ]]\n"
# Ensure that target machines are defined
if [ -z "$MACHINE_LIST" ]
then
   echo -e "\nERROR: No machines are defined to copy data to..."
   echo "... Unable to continue... Exiting... "
   exit 4
fi
```

```
# Ensure the remote machines are reachable through
# the network by sending 1 ping.
echo -e "Verifying the target machines are pingable...\n"
for M in $MACHINE LIST
do
    echo "Pinging $M..."
    ping -c1 $M >/dev/null 2>&1
    if (( $? != 0 ))
    then
        echo -e "\nERROR: $M host is not pingable...cannot continue..."
        echo -e "...EXITING...\n"
        exit 2
    else
        echo "Pinging $M succeeded..."
    fi
done
# Start all of the rsync copy sessions at once by looping
# through each of the mount points and issuing an rsync
# command in the background, and incrementing a counter.
echo -e "\nStarting a bunch of rsync sessions...\n"
for M in $MACHINE_LIST
do
   # Loop through each filesystem that matches the regular
   # expression point to by "$FS_PATTERN". It is
   # important that this variable is enclosed within double
   # quotes. Also note the column for the mount point is 6;
   # in AIX this should be changed to 7.
   for MP in $(df | grep "$FS_PATTERN" | awk '{print $6}')
   do
        # Start the rsync session in the background
        time rsync -avz ${MP}/ ${M}:${MP} 2>&1 &
        # Keep a running total of the number of rsync
        # sessions started
        ((TOTAL = TOTAL + 1))
   done
done
# Sleep a few seconds before monitoring processes
sleep 10
```

```
# Find the number of rsync sessions that are still running
REM_SESSIONS=$(ps x | grep "rsync -avz" | grep "$FS_PATTERN" \
               grep -v grep | awk {print $1}' | wc -1)
# Give some feedback to the end user.
if (( REM SESSIONS > 0 ))
then
   echo -e "\n$REM_SESSIONS of $TOTAL rsync copy sessions require \
   further updating ... "
   echo -e "\nProcessing rsync copies from $THIS_HOST to both \
   $MACHINE_LIST machines"
   echo -e "\nPlease be patient, this process may take a very long \
   time...\n"
   echo -e "Rsync is running [Start time: $(date)]\c"
else
   echo -e "\nAll files appear to be in sync...verifying file sizes...\
   Please wait...\n"
fi
# While the rsync processes are executing this loop
# will place a . (dot) every 60 seconds as feedback
# to the end user that the background rsync copy
# processes are still executing. When the remaining
# rsync sessions are less than the total number of
# sessions (normally 36) this loop will give feedback
# as to the number of remaining rsync session, as well
# as the elapsed time of the processing, every 5 minutes..
SECONDS=10
MIN_COUNTER=0
# Loop until all of the local rsync sessions have completed
until (( REM_SESSIONS == 0 ))
do
    # sleep 60 seconds between loop iterations
    sleep 60
    # Display a dot on the screen every 60 seconds
    echo -e ".\c"
    # Find the number of remaining rsync sessions
```

```
Listing 7-6 (continued)
```

```
REM_SESSIONS=$(ps x | grep "rsync -avz" | grep "$FS_PATTERN" \
                   | grep -v grep | '{print $1}'| wc -1)
    # Have any of the sessions completed? If so start giving
    # the user updates every 5 minutes, specifying the number
    # remaining rsync sessions and the elapsed time.
    if (( REM SESSIONS < TOTAL ))
    then
        # Count every 5 minutes
        (( MIN_COUNTER = MIN_COUNTER + 1 ))
        # 5 minutes timed out yet?
        if (( MIN_COUNTER >= 5 ))
        then
           # Reset the minute counter
           MIN_COUNTER=0
           # Update the user with a new progress report
           echo -e "\n$REM_SESSIONS of $TOTAL rsync
           sessions remaining $(elapsed_time $SECONDS)\c"
           # Have three-fourths of the rsync sessions completed?
           if (( REM_SESSIONS <= $(( TOTAL / 4 )) ))
           then
              # Display the list of remaining rsync sessions
              # that continue to run.
              echo -e "\nRemaining rsync sessions include:\n"
              ps x | grep "rsync -avz" | grep "$FS_PATTERN" \
                     grep -v grep
              echo
           fi
        fi
    fi
done
echo -e "\n...Local rsync processing completed on $THIS_HOST...$(date)"
echo -e "\n...Checking remote target machines: $MACHINE_LIST..."
# Just because the local rsync processes have completed does
# not mean that the remote rsync processes have completed
# execution. This loop verifies that all of the remote
# rsync processes completed execution. The end user will
# receive feedback if any of the remote processes are running.
```

```
for M in $MACHINE_LIST
do
   # Loop through each matching filesystem
   for MP in $(df | grep "$FS_PATTERN" | awk '{print $7}')
   do
      # Find the remote process Ids.
      RPID=$(rsh $M ps x | grep rsync | grep $MP | grep -v grep \
             awk '{print $1}')
      # Loop while remote rsync sessions continue to run.
      # NOTE: I have never found remote rsync sessions running
      # after the local sessions have completed. This is just
      # an extra sanity check
      until [ -z "$RPID" ]
      do
          echo "rsync is processing MP on M...
  sleeping one minute..."
          sleep 60
          RPID=$(rsh $M ps x | grep rsync | grep $MP | grep -v grep \
                 | awk '{print $1}')
      done
  done
done \
echo -e "\n...Remote rsync processing completed $(date)\n"
# Verify the copy process
verify_copy
# Check the verify_copy return code
if (( $? != 0 ))
then
   exit 3
fi
echo -e "\nRsync copy completed $(date)"
echo -e "\n[[ $THIS_SCRIPT completed execution $(date) ]]\n"
echo -e "\n[[ Elapsed Time: $(elapsed_time $SECONDS) ]]\n"
} | 2>&1 tee -a $LOGFILE
*****
# END OF SCRIPT
*****
```

We begin the generic_FS_rsync_copy.Bash shell script in Listing 7-6 by defining some files and variables. The MACHINE_LIST is the list of target machines to replicate the data to. More than one machine can be a target. The format this script is expecting is a list of hostnames separated by white space, with the entire list enclosed in double quotes.

The FS_PATTERN is where we define the regular expression that defines the filesystems we want to replicate — for our example, FS_PATTERN="/data0[1-5]".

Before we start the rsync copy, we ping the target machines to verify they are reachable through the network. At this point, we are ready to start the rsync copy process. We use the MACHINE_LIST variable to loop through all the target machines, and then all the filesystems defined by the regular expression shown here:

/data0[1-5]

Once all the rsync sessions have started, we sleep for 10 seconds to allow each of the sessions to start. After this 10-second interval, we scan the process table to identify how many rsync sessions remain executing of the total that started. If all the sessions completed in 10 seconds, all the files are up-to-date and no rsync copy will take place. Otherwise, we give the user feedback as to how many sessions require updating, show the start time, and start a loop to monitor all the rsync sessions every minute. Until the remaining sessions executing is less than the total sessions started, we just print a dot on the screen every minute as feedback that processing continues. As soon as the remaining sessions drop below the \$TOTAL sessions, we print the elapsed time every 5 minutes and give an update on how many sessions remain executing. Then, when the remaining sessions in addition to the elapsed time and number of remaining sessions. Users always want to know "What's taking so long?" Well, this script tells them.

When local rsync processing completes execution on the source rsync copy server, we check the target machines to verify that all remote rsync processing is also complete. I have never found remote rsync sessions executing after the local rsync sessions have completed execution. This is just an extra sanity check. After we verify that all rsync processing has completed both locally and remotely, we verify that the rsync copy was successful by comparing the file sizes on the source with the file sizes on the target machines. If the file sizes match, we have a successful rsync copy.

The generic_FS_rsync_copy.Bash shell script in Listing 7-6 is shown in action in Listings 7-7 and 7-8. The initial copy can take quite a while to replicate because the files do not yet exist on the target(s). However, after the files are copied the first time, the rsync remote-update protocol allows for much faster replication. Listing 7-7 shows a first-time rsync copy session.

```
[[ generic_FS_rsync_copy.Bash started execution Mon Jan 14 15:22:48
    EST 2008 ]]
```

```
Verifying the target machines are pingable...
```

Listing 7-7 generic_FS_rsync_copy.Bash in action performing an initial copy

```
Pinging fred...
Pinging fred succeeded...
Starting a bunch of rsync sessions...
building file list ... building file list ... building file list ...
  building file list ... building file list ... done
done
done
done
done
./
./
./
./
./
random_file.out2
random_file.out4
random_file.out3
random_file.out1
random_file.out5
      5 of 5 rsync copy sessions require further updating...
Processing rsync copies from booboo to both fred machines
Please be patient, this process may take a very long time...
Rsync is running [Start time: Mon Jan 14 15:22:59 EST 2008]...
.....wrote 55772241 bytes read 36 bytes
   25038.60 bytes/sec
total size is 74158800 speedup is 1.33
...wrote 55772241 bytes read 36 bytes 23070.23 bytes/sec
total size is 74158800 speedup is 1.33
. .
      3 of 5 rsync sessions remaining [Elapsed time: 41 min 33 sec]
  wrote 55772241 bytes read 36 bytes 21961.91 bytes/sec
total size is 74158800 speedup is 1.33
. . . . .
      2 of 5 rsync sessions remaining [Elapsed time: 46 min 36 sec].
  wrote 34076139 bytes read 36 bytes 11887.73 bytes/sec
total size is 45309952 speedup is 1.33
.wrote 55772241 bytes read 36 bytes 18737.54 bytes/sec
total size is 74158800 speedup is 1.33
...Local rsync processing completed on booboo...Mon Jan 14 16:12:27
   EST 2008
```

```
Listing 7-7 (continued)
```

```
... Checking remote target machines: fred...
...Remote rsync processing completed Mon Jan 14 16:12:28 EST 2008
Checking File: /data01/random_file.out1
Local booboo size: 74158800
Remote fred size: 74158800
Checking File: /data02/random file.out2
Local booboo size: 74158800
Remote fred size: 74158800
Checking File: /data03/random_file.out3
Local booboo size: 74158800
Remote fred size: 74158800
Checking File: /data04/random_file.out4
Local booboo size: 74158800
Remote fred size: 74158800
Checking File: /data05/random_file.out5
Local booboo size: 45309952
Remote fred size: 45309952
SUCCESS: Rsync copy completed successfully ...
All file sizes match...
Rsync copy completed Mon Jan 14 16:12:29 EST 2008
[[ generic_FS_rsync_copy.Bash completed execution Mon Jan 14 16:12:29
  EST 2008 ]]
[[ Elapsed Time: [Elapsed time: 49 min 40 sec] ]]
```

As we can see, it took almost an hour to complete the initial rsync copy. Listing 7-8 shows an rsync replication process after the files exist on the target(s).

```
[[ generic_FS_rsync_copy.Bash started execution Mon Jan 14 16:13:46
    EST 2008 ]]
Verifying the target machines are pingable...
Pinging fred...
Pinging fred succeeded...
Starting a bunch of rsync sessions...
```

Listing 7-8 generic_FS_rsync_copy.Bash in action when files exist on the target

```
building file list ... building file list ... building file list ...
  building file list ... building file list ... done
done
done
done
done
wrote 133 bytes read 20 bytes 306.00 bytes/sec
total size is 74158800 speedup is 484698.40
wrote 133 bytes read 20 bytes 306.00 bytes/sec
total size is 74158800 speedup is 484698.40
wrote 133 bytes read 20 bytes 306.00 bytes/sec
total size is 74158800 speedup is 484698.40
wrote 133 bytes read 20 bytes 306.00 bytes/sec
total size is 45309952 speedup is 296143.48
wrote 133 bytes read 20 bytes 306.00 bytes/sec
total size is 74158800 speedup is 484698.40
All files appear to be in sync...verifying file sizes...Please wait...
...Local rsync processing completed on booboo...Mon Jan 14 16:13:56
  EST 2008
... Checking remote target machines: fred...
... Remote rsync processing completed Mon Jan 14 16:13:58 EST 2008
Checking File: /data01/random file.out1
Local booboo size:
                   74158800
Remote fred size: 74158800
Checking File: /data02/random_file.out2
Local booboo size: 74158800
Remote fred size: 74158800
Checking File: /data03/random_file.out3
Local booboo size: 74158800
Remote fred size: 74158800
Checking File: /data04/random_file.out4
Local booboo size: 74158800
Remote fred size: 74158800
Checking File: /data05/random_file.out5
Local booboo size: 45309952
Remote fred size: 45309952
SUCCESS: Rsync copy completed successfully...
All file sizes match...
```

Listing 7-8 (continued)

```
Rsync copy completed Mon Jan 14 16:13:59 EST 2008
[[ generic_FS_rsync_copy.Bash completed execution Mon Jan 14 16:13:59
    EST 2008 ]]
[[ Elapsed Time: [Elapsed time: 13 seconds] ]]
```

Again, we can see that after the initial data replication, the rsync remote-update protocol found all the files to be in sync and completed the verification in 13 seconds. rsync is a very nice tool for replicating data.

Replicating an Oracle Database with rsync

Up to this point, we have been able to execute the rsync copy immediately when we type in the command and press Enter. In many situations, we must wait for some event to happen — for example, if we have a requirement to stop an application before we start the rsync copy process, or, as in this particular example, if we need to wait for the database to be put into read-only mode before we start the rsync sessions. This section looks at some techniques to allow different teams to communicate by way of *status files* that indicate when each step is allowed to proceed. Sit back — this is a fun section.

Here is the scenario of the problem we want to solve in this section. We have a master Oracle database on one machine that we want to share with two remote machines, like Oracle RAC allows concurrent access to a single database by multiple servers. The remote machines are used by production accounts to access the database in real time. This is called *online transaction processing* (OLTP). Our task is to refresh the data on multiple OLTP servers every day without interrupting production access to the database. To solve this problem, we came up with a *day 1 data* and *day 2 data* strategy. When we update the data each day, the Oracle DBA Team's script changes a synonym in the database to point to the new current day's data. (We are not covering the DBA scripts here.) Please stick with me; this is not that difficult if you keep following along.

rsync was chosen as the data-replication tool because of its stability, relatively low network load, and ability to handle only file changes, on certain file types. This rsync copy process is part UNIX Systems Administrator (SA) task, and part Oracle DBA task. However, we are covering only the SA scripts in this book. For this method of copying database files to work, the Oracle database must be put into read-only mode so that no changes are made to the .dbf database files until the copy process is complete. To facilitate the communications between the Oracle DBA Team's scripts and the SA Team's scripts, we use flat files to signal when the rsync copy can start, then another file to signal that the rsync copy process has completed execution, and a third file if the rsync copy process fails.

Filesystem Structures

We need to understand the filesystem structures for both the master database server, gamma, which contains the master Oracle data, and the OLTP servers, alpha and bravo, which are used for real-time production-database access. The master database server, gamma, has one set of 18 filesystems that contains the entire database. The two OLTP production access servers, alpha and bravo, have two sets of 18 database filesystems, one set of 18 filesystems for day 1 data, and one set of 18 filesystems for day 2 data. The purpose of the day 1 and day 2 filesystems is to allow for alternating days of fresh data without the need to interrupt production access while the data is refreshed each day. The master database server, gamma, processes a fresh set of data each night, which can take as long as 12 or more hours to complete. Upon completion of overnight processing, an Oracle script, which is started each night via a cron table entry, puts the database in read-only mode, then signals, with a file, for the rsync process to start. The file, readytocopy.txt, contains only the number 1 or 2, indicating the target set of filesystems to copy the master data to. Listings 7-9 and 7-10 show the filesystem layouts on the master database server (gamma) and the OLTP servers (alpha and bravo), respectively.

/dev/datadmlv1	102760448	59735296	42%	6	1%	/dba/oradata_dm_01
/dev/datadmlv2	102760448	59735288	42%	6	1%	/dba/oradata_dm_02
/dev/datadmlv3	102760448	59735280	42%	6	1%	/dba/oradata_dm_03
/dev/datadmlv4	102760448	59735272	42%	6	1%	/dba/oradata_dm_04
/dev/datadmlv5	102760448	59735256	42%	6	1%	/dba/oradata_dm_05
/dev/datadmlv6	102760448	59735240	42%	6	1%	/dba/oradata_dm_06
/dev/datadmlv7	102760448	59735432	42%	6	1%	/dba/oradata_dm_07
/dev/datadmlv8	102760448	59735240	42%	6	1%	/dba/oradata_dm_08
/dev/datadmlv9	102760448	59735272	42%	6	1%	/dba/oradata_dm_09
/dev/datadmlv10	102760448	59735240	42%	6	1%	/dba/oradata_dm_10
/dev/datadmlv11	102760448	59735280	42%	6	1%	/dba/oradata_dm_11
/dev/datadmlv12	102760448	59735264	42%	6	1%	/dba/oradata_dm_12
/dev/datadmlv13	102760448	59735248	42%	6	1%	/dba/oradata_dm_13
/dev/datadmlv14	102760448	59735264	42%	6	1%	/dba/oradata_dm_14
/dev/datadmlv15	102760448	59735200	42%	6	1%	/dba/oradata_dm_15
/dev/datadmlv16	102760448	59735224	42%	6	1%	/dba/oradata_dm_16
/dev/datadmlv17	102760448	61783272	40%	6	1%	/dba/oradata_dm_17
/dev/datadmlv18	102760448	61783256	40%	6	1%	/dba/oradata_dm_18

Listing 7-9 Master database server filesystem layout

/dev/rsync_d1_01 104595456 83075104 21% 6 1% /dba/oradata_d1_01 /dev/rsync_d1_02 104595456 83075104 21% 6 1% /dba/oradata_d1_02 /dev/rsync_d1_03 104595456 83075104 21% 6 1% /dba/oradata_d1_03

Listing 7-10 OLTP database server filesystem layout

/dev/rsync_d1_04	104595456	83075104	21%	6	1%	/dba/oradata_d1_04
/dev/rsync_d1_05	104595456	83075104	21%	6	1%	/dba/oradata_d1_05
/dev/rsync_d1_06	104595456	83075104	21%	6	1%	/dba/oradata_d1_06
/dev/rsync_d1_07	104595456	83075104	21%	6	1%	/dba/oradata_d1_07
/dev/rsync_d1_08	104595456	83075104	21%	6	1%	/dba/oradata_d1_08
/dev/rsync_d1_09	104595456	83075104	21%	6	1%	/dba/oradata_d1_09
/dev/rsync_d1_10	104595456	83075104	21%	6	1%	/dba/oradata_d1_10
/dev/rsync_d1_11	104595456	83075104	21%	6	1%	/dba/oradata_d1_11
/dev/rsync_d1_12	104595456	83075104	21%	6	1%	/dba/oradata_d1_12
/dev/rsync_d1_13	104595456	83075104	21%	6	1%	/dba/oradata_d1_13
/dev/rsync_d1_14	104595456	83075104	21%	6	1%	/dba/oradata_d1_14
/dev/rsync_d1_15	104595456	83075104	21%	6	1%	/dba/oradata_d1_15
/dev/rsync_d1_16	104595456	83075104	21%	6	1%	/dba/oradata_d1_16
/dev/rsync_d1_17	104595456	86060540	18%	6	1%	/dba/oradata_d1_17
/dev/rsync_d1_18	104595456	86168976	18%	6	1%	/dba/oradata_d1_18
/dev/rsync_d2_01	104595456	83075104	21%	6	1%	/dba/oradata_d2_01
/dev/rsync_d2_02	104595456	83075104	21%	6	1%	/dba/oradata_d2_02
/dev/rsync_d2_03	104595456	83075104	21%	6	1%	/dba/oradata_d2_03
/dev/rsync_d2_04	104595456	83075104	21%	6	1%	/dba/oradata_d2_04
/dev/rsync_d2_05	104595456	83075104	21%	6	1%	/dba/oradata_d2_05
/dev/rsync_d2_06	104595456	83075104	21%	6	1%	/dba/oradata_d2_06
/dev/rsync_d2_07	104595456	83075104	21%	6	1%	/dba/oradata_d2_07
/dev/rsync_d2_08	104595456	83075104	21%	6	1%	/dba/oradata_d2_08
/dev/rsync_d2_09	104595456	83075104	21%	6	1%	/dba/oradata_d2_09
/dev/rsync_d2_10	104595456	83075104	21%	6	1%	/dba/oradata_d2_10
/dev/rsync_d2_11	104595456	83075104	21%	6	1%	/dba/oradata_d2_11
/dev/rsync_d2_12	104595456	83075104	21%	6	1%	/dba/oradata_d2_12
/dev/rsync_d2_13	104595456	83075104	21%	6	1%	/dba/oradata_d2_13
/dev/rsync_d2_14	104595456	83075104	21%	6	1%	/dba/oradata_d2_14
/dev/rsync_d2_15	104595456	83075104	21%	6	1%	/dba/oradata_d2_15
/dev/rsync_d2_16	104595456	83075104	21%	6	1%	/dba/oradata_d2_16
/dev/rsync_d2_17	104595456	84099104	20%	6	1%	/dba/oradata_d2_17
/dev/rsync_d2_18	104595456	84099104	20%	6	1%	/dba/oradata_d2_18

Listing 7-10 (continued)

Both listings show a df -k output of only the database filesystems. As you can see, these are huge filesystems. Daily we are moving 400 GB of data to two servers at the same time, for a total of 800 GB of data transfer.

NOTE The destination filesystems need the capacity to hold twice the source data. The reason is to ensure that if the file already exists on the destination, the filesystem has enough space to hold the temporary file needed to handle the rsync remote-update protocol. If you do not have the disk space available, you can delete the destination files before starting the rsync copy process.

rsync Copy Shell Script

The rsync copy process is performed by executing the shell script rsync_daily_ copy.ksh (shown later in Listing 7-13), which is located in /usr/local/bin on the gamma Oracle master database server. The rsync_daily_copy.ksh shell script is executed via root user cron table entries, as shown in Listing 7-11.

```
# Daily rsync copy process
0 2 * * 0 /usr/local/bin/rsync_daily_copy.ksh > \
    /usr/local/bin/daily_rsync_stdout_sun.log 2>&1
0 2 * * 1 /usr/local/bin/rsync_daily_copy.ksh > \
    /usr/local/bin/daily_rsync_stdout_mon.log 2>&1
0 2 * * 2 /usr/local/bin/rsync_daily_copy.ksh > \
    /usr/local/bin/daily_rsync_stdout_tue.log 2>&1
0 2 * * 3 /usr/local/bin/rsync_daily_copy.ksh > \
    /usr/local/bin/daily_rsync_stdout_wed.log 2>&1
0 2 * * 4 /usr/local/bin/rsync_daily_copy.ksh > \
    /usr/local/bin/daily_rsync_stdout_thu.log 2>&1
0 2 * * 5 /usr/local/bin/rsync_daily_copy.ksh > \
    /usr/local/bin/daily_rsync_stdout_thu.log 2>&1
0 2 * * 6 /usr/local/bin/rsync_daily_copy.ksh > \
    /usr/local/bin/daily_rsync_stdout_fri.log 2>&1
0 2 * * 6 /usr/local/bin/rsync_daily_copy.ksh > \
    /usr/local/bin/daily_rsync_stdout_sat.log 2>&1
```

Listing 7-11 Root cron table to execute rsync_daily_copy.ksh daily

This set of cron table entries in Listing 7-11 starts the rsync_daily_copy.ksh shell script executing every day at 2:00 a.m. Notice that the log filename changes every day, so we can keep one week of log files to troubleshoot any problems.

We begin by defining files and variables for this script, as shown In Listing 7-12.

Listing 7-12 File and variable definitions

```
BG_PID_LIST=
TOTAL=0
THIS_HOST=$(hostname)
[[ $THIS_HOST = gamma ]] && MACHINE_LIST="alpha-rsync bravo-rsync"
[[ $THIS_HOST = gamma-dg ]] && MACHINE_LIST="alpha-rsync bravo-rsync"
# Set up the correct echo command usage. Many Linux
# distributions will execute in Bash even if the
# script specifies Korn shell. Bash shell requires
# we use echo -e when we use \n, \c, etc.
case $SHELL in
*/bin/Bash) alias echo="echo -e"
;;
esac
```

In Listing 7-12 we define the following files and global variables:

- typeset -i DAY Typesets the DAY variable to be an integer.
- EMAIL_FROM Specifies the "from" address for emails.
- PATH Adds /usr/local/bin to the PATH.
- WORK_DIR Defines the work directory as /usr/local/bin.
- LOGFILE Defines the shell script log file.
- SEARCH_DIR Defines a directory to search for status files.
- READYTORUN_FILE Indicates the file specifying the script can start copying.
- COMPLETE_FILE Indicates the file specifying the copy process has completed.
- RSYNCFAILED_FILE Indicates the file specifying the copy process failed.
- MAILMESSAGEFILE Defines the file that contains the mail message body.

The last step here is to use the hostname of the machine, defined as THIS_HOST, to determine the target machines to receive the rsync data. Notice that we have two gamma machines defined, with the second named gamma-dg. This is an Oracle Data Guard server used as a backup in case of primary-database failure. The final task is to verify that the echo command executes properly. Many Linux distributions will execute in Bash shell even through we declare a Korn shell on the very first line of the shell script. The following code segment corrects this problem by setting up an alias if \$SHELL is */bin/Bash:

```
# Set up the correct echo command usage. Many Linux
```

```
# distributions will execute in Bash even if the
```

```
# script specifies Korn shell. Bash shell requires
```

```
\# we use echo -e when we use \n, \c, etc.
```

```
case $SHELL in
*/bin/Bash) alias echo="echo -e"
;;
esac
```

The rsync_daily_copy.ksh shell script, starting at the BEGINNING OF MAIN, checks if any other versions of itself are currently executing. If another instance of the shell script is running, this script kills the previous session, and then sleeps for 60 seconds to allow any running rsync sessions to die.

The next step in the rsync_daily_copy.ksh shell script is where we monitor the search directory /orabin/apps/oracle/dbadm/general/bin for a file named readytocopy.txt. This is the file created by the DBA Team's shell script that signals that the Oracle database has been put into read-only mode and the rsync copy process can start. The readytocopy.txt file contains only one character, 1 or 2, that specifies the target day's destination filesystems. The number is assigned to the DAY variable. In the next step, we verify that the \$DAY variable is valid.

Before we start the rsync copy, we ping the target machines to verify that they are reachable through the network. When verified, we send a notification email to all users defined in the mail alias data_support that the rsync copy process started execution.

At this point we are ready to start the rsync copy process. We use the MACHINE_LIST variable to loop through all the target machines, and then all the filesystems defined by the regular expression shown here:

```
oradata_dm_[0-2][0-9]
```

Notice that this regular expression supports valid values of oradata_dm_00 through oradata_dm_29. Even though we currently have only 18 filesystems, starting at oradata_dm_01, this allows room for growth without modifying the shell script in the near term. For each Oracle filesystem on the master database server, we use a sed statement to convert the dm to the specified \$DAY, d1 or d2, to match the OLTP filesystems, as shown here:

```
# This sed statement changes the "m" to $DAY
REM_FL=$(echo $FL | sed s/oradata_dm_/oradata_d${DAY}_/g)
```

As we loop through all the filesystems, we execute an rsync session in the background, for a total of 36 sessions, 18 to each target machine. To facilitate the bandwidth to copy 400 GB of data to two servers, we use a private network using a dedicated switch and dedicated NIC adapter/IP address. We define the hostnames for the new private network interfaces as follows:

```
gamma-rsync, alpha-rsync, bravo-rsync
```

Once all the rsync sessions have started, we sleep for 10 seconds to allow each of the sessions to start. After this 10-second interval, we scan the process table to identify how many rsync sessions remain executing of the total that started. If all the sessions completed in 10 seconds, all the files are up-to-date and no rsync copy will take place. Otherwise, we give the user feedback as to how many sessions require updating, show the start time, and start a loop to monitor all the rsync sessions every minute. Until

the remaining rsync sessions are less than the total sessions started, we just print a dot on the screen every minute as feedback that processing continues. As soon as the remaining rsync sessions drop below the total sessions, we print the elapsed time every 5 minutes and give an update on how many sessions remain executing. Then, when the remaining rsync sessions reaches one-fourth of the total sessions started, we list the remaining filesystem sessions in addition to the elapsed time, and number of remaining sessions. Users always want to know "What's taking so long?" Well, this script tells them.

When local rsync processing completes execution on the master database server, we check the target machines to verify that all remote rsync processing is also complete. After we verify that all rsync processing has completed both locally and remotely, we verify that the rsync copy was successful by comparing the file sizes on the source with the file sizes on the target machines. If the file sizes match, we have a successful rsync copy.

The last two tasks are to put the signal file named copycomplete.txt on the system so that the Oracle DBA Team's script can put the master database back in read-write mode. On the OLTP server's Oracle scripts, rename the database to the specified \$DAY and swing the synonym to point to the correct \$DAY OLTP database. When the rsync copy portion of the process completes, the script sends an email to notify everyone that the rsync copy completed successfully.

Should there be a failure during the rsync process, an email stating the exact failure is sent, and an error file named copyfailed.txt is placed on the system to let the Oracle DBA Team's script know that the rsync process failed. Listing 7-23, shown later in this chapter, shows the rsync_daily_copy.ksh shell script in action. For now, study Listing 7-13 and pay particular attention to the boldface type.

```
#!/bin/ksh
#
# SCRIPT: rsync_daily_copy.ksh
# AUTHOR: Randy Michael
# DATE: 11/10/2007
# REV: 3.2.Prod
#
# PURPOSE: This script is used to replicate Oracle .dbf
     files between the "master" DB server and the two
#
#
     OLTP Oracle servers. The copy method used is
     rsync. For this method to work, the Oracle DBA must
#
#
    first put the database tables that reside in the
     copy filesystems into READ-ONLY mode. Before starting
#
#
    rsync copy sessions, this script searches for a file,
#
     defined by the $READYTORUN FILE variable, that is
     placed on the system by the Oracle DBA team; when
#
     the file is found this script will execute
     all 36 rsync sessions, 18 to each OLTP server,
#
     at the same time, then wait for all sessions to
#
```

Listing 7-13 rsync_daily_copy.ksh shell script

complete, both locally and on the remote servers. # After the rsync copy sessions complete the copy is # verified by matching file sizes between the master # copy file and the target copy files. When verified # successful this script writes a file to the system, # defined by the \$COMPLETE_FILE variable, to signal # the Oracle DBAs to put the DB back into READ-WRITE # mode, copy the metadata over, build the tables and # attach the DB on the OLTP side servers. Should a # failure occur a file, defined by the \$RSYNCFAILED_FILE variable, is written to the system to signal to the # # Oracle DBA team an rsync copy process failure. # # # EXIT CODES: # 0 ==> Normal execution. # 1 ==> The value assigned to DAY is not # an integer 1 or 2. # 2 ==> Remote host is not pingable. 3 ==> Copy verification failed. One or # more local files are not the same # size as the remote files. # 4 ==> No machines are defined to copy # # data to. 5 ==> Exited on a trapped signal. # set -x # Uncomment to debug this script # set -n # Uncomment to check the script's syntax # without any execution. Do not forget to # recomment this line! **** # REVISION LIST: # IF YOU MODIFY THIS SCRIPT, DOCUMENT THE CHANGE(S) !!! ***** # Revised by: Randy Michael # Revision Date: 7/23/2007 # Revision: Changed the script to process # all 36 mount points at a time. ***** # Revised by:

```
# Revision Date:
# Revision:
*******
# DEFINE FILES AND GLOBAL VARIABLES HERE
****
typeset -i DAY
EMAIL_FROM=data_support@gamma
export PATH=$PATH:/usr/local/bin
WORK_DIR=/usr/local/bin
LOGFILE=${WORK_DIR}/rsync_daily_copy.log
SEARCH_DIR=/orabin/apps/oracle/dbadm/general/bin
READYTORUN_FILE=${SEARCH_DIR}/readytocopy.txt
COMPLETE_FILE=${SEARCH_DIR}/copycomplete.txt
RSYNCFAILED_FILE=${SEARCH_DIR}/copyfailed.txt
MAILMESSAGEFILE=${WORK_DIR}/email_message.out
THIS_SCRIPT=$(basename $0)
BG_PID_LIST=
TOTAL=0
THIS HOST=$(hostname)
[[ $THIS_HOST = gamma ]] && MACHINE_LIST="alpha-rsync bravo-rsync"
[[ $THIS_HOST = gamma-dg ]] && MACHINE_LIST="alpha-rsync bravo-rsync"
# Set up the correct echo command usage. Many Linux
# distributions will execute in Bash even if the
# script specifies Korn shell. Bash shell requires
# we use echo -e when we use n, c, etc.
case $SHELL in
*/bin/Bash) alias echo="echo -e"
          ;;
esac
*******
# DEFINE FUNCTIONS HERE
*****
usage ()
{
echo "\nUSAGE: $THIS_SCRIPT Day"
echo "\nWhere Day is 1 or 2\n"
}
****
cleanup_exit ()
{
```

```
# If this script is executing, then something failed!
[ $1 ] && EXIT_CODE=$1
echo "\n$THIS_SCRIPT is exiting on non-zero exit code: $EXIT_CODE"
echo "\nPerforming cleanup..."
echo "Removing $READYTORUN_FILE"
rm -f $READYTORUN FILE >/dev/null 2>&1
echo "Removing $COMPLETE_FILE"
rm -f $COMPLETE_FILE >/dev/null 2>&1
echo "\nCreating $RSYNCFAILED_FILE"
echo "\nRsync failed on $THIS_HOST with exit code $EXIT_CODE $(date)\n"\
      | tee -a $RSYNCFAILED_FILE
echo "\nCleanup Complete...Exiting..."
exit $EXIT_CODE
}
*******
trap_exit ()
{
echo "\nERROR: EXITING ON A TRAPPED SIGNAL!\n"
echo "\nRSYNC ERROR: $THIS HOST -- Rsync process exited abnormally \
   on a trapped signal $(date)!" > $MAILMESSAGEFILE
mailx -r "$EMAIL_FROM" -s "SYNC ERROR: $THIS_HOST -- Rsync process \
   exited abnormally on a trapped signal!" \
           data_support < $MAILMESSAGEFILE</pre>
sleep 2 # Allow the email to go out
cleanup_exit 5
exit 5
}
*****
verify_copy ()
{
#set -x
MYLOGFILE=${WORK_DIR}/verify_rsync_copy_day${DAY}.log
>$MYLOGFILE
ERROR=0
# Enclose this loop so we can redirect output to the log file
# with one assignment at the bottom of the function
# Put a header for the verification log file
echo "\nRsync copy verification between $THIS_HOST and machines
```

Listing 7-13 (continued)

```
$MACHINE_LIST\n\n"\
       >$MYLOGFILE
for M in $MACHINE LIST
do
   for LOC_MP in $(df | grep oradata_dm_[0-2][0-9] | awk '{print $7}')
   do
       LS_FILES=$(find $LOC_MP -type f)
       for FL in $LS_FILES
       do
           LOC_FS=$(1s -1 $FL | awk '{print $5}' 2>&1)
           # This sed statement changes the "m" to $DAY
           REM_FL=$(echo $FL | sed s/oradata_dm_/oradata_d${DAY}_/g)
           REM_FS=$(rsh $M ls -l $REM_FL | awk '{print $5}' 2>&1)
           echo "Checking File: $FL"
           echo "Local $THIS_HOST size:\t$LOC_FS"
           echo "Checking Remote File: $REM_FL"
           echo "$M size:\t$REM_FS"
           if [ "$LOC_FS" -ne "$REM_FS" ]
           then
              echo "ERROR: File size mismatch between $THIS_HOST and $M"
              echo "File is: $FL"
              ERROR=1
           fi
       done
   done
done
if (( ERROR != 0 ))
then
    # Record the failure in the log file
    echo "\n\nRSYNC ERROR: $THIS_HOST Rsync copy failed...file \
   size mismatch...\n\n" | tee -a $MYLOGFILE
    # Send email notification with file size log
   mailx -r "$EMAIL_FROM" -s "RSYNC ERROR: $THIS_HOST Rsync \
   copy failed...file size mismatch -- log attached" \
                 data_support < $MYLOGFILE</pre>
    echo "\nERROR: Rsync copy Failed!"
    echo "\n\nCheck log file: $MYLOGFILE\n\n"
    echo "\n...Exiting...\n"
    cleanup_exit 3
    return 3
```

```
exit 3
else
   echo "\nSUCCESS: Rsync copy completed successfully..."
   echo "\nAll file sizes match...\n"
fi
} | tee -a $MYLOGFILE
}
*****
ready_to_run ()
{
# set -x
# This function looks for a file on the system
# defined by the $READYTORUN_FILE variable. The
# presence of this file indicates we are ready
# to run this script. This file will contain a
# number 1 or 2 identifying the day we are
# working with.
if [ -r ${READYTORUN_FILE} ]
then
    cat ${READYTORUN FILE}
else
    echo "NOT_READY"
fi
}
****
elapsed_time ()
{
SEC=$1
(( SEC < 60 )) && echo "[Elapsed time: $SEC seconds]\c"
(( SEC >= 60 && SEC < 3600 )) && echo "[Elapsed time:
  $(( SEC / 60 )) min $(( SEC % 60 )) sec]\c"
(( SEC > 3600 )) && echo "[Elapsed time: $(( SEC / 3600 ))
  hr $(( (SEC % 3600) / 60 )) min $(( (SEC % 3600) % 60 )) sec]\c"
}
*****
# BEGINNING OF MAIN
*****
# Set a trap
trap 'trap_exit' 1 2 3 5 6 11 14 15 17
```

```
# Save the old log file
cp -f $LOGFILE ${LOGFILE}.yesterday \
      >$LOGFILE
# Enclose the entire main part of the script in
# curly braces so we can redirect all output of
# the shell script with a single redirection
# at the bottom of the script to the $LOGFILE
echo "\n[[ $THIS_SCRIPT started execution $(date) ]]\n"
# Ensure that target machines are defined
if [ -z "$MACHINE_LIST" ]
then
    echo "\nERROR: No machines are defined to copy data to..."
    echo "\nRSYNC ERROR: $THIS_HOST has no machines are defined to \
   copy data to... " > $MAILMESSAGEFILE
   mailx -r "$EMAIL_FROM" -s "RSYNC ERROR: $THIS_HOST has no machines \
   defined to copy data to" data_support < $MAILMESSAGEFILE
    echo "....Unable to continue....Exiting...."
    cleanup_exit 4
    exit 4
fi
# Checking for currently running versions of this script
echo "Checking for Currently Running Versions of this Script"
MYPID=$$ # Capture this script's PID
MYOTHERPROCESSES=$(ps -ef | grep $THIS_SCRIPT | grep -v $MYPID \
                   grep -v grep | awk '{print $2}')
if [[ "$MYOTHERPROCESSES" != "" ]]
then
    echo "\WARNING: Another version of this script is running...\
   Killing it!"
    echo "Killing the following process(es)...$MYOTHERPROCESSES"
   kill -9 $MYOTHERPROCESSES
    echo "\nNOTICE: Sleeping for 1 minute to allow both local \
    and remote rsync sessions to terminate, if they exist...\n"
   sleep 60 # Allow any rsync sessions to stop both locally and remotely
else
```

```
echo "No other versions running...proceeding"
fi
# Remove the file that indicates the rsync copy
# is mounted and ready to use, if it exists.
rm -f $COMPLETE_FILE >/dev/null 2>&1
# Remove the file that indicates the rsync copy
# failed and exited on a non-zero exit code, if
# it exists.
rm -f $RSYNCFAILED_FILE >/dev/null 2>&1
# Search for the file indicating we are ready to proceed
# Send notification
echo "\nDaily Rsync Copy Process Began on Host $THIS_HOST $(date)" \
   > $MAILMESSAGEFILE
echo "\nStarted looking for $READYTORUN_FILE" >> $MAILMESSAGEFILE
mailx -r "$EMAIL_FROM" -s "Rsync Copy Process Began Looking \
   for Startup File on $THIS_HOST" \
      data_support < $MAILMESSAGEFILE</pre>
echo "\nSearching for the file: ${READYTORUN_FILE}"
RUN_STATUS=$(ready_to_run)
until [[ $RUN_STATUS != "NOT_READY" ]]
do
    date
    echo "$READYTORUN_FILE is not present...Sleeping 5 minutes..."
    sleep 300
    RUN_STATUS=$(ready_to_run)
done
date
echo "Found file: $READYTORUN_FILE -- Ready to proceed..."
DAY=$RUN STATUS
# Test the value assigned to the DAY variable
echo "Testing variable assignment for the DAY variable"
if (( DAY != 1 && DAY != 2 ))
then
    echo "\nRSYNC ERROR: $THIS_HOST -- The value assigned to the DAY \
```

```
variable" > $MAILMESSAGEFILE
   echo "==> $DAY - is not an integer 1 or 2...Exiting..." \
   >> $MAILMESSAGEFILE
   mailx -r "$EMAIL_FROM" -s "ERROR: $THIS_HOST -- Value assigned \
   to the DAY variable $DAY is invalid" data_support < $MAILMESSAGEFILE
   usage
    cleanup exit
    exit 1
fi
# Ensure the remote machines are reachable through
# the network by sending 1 ping.
echo "Verifying the target machines are pingable...\n"
for M in $MACHINE_LIST
do
    echo "Pinging $M..."
   ping -c1 $M >/dev/null 2>&1
    if (( $? != 0 ))
    then
        echo "RSYNC ERROR: $M is not pingable from $THIS_HOST" \
   > $MAILMESSAGEFILE
        echo "The rsync copy process cannot continue...Exiting" \
   >> $MAILMESSAGEFILE
        mailx -r "$EMAIL_FROM" -s "RSYNC ERROR: $M is not pingable \
   from $THIS_HOST" data_support < $MAILMESSAGEFILE</pre>
        echo "\nERROR: $M host is not pingable...cannot continue..."
        echo "...EXITING...\n"
        sleep 2
       cleanup_exit 2
        exit 2
    Pleb
        echo "Pinging $M succeeded..."
    fi
done
# Notify start of rsync processing by email
echo "Rsync Copy Process for Day $DAY Starting Execution on \
   $THIS_HOST $(date)" > $MAILMESSAGEFILE
mailx -r "$EMAIL_FROM" -s "Rsync copy process for day $DAY starting \
   on $THIS_HOST" data_support < $MAILMESSAGEFILE
# Start all of the rsync copy sessions at once by looping
# through each of the mount points and issuing an rsync
# command in the background, and incrementing a counter.
```

```
echo "\nStarting a bunch of rsync sessions...\n"
# This script copies from the DM filesystems to the
# Day 1 or Day 2 filesystems on the OLTP servers.
for M in $MACHINE_LIST
do
   for LOC_MP in $(df | grep oradata_dm_[0-2][0-9] | awk '{print $7}')
   do
        # This sed statement changes the "m" to $DAY
       REM_MP=$(echo $LOC_MP | sed s/m/$DAY/g)
        time rsync -avz ${LOC_MP}/ ${M}:${REM_MP} 2>&1 &
        ((TOTAL = TOTAL + 1))
   done
done
# Sleep a few seconds before monitoring processes
sleep 10
# Give some feedback to the end user.
REM_SESSIONS=$(ps -ef | grep "rsync -avz" | grep oradata_dm_[0-2][0-9] \
               grep -v grep | awk '{print $2}' | wc -1)
if (( REM_SESSIONS > 0 ))
then
   echo "\n$REM_SESSIONS of $TOTAL rsync copy sessions \
   require further updating ... "
   echo "\nProcessing rsync copies from $THIS_HOST to both \
   $MACHINE_LIST machines"
   echo "\nPlease be patient, this process may take longer than \
   two hours...\n"
   echo "Rsync is running [Start time: $(date)]\c"
else
   echo "\nAll files appear to be in sync...verifying file sizes... \
   Please wait...\n"
fi
# While the rsync processes are executing this loop
# will place a . (dot) every 60 seconds as feedback
# to the end user that the background rsync copy
# processes are still executing. When the remaining
# rsync sessions are less than the total number of
# sessions (normally 36) this loop will give feedback
# as to the number of remaining rsync session, as well
# as the elapsed time of the processing, every 5 minutes.
```

```
SECONDS=10
MIN_COUNTER=0
until (( REM_SESSIONS == 0 ))
do
    sleep 60
    echo ".\c"
    REM_SESSIONS=$(ps -ef | grep "rsync -avz" \
                   grep oradata_dm_[0-2][0-9] \
                   | grep -v grep | awk '{print $2}' | wc -1)
    if (( REM_SESSIONS < TOTAL ))
    then
        (( MIN COUNTER = MIN COUNTER + 1 ))
        if (( MIN_COUNTER >= 5 ))
        then
           MIN_COUNTER=0
           echo "\n$REM_SESSIONS of $TOTAL rsync sessions \
   remaining $(elapsed_time $SECONDS)\c"
           if (( REM_SESSIONS <= $(( TOTAL / 4 )) ))
           then
              echo "\nRemaining rsync sessions include:\n"
              ps -ef | grep "rsync -avz" | grep oradata_dm_[0-2][0-9] \
                     grep -v grep
              echo
           fi
        fi
    fi
done
echo "\n...Local rsync processing completed on $THIS_HOST...$(date)"
echo "\n...Checking remote target machines: $MACHINE_LIST..."
# Just because the local rsync processes have completed does
# not mean that the remote rsync processes have completed
# execution. This loop verifies that all of the remote
# rsync processes completed execution. The end user will
# receive feedback if any of the remote processes are running.
for M in $MACHINE_LIST
do
   for LOC_MP in $(df | grep oradata_dm_[0-2][0-9] | awk '{print $7}')
   đo
       # This sed statement changes the "m" to $DAY
       REM_MP=$(echo $LOC_MP | sed s/m/$DAY/g)
       RPID=$(rsh $M ps -ef | grep rsync | grep $REM_MP \
              grep -v grep | awk '{print $2}')
       until [ -z "$RPID" ]
```

Listing 7-13 (continued)

```
do
          echo "rsync is processing ${REM_MP} on ${M}... \
   sleeping one minute ... "
          sleep 60
          RPID=$(rsh $M ps -ef | grep rsync | grep $REM_MP \
               grep -v grep | awk '{print $2}')
      done
  done
done
echo "\n...Remote rsync processing completed $(date)\n"
# Verify the copy process
verify_copy
if (( $? != 0 ))
then
   exit 3
fi
# Write to the $COMPLETE_FILE to signal the Oracle DBAs
# that the copy has completed
echo "Rsync copy from $THIS_HOST to machines $MACHINE_LIST \
completed successfully $(date)" | tee -a $COMPLETE_FILE
# Remove the ready to run file
rm -f $READYTORUN FILE >/dev/null 2>&1
# Notify completion by email
echo "Rsync Copy for Day $DAY Completed Successfully Execution \
  on $THIS_HOST $(date)\n" > $MAILMESSAGEFILE
elapsed_time $SECONDS >> $MAILMESSAGEFILE
mailx -r "$EMAIL_FROM" -s "Rsync copy for day $DAY completed \
   successfully on $THIS_HOST" data_support < $MAILMESSAGEFILE
echo "\nRsync copy completed $(date)"
echo "\n[[ $THIS_SCRIPT completed execution $(date) ]]\n"
exit 0
} | tee -a $LOGFILE 2>&1
****
# END OF SCRIPT
*****
```

You are probably thinking that Listing 7-13 is intuitively obvious! But, if not, let's cover a few points in more detail. I know this is a bit complicated in that we have to monitor for the existence of a file, and verify that the file contains valid data. However, this is a simple method of relaying information between two separate shell scripts. Remember, the Oracle DBA Team has a shell script monitoring for the nightly processing to complete. When that processing is complete, the database is put into read-only mode, the metadata is captured, and then the readytocopy.txt file is created with the specific target day, 1 or 2.

Let's cover Listing 7-13 in logical execution order starting with BEGINNING OF MAIN. The first thing we do is set a trap for exit signals 1, 2, 3, 5, 6, 11, 14, 15, and 17. Remember that we cannot trap kill -9. With a trap we can log a failure, send an email, and clean up any leftover files before exiting. After setting the trap, we copy the \$LOGFILE to \${LOGFILE}.yesterday. This log file is an internal script log file, not to be confused with the log files defined in the cron table earlier.

Now we are ready to start looking for the file that signals that we can start the rsync copy process. Listing 7-14 shows this section of script code.

```
# Checking for currently running versions of this script
echo "Checking for Currently Running Versions of this Script"
MYPID=$$ # Capture this script's PID
MYOTHERPROCESSES=$(ps -ef | grep $THIS_SCRIPT | grep -v $MYPID \
                   grep -v grep | awk '{print $2}')
if [[ "$MYOTHERPROCESSES" != "" ]]
then
   echo "\WARNING: Another version of this script is running... \
  Killing it!"
    echo "Killing the following process(es)...$MYOTHERPROCESSES"
   kill -9 $MYOTHERPROCESSES
    echo "\nNOTICE: Sleeping for 1 minute to allow both local \
   and remote rsync sessions to terminate, if they exist...\n"
    sleep 60 # Allow any rsync sessions to stop both locally and \
remotely else
    echo "No other versions running...proceeding"
fi
```

Listing 7-14 Script code to check for other sessions

We capture the shell script's process ID (PID) using the \$\$ shell variable. With this PID we can scan the process table looking for other instances of itself without killing this executing shell script. If we find other script sessions running, we issue a kill -9 on each of the PIDs. If we need to kill other rsync processes, we sleep for 60 second to allow time for the killed sessions to exit.

Next we remove any leftover files indicating the shell script completed or failed, if they exist. At this point, we are starting to look for the readytocopy.txt file, so we send a notification email and enter the loop shown in Listing 7-15.

Listing 7-15 Loop to wait for a startup file

Notice in Listing 7-15 that we call a function named ready_to_run. This function simply looks for the presence of a file defined by the variable \$READYTORUN_FILE. The ready_to_run function is shown in Listing 7-16.

```
ready_to_run ()
{
# set -x
# This function looks for a file on the system
# defined by the $READYTORUN_FILE variable. The
# presence of this file indicates we are ready
# to run this script. This file will contain a
# number 1 or 2 identifying the day we are
# working with.
if [ -r ${READYTORUN_FILE} ]
then
    cat ${READYTORUN_FILE}
else
   echo "NOT_READY"
fi
}
```

Listing 7-16 ready_to_run function

We use the -r test operator to look for a readable file defined by \${READYTORUN_ FILE}. If we find the file, we display back the file contents; otherwise, we just echo back "NOT_READY". When we find the startup file, we test to ensure the value is a 1 or

2. If this test passes, we send another email stating that we found the startup file and are starting the rsync sessions.

As we loop through the filesystems defined by the regular expression oradata_ dm_[0-2][0-9], we use the DAY variable to translate the master data filesystems to the corresponding OLTP day 1 or 2 filesystems. This substitution is performed with a sed statement and assigned to the REM_MP variable, for remote mount point. Using the LOC_MP and REM_MP variables, we start the rsync sessions in the background, and time the execution of each session with the shell time command, not the system's /usr/bin/time command. The shell's time program gives a more granular detail for user, system, and kernel execution times. The last step is to keep a running total of how many rsync sessions we start. We need this value as feedback to the end user. This loop to start all the rsync sessions is shown in Listing 7-17.

```
for M in $MACHINE_LIST
do
for LOC_MP in $(df | grep oradata_dm_[0-2][0-9] | awk '{print $7}')
do
    # This sed statement changes the "m" to $DAY
    REM_MP=$(echo $LOC_MP | sed s/m/$DAY/g)
    time rsync -avz ${LOC_MP}/ ${M}:${REM_MP} 2>&1 &
    (( TOTAL = TOTAL + 1 ))
    done
done
```

Listing 7-17 Loop to start all the rsync sessions

We sleep for 10 seconds after this loop in Listing 7-17 to allow time for all the rsync sessions to start. After this 10-second wait period, we check the process table to see how many rsync sessions remain executing *locally*. Just remember that we will also check the remote machines' process table when all the local rsync sessions complete.

Listing 7-18 shows the code segment to check for the remaining local rsync sessions.

Listing 7-18 Code to check for remaining rsync sessions

```
echo "\nPlease be patient, this process may take longer than \
  two hours...\n"
  echo "Rsync is running [Start time: $(date)]\c"
else
  echo "\nAll files appear to be in sync...verifying file sizes... \
  Please wait...\n"
fi
```

Notice that we give the end user feedback as to the number of sessions that remain executing out of the total number of sessions started. We also let the user know that this could be a long process and give a startup time stamp.

If there are no remaining sessions executing after 10 seconds, all the files are up-to-date and there is nothing to copy. This advanced ability to check the source and target files is what makes rsync an excellent choice to replicate data.

While the rsync processes are executing, this loop will place a . (dot) every 60 seconds as feedback to the end user that the background rsync copy processes are still executing. When the remaining rsync sessions are less than the total number of sessions (normally 36 for our example), this loop will give feedback as to the number of remaining rsync sessions, as well as the elapsed time of the processing, every 5 minutes. This feedback loop is shown in Listing 7-19.

```
SECONDS=10
MIN COUNTER=0
until (( REM_SESSIONS == 0 ))
do
    sleep 60
    echo ".\c"
    REM_SESSIONS=$(ps -ef | grep "rsync -avz" \
                   grep oradata_dm_[0-2][0-9] \
                   | grep -v grep | awk '{print $2}' | wc -1)
    if (( REM_SESSIONS < TOTAL ))
    then
        (( MIN_COUNTER = MIN_COUNTER + 1 ))
        if (( MIN_COUNTER >= 5 ))
        then
           MIN COUNTER=0
           echo "\n$REM_SESSIONS of $TOTAL rsync sessions \
   remaining $(elapsed time $SECONDS)\c"
           if (( REM_SESSIONS <= $(( TOTAL / 4 )) ))
           then
              echo "\nRemaining rsync sessions include:\n"
              ps -ef | grep "rsync -avz" | grep oradata_dm_[0-2][0-9] \
```

Listing 7-19 Loop to monitor remaining rsync sessions

```
| grep -v grep
echo
fi
fi
fi
done
```

Notice that we start this code segment in Listing 7-19 by initializing the shell's SECONDS variable to 10 seconds. Remember that 10-second sleep time? We make up for it here. The nice thing about using the shell's SECONDS variable is that it increments automatically and we can initialize it to any integer value.

Until the remaining executing rsync sessions are 5 sessions or less, we just put a dot on the screen every 60 seconds. Once we reach one-fourth of the total session, we list the remaining session(s) to let the user know the answer to that all-important question, "What's taking so long?" This script gives the user that feedback every 5 minutes, and still puts a dot on the screen every 60 seconds.

Just because the local rsync processes have completed does not necessarily mean that the remote rsync processes have completed execution. This loop verifies that all the remote rsync processes completed execution. The end user will receive feedback if any of the remote processes are running. The code is shown Listing 7-20.

```
for M in $MACHINE LIST
do
   for LOC_MP in $(df | grep oradata_dm_[0-2][0-9] | awk '{print $7}')
   do
       # This sed statement changes the "m" to $DAY
       REM_MP=$(echo $LOC_MP | sed s/m/$DAY/g)
       RPID=$(rsh $M ps -ef | grep rsync | grep $REM_MP \
              grep -v grep | awk '{print $2}')
       until [ -z "$RPID" ]
       do
          echo "rsync is processing ${REM_MP} on ${M}... \
   sleeping one minute ... "
          sleep 60
           RPID=$(rsh $M ps -ef | grep rsync | grep $REM_MP \
                grep -v grep | awk '{print $2}')
       done
   done
done
echo "\n...Remote rsync processing completed $(date)\n"
```

Listing 7-20 Code segment to verify remote rsync sessions completed

NOTE Notice that this implementation uses remote shell (*x*sh) to perform the remote process table query. For security reasons you may want to install OpenSSH and set up the encryption key pairs so that this data transfer is encrypted.

Now that the rsync copy process has completed, we are ready to verify the file sizes, both locally and remotely, and to send email notification of the success or failure. Take a look at Listing 7-21 and we will cover the details at the end.

```
verify_copy
if (( $? != 0 ))
then
   exit 3
fi
# Write to the $COMPLETE_FILE to signal the Oracle DBAs
# that the copy has completed
echo "Rsync copy from $THIS_HOST to machines $MACHINE_LIST \
completed successfully $(date)" | tee -a $COMPLETE_FILE
# Remove the ready to run file
rm -f $READYTORUN_FILE >/dev/null 2>&1
# Notify completion by email
echo "Rsync Copy for Day $DAY Completed Successfully Execution \
   on $THIS_HOST $ (date) \n" > $MAILMESSAGEFILE
elapsed_time $SECONDS >> $MAILMESSAGEFILE
mailx -r "$EMAIL FROM" -s "Rsync copy for day $DAY completed \
   successfully on $THIS_HOST" data_support < $MAILMESSAGEFILE
echo "\nRsync copy completed $(date)"
echo "\n[[ $(basename $0) completed execution $(date) ]]\n"
exit 0
} | tee -a $LOGFILE 2>&1
```

Listing 7-21 Final verification and notification code

The first thing we do in this final step of the rsync_daily_copy.ksh shell script is to call the shell script's internal function verify_copy. This verification function compares the local (master) file size(s) to the remote (OLTP) server(s) file size(s). If the

size of each file matches, we assume that the copy completed successfully. However, because we are working with Oracle database files (*.dbf), the size of the file is not likely to change. It is the DBA Team's responsibility to verify the copy process by querying the OLTP databases to ensure they are usable. I tried to do a checksum on the Oracle *.dbf files, but it just takes too long. I've never had a problem transferring data with rsync; it is a very reliable program. This verify_copy function is a bit too long to repeat here, so please study Listing 7-13 for details.

If the copy verification fails (the file sizes do not match), we write a timestamp to the \$RSYNCFAILED_FILE file, which signals the DBA Team's script that the copy failed, and exit the script with a return code of 3. Otherwise, we create the \$COMPLETE_FILE that signals the DBA Team's shell script that the copy completed successfully.

Next we remove the \$READYTORUN_FILE file so that we will not automatically start this script again without the DBA Team creating a new \$READYTORUN_FILE file.

Before we send the final email notification, we create the body of the mail message by creating a file with the text we want to send. One part of this final message is to calculate the elapsed time by executing the elapsed_time function, as shown in Listing 7-22.

```
elapsed_time ()
{
SEC=$1
(( SEC < 60 )) && echo "[Elapsed time: $SEC seconds]\c"
(( SEC >= 60 && SEC < 3600 )) && echo "[Elapsed time:
    $(( SEC / 60 )) min $(( SEC % 60 )) sec]\c"
(( SEC > 3600 )) && echo "[Elapsed time: $(( SEC / 3600 ))
    hr $(( (SEC % 3600) / 60 )) min $(( (SEC % 3600) % 60 )) sec]\c"
}
```

Listing 7-22 elapsed_time function

The elapsed_time function in Listing 7-22 expects as ARG1 the number of seconds to convert to a more readable, user-friendly format. Specifically, this function supports feedback in hours, minutes, and seconds. To add "days" ability, we need only add a fourth test and calculation. This modification process can be repeated for months, years, and so on.

With the elapsed time added to the mail message, we send the email to all users defined by the data_support email alias.

Notice how we end the shell script with an exit 0 and close the ending brace, which uses the tee -a \$LOGFILE command to both capture the log data in the \$LOGFILE, and to display the data on the screen at the same time. The braces around the main body of the scripts allow us to only have this single \$LOGFILE output redirection.

After all this explanation, I hope you are ready to see the rsync_daily_copy.ksh shell script in action. Check out Listing 7-23 to see how the user is continuously updated on the status of this extremely long-running copy process.

```
[[ rsync_daily_copy.ksh started execution Thu Oct 25 02:00:00 EDT 2007 ]]
Checking for Currently Running Versions of this Script
No other versions running...proceeding
Searching for the file: /orabin/apps/oracle/dbadm/general/bin/
   readytocopy.txt
Thu Oct 25 02:00:02 EDT 2007
/orabin/apps/oracle/dbadm/general/bin/readytocopy.txt is not present...
   Sleeping 5 minutes...
Thu Oct 25 02:05:15 EDT 2007
/orabin/apps/oracle/dbadm/general/bin/readytocopy.txt is not present...
   Sleeping 5 minutes...
Thu Oct 25 02:10:18 EDT 2007
. . .
/orabin/apps/oracle/dbadm/general/bin/readytocopy.txt is not present...
   Sleeping 5 minutes...
Thu Oct 25 10:00:19 EDT 2007
/orabin/apps/oracle/dbadm/general/bin/readytocopy.txt is not present...
   Sleeping 5 minutes...
Thu Oct 25 10:05:19 EDT 2007
/orabin/apps/oracle/dbadm/general/bin/readytocopy.txt is not present...
   Sleeping 5 minutes...
Thu Oct 25 10:10:19 EDT 2007
Found file: /orabin/apps/oracle/dbadm/general/bin/readytocopy.txt --
   Ready to proceed ...
Testing variable assignment for the DAY variable
Verifying the target machines are pingable ...
Pinging alpha-rsync...
Pinging alpha-rsync succeeded...
Pinging bravo-rsync...
Pinging bravo-rsync succeeded...
Starting a bunch of rsync sessions...
building file list ... building file list ... building file list ...
building file list ... building file list ... building file list ...
building file list ... building file list ... building file list ...
building file list ... building file list ... building file list ...
building file list ... building file list ... building file list ...
building file list ... building file list ... building file list ... done
done
done
done
```

Listing 7-23 rsync_daily_copy.ksh shell script in action

done
done
./
./
./
done
./
./
oradata/
./
./
./
./
./
oradata/
./
./
./
./
done
./
./
building file list building file list building file list
building file list building file list building file list
building file list building file list building file list done
done
building file list done
done
done
done
done
building file list done
building file list done
done
done
done
dataset_master_12.dbf
oradata/iddata_master_02.dbf

dataset_master_08.dbf dataset_master_15.dbf dataset_master_16.dbf dataset_master_13.dbf dataset_master_14.dbf dataset_master_03.dbf dataset_master_07.dbf oradata/iddata_master_01.dbf dataset_master_04.dbf dataset_master_10.dbf dataset_master_09.dbf dataset_master_02.dbf dataset_master_11.dbf dataset_master_06.dbf dataset_master_05.dbf dataset_master_01.dbf ./ ./ ./ ./ ./ . / ./ . / ./ ./ ./ oradata/ dataset_master_04.dbf building file list ... building file list ... oradata/ iddata_master_01.dbf done done dataset_master_05.dbf building file list ... building file list ... dataset_master_10.dbf dataset_master_01.dbf done done building file list ... building file list ... dataset_master_06.dbf done dataset_master_14.dbf done dataset_master_08.dbf dataset_master_12.dbf dataset_master_11.dbf ./ ./ ./

```
dataset_master_09.dbf
./
./
dataset_master_07.dbf
oradata/
dataset_master_16.dbf
dataset_master_03.dbf
oradata/iddata master 02.dbf
dataset_master_02.dbf
dataset_master_13.dbf
dataset_master_15.dbf
     36 of 36 rsync copy sessions require further updating ...
Processing rsync copies from gamma to both alpha-rsync
  bravo-rsync machines
Please be patient, this process may take longer than two hours...
Rsync is running [Start time: Thu Oct 25 10:10:30 EDT 2007]...
   ...wrote 1679091467 bytes read 36 bytes 334647.40 bytes/sec
total size is 21495840768 speedup is 12.80
real 1h23m37.30s
user 22m53.20s
sys 0m12.17s
.wrote 1700381810 bytes read 36 bytes 336476.80 bytes/sec
total size is 21495840768 speedup is 12.64
real 1h24m12.82s
user 22m54.44s
sys 0m12.25s
wrote 1697703683 bytes read 36 bytes 335614.60 bytes/sec
total size is 21495840768 speedup is 12.66
real 1h24m17.93s
user 23m2.73s
sys 0m12.34s
wrote 1689385648 bytes read 36 bytes 333705.81 bytes/sec
total size is 21495840768 speedup is 12.72
real 1h24m21.60s
user 22m58.08s
sys 0m12.27s
.wrote 1725237799 bytes read 36 bytes 336139.86 bytes/sec
total size is 22020128768 speedup is 12.76
```

```
real 1h25m31.75s
user 23m28.60s
sys 0m12.48s
wrote 1728575265 bytes read 36 bytes 335939.23 bytes/sec
total size is 22020128768 speedup is 12.74
real 1h25m45.23s
user 23m28.51s
sys 0m12.68s
.wrote 1722668291 bytes read 36 bytes 334466.23 bytes/sec
total size is 22020128768 speedup is 12.78
real 1h25m49.70s
user 23m28.28s
sys 0m12.60s
wrote 1756094117 bytes read 36 bytes 340889.87
wrote 1689385648 bytes read 36 bytes 326041.82 bytes/sec
total size is 21495840768 speedup is 12.72
real 1h26m21.51s
user 22m57.00s
sys 0m12.30s
wrote 1679091467 bytes read 36 bytes 323867.59 bytes/sec
total size is 21495840768 speedup is 12.80
real 1h26m24.50s
user 22m51.34s
sys 0m12.16s
wrote 1697703683 bytes read 36 bytes 327079.30 bytes/sec
total size is 21495840768 speedup is 12.66
real 1h26m30.66s
user 22m56.29s
sys 0m12.25s
wrote 1772720217 bytes read 36 bytes 340416.76 bytes/sec
total size is 22020128768 speedup is 12.42
real 1h26m47.00s
user 23m44.54s
sys 0m12.51s
.wrote 1700381810 bytes read 36 bytes 326149.77 bytes/sec
total size is 21495840768 speedup is 12.64
real 1h26m52.65s
user 22m59.09s
sys 0m12.40s
wrote 1750600698 bytes read 36 bytes 335460.52 bytes/sec
total size is 22020128768 speedup is 12.58
```

Listing 7-23 (continued)

```
real 1h26m57.61s
user 23m37.81s
sys 0m12.56s
     22 of 36 rsync sessions remaining [Elapsed time: 1 hr 27 min 49 sec]
  wrote 1722668291 bytes read 36 bytes 325985.11 bytes/sec
total size is 22020128768 speedup is 12.78
real 1h28m3.83s
user 23m29.03s
sys 0m12.59s
wrote 1985371235 bytes read 36 bytes 374916.68 bytes/sec
total size is 22020128768 speedup is 11.90
real 1h28m15.61s
user 25m37.23s
sys 0m12.89s
wrote 1985371235 bytes read 36 bytes 374916.68 bytes/sec
total size is 22020128768 speedup is 11.90
real 1h28m15.59s
user 25m28.99s
sys 0m12.91s
wrote 1728575265 bytes read 36 bytes 325808.18 bytes/sec
total size is 22020128768 speedup is 12.74
real 1h28m25.05s
user 23m24.20s
sys 0m12.50s
wrote 1725237799 bytes read 36 bytes 324201.42 bytes/sec
total size is 22020128768 speedup is 12.76
real 1h28m41.40s
user 23m29.53s
sys 0m12.65s
.wrote 1756094117 bytes read 36 bytes 328149.89 bytes/sec
total size is 22020128768 speedup is 12.54
real 1h29m10.67s
user 23m41.94s
sys 0m12.56s
wrote 2029784980 bytes read 36 bytes 377739.84 bytes/sec
total size is 22544416768 speedup is 11.11
real 1h29m32.77s
user 26m1.22s
sys 0m13.06s
wrote 2029784980 bytes read 36 bytes 377739.84 bytes/sec
```

```
total size is 22544416768 speedup is 11.11
real 1h29m33.07s
user 26m5.36s
sys 0m13.05s
wrote 1772720217 bytes read 36 bytes 329839.10 bytes/sec
total size is 22020128768 speedup is 12.42
real 1h29m33.75s
user 23m36.80s
sys 0m12.40s
.wrote 1750600698 bytes read 36 bytes 324335.48 bytes/sec
total size is 22020128768 speedup is 12.58
real 1h29m57.29s
user 23m39.67s
sys 0m12.64s
wrote 1990832602 bytes read 36 bytes 368093.30 bytes/sec
total size is 22544416768 speedup is 11.32
real 1h30m8.23s
user 26m9.25s
sys 0m13.12s
.wrote 1986846605 bytes read 36 bytes 362067.73 bytes/sec
total size is 22020128768 speedup is 11.80
real 1h31m27.24s
user 25m38.40s
sys 0m13.08s
wrote 1986846605 bytes read 36 bytes 361474.87 bytes/sec
total size is 22020128768 speedup is 11.80
real 1h31m36.02s
user 25m40.54s
sys 0m13.05s
.wrote 2024116016 bytes read 36 bytes 366920.34 bytes/sec
total size is 22544416768 speedup is 11.14
real 1h31m56.00s
user 26m9.30s
sys 0m13.13s
wrote 2024116016 bytes read 36 bytes 365925.35 bytes/sec
total size is 22544416768 speedup is 11.14
real 1h32m11.50s
user 26m10.12s
sys 0m13.25s
wrote 1990832602 bytes read 36 bytes 357773.86 bytes/sec
```

Listing 7-23 (continued)

```
total size is 22544416768 speedup is 11.32
real 1h32m44.39s
user 26m19.03s
sys 0m13.30s
      6 of 36 rsync sessions remaining [Elapsed time: 1 hr 32 min 51 sec]
Remaining rsync sessions include:
   root 13217834 13279464 120 10:10:20
                                         - 15:31 rsync -avz /dba/
   oradata_dm_15/alpha-rsync:/dba/oradata_d1_15
   root 13869244 13893768 120 10:10:20 - 15:15 rsync -avz /dba/
   oradata_dm_15/ bravo-rsync:/dba/oradata_d1_15
   root 14028924 14758450 120 10:10:20 - 39:51 rsync -avz /dba/
   oradata_dm_17/ alpha-rsync:/dba/oradata_d1_17
   root 27713858 14320212 111 10:10:20 - 39:43 rsync -avz /dba/
   oradata_dm_18/ alpha-rsync:/dba/oradata_d1_18
   root 28176732 13754940 102 10:10:20 - 27:42 rsync -avz /dba/
  oradata_dm_17/ bravo-rsync:/dba/oradata_d1_17
   root 13873778 19423708 92 10:10:20 - 27:34 rsync -avz /dba/
   oradata_dm_18/ bravo-rsync:/dba/oradata_d1_18
      6 of 36 rsync sessions remaining [Elapsed time: 1 hr 37 min 54 sec]
Remaining rsync sessions include:
   root 13217834 13279464 93 10:10:20
                                         - 18:24 rsync -avz /dba/
   oradata_dm_15/ alpha-rsync:/dba/oradata_d1_15
   root 13869244 13893768 90 10:10:20 - 18:01 rsync -avz /dba/
   oradata_dm_15/ bravo-rsync:/dba/oradata_d1_15
   root 14028924 14758450 105 10:10:20 - 43:36 rsync -avz /dba/
   oradata_dm_17/ alpha-rsync:/dba/oradata_d1_17
   root 27713858 14320212 120 10:10:20 - 43:25 rsync -avz /dba/
   oradata_dm_18/ alpha-rsync:/dba/oradata_d1_18
   root 28176732 13754940 120 10:10:20 - 31:14 rsync -avz /dba/
  oradata_dm_17/ bravo-rsync:/dba/oradata_d1_17
   root 13873778 19423708 98 10:10:20 - 31:06 rsync -avz /dba/
   oradata_dm_18/ bravo-rsync:/dba/oradata_d1_18
....wrote 7183466709 bytes read 36 bytes 1163785.62 bytes/sec
total size is 20971552768 speedup is 2.92
real 1h42m52.05s
user 45m59.38s
sys 0m19.95s
wrote 7174332611 bytes read 36 bytes 1162117.54 bytes/sec
total size is 20971552768 speedup is 2.92
```

```
real 1h42m53.63s
user 45m51.40s
sys 0m20.11s
      4 of 36 rsync sessions remaining [Elapsed time: 1 hr 42 min 56 sec]
Remaining rsync sessions include:
   root 13217834 13279464 107 10:10:20
                                         - 21:20 rsync -avz /dba/
   oradata_dm_15/ alpha-rsync:/dba/oradata_d1_15
   root 13869244 13893768 85 10:10:20
                                         - 20:52 rsync -avz /dba/
   oradata_dm_15/ bravo-rsync:/dba/oradata_d1_15
   root 28176732 13754940 115 10:10:20 - 34:51 rsync -avz /dba/
   oradata_dm_17/ bravo-rsync:/dba/oradata_d1_17
   root 13873778 19423708 114 10:10:20
                                         - 34:41 rsync -avz /dba/
   oradata_dm_18/ bravo-rsync:/dba/oradata_d1_18
      4 of 36 rsync sessions remaining [Elapsed time: 1 hr 47 min 58 sec]
Remaining rsync sessions include:
   root 13217834 13279464 99 10:10:20 - 24:34 rsync -avz /dba/
   oradata_dm_15/ alpha-rsync:/dba/oradata_d1_15
   root 13869244 13893768 80 10:10:20
                                       - 24:02 rsync -avz /dba/
   oradata_dm_15/ bravo-rsync:/dba/oradata_d1_15
   root 28176732 13754940 88 10:10:20 - 38:24 rsync -avz /dba/
   oradata_dm_17/ bravo-rsync:/dba/oradata_d1_17
   root 13873778 19423708 117 10:10:20 - 38:16 rsync -avz /dba/
   oradata_dm_18/ bravo-rsync:/dba/oradata_d1_18
wrote 1715967444 bytes read 36 bytes 263165.10 bytes/sec
total size is 22020128768 speedup is 12.83
real 1h48m40.22s
user 24m47.92s
sys 0m13.87s
.wrote 1715967444 bytes read 36 bytes 260607.10 bytes/sec
total size is 22020128768 speedup is 12.83
real 1h49m43.92s
user 24m49.36s
sys 0m13.95s
. . . .
      2 of 36 rsync sessions remaining [Elapsed time: 1 hr 53 min 0 sec]
Remaining rsync sessions include:
   root 28176732 13754940 82 10:10:20 - 41:59 rsync -avz /dba/
   oradata_dm_17/ bravo-rsync:/dba/oradata_d1_17
   root 13873778 19423708 120 10:10:20 - 41:50 rsync -avz /dba/
```

Listing 7-23 (continued)

```
oradata_dm_18/ bravo-rsync:/dba/oradata_d1_18
      2 of 36 rsync sessions remaining [Elapsed time: 1 hr 58 min 3 sec]
Remaining rsync sessions include:
                                          - 45:38 rsync -avz /dba/
   root 28176732 13754940 71 10:10:20
   oradata dm 17/ bravo-rsync:/dba/oradata d1 17
    root 13873778 19423708 120 10:10:20
                                          - 45:31 rsync -avz /dba/
   oradata_dm_18/ bravo-rsync:/dba/oradata_d1_18
....wrote 7183466709 bytes read 36 bytes 977010.10 bytes/sec
total size is 20971552768 speedup is 2.92
real 2h2m31.79s
user 47m36.77s
sys 0m21.09s
wrote 7174332611 bytes read 36 bytes 974839.68 bytes/sec
total size is 20971552768 speedup is 2.92
real 2h2m39.11s
user 47m35.59s
sys 0m21.33s
      0 of 36 rsync sessions remaining [Elapsed time: 2 hr 3 min 5 sec]
Remaining rsync sessions include:
...Local rsync processing completed on gamma...Thu Oct 25 12:13:25
  EDT 2007
... Checking remote target machines: alpha-rsync bravo-rsync...
... Remote rsync processing completed Thu Oct 25 12:13:32 EDT 2007
Checking File: /dba/oradata_dm_01/dataset_master_01.dbf
Local gamma size: 22020128768
Checking Remote File: /dba/oradata_d1_01/dataset_master_01.dbf
alpha-rsync size: 22020128768
Checking File: /dba/oradata_dm_02/dataset_master_02.dbf
Local gamma size: 22544416768
Checking Remote File: /dba/oradata_d1_02/dataset_master_02.dbf
alpha-rsync size: 22544416768
Checking File: /dba/oradata_dm_03/dataset_master_03.dbf
Local gamma size: 22544416768
Checking Remote File: /dba/oradata_d1_03/dataset_master_03.dbf
alpha-rsync size: 22544416768
```

Listing 7-23 (continued)

Checking File: /dba/oradata_dm_04/dataset_master_04.dbf Local gamma size: 22020128768 Checking Remote File: /dba/oradata_d1_04/dataset_master_04.dbf alpha-rsync size: 22020128768 Checking File: /dba/oradata_dm_05/dataset_master_05.dbf Local gamma size: 22020128768 Checking Remote File: /dba/oradata_d1_05/dataset_master_05.dbf alpha-rsync size: 22020128768 Checking File: /dba/oradata_dm_06/dataset_master_06.dbf Local gamma size: 22544416768 Checking Remote File: /dba/oradata_d1_06/dataset_master_06.dbf alpha-rsync size: 22544416768 Checking File: /dba/oradata_dm_07/dataset_master_07.dbf Local gamma size: 22020128768 Checking Remote File: /dba/oradata_d1_07/dataset_master_07.dbf alpha-rsync size: 22020128768 Checking File: /dba/oradata_dm_08/dataset_master_08.dbf Local gamma size: 22020128768 Checking Remote File: /dba/oradata_d1_08/dataset_master_08.dbf alpha-rsync size: 22020128768 Checking File: /dba/oradata_dm_09/dataset_master_09.dbf Local gamma size: 21495840768 Checking Remote File: /dba/oradata_d1_09/dataset_master_09.dbf alpha-rsync size: 21495840768 Checking File: /dba/oradata_dm_10/dataset_master_10.dbf Local gamma size: 21495840768 Checking Remote File: /dba/oradata_d1_10/dataset_master_10.dbf alpha-rsync size: 21495840768 Checking File: /dba/oradata_dm_11/dataset_master_11.dbf Local gamma size: 21495840768 Checking Remote File: /dba/oradata_d1_11/dataset_master_11.dbf alpha-rsync size: 21495840768 Checking File: /dba/oradata_dm_12/dataset_master_12.dbf Local gamma size: 21495840768 Checking Remote File: /dba/oradata_d1_12/dataset_master_12.dbf alpha-rsync size: 21495840768 Checking File: /dba/oradata_dm_13/dataset_master_13.dbf Local gamma size: 22020128768 Checking Remote File: /dba/oradata_d1_13/dataset_master_13.dbf alpha-rsync size: 22020128768 Checking File: /dba/oradata_dm_14/dataset_master_14.dbf Local gamma size: 22020128768 Checking Remote File: /dba/oradata_d1_14/dataset_master_14.dbf alpha-rsync size: 22020128768 Checking File: /dba/oradata_dm_15/dataset_master_15.dbf Local gamma size: 22020128768 Checking Remote File: /dba/oradata_d1_15/dataset_master_15.dbf alpha-rsync size: 22020128768

Listing 7-23 (continued)

Checking File: /dba/oradata_dm_16/dataset_master_16.dbf Local gamma size: 22020128768 Checking Remote File: /dba/oradata_d1_16/dataset_master_16.dbf alpha-rsync size: 22020128768 Checking File: /dba/oradata_dm_17/oradata/iddata_master_01.dbf Local gamma size: 20971552768 Checking Remote File: /dba/oradata_d1_17/oradata/iddata_master_01.dbf alpha-rsync size: 20971552768 Checking File: /dba/oradata_dm_18/oradata/iddata_master_02.dbf Local gamma size: 20971552768 Checking Remote File: /dba/oradata_d1_18/oradata/iddata_master_02.dbf alpha-rsync size: 20971552768 Checking File: /dba/oradata_dm_01/dataset_master_01.dbf Local gamma size: 22020128768 Checking Remote File: /dba/oradata_d1_01/dataset_master_01.dbf bravo-rsync size: 22020128768 Checking File: /dba/oradata_dm_02/dataset_master_02.dbf Local gamma size: 22544416768 Checking Remote File: /dba/oradata_d1_02/dataset_master_02.dbf bravo-rsync size: 22544416768 Checking File: /dba/oradata_dm_03/dataset_master_03.dbf Local gamma size: 22544416768 Checking Remote File: /dba/oradata_d1_03/dataset_master_03.dbf bravo-rsync size: 22544416768 Checking File: /dba/oradata_dm_04/dataset_master_04.dbf Local gamma size: 22020128768 Checking Remote File: /dba/oradata_d1_04/dataset_master_04.dbf bravo-rsync size: 22020128768 Checking File: /dba/oradata_dm_05/dataset_master_05.dbf Local gamma size: 22020128768 Checking Remote File: /dba/oradata_d1_05/dataset_master_05.dbf bravo-rsync size: 22020128768 Checking File: /dba/oradata_dm_06/dataset_master_06.dbf Local gamma size: 22544416768 Checking Remote File: /dba/oradata_d1_06/dataset_master_06.dbf bravo-rsync size: 22544416768 Checking File: /dba/oradata_dm_07/dataset_master_07.dbf Local gamma size: 22020128768 Checking Remote File: /dba/oradata_d1_07/dataset_master_07.dbf bravo-rsync size: 22020128768 Checking File: /dba/oradata_dm_08/dataset_master_08.dbf Local gamma size: 22020128768 Checking Remote File: /dba/oradata_d1_08/dataset_master_08.dbf bravo-rsync size: 22020128768 Checking File: /dba/oradata_dm_09/dataset_master_09.dbf Local gamma size: 21495840768 Checking Remote File: /dba/oradata_d1_09/dataset_master_09.dbf bravo-rsync size: 21495840768

Listing 7-23 (continued)

```
Checking File: /dba/oradata_dm_10/dataset_master_10.dbf
Local gamma size: 21495840768
Checking Remote File: /dba/oradata_d1_10/dataset_master_10.dbf
bravo-rsync size: 21495840768
Checking File: /dba/oradata_dm_11/dataset_master_11.dbf
Local gamma size: 21495840768
Checking Remote File: /dba/oradata_d1_11/dataset_master_11.dbf
bravo-rsync size: 21495840768
Checking File: /dba/oradata_dm_12/dataset_master_12.dbf
Local gamma size: 21495840768
Checking Remote File: /dba/oradata_d1_12/dataset_master_12.dbf
bravo-rsync size: 21495840768
Checking File: /dba/oradata_dm_13/dataset_master_13.dbf
Local gamma size: 22020128768
Checking Remote File: /dba/oradata_d1_13/dataset_master_13.dbf
bravo-rsync size: 22020128768
Checking File: /dba/oradata_dm_14/dataset_master_14.dbf
Local gamma size: 22020128768
Checking Remote File: /dba/oradata_d1_14/dataset_master_14.dbf
bravo-rsync size: 22020128768
Checking File: /dba/oradata_dm_15/dataset_master_15.dbf
Local gamma size: 22020128768
Checking Remote File: /dba/oradata_d1_15/dataset_master_15.dbf
bravo-rsync size: 22020128768
Checking File: /dba/oradata_dm_16/dataset_master_16.dbf
Local gamma size: 22020128768
Checking Remote File: /dba/oradata_d1_16/dataset_master_16.dbf
bravo-rsync size: 22020128768
Checking File: /dba/oradata_dm_17/oradata/iddata_master_01.dbf
Local gamma size: 20971552768
Checking Remote File: /dba/oradata_d1_17/oradata/iddata_master_01.dbf
bravo-rsync size: 20971552768
Checking File: /dba/oradata_dm_18/oradata/iddata_master_02.dbf
Local gamma size: 20971552768
Checking Remote File: /dba/oradata_d1_18/oradata/iddata_master_02.dbf
bravo-rsync size: 20971552768
SUCCESS: Rsync copy completed successfully ...
All file sizes match...
Rsync copy from gamma to machines alpha-rsync bravo-rsync
completed successfully Thu Oct 25 12:13:40 EDT 2007
Rsync copy completed Thu Oct 25 12:13:40 EDT 2007
[[ rsync_daily_copy.ksh completed execution Thu Oct 25 12:13:40
   EDT 2007 ]]
```

Listing 7-23 (continued)

This is an example of a successful rsync copy process. You can easily follow the process of checking for other copies of this script executing, monitoring for the startup file, verifying the DAY variable, pinging the destination servers, starting the replication process, monitoring the rsync processes, and giving the user feedback on the progress of the copy process. After all local rsync sessions have completed, we query the OLTP server process tables remotely for any leftover rsync sessions. Although very unlikely, I still like to check. The last step verifies the file sizes are equal on each server, and then sends notification to that effect.

Summary

I hope you picked up some tips and tricks in this chapter. As with any scripts we write, there is always room for improvement. Using rsync is a smart method to replicate data because the rsync checksum search algorithm is so fast and accurate. You have a number of options, including using compression, limiting bandwidth, and using the rsyncd protocol to replicate data through specific TCP ports.

To get the most out of rsync, you need to study the manual page in detail. You can send the man page directly to the printer with the following command:

man rsync | lp -d queue_name

where queue_name is a defined print queue on your system.

We used remote shell (rsh) for this data transfer. Depending on the sensitivity of your data, you may want to use Open Secure Shell (OpenSSH) so that the data is encrypted, and set up the encryption keys so that a password is not required. You can download OpenSSH from www.openssh.org.

We will use rsync again in Chapter 27, "Using Dirvish with rsync to Create Snapshot-Type Backups." Dirvish is a wrapper program around rsync that does snapshot-type backups. This is a powerful and efficient backup method, so be sure to check it out.

In the next chapter we will look automating interactive programs with expect and using autoexpect to easily, and automatically, create an expect script to interact with a program.

Lab Assignments

 Rewrite the shell script in Listing 7-1 so that the source file/directory and destination file/directory are specified on the command line as ARG1 and ARG2, respectively.

Example: generic_rsync.Bash yogi:/scripts//scripts

2. Write two shell scripts. Shell script A should do an archive-compressed rsync copy of some large directory structure to a remote system. Shell script B should rsync a tarred-compressed file of the same directory structure (somedir. tar.Z) to the remote machine.

Is it faster to just let rsync do the compression, or does the compressed file transfer faster with a higher level of compression? Time each method. NOTE: Do not store the tarred-compressed file in the same directory structure. That would change the results.

CHAPTER 8

Automating Interactive Programs with Expect and Autoexpect

Need a robot to take care of a few tasks for you every day in your environment? If so, Expect just may be your answer. Expect is a programming language that "talks" with an interactive program or script that requires user interaction for input. Expect works by *expecting* input, and upon receiving the expected input, the Expect script will *send* a response, just like magic.

To make it easy to create Expect scripts, and to help users learn how to use Expect to interact with a program, the programmers came up with the autoexpect command. By typing **autoexpect** and pressing Enter, we can create the script automatically by just interacting with the program, *without making a mistake*, as we normally would, and save the interaction in an Expect script file. When we finish interacting with the program, we simply press Ctrl+D to save the new Expect script. The target command or application can also be supplied as an argument to autoexpect, and Expect will *spawn* the process for us. We will get to more details in the coming pages. For now, just relax and let's get started.

Downloading and Installing Expect

Expect requires Tcl to be installed before Expect will work to interact with a program. If you need to install Tcl, you can download the code from http://www.tcl.tk/ software/tcltk/. The simplest method of installing Tcl and Expect is to use yum or apt to install the file sets from one of many mirror sites around the world. You can also go to the official Expect web page, http://expect.nist.gov/, and download the source code to compile on your system.

I like using yum to install packages on my Linux machine. Listing 8-1 shows a typical yum installation for Tcl and Expect.

```
[root@localhost scripts] # yum install expect
Loading "installonlyn" plugin
Setting up Install Process
Parsing package install arguments
updates
                  100%
|=======| 1.9 kB 00:00
primary.sqlite.bz2 100%
freshrpms
                  100%
|======| 1.9 kB
                         00:00
primary.sglite.bz2 100%
100%
fedora
primary.sqlite.bz2 100%
Resolving Dependencies
--> Running transaction check
---> Package expect.i386 0:5.43.0-8 set to be updated
---> Package expect.x86_64 0:5.43.0-8 set to be updated
--> Processing Dependency: libtcl8.4.so()(64bit) for package: expect
--> Processing Dependency: libtcl8.4.so for package: expect
--> Restarting Dependency Resolution with new changes.
--> Running transaction check
---> Package tcl.i386 1:8.4.13-16.fc7 set to be updated
---> Package tcl.x86_64 1:8.4.13-16.fc7 set to be updated
---> Package expect.i386 0:5.43.0-8 set to be updated
---> Package expect.x86_64 0:5.43.0-8 set to be updated
Dependencies Resolved
Arch Version Repository Size
Package
Installing:
             i386
                    5.43.0-8 fedora
                                           262 k
expect
             x86_64 5.43.0-8
                                fedora
                                           263 k
expect
Installing for dependencies:
             i386 1:8.4.13-16.fc7 fedora
                                           1.8 M
t.c1
             x86_64
tcl
                    1:8.4.13-16.fc7 fedora
                                           1.8 M
Transaction Summary
_____
Install 4 Package(s)
Update
       0 Package(s)
Remove 0 Package(s)
Total download size: 4.1 M
```

Listing 8-1 Installing Tcl and Expect with yum

```
Is this ok [y/N]: y
Downloading Packages:
(1/4): expect-5.43.0-8.x8 100%
|=======| 263 kB 00:00
(2/4): expect-5.43.0-8.i3 100%
|=======| 262 kB 00:00
(3/4): tcl-8.4.13-16.fc7. 100%
|======| 1.8 MB 00:02
(4/4): tcl-8.4.13-16.fc7. 100%
|======| 1.8 MB 00:02
Running Transaction Test
Finished Transaction Test
Transaction Test Succeeded
Running Transaction
 Installing: tcl
                             Installing: tcl
                             Installing: expect
                             Installing: expect
                             Installed: expect.i386 0:5.43.0-8 expect.x86_64 0:5.43.0-8
Dependency Installed: tcl.i386 1:8.4.13-16.fc7
tcl.x86_64 1:8.4.13-16.fc7
Complete!
```

Listing 8-1 (continued)

As you can see in Listing 8-1, yum found the Tcl requirement for Expect and installed the required Tcl package before installing Expect. For more information on setting up yum or apt, see the manual pages, man yum and man apt, respectively.

The Basics of Talking to an Interactive Script or Program

With Expect and Tcl installed, we can now start *talking* to our interactive programs and scripts.

NOTE For our discussion, I will refer to the combined installation of Tcl and Expect as *Expect* in this chapter.

To get started, we need to look at the basic commands that Expect uses to interact with a program or script:

The expect command is used to wait for specific output from a process or program, just as read waits for input in a shell script. The -exact command switch tells Expect to recognize only the *exact* specified output, including spaces, line feeds, and carriage returns. This is why I like to say that when we program in Expect, *we should not expect too much*. If you are running an Expect script and it hangs on one step, it may be caused by the use of the -exact switch.

- We use the send command to send a reply to a process or program. This works like echo in our shell scripts. You must specify the proper cursor control when talking with an interactive application, using \r for carriage return and \n for each new line. Carriage returns and line feeds are not automatically implied.
- We use the spawn command to start up a process or application, such as the default shell, ftp, telnet, ssh, rcp, and so on. The -noecho switch suppresses screen output of the spawned command. This is useful when you do not want usernames, passwords, or other output displayed on the screen.
- Then we have the interact command, which enables us to program a predefined user interaction point in our Expect scripts. For example, we might want have an Expect script that will log us in to an FTP site and then turn control over to the end user.

Table 8-1 summarizes the most basic Expect commands.

COMMAND	ACTION
spawn	Starts a process or program
send	Sends a reply to a process or program
expect	Waits for output from a process or program
interact	Allows you in interact with a process or program

Table 8-1 Expect Commands

I wrote a simple Bash shell script that asks questions, and an Expect script that answers the questions automatically. The lunch_time.Bash shell script asks Bob if he wants to go to lunch. The talk_back.exp Expect script sends the replies to the Bash script. Listing 8-2 shows the lunch_time.Bash shell script.

```
#!/bin/Bash
#
# SCRIPT: lunch_time.Bash
#
# This script is used to test an Expect script
#
echo "Hey Bob, you in?"
read REPLY
echo "Have you eaten yet?"
read REPLY
echo "You want to?"
read REPLY
echo "How about that Italian place?"
read REPLY
echo "Meet you in the lobby"
read REPLY
```

Listing 8-2 Simple Bash script to talk with Expect

As you can see in Listing 8-2, we are not testing any of the replies. Any reply will cause the script to ask the next question. Listing 8-3 shows the talk_back.exp Expect script.

```
#!/usr/local/bin/expect -f
# SCRIPT: talk_back.exp
# This is an example Expect script starting up and
# talking to a Bash script.
set timeout -1
spawn $env(SHELL)
expect "\r*"
send -- "./lunch_time.Bash\r"
expect "lunch_time.Bash\r"
expect "Hey Bob, you in?\r"
send -- "Yes\r"
expect -exact "Yes\r"
expect "Have you eaten yet?\r"
send -- "No\r"
expect "You want to?\r"
send -- "Where at?\r"
expect "How about that Italian place?\r"
send -- "Works for me\r"
expect "Meet you in the lobby\r"
send -- "Groovy man\r"
expect -exact "Groovy man\r"
expect eof
```

Listing 8-3 Simple Expect script that starts and talks to a Bash script

Let's look at the Expect script in Listing 8-3 from the top. As with all scripts, we need to define the interpreter on the very first line. Normally, we need to define the executing shell, but in this case #!/usr/local/bin/expect specifies the path to Expect. The same is true for Perl scripts.

NOTE If you get a *bad interpreter* error, you have defined the wrong path to Expect, or Expect is not installed. Expect resides in /usr/bin/expect or /usr/local/bin/expect.

On the first line of code, we disable timeouts by setting the Expect timeout variable to -1. Next we start the default UNIX shell with the command spawn \$env(SHELL). You can also use spawn to initiate scripts, programs, ftp, telnet, and other interactive programs. Next we send to the shell the command ./lunch_time.Bash, which starts our interactive Bash script. The remaining parts of the talk_back.exp Expect script are the interactions with the lunch_time.Bash shell script. Notice that we end the Expect script with expect eof. This last Expect statement tells Expect the end of file is expected and the script ends. Listing 8-4 shows both scripts interacting.

[root@yogi scripts]# ./talk_back.exp	Command-line starting		
	Expect script		
spawn /bin/Bash	Expect spawning the		
	default UNIX shell		
[root@yogi scripts]# ./lunch_time.Bash	Expect starting Bash		
	shell script		
Hey Bob, you in?	Question from Bash		
	script		
Yes	Expect script reply		
Have you eaten yet?	Question from Bash		
	script		
No	Expect script reply		
You want to?	Question from Bash		
	script		
Where at?	Expect script reply		
How about that Italian place?	Question from Bash		
	script		
Works for me	Expect script reply		
Meet you in the lobby	Question from Bash		
	script		
Groovy man	Expect script reply		
[root@yogi scripts]#	Command prompt		

Listing 8-4 Example of an Expect script interacting with a Bash shell script

Listing 8-4 shows the how the interaction takes place between the Expect script and the Bash shell script. On the command line, we started the Expect script talk_back.exp located in the current directory. On the next line, Expect spawns a new default shell, which is Bash in this case.

NOTE We can also spawn the Bash shell script directly with spawn ./lunch_time.Bash, instead of first spawning a default shell and using send

"./lunch_time.Bash\r" to start the Bash shell script.

Next the Expect script starts the Bash shell script. Had we added the -noecho switch to the spawn command, this particular output would have been suppressed. Once the Bash script starts asking questions, the Expect script responds. You can see the interaction descriptions on the right side of Listing 8-4.

In this example we had the Expect script start the Bash shell script, but we could just as easily have had the Bash shell script start the Expect script. It just depends on the particular application.

Using autoexpect to Automatically Create an Expect Script

To create an Expect script quickly and automatically, we can use autoexpect. autoexpect works the same way the **script** command works. We enter **autoexpect** on the command line, and a new script-type session is started that saves all the keystrokes

in a file until the application ends. Or we can press Ctrl+D to end the autoexpect session. The new Expect script is saved to the default filename script.exp, and it is saved in the current directory. You can also have autoexpect spawn the process for you by specifying the name on the command line, as follows:

```
# autoexpect ftp yogi
```

This command spawns an FTP session to the server yogi and starts an autoexpect session to save all the keystrokes. Using this technique, let's use autoexpect to create the new Expect script, and we will look at the differences in the script autoexpect creates and the script in Listing 8-4. Listing 8-5 shows how to create the new Expect script using autoexpect.

```
[root@yogi scripts]# autoexpect ./lunch_time.Bash
autoexpect started, file is script.exp Autoexpect session
                                      started
Hey Bob, you in?
                                      Bash script question
                                      Command-line reply
Yes
                                      Bash script question
Have you eaten yet?
                                      Command-line reply
No
                                      Bash script question
You want to?
Where at?
                                      Command-line reply
How about that Italian place?
                                     Bash script question
Works for me
                                      Command-line reply
Meet you in the lobby
                                      Bash script question
Groovy man
                                       Command-line reply
autoexpect done, file is script.exp
                                     Bash scripts ends and
                                       autoexpect saves file
[root@yogi scripts]#
                                       Command prompt
```

Listing 8-5 Example of using autoexpect

Notice in Listing 8-5 that we specified the interactive program or script on the command line, and then there is the interaction between the Bash script and my typing on the command line to reply. When the Bash script ends, the autoexpect session automatically ends and the new Expect script is saved in the file script.exp located in the current directory. So, don't forget to rename the file to something meaningful.

Now let's look at the Expect script that autoexpect created, as shown in Listing 8-6.

```
#!/usr/local/bin/expect -f
#
# This Expect script was generated by autoexpect on Wed
Dec 26 15:16:05 2007
# Expect and autoexpect were both written by Don Libes, NIST.
#
```

Listing 8-6 Expect script created with autoexpect

```
# Note that autoexpect does not guarantee a working script. It
# necessarily has to guess about certain things. Two reasons a script
# might fail are:
# 1) timing - A surprising number of programs (rn, ksh, zsh, telnet,
# etc.) and devices discard or ignore keystrokes that arrive "too
# quickly" after prompts. If you find your new script hanging up at
# one spot, try adding a short sleep just before the previous send.
# Setting "force_conservative" to 1 (see below) makes Expect do this
# automatically - pausing briefly before sending each character. This
# pacifies every program I know of. The -c flag makes the script do
# this in the first place. The -C flag allows you to define a
# character to toggle this mode off and on.
set force_conservative 0 ;# set to 1 to force conservative mode even if
                      ;# script wasn't run conservatively originally
if {$force_conservative} {
     set send_slow {1 .1}
     proc send {ignore arg} {
            sleep .1
            exp_send -s -- $arg
     }
}
# 2) differing output - Some programs produce different output each time
# they run. The "date" command is an obvious example. Another is
# ftp, if it produces throughput statistics at the end of a file
# transfer. If this causes a problem, delete these patterns or replace
# them with wildcards. An alternative is to use the -p flag (for
# "prompt"), which makes Expect only look for the last line of output
# (i.e., the prompt). The -P flag allows you to define a character to
# toggle this mode off and on.
# Read the man page for more info.
# -Don
set timeout -1
spawn ./lunch_time.Bash
match_max 100000
expect -exact "Hey Bob, you in?\r
send -- "Yes\r"
expect -exact "Yes\r
Have you eaten yet?\r
```

Listing 8-6 (continued)

```
send -- "No\r"
expect -exact "No\r
You want to?\r
"
send -- "Where at?\r"
expect -exact "Where at?\r
How about that Italian place?\r
"
send -- "Works for me\r"
expect -exact "Works for me\r
Meet you in the lobby\r
"
send -- "Groovy man\r"
expect eof
```

Listing 8-6 (continued)

As shown in Listing 8-6, autoexpect adds a lot of text at the top of the Expect script. One thing to notice is the setting for force_conservative mode. Many programs and applications will ignore any data that arrives too quickly. This can be a problem and is one cause of an Expect script hanging. If you set force_conservative to 1, Expect will pause briefly before sending each reply. Of course, this can slow down the script's execution time.

We need to remember that autoexpect *does not guarantee* the new Expect script will work. There are many commands that use timestamps in output or show transfer statistics, as is the case with ftp, and as these values change, the Expect script can fail or hang. If you get this type of output when using autoexpect, either delete the data that changes from the Expect script or use wildcards to represent the data. We can also use the -p (for prompt) autoexpect switch to have Expect look only at the last line of output, which may be the command prompt. You can also add the -P switch to specify a character to toggle this mode on and off. For example, use $-P \land P$ to specify Control+P as the toggle character.

Speaking of file-transfer statistics, let's take a look at another Expect script created by autoexpect. The goal of the ftp-get-file.exp Expect script is to ftp a file named random_file.out, located on the remote server named yogi in the /scripts directory, to the current directory on the local machine. The username for this file transfer is prodmgr and the password on yogi is Abc1234. Check out how we use autoexpect to create the ftp-get-file.exp Expect script in Listing 8-7.

```
[root@booboo ~]# autoexpect ftp yogi
autoexpect started, file is script.exp
Connected to yogi (192.168.1.100).
220 yogi FTP server (Version 4.1 Sat Feb 23 00:11:36 CST 2002) ready.
Name (yogi:root): prodmgr
331 Password required for prodmgr.
```

Listing 8-7 Using autoexpect to create the ftp-get-file.exp Expect script

```
Password:
230 User prodmgr logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd /scripts
250 CWD command successful.
ftp> get random_file.out
local: random_file.out remote: random_file.out
227 Entering Passive Mode (192,168,1,100,128,45)
150 Opening data connection for random_file.out (2185 bytes).
226 Transfer complete.
2185 bytes received in 0.0149 secs (1.4e+02 Kbytes/sec)
ftp> bye
221 Goodbye.
autoexpect done, file is script.exp
```

Listing 8-7 (continued)

We begin the autoexpect session by specifying ftp yogi on the command line. As you will see in the ftp-get-file.exp Expect script, this causes Expect to spawn a new ftp yogi process instead of spawning a new default shell, as we did earlier when we just typed **autoexpect** on the command line and pressed Enter, and did not specify the program to interact with on the command line.

Next we supply the username and password to get logged in. Now we change directory to /scripts, where the random_file.out file resides. The next command is get random_file.out, which transfers the file to the local server and current directory. Let's look at the code that autoexpect created in Listing 8-8.

```
#!/usr/bin/expect -f
#
# This Expect script was generated by autoexpect on Wed
Dec 26 08:24:28 2007
# Expect and autoexpect were both written by Don Libes, NIST.
#
# Note that autoexpect does not guarantee a working script. It
# necessarily has to guess about certain things. Two reasons a script
# might fail are:
#
# 1) timing - A surprising number of programs (rn, ksh, zsh, telnet,
# etc.) and devices discard or ignore keystrokes that arrive "too
# quickly" after prompts. If you find your new script hanging up at
# one spot, try adding a short sleep just before the previous send.
# Setting "force_conservative" to 1 (see below) makes Expect do this
# automatically - pausing briefly before sending each character. This
# pacifies every program I know of. The -c flag makes the script do
# this in the first place. The -C flag allows you to define a
```

Listing 8-8 Expect script to FTP a file that autoexpect created

```
# character to toggle this mode off and on.
set force_conservative 0 ;# set to 1 to force conservative mode even if
                      ;# script wasn't run conservatively originally
if {$force_conservative} {
    set send_slow {1 .1}
     proc send {ignore arg} {
            sleep .1
            exp_send -s -- $arg
     }
}
# 2) differing output - Some programs produce different output each time
# they run. The "date" command is an obvious example. Another is
# ftp, if it produces throughput statistics at the end of a file
# transfer. If this causes a problem, delete these patterns or replace
# them with wildcards. An alternative is to use the -p flag (for
# "prompt"), which makes Expect only look for the last line of output
# (i.e., the prompt). The -P flag allows you to define a character to
# toggle this mode off and on.
# Read the man page for more info.
#
# -Don
set timeout -1
spawn ftp yogi
match_max 100000
expect -exact "Connected to yogi (192.168.1.100).\r
220 yogi FTP server (Version 4.1 Sat Feb 23 00:11:36 CST 2002) ready.\r
Name (yogi:root): "
send -- "prodmgr\r"
expect -exact "prodmgr\r
331 Password required for prodmgr.\r
Password:"
send -- "Abc1234\r"
expect -exact "\r
230 User prodmgr logged in.\r
Remote system type is UNIX.\r
Using binary mode to transfer files.\r
ftp> "
send -- "cd /scripts\r"
expect -exact "cd /scripts\r
250 CWD command successful.\r
ftp> "
send -- "get random_file.out"
```

Listing 8-8 (continued)

```
expect -exact "random_file.out"
send -- "\r"
expect -exact "\r
local: random_file.out remote: random_file.out\r
227 Entering Passive Mode (192,168,1,100,128,45)\r
150 Opening data connection for random_file.out (2185 bytes).\r
226 Transfer complete.\r
2185 bytes received in 0.0149 secs (1.4e+02 Kbytes/sec)\r
ftp> "
send -- "bye\r"
expect eof
```

Listing 8-8 (continued)

How much of the code in Listing 8-8 is *really* needed? As it turns out, not a whole lot. Let's strip out the unneeded code that autoexpect created to streamline this Expect script. Check out the new Expect script in Listing 8-9 and we will cover the details at the end.

```
#!/usr/bin/expect -f
#
# SCRIPT: ftp-get-file.exp
#
set force_conservative 0 ;# set to 1 to force conservative mode even if
                      ;# script wasn't run conservatively originally
if {$force_conservative} {
    set send_slow {1 .1}
     proc send }ignore arg} {
            sleep .1
            exp_send -s -- $arg
     }
}
set timeout -1
spawn ftp yogi
match_max 100000
expect "Name *: "
send -- "prodmgr\r"
expect "Password:"
send -- "Abc1234\r"
expect "ftp>"
send -- "cd /scripts\r"
expect "ftp>"
```

Listing 8-9 New ftp-get-file.exp Expect script

```
send -- "get random_file.out\r"
expect "ftp>"
send -- "bye\r"
expect eof
```

Listing 8-9 (continued)

In Listing 8-9 we removed all the "extra stuff" and only test for, or *expect*, specific prompts. We use each of these prompts as signal points that the program's interaction requires more input. Notice that expect -exact is not used anywhere in the ftp-get-file.exp Expect script.

If we initially create an Expect script using autoexpect to save the keystrokes, we can easily strip out the extra output and extra send data, thus limiting the recognition to specific prompts and saving us a lot of trouble trying to match varying output. The new ftp-get-file.exp Expect script is shown in action in Listing 8-10.

```
[root@booboo ~]# ./ftp-get-file.exp
spawn ftp yogi
Connected to yogi (192.168.1.100).
220 yogi FTP server (Version 4.1 Sat Feb 23 00:11:36 CST 2002) ready.
Name (yogi:root): prodmgr
331 Password required for prodmgr.
Password:
230 User root logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd /scripts
250 CWD command successful.
ftp> get random_file.out
local: random_file.out remote: random_file.out
227 Entering Passive Mode (192,168,1,100,128,46)
150 Opening data connection for random_file.out (2185 bytes).
226 Transfer complete.
2185 bytes received in 0.00364 secs (5.9e+02 Kbytes/sec)
ftp> bye
221 Goodbye.
[root@booboo ~]#
```

Listing 8-10 ftp-get-file.exp shell script in action

In Listing 8-10 we have six instances where we must supply input to the ftp process. We first spawn the new ftp session to the server yogi. Next we supply the username and password to get logged in. Once we are logged in to the system we change to the /scripts directory and execute the get random_file.out command to retrieve the remote file. After the transfer we again get the ftp> prompt and send bye to end the ftp session. At each step, notice how we use the specific known prompt as a signal point to provide more input.

Working with Variables

Our little ftp Expect script works okay, but what about letting us specify the remote host and file on the command line as arguments to the Expect script? This is easy using the following technique.

The Expect command set is used to define variables and assign values. For example, to assign the value 5 to the variable VAR, we use the following syntax:

```
set VAR 5
```

To access the variable, we use the same technique as we do in a normal shell script — add a \$ in front of the variable name (\$VAR).

To define command-line arguments to Expect script variables, we use the following syntax:

set VAR [lindex \$argv 0]

The set command defines the variable VAR as a pointer to the first command-line argument (\$argv 0). For our ftp shell script, we want to define the first command-line argument as the remote host, rem_host. The second command-line argument is assigned to the remote directory that contains the file, rem_dir, and the third command-line argument is assigned to the remote file, rem_file, variable. In our case we make these definitions using the following declarations:

```
set rem_host [lindex $argv 0]
set rem_dir [lindex $argv 1]
set rem_file [lindex $argv 2]
```

With these new definitions, let's modify our ftp-get-file.exp script to allow us to create a more flexible Expect script. Check out the code in Listing 8-11 and we will study the details at the end.

```
#!/usr/local/bin/expect -f
#
# SCRIPT: ftp-get-file-cmd-line.exp
#
set force_conservative 0 ;# set to 1 to force conservative mode even if
         ;# script wasn't run conservatively originally
if {$force_conservative} {
        set send_slow {1 .1}
        proc send {ignore arg} {
            sleep .1
exp_send -s -- $arg
        }
}
```

Listing 8-11 ftp-get-file-cmd-line.exp script

```
set rem_host [lindex $argv 0]
set rem_dir [lindex $argv 1]
set rem_file [lindex $argv 2]
set timeout -1
spawn ftp $rem_host
match_max 100000
expect "Name *: "
send -- "prodmgr\r"
expect "Password:"
send -- "Abc1234\r"
expect "ftp>"
send -- "cd $rem_dir \r"
expect "ftp>"
send -- "get $rem_file \r"
expect "ftp>"
send -- "bye\r"
expect eof
```

Listing 8-11 (continued)

The only difference between Listings 8-9 and 8-11 is the use of command-line arguments. Instead of hard-coding the script for a single use, we now have a flexible Expect script that lets us specify the remote host, remote directory, and remote file that we want to transfer.

Notice in Listing 8-11 that we use the set command to define three variables as command-line arguments: rem_host, rem_dir, and rem_file. With these definitions we now spawn the ftp session to \$rem_host, then we cd to the \$rem_dir directory, and then get the \$rem_file file, with every variable defined as command-line arguments. Listing 8-12 shows the ftp-get-file-cmd-line.exp Expect script in action.

```
[root@booboo ~]# ./ftp-get-file-cmd-line.exp yogi
/scripts random_file.out
spawn ftp yogi
Connected to yogi (192.168.1.100).
220 yogi FTP server (Version 4.1 Sat Feb 23 00:11:36 CST 2002) ready.
Name (yogi:root): prodmgr
331 Password required for prodmgr.
Password:
230 User root logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd /scripts
250 CWD command successful.
ftp> get random_file.out
```

Listing 8-12 ftp-get-file-cmd-line.exp Expect script in action

```
local: random_file.out remote: random_file.out
227 Entering Passive Mode (192,168,1,100,128,46)
150 Opening data connection for random_file.out (2185 bytes).
226 Transfer complete.
2185 bytes received in 0.00352 secs (5.9e+02 Kbytes/sec)
ftp> bye
221 Goodbye.
[root@booboo ~]#
```

Listing 8-12 (continued)

Listings 8-10 and 8-12 are identical, and they should be because we are executing the exact same command, but referenced through variable assignments.

The only bad thing about these scripts is the fact that the usernames and passwords are visible in plain text in the script. Therefore, you need to lock down the file permissions to keep others from getting the password. A better solution is to create an environment file in an obscure location that has username and password information, and source, or execute, the environment file at the beginning of the script, and maybe even encrypt the password file and have the Expect script decrypt the password before use. Just remember to never export the username and password script variables or they will be viewable in the process table in plain text.

What about Conditional Tests?

Many times what we are expecting varies. For example, if we ssh to a server and the remote host is not listed in the <code>\$HOME/.ssh/known_hosts</code> file, we are prompted to add the remote host's public key. In this case, we first need to type **yes** for the login to proceed. Other times we get the <code>Password</code>: prompt. So, how does Expect handle this type of conditional test? One method is with braces. Let's take a closer look.

Expect's Version of a case Statement

Let's look at two examples of using ssh to log in to a remote server. In the first example, the remote server is not in the user's known_hosts file, and we get an extra prompt to permanently add the server to the ssh configuration. The second example shows an ssh login to a known host. Listing 8-13 shows an ssh session to an unknown host.

```
[root@yogi ~]# ssh prodmgr@wilma
The authenticity of host 'wilma (192.168.1.104)' can't be established.
RSA key fingerprint is de:2f:e1:42:96:68:fe:4c:4f:2a:25:97:0a:22:46:e4.
Are you sure you want to continue connecting (yes/no)? yes
Warning:Permanently added 'wilma,192.168.1.104' (RSA)
to the list of known hosts.
```

Listing 8-13 SSH login to an unknown host

```
prodmgr@wilma's password:
Last unsuccessful login: Thu Nov 29 10:53:04 EST 2007 on ssh from yogi
Last login: Mon Dec 24 17:36:50 EST 2007 on /dev/pts/0 from fred
You are logging on to wilma
$
```

Listing 8-13 (continued)

Notice in Listing 8-13 that ssh does not recognize the host wilma as a known host because the public key for wilma is not in the local <code>\$HOME/.ssh/known_hosts</code> file. So, ssh prompts us to add the key for wilma into the <code>\$HOME/.ssh/known_hosts</code> file and we respond with yes, as shown here:

```
Are you sure you want to continue connecting (yes/no)? yes
```

With a positive reply we are prompted for the prodmgr password and we get logged in. In Listing 8-14 notice that we are prompted for only the prodmgr password this time.

```
[root@yogi ~]# ssh prodmgr@wilma
prodmgr@wilma's password:
Last unsuccessful login: Thu Nov 29 10:53:04 EST 2007 on ssh from yogi
Last login: Thu Dec 27 13:44:11 EST 2007 on /dev/pts/0 from yogi
You are logging on to wilma
$
```

Listing 8-14 SSH login to a known host

In Listing 8-14 the host wilma is known, so ssh does not prompt us to add the key; we just get the Password: prompt. With Expect, we use the syntax shown in Listing 8-15 to perform a conditional test.

```
expect {
    "*re you sure you want to continue connecting*"
    {
        send "yes\n"
        expect {
            "*assword*"
            {
            send "Abc1234\n"
        }
        }
    }
}
```

Listing 8-15 Example syntax for an Expect conditional test

```
"*assword*"
{
send "Abc1234\n"
}
}
```

Listing 8-15 (continued)

Listing 8-15 illustrates Expect's version of a case statement. The first expect has two possible expected responses:

"*re you sure you want to continue connecting*"

or

"*assword*"

Notice that we are using wildcards because different versions of SSH and UNIX flavors produce varied responses. Sometimes the A and the P are capitalized and sometimes not. Sometimes there is extra text that we do not care about, so we use wildcards to match the data we are not sure about because it can vary.

If the reply is *re you sure you want to continue connecting*, we respond with yes, and then we expect the Password: prompt, and we respond with the password. Otherwise, we get only the "*assword*" prompt and we respond with the password. This is like a case statement in a shell script.

Let's look at the Expect script in Listing 8-16 to see how to do this. The etc-hosts-copy.exp Expect script is used to copy the master /etc/hosts file, located on the server yogi, to a remote machine. Follow through the script and pay attention to the bold text and the comments throughout Listing 8-16.

```
#!/usr/local/bin/expect -f
#
# SCRIPT: etc-hosts-copy.exp
# Set the timeout to 3 seconds
set timeout 3
# Assign the first command line argument to the
# "host" variable
set rem_host [lindex $argv 0]
# Spawn an ssh session to $host as the prodmgr user
spawn /usr/bin/ssh -1 prodmgr ${rem_host}
# When we ssh to a remote server we can have two
# possible responses, 1) "*re you sure you want to continue
```

Listing 8-16 etc-hosts-copy.exp Expect script

Chapter 8 Automating Interactive Programs with Expect and Autoexpect 309

```
# connecting*", which we must reply to with "yes", and
# 2) Password: prompt. This next expect statement
# will perform the correct action based on the
# response.
expect {
        "*re you sure you want to continue connecting*"
            {
                send "yes\n"
                expect {
                        "*assword*"
                            {
                                send "Abc1234\n"
                            }
                       }
            }
        "*assword*"
            {
                send "Abc1234\n"
            }
       }
# Sleep for 1 second before proceeding
sleep 1
# Set the command-line prompt for the prodmgr user
# to the hash mark, #
send "PS1\=\"\# \"\n "
# Now we know what to expect as a command-line prompt
expect "^#*"
# Copy the master /etc/hosts file from yogi
# to this machine.
# First su to root
send "su - \n"
expect "*assword:*"
send "ABCD1234\n"
sleep 1
# Set the command-line prompt for the root user
# to the hash mark, #
send "PS1\=\"\# \"\n "
```

Listing 8-16 (continued)

```
# Now we know what the command-line prompt is for
# the root user
expect "^#*"
# Save the current version of the /etc/hosts file
send "cp /etc/hosts /etc/hosts$$\n"
expect {
        "*re you sure you want to continue connecting*"
            {
                send "yes\n"
                expect {
                         "*assword*"
                             ł
                                 send "Abc1234\n"
                             }
                        }
            }
        "*assword*"
            {
                send "Abc1234\n"
            3
        " ^ # * "
                          {
                send "\n"
            }
       }
# Copy the master /etc/hosts file from yogi to
# this server.
send "rcp yogi:/etc/hosts /etc/hosts \n"
expect "^#*"
# Logout from $host
send "exit\n"
expect eof
```

Listing 8-16 (continued)

We start out Listing 8-16 by setting the timeout to three seconds. We do not want to wait too long if a host is not reachable. Then we assign the first command-line argument, <code>\$argv 0</code>, to the variable <code>rem_host</code>. Next we spawn the <code>ssh</code> session to the remote <code>\$rem_host</code> as the <code>prodmgr</code> user. At this point we are expecting two possible replies:

"*re you sure you want to continue connecting*"

```
or
```

"*assword*"

Depending on the reply, we either respond with yes and wait for the Password: prompt, or we have the Password: prompt and we respond with the prodmgr's password. At this point, by either path, we should be logged in to the remote host.

At this point we have no clue what the command-line prompt is. It could be anything, so let's set the prompt to the hash mark, #, as shown here:

```
send "PS1=\"\ \# \"\n "
```

Now we know what to expect as a command-line prompt, so we can just expect a line that begins with the hash mark, ^#, as shown here:

expect "^#*"

To copy the /etc/hosts file, we need root authority, and in my shop you cannot log in remotely to any server as root. You have to be on the system console to log in directly as root, so we have to su to root, as follows:

```
send "su - \n"
expect "*assword:*"
send "ABCD1234\n"
sleep 1
```

As you can see, the password is in plain text, so be sure to lock down the file permissions on this Expect script to keep from revealing the passwords.

After we su to root, we need to set the prompt to something we know, so let's set the command-line prompt to the hash mark, as we did for the prodmgr user.

Now that we have root authority on the remote machine, we can make a backup copy of the current /etc/hosts file with the current process ID, specified by the shell's \$\$ variable, as a suffix, and then remote-copy the master /etc/hosts file located on the server yogi to the remote server:

```
# Save the current version of the /etc/hosts file
send "cp /etc/hosts /etc/hosts$$\n"
# Copy the master /etc/hosts file from yogi to
# this server.
send "rcp yogi:/etc/hosts /etc/hosts \n"
expect {
        "*re you sure you want to continue connecting*"
            {
                send "yes\n"
                expect {
                        "*assword*"
                            {
                              send "Abc1234\n"
                            }
                       }
            }
```

```
"*assword*"
{
send "Abc1234\n"
}
"^#*" {
send "\n"
}
}
```

Notice that even though we are using remote copy (rcp) to copy the master /etc/hosts file, some UNIX flavors will execute secure copy (scp) instead. We added another Expect case statement to test for a secure shell login. Otherwise, we get the PS1 prompt we specified. With the tasks complete, we exit. Listing 8-17 shows the etc-hosts-copy.exp script in action.

```
[root@yogi scripts]# ./etc-hosts-copy.exp booboo
spawn /usr/bin/ssh -1 prodmgr booboo
prodmgr@booboo's password:
You are logging on to booboo
Stats: Linux booboo 2.4.22-1.2199.nptl #1 Wed Aug 4
12:21:48 EDT 2004 i686 i686 i386 GNU/Linux
PS1="# "
  [prodmgr@booboo ~]$ PS1="# "
# su -
Password:
[root@booboo ~]# PS1="# "
# cp /etc/hosts /etc/hosts$$
# rcp li01s:/etc/hosts /etc/hosts
exit
[root@yogi scripts]#
```

Listing 8-17 etc-hosts-copy.exp Expect script in action

Notice that the system suppresses echoing the password on the terminal. A good way to use this Expect script is to create a short shell script with a while read loop that reads a list of hostnames. As we read each hostname, we execute the etc-hosts-copy.exp Expect script to copy the master /etc/hosts file to each remote machine in the list, as shown here:

```
while read SERVER
do
    ./etc-hosts-copy.exp $SERVER
done < ./host-update.lst</pre>
```

This while loop reads the host-update.lst file line-by-line and on each loop iteration executes the etc-hosts-copy.exp Expect script using the current \$SERVER as the remote host.

Expect's Version of an if...then...else Loop

Expect also provides if...then...else and other basic control structures. Just like Expect's version of a case statement, the if...then...else structure uses curly braces, as shown here in an Expect code segment:

```
set TOTAL 5
if ( $TOTAL < 10 ) {
    puts ''TOTAL is less than 10''
} else {
    puts "TOTAL greater than or equal to 10"
}</pre>
```

We assign the variable TOTAL the value 5. If the \$TOTAL variable's value is less than 10 we use the puts command, like we use echo in a shell script, to output user data to the terminal. In our case, this code segment will return the following:

```
TOTAL is less than 10
```

Let's expand this if...then...else statement to add in an elseif to extend our test of the \$TOTAL value, as shown in Listing 8-18.

```
#!/usr/local/bin/expect -f
set TOTAL 1
if { $TOTAL < 10 } {
    puts "\nTOTAL is less than 10\n"
} elseif { $TOTAL > 10 } {
    puts "\nTOTAL greater than 10\n"
} else {
    puts "\nTOTAL is equal to 10\n"
}
```

Listing 8-18 if-then-else-test.exp Expect script

Other than the use of the curly braces, this if...then...else works the same way it does in a shell script.

NOTE The opening brace *must* appear on the same line as the if, elseif, and else statements.

Notice that we added new lines before and after the text we output with the puts command. This is the same technique we use with the echo command.

Expect's Version of a while Loop

Expect uses similar syntax for a while loop as our shell scripts use. But again, we are using braces to contain the expression. Carrying over the previous example of testing the value of the variable TOTAL, this time using a COUNT variable, we will add an incremental counter as we have the COUNT variable count from 0 to 10, as shown in Listing 8-19.

```
#!/usr/local/bin/expect -f
set COUNT 0
while { $COUNT <= 10 } {
    puts "\nCOUNT is currently at $COUNT"
    set COUNT [ expr $COUNT + 1 ]
}
puts ""</pre>
```

Listing 8-19 while_test.exp Expect script

In Listing 8-19 we initialize the variable COUNT to 0. Then we start a while loop that continues while \$COUNT is less than or equal to 10. On each loop iteration we use the puts command to display the current value of \$COUNT and increment the COUNT by adding 1, until \$COUNT is greater than 10. The output of this while loop is shown in Listing 8-20.

```
[root@yogi scripts]# ./while_test.exp
COUNT is currently at 0
COUNT is currently at 1
COUNT is currently at 2
COUNT is currently at 3
COUNT is currently at 4
COUNT is currently at 5
COUNT is currently at 6
COUNT is currently at 7
```

Listing 8-20 while_test.exp Expect script in action

COUNT is currently at 8 COUNT is currently at 9 COUNT is currently at 10 [root@yogi scripts]#

Listing 8-20 (continued)

Notice in Listing 8-20 there is a new line after the line COUNT is currently at 10, and before the ending command prompt, [root@yogi scripts]#. This last new line is the result of the last puts "" at the end of the while_test.exp Expect script in Listing 8-19, which adds the final new line before exiting when the count reaches 10.

Expect's Version of a for Loop

The for loop in Expect works a little differently from a for loop in a shell script. An Expect for loop requires three fields to be specified with the following syntax:

```
for initial_value conditional_exression incremental_step {
    Do something here
}
```

As an example, let's use an Expect for loop to count from 0 to 10, as shown in Listing 8-21.

```
#!/usr/local/bin/expect -f
for {set COUNT 0} {$COUNT <= 10} {incr COUNT} {
    puts "\nCOUNT is currently at $COUNT"
}
puts ""</pre>
```

Listing 8-21 for_text.exp Expect script

This for loop begins by assigning the variable COUNT an initial value of 0. The second part of the for statement is the conditional expression that determines when to exit the loop. The last part of the for loop statement defines how we increment or decrement the COUNT variable on each loop iteration. Notice that we introduced a new Expect operator, incr. The incr operator will increment the variable given as an

argument. By default incr will increment by 1, but we can override this behavior by adding a second argument to incr specifying the increment to add or subtract. For example, if we want to count down from 10 to 0, we would use the following syntax:

```
for {set COUNT 10} {$COUNT <= 0} {incr COUNT -1} {
```

Notice that the second argument to incr is -1, telling the incr operator to subtract 1 instead of adding 1. We can also count by 2 or 10 or any other integer values.

The output that the for loop in Listing 8-21 produced is shown in Listing 8-22.

```
[root@yogi bin]# ./for_test.exp
COUNT is currently at 0
COUNT is currently at 1
COUNT is currently at 2
COUNT is currently at 3
COUNT is currently at 4
COUNT is currently at 5
COUNT is currently at 6
COUNT is currently at 7
COUNT is currently at 7
COUNT is currently at 8
COUNT is currently at 9
COUNT is currently at 10
[root@yogi bin]#
```

Listing 8-22 for_test.exp Expect script in action

This is the same output that our while loop produced. Instead of using the **incr** Expect operator, we can also use a mathematical expression in the for loop. Expressions are enclosed in square brackets, [*expression*], to tell Expect to evaluate the expression first. Adding 1 to the COUNT variable using an expression is shown here:

```
[ expr $COUNT + 1 ]
```

If we use this expression along with the set command, we can increment the COUNT variable on each loop iteration, as shown in Listing 8-23.

```
#!/usr/local/bin/expect -f
for {set COUNT 0} {$COUNT <= 10} {set COUNT [expr $COUNT + 1 ]} {
    puts "\nCOUNT is currently at $COUNT"
}
puts ""</pre>
```

Listing 8-23 for_test2.exp example for loop using an expression to increment

On each loop iteration we use the set command to assign the COUNT variable a newly calculated value. The square brackets ensure that the mathematical expression is evaluated before the new value is assigned to the COUNT variable.

Expect's Version of a Function

Expect supports functions using the **proc** Expect operator instead of the word **function**. We can define arguments to a proc but it is not a requirement, just as in a shell script. Let's modify our counting Expect script to use a function, or proc, to increment the COUNT variable. The modified version is called function_test.exp and is shown in Listing 8-24.

```
#!/usr/local/bin/expect -f
#
# SCRIPT: function_test.exp
****
# DEFINE VARIABLES HERE
****
set COUNT 0
****
# DEFINE FUNCTIONS HERE
****
proc increment_by_1 { MY_COUNT } {
  set MY_COUNT [expr $MY_COUNT + 1]
  return "$MY COUNT"
}
****
# BEGINNING OF MAIN
*****
```

Listing 8-24 function_test.exp example using a function in Expect

```
while {$COUNT <= 10} {
    puts "\nCOUNT is currently at $COUNT"
    set COUNT [increment_by_1 $COUNT]
}
puts ""</pre>
```

Listing 8-24 (continued)

Listing 8-24 starts by initializing the variable COUNT to 0. In the next section we define our function, or proc, in pretty much the same way we do in a shell script. Of course, you still need to define the function in the code before the point where it is first used. Our proc is named increment_by_1 and this proc is expecting one argument, and we assign the value of the argument to the variable MY_COUNT.

In the body of the proc we use the set command with an expr expression to increment \$COUNT by 1. To send the value back to the script, we use the return command specifying the new \$MY_COUNT value.

In the main body of the Expect script we use a while loop to loop until \$COUNT is greater than 10. On each loop iteration we use the puts command to display text on the screen, and then use the set command to call the increment_by_1 proc, giving \$COUNT as the argument to the proc, and the proc returns \$COUNT incremented by 1.

This is a very simple example of using a proc in an Expect script, but it shows the proper syntax and techniques to use functions, or a proc, in Expect.

Using Expect Scripts with Sun Blade Chassis and JumpStart

I was playing around with Expect one day while working on some Sun Blade servers. I needed to do a few things with the blades but I had to go through the Blade Chassis to get console access. The first thing I want is to show the Blade Chassis configuration using the showplatform -v Blade Chassis command. But to do this I had to first log in to the Sun Blade Chassis. I used autoexpect to initially create the showplatform.exp Expect script, and then removed the unneeded text. Follow through Listing 8-25 and we will cover the details at the end.

```
#!/usr/local/bin/expect -f
#
# This Expect script was generated by autoexpect on Mon
Apr 2 09:44:34 2007
# Expect and autoexpect were both written by Don Libes, NIST.
#
# Note that autoexpect does not guarantee a working script. It
# necessarily has to guess about certain things. Two reasons a script
# might fail are:
```

Listing 8-25 showplatform.exp Expect script

```
# 1) timing - A surprising number of programs (rn, ksh, zsh, telnet,
# etc.) and devices discard or ignore keystrokes that arrive "too
# quickly" after prompts. If you find your new script hanging up at
# one spot, try adding a short sleep just before the previous send.
# Setting "force_conservative" to 1 (see below) makes Expect do this
# automatically - pausing briefly before sending each character. This
# pacifies every program I know of. The -c flag makes the script do
# this in the first place. The -C flag allows you to define a
# character to toggle this mode off and on.
set force conservative 0 ;# set to 1 to force conservative mode even if
                     ;# script wasn't run conservatively originally
if {$force_conservative} {
    set send_slow {1 .1}
     proc send {ignore arg} {
            sleep .1
            exp_send -s -- $arg
     }
}
# 2) differing output - Some programs produce different output each time
# they run. The "date" command is an obvious example. Another is
# ftp, if it produces throughput statistics at the end of a file
# transfer. If this causes a problem, delete these patterns or replace
# them with wildcards. An alternative is to use the -p flag (for
# "prompt"), which makes Expect only look for the last line of output
\# (i.e., the prompt). The -P flag allows you to define a character to
# toggle this mode off and on.
# Read the man page for more info.
# -Don
set chassis [lindex $argv 0]
set timeout -1
spawn $env(SHELL)
send -- "telnet ${chassis}\r"
expect -exact "telnet ${host}\r"
expect -exact "username: "
send -- "prodmgr\r"
expect -exact "prodmgr\r
password: "
send -- "Abc1234\r"
expect -exact "\r
${host}>"
send -- "showplatform -v\r"
```

Listing 8-25 (continued)

```
expect -exact "showplatform -v\r"
send -- "logout\r"
expect -exact "logout\r"
exit
expect eof
```

Listing 8-25 (continued)

In the showplatform.exp Expect script in Listing 8-25, we first assign the first command-line argument to the variable chassis. Next we disable timeouts by setting timeout to -1. Now we spawn a default shell and telnet to \$chassis, which is the hostname of the Blade Chassis.

Next we expect the username and password prompts, and send the correct response to get logged in to the Chassis. Now we can execute the showplatform -v command and log out of the Blade Chassis. A sample output from showplatform.exp is shown in Listing 8-26.

```
wilma:> ./showplatform.exp sunchas1
spawn /sbin/sh
wilma:> telnet sunchas1
Trying 192.168.1.142...
Connected to sunchas1.
Escape character is '^]'.
Sun Advanced Lights Out Manager for Blade Servers 1.2
Copyright 2003 Sun Microsystems, Inc. All Rights Reserved.
ALOM-B 1.2
username: prodmgr
password:
sunchas1>showplatform -v
FRU Status Type Part No. Serial No.
----- ------
                                                              _____
                                                                              _____
            Not Present ***
                                                            * * *
                                                                              * * *
S0
             Not Present ***
                                                            * * *
                                                                             * * *
S1

        Not Present
        ***
        ***
        ***

        Faulty
        SF B100s
        5405078
        009897

        OK
        SF B100s
        5405079
        013116

        Faulty
        SF B100s
        5405078
        009973

        Faulty
        SF B100s
        5405078
        010562

        Faulty
        SF B100s
        5405078
        010111

        Faulty
        SF B100s
        5405079
        010678

        Standby
        SF B100s
        5405079
        029965

        OK
        SF B100s
        5405079
        029854

S2
S3
S4
S5
S6
S7
S8
                                                           5405079 029854
5405079 029698
                                     SF B100s
S9
             OK
                        SF B100s
S10
             OK
            Not Present ***
Not Present ***
                                                            * * *
                                                                            * * *
S11
                                                              * * *
S12
                                                                              * * *
                                                              * * *
             Not Present ***
                                                                              * * *
S13
```



S14	Not Present	* * *	* * *	* * *		
S15	Not Present	* * *	* * *	* * *		
SSC0	OK	SF B1600 SSC	5405185	0004703-0308000162		
SSC0/SWT						
SSC0/SC						
SSC1	OK	SF B1600 SSC	5405185	0004703-0407005951		
SSC1/SWT						
SSC1/SC						
PS0	OK	SF B1600 PSU	3001544	0017527-0324P02770		
PS1	OK	SF B1600 PSU	3001544	0017527-0324P02700		
CH	OK	SF B1600	5405082	001635		
Domain	Status	MAC Add	ress			
	OS Running		b:84:4c:d0			
	OS Running		b:8c:ea:bc			
	OS Running		b:59:fe:e8			
	OS Running		b:84:33:62			
	OS Running		b:84:37:c2	-		
	OS Running		b:84:2e:80			
S8	Standby		b:44:4f:5c			
	OS Running		b:34:42:1e			
	OS Running	00:03:bi	b:34:40:88	sunbldd1		
SSC0/SWT	OS Running		b:3b:8b:e3			
SSC1/SWT	OS Running	00:03:b	b:39:7d:ac			
SSC0/SC	OS Running (Ac	tive) 00:03:b	b:8b:88:fc			
SSC1/SC	OS Running	00:03:b	b:19:7b:c5			
sunchas1>logout						
Connection to sunchas1 closed by foreign host.						
wilma:>						

Listing 8-26 (continued)

In Listing 8-26 we log in to the remote Sun Blade Chassis sunchas1 with the username prodmgr. After logging in, we execute the showplatform -v chassis command and log out of sunchas1.

How else can we use this Expect script? Many times we need to know which slot a particular blade server is located in. If we are logged in to the Sun Blade Chassis and we want console access to a specific blade, we use the slot number, not the hostname, to gain console access. Check out the findslot Bash script in Listing 8-27 to see how we use the showplatform.exp Expect script.

```
#!/usr/bin/Bash
#
# SCRIPT: findslot
# PURPOSE: This script is used to locate a blade server
# within the specified SUN blade chassis.
```

Listing 8-27 findslot Bash Script

Listing 8-27 (continued)

The findslot Bash script in Listing 8-27 expects two command-line arguments: the chassis hostname, CHAS, and the hostname of the blade for which we want the slot number, HOST. Using this information we execute the showplatform.exp Expect script and grep for the blade's hostname, and extract the first field. Then we echo the \$SLOT slot number.

Now that we can easily find the slot a blade is located in, we can write an Expect script that will rebuild a failed blade using JumpStart. It is beyond the scope of this book to cover the details of JumpStart, but at a high level JumpStart is a Sun Microsystems product that enables us to do network OS installations. We are making the assumption that JumpStart is already configured to boot a particular blade and install a new OS image.

The key here is that the blade must be on the blade's console at the ok prompt, and then we execute the following command:

boot net - install

Now we are adding a second layer of logins to this Expect script because we need to be on the blade's console. First we log in to the Blade Chassis, and then, using a known slot number, gain console access to the specific blade server with the chassis command console S#, where S# is the specific slot where the blade resides — for example console S1. At this point we are expecting ok, and then we execute boot net - install to kick off the JumpStart installation. Take a close look at boot-net-install.exp in Listing 8-28.

Listing 8-28 boot-net-install.exp Expect script

```
set chassis [lindex $argv 0]
set blade [lindex $argv 1]
set timeout -1
spawn -noecho telnet $chassis
log_user 0
match_max 100000
expect "username: "
send -- "prodmgr\r"
expect -exact "prodmgr\r
password: "
send -- "Abc1234\r"
expect ">"
send -- "console -f $blade\r"
send -- "\r"
expect "ok"
send -- "boot net - install\r"
log_user 1
expect ""
send -- "logout\r"
send_user "\r\n"
exit
expect eof
```

Listing 8-28 (continued)

In Listing 8-28 we start by assigning the first command-line argument to the variable chassis, and the second command-line argument to the variable blade, which is a *blade slot, not the blade hostname*. Next we disable timeouts by setting timeout to -1. Then we spawn a telnet session to the blade chassis, \$chassis, with the -noecho option. This spawn switch option suppresses telnet output. Then we provide the username and password to log in to the blade chassis. Now that we are on the chassis, we can get console access to the specific blade server by knowing the slot it resides in. When we gain console access we are expecting the ok prompt. At the ok prompt we execute boot net - install and the blade starts a network boot for the JumpStart installation.

There is a lot to JumpStart, but I will leave the details up to you to research on your own.

Summary

There is no way I could cover everything about Expect in a single chapter in this book. I included this chapter as a primer of what Expect can do and how to use variables, perform conditional tests, and how to use functions.

To see a great Expect script, download the passmass Expect script from the official Expect web site at http://expect.nist.gov/. The passmass Expect script is used to change passwords on a large number of UNIX servers of varying UNIX flavors.

If you study this passmass Expect script, you will pick up some good techniques for Expect.

I tried to provide as many examples as I could to get you familiar with using Expect and autoexpect. autoexpect is a nice tool to help you build the quick and dirty Expect script, then you can go back and remove the extra text in the replies.

In the next chapter we are going to study some techniques to find large files, and files of a specific type and size. When filesystems fill up, it is sometimes difficult to find the source of the problem. The next chapter will help to resolve the issue.

Lab Assignments

- 1. Write an Expect script that changes your password on the local UNIX machine. Remember: the new password must be entered twice.
- 2. Write an Expect script that changes your password on a remote UNIX machine.
- 3. Write a Bash script that utilizes the Expect script you created in Lab Assignment 2 to change your password on a list of remote UNIX machines. The list of remote machines should reside in a flat text file in the current directory where the script executes.

CHAPTER 9

Finding Large Files and Files of a Specific Type

Filesystem alert! We all hate to get full-filesystem alerts, especially at quitting time on Friday when the developers are trying to meet a deadline. The usual culprit is one or more *large* files that were just created, compiled, or loaded. Determining the definition of a large file varies by system environment, but a "large" file can fill up a filesystem quickly, especially in a large database or development environment. To find the files that quickly fill up a filesystem and/or other large files, we need a flexible tool that will search for files larger than a user-defined value. The **find** command is your friend when a filesystem search is needed.

The find command is one of the most flexible and powerful commands on the system. Before we get started, I want you to have a copy of the find command manual page. You have two options to save a copy of the man page for the find command. The first option uses paper (you know, in a printer), and the second option saves a bunch of trees. To send the manual page for the find command to a specific print queue (in other words, burn up some trees), enter the following command:

```
man find | lp -d print_queue_name
```

If you are interested in saving the planet, you can archive the find command manual page by executing the following command:

man find | col -b >> find.txt

The previous command will print the manual page output to the printer defined by print_queue_name, or save the manual page for future viewing in a file named find.txt. If you take the time to study the find command manual page in detail, you can see that the find command is the most flexible command on the system. You can find files by modification/creation time, last access time, exact size, greater-than

size, owner, group, permission, and a boatload of other options. You can also execute a command to act on the file(s) using the -exec command switch. For this chapter we are going to concentrate on two topics. The first is finding files larger than an integer value specified on the command line. The second is finding files recently created and/or modified. I will also give some examples of techniques to quickly narrow down the search to find the offending file. As with all of our shell scripts, we first need to get the correct command syntax for our task.

Syntax

We are going to use the -size option for the find command to search the system for "large" files. There are two things to consider when using the -size command switch to search the system. We must supply an argument for this switch, and the argument must be an integer. But the integer number alone as an argument to the -size switch specifies an *exact* value, and the value is expressed, by default, in 512-byte blocks instead of measuring the file size in bytes, kilobytes, megabytes, or gigabytes. For a more familiar measurement we would like to specify our search value in megabytes (MB) for this example. To specify the value in bytes instead of 512-byte blocks, we add the character c as a suffix to the integer value, and to get to MB we can just add six zeros. We also want to look for file sizes *greater than* the command-line integer value. To specify greater than, we add a + (plus sign) as a prefix to the integer value. With these specifications the command will look like the following:

find \$SEARCH_PATH -size +integer_valuec -print

To search for files greater than 5 MB we can use the following command:

find \$SEARCH_PATH -size +5000000c -print

The + (plus sign) specifies greater than, and the c denotes bytes. Also notice in the previous command that we specified a path to search using the variable SEARCH_PATH. The find command requires a search path to be defined in the first argument to the command. We also added the -print switch at the end of the command line. If you omit the -print, you cannot guarantee that any output will be produced if you use the -exec option. The command will return the appropriate return code but may not give any output, even if the files were found! I have found this to be a combination of UNIX flavor, release, and executing shell. Just always remember to add the -print switch to the find command if you need to see what file is being acted upon, and you will not be surprised.

For ease of using this shell script we are going to assume that the search will always begin in the current directory. The **pwd** command, or *print working directory* command, will display the full pathname of the current directory. Using our script this way requires that the shell script is located in a directory that is in the user's \$PATH, or you must use the full script pathname any time you use the shell script. I typically put all of my scripts in the /usr/local/bin directory and add this directory to my \$PATH

environment. You can add a directory to your path using the following command syntax:

```
PATH=$PATH:/usr/local/bin
export PATH
```

Remember That File and Directory Permissions Thing

If you are not executing the find command as root, either by logging in as root, using su – root to gain root access, executing a command as root using su – root "-c some_command", or using sudo command to execute a single command as root, file permission errors may scroll across your screen. If the filesystem that filled up is an application or development filesystem, you may have permissions if you have the ability to log in as the *application owner*, or an account used by the application or development, database, and other teams.

Don't Be Shocked by the Size of the Files

If you do a search on a system that has a database, be aware that database files are *huge*! When I first wrote this book in 2000–2001, a *large* disk drive was 32 GB. Now 1TB disk drives are available! Multi-terabyte transactional databases are the norm. Then there are the processing times of 12–24+ hours to massage the data into a usable format to make money.

Just remember to limit your search. Starting a search from the root directory, /, implies a full system search. That means all filesystems are searched, and that can take quite a while, depending on the attached storage.

Creating the Script

We have the basic idea of the find command syntax, so let's write a script. The only argument that we want from the user is a positive integer representing the number of megabytes (MB) to trigger the search on. We will add the extra six zeros inside the shell script. As always, we need to confirm that the data supplied on the command line is valid and usable. For our search script we are expecting exactly one argument; therefore, the shell variable \$# must equal one. We are also expecting the argument to be an integer, so the regular expression + ([0-9]) should be true.

NOTE Bash shell does not support the regular expression + ([0-9]).

We will use this regular expression in a case statement to confirm that we have an integer. The integer specified must also be a positive value, so the value given must be

greater than zero. If all three tests are true, we have a valid value to trigger our search. All these requirements are not difficult, so please hang with me.

I can envision this script producing a daily report at some shops. To facilitate the reporting we need some information from the system. I would like to know the hostname of the machine that the report represents. The **hostname** command will provide this information. A date and timestamp would be nice to have also. The **date** command has plenty of options for the timestamp, and because this is going to be a report, we should store the data in a file for printing and future review. We can just define a file, pointed to by *SOUTPUT*, to store our report on disk.

Everyone needs to understand that this script always starts the search from the *current working directory*, defined by the system environment variable \$PWD and by executing the pwd command. We are going to use the pwd command and assign the output to the SEARCH_PATH script variable. The only other thing we want to do before starting the search is to create a report header for the \$OUTFILE file. For the header information we can append all of the pertinent system data we have already gathered from the system to the \$OUTFILE.

We are now ready to perform the search starting from the current directory. Starting a search from the current directory implies, again, that this script's filename must be in the \$PATH for the user who is executing the script, or the full pathname to the script must be used instead.

Study the findlarge.ksh shell script in Listing 9-1, and pay attention to the bold type.

Listing 9-1 findlarge.ksh shell script

```
*****
function usage
{
echo "\n\nUSAGE:
           findlarge.ksh [Number_Of_Meg_Bytes]"
echo "\nEXAMPLE: filelarge.ksh 5"
echo "\n\nWill Find Files Larger Than 5 Mb in, and Below, the \backslash
 Current Directory..."
echo "\n\nEXITING...\n"
exit
}
*****
function cleanup
{
echo "\n\nEXITING ON A TRAPPED SIGNAL..."
exit
}
*****
# Set a trap to exit. REMEMBER - CANNOT TRAP ON kill -9 !!!!
trap 'cleanup' 1 2 3 15
****
# Check for the correct number of arguments and a number
# Greater than zero
if [ $# -ne 1 ]
then
    usage
fi
if [ $1 -1t 1 ]
then
    usage
fi
****
# Define and initialize files and variables here...
```

Listing 9-1 (continued)

```
THISHOST=`hostname` # Hostname of this machine
DATESTAMP=`date +"%h%d:%Y:%T"`
SEARCH_PATH=`pwd` # Top level directory to search
MEG_BYTES=$1
                 # Number of Mb for file size trigger
DATAFILE="/tmp/filesize_datafile.out" # Data storage file
>$DATAFILE
                  # Initialize to a null file
OUTFILE="/tmp/largefiles.out" # Output user file
>$OUTFILE
                   # Initialize to a null file
HOLDFILE="/tmp/temp_hold_file.out" # Temporary storage file
                  # Initialize to a null file
>$HOLDFILE
**********
# Prepare the output user file
echo "\n\nSearching for Files Larger Than ${MEG_BYTES}Mb starting in:"
echo "\n==> $SEARCH PATH"
echo "\n\nPlease Standby for the Search Results..."
echo "\n\nLarge Files Search Results:" >> $OUTFILE
echo "\nHostname of Machine: $THISHOST" >> $OUTFILE
echo "\nTop Level Directory of Search:" >> $OUTFILE
echo "\n==> $SEARCH_PATH" >> $OUTFILE
echo "\nDate/Time of Search: 'date'" >> $OUTFILE
echo "\n\nSearch Results Sorted by Time:" >> $OUTFILE
******
# Search for files > $MEG_BYTES starting at the $SEARCH_PATH
find $SEARCH_PATH -type f -size +${MEG_BYTES}000000c \
     -print > $HOLDFILE
# How many files were found?
if [ -s $HOLDFILE ]
then
   NUMBER_OF_FILES=`cat $HOLDFILE | wc -1`
   echo "\n\nNumber of Files Found: ==> \
   $NUMBER_OF_FILES\n\n" >> $OUTFILE
    # Append to the end of the Output File...
```

Listing 9-1 (continued)

```
ls -lt `cat $HOLDFILE` >> $OUTFILE
# Display the Time Sorted Output File...
more $OUTFILE
echo "\n\nThese Search Results are Stored in ==> $OUTFILE"
echo "\n\nSearch Complete...EXITING...\n\n"
else
cat $OUTFILE
echo "\n\nNo Files were Found in the Search Path that"
echo "are Larger than ${MEG_BYTES}Mb\n\n"
echo "\nEXITING...\n\n"
fi
```

Listing 9-1 (continued)

Let's review the findlarge.ksh shell script in Listing 9-1 in a little more detail. We added two functions to our script. We always need a usage function, and in case Ctrl+C is pressed, we added a trap_exit function. The trap_exit function is executed by the trap for exit signals 1, 2, 3, and 15 and will display EXITING ON A TRAPPED SIGNAL before exiting with a return code of 2. The usage function is executed if any of our three previously discussed data tests fail and the script exits with a return code of 1, one, indicating a script usage error.

In the next block of code we query the system for the hostname, date/timestamp, and the search path (the current directory!) for the find command. All of this system data is used in the file header for the \$OUTFILE. For the search path we could have just used a dot to specify the current directory, but this short notation would result in a *relative pathname* in our report. The *full pathname*, which begins with a forward slash (/), provides much clearer information and results in an easier-to-read file report. To get the full pathnames for our report, we use the pwd command output assigned to the SEARCH_PATH variable.

We define two files for processing the data. The \$HOLDFILE holds the search results of the find command's output. The \$OUTFILE contains the header data, and the search results of the find command are appended to the end of the \$OUTFILE file. If the \$HOLDFILE is zero-sized, then the find command did not find any files larger than \$MEG_BYTES, which is the value specified by the \$1 shell positional parameter on the command line. If the \$HOLDFILE is not empty, we count the lines in the file with the command statement NUMBER_OF_LINES=`wc -1 \$HOLDFILE`. Notice that we used back tics for command substitution, `command`. This file count is displayed along with the report header information in our output file. The search data from the find command, stored in \$HOLDFILE, consists of full pathnames of each file that has exceeded our limit. In the process of appending the \$HOLDFILE data to our \$OUTFILE, we do a long listing sorted by the modification time of each file. This long listing is produced using the 1s -1t \$HOLDFILE command. A long listing is needed in the report so that we can see not only the modification date/time, but also the file owner and group as well as the size of each file.

All the data in the *\$OUTFILE* is displayed by using the **more** command so that we display the data one page at a time. The findlarge.ksh shell script is in action in Listing 9-2.

```
Searching for Files Larger Than 1Mb starting in:
==> /scripts
Please Standby for the Search Results...
Large Files Search Results:
Hostname of Machine: yogi
Top Level Directory of Search:
==> /scripts
Date/Time of Search: Thu Nov 8 10:46:21 EST 2001
Search Results Sorted by File Modification Time:
Number of Files Found: ==>
                            4
-rwxrwxrwx 1 root sys 3490332 Oct 25 10:03 /scripts/
   sling_shot621.tar
-rwxrwxrwx 1 root sys 1280000 Aug 27 15:33 /scripts/
   sudo/sudo-1.6.tar
-rw-r--r-- 1 root sys 46745600 Jul 27 09:48 /scripts/scripts.tar
-rw-r--r-- 1 root system 10065920 Apr 20 2001 /scripts/
  exe/exe files.tar
These Search Results are Stored in ==> /tmp/largefiles.out
Search Complete...EXITING...
```

Listing 9-2 findlarge.ksh shell script in action

The output in Listing 9-2 is a listing of the entire screen output, which is also the contents of the *\$OUTFILE*. The user is informed of the trigger threshold for the search, the top-level directory for the search, the hostname of the machine, the date and time of the search, and the number of files found to exceed the threshold. The long listing of each file is displayed; the long listing has the file owner and group, the size of the file in bytes, the modification time, and the full path to the file. The long listing is very helpful in large shops with thousands of users!

Narrowing Down the Search

If we just got a filesystem alert, that tells us that at least one file in that particular filesystem was recently created and/or modified. So, a good starting point is to look for newly modified/created files. You can add the following command as a cross-reference to find the true cause of the filesystem filling up. This command will search the \$SEARCH_PATH for all files modified, or created, less than 10 minutes ago:

```
find $SEARCH_PATH -mmin 10 -type f
```

This command will find all files that have been modified or created in the last 10 minutes. You can redirect this output to a file and do a cross-reference to discover the files and users that actually caused the filesystem to fill up.

We can also look at the files most recently accessed by using the *-amin* switch to the find command. This search will be more extensive.

find \$SEARCH_PATH -amin 10 --type f

There are many more options to the find command for you to try out. Some other find command switches are shown in Table 9-1.

This is only a very small sample of find command options.

SWITCH	FIND OPTION
-xdev	Do not descend directories on other filesystems
-amin n	List files accessed within n minutes
-mmin n	List files modified or created with n minutes
-atime n	List files accesses within n 24 hour periods
-mtime n	List files modified or created within n 24 hour periods

Table 9-1 More find Command Search Options

Other Options to Consider

The findlarge.ksh shell script is simple and does all of the basics for the system reporting, but it can be improved and customized for your particular needs. I think you will be interested in the following ideas and tasks:

1. When we specify our search value we are just adding six zeros to the usersupplied integer value. But we are back to a basic question: Is 1MB equal to 1,000,000 or 1,024,000? Because a Systems Administrator may not be the one reading the report, (maybe a manager will be), I used the mathematical 1,000,000 and not the system-reported power-of-2 value. This is really a toss-up, so you make the decision on the value you want to use. The value is easy to change by doing a little math to multiply the user-supplied value by 1,024,000.

2. The Lab Assignments tackle some other options to consider.

Summary

In this chapter we have shown how to search the system for large files and create a machine-specific report. There are many ways to do the same task, and as always we have other options to consider. This chapter, along with Chapter 17, "Filesystem Monitoring," can help keep filesystem surprises to a minimum.

In the next chapter we are going to study techniques to monitor processes and applications as well as to write scripts to allow pre-, startup, and post-processing.

I hope you gained some knowledge in this chapter, and I will see you in the next chapter!

Lab Assignments

- 1. Modify the findlarge.ksh shell script in Listing 9-1 to add a second commandline argument so that you can specify a search path other than the current directory. You should add this user-supplied search path as an option, and if a search path is omitted, you use the current directory to start the search. This adds a little more flexibility to the original shell script.
- 2. When searching large filesystems, the search may take a very long time to complete. To give the user feedback that the search process is continuing, add one of the progress indicators studied in Chapter 4, "Progress Indicators Using a Series of Dots, a Rotating Line, or Elapsed Time." Each of the three progress indicators would be appropriate; refer to Chapter 4 for details. Name the modified file so that the method selected is intuitively obvious.
- 3. Each time we run the findlarge.ksh shell script, we overwrite the \$OUTFILE. Modify the script to keep a month's worth of reports on the system. An easy way to keep one month of reports is to use the date command and extract the day of the month, and then add this value as a suffix to the \$OUTFILE filename definition. Over time our script will result in filenames largefile.out.01 through largefiles.out.31, depending on the month. Simulate the date to test your script to ensure it will produce files .01 through .31.
- 4. Modify your last script from Lab Assignment 3 to execute in Bash shell. Hint: it is all about the echo command, and testing for an integer.

CHAPTER 10 Process Monitoring and Enabling Pre-Processing, Startup, and Post-Processing Events

All too often a program or script will die, hang, or error out during execution or fail to even start up. This type of problem can be hard to nail down due to the unpredictable behavior and the timing required to catch the event as it happens. We also sometimes want to execute some commands before a process starts, as the process starts (or as the monitoring starts), or as a post event when the process dies. Timing is everything! Instead of reentering the same command over and over to monitor a process, we can write scripts to wait for a process to start or end and record the timestamps, or we can perform some other function as a pre-startup or post-event. To monitor the process we are going to use grep to grab one or more matched patterns from the process list output. Because we are going to use grep, there is a need for the process to be unique in some way — for example, by process name, user name, process ID (PID), parent process ID (PPID), or even a date/time.

In this chapter we cover four scripts that do the following:

- Monitor for a process (one or more) to start execution.
- Monitor for a process (one or more) to stop execution.
- Monitor as the process(es) stops and starts, and log the events as they happen with a timestamp.
- Monitor as the process(es) starts and stops while keeping track of the current number of active processes, giving user notification with a timestamp, and listing of all the active PIDs. We also add pre-startup or post-event capabilities. Two examples of using one of these functions are waiting for a backup to finish before rebooting the system and sending an email as a process starts up.

Syntax

As with all our scripts, we start out by getting the correct command syntax. To look at the system processes, we want to look at *all* the processes, not a limited view for a particular user. To list all the processes, we use the **ps** command with the BSD aux switch (notice there is no hyphen prefix). Using grep with the ps aux command requires us to filter the output. I use ps aux particularly because OpenBSD does not support ps -ef. The grep command will produce two additional lines of output. One line will result from the grep command, and the other will result from the script name, which is doing the grepping. To remove both of these we can use either grep -v or egrep -v to exclude this output. From this specification, and using variables, we came up with the following command syntax:

```
ps aux | grep $PROCESS | grep -v "grep $PROCESS" | grep -v $SCRIPT_NAME
```

The preceding command will give a full process listing while excluding the shell script's name and the grepping for the target process. This will leave only the actual processes that we are interested in monitoring. The return code for this command is 0, if at least one process is running, and it will return a nonzero value if no process, specified by the \$PROCESS variable, is currently executing. To monitor a process to start or stop we need to remain in a tight loop until there is a transition from running to end of execution, and vice versa.

Monitoring for a Process to Start

Now that we have the command syntax we can write the script to wait for a process to start. This shell script is pretty simple because all it does is run in a loop until the process starts. The first step is to check for the correct number of arguments: one — the process to monitor. If the process is currently running, we will just notify the user and exit. Otherwise, we will loop until the target process starts and then display the process name that started, and exit. The loop is listed in Listing 10-1.

```
RC=1
```

Listing 10-1 Process startup loop

There are a few things to point out in Listing 10-1. First, notice that we are using the numeric tests, which are specified by the double parentheses: ((numeric_expression)). The numeric tests can be seen in the if and until control structures. When using the double-parentheses numeric testing method, we do *not* need to reference any userdefined numeric variables with a dollar sign — that is, \$RC. The double-parentheses mathematical test knows that anything that is not a number is a variable. Just leave out the \$. We still must use the \$ reference for system and shell variables — for example, \$? and \$#. Also notice that we use double equal signs when making an equality test — for example, until ((RC == 0)). If you use only one equal sign it is assumed to be an assignment, not an equality test! Failure to use double equal signs is one of the most common mistakes, and it is very hard to find during troubleshooting. Also notice in Listing 10-1 that we sleep on each loop iteration. If we do not have a sleep interval, the load on the CPU can be tremendous. Try programming a loop with and without the sleep interval and monitor the CPU load with either the **uptime** or **vmstat** commands. You can definitely see a big difference in the load on the system. What does this mean for our monitoring? The process must remain running for at least the length of time that the sleep is executing on each loop iteration. If you need an interval of less than 1 second, you can try setting the sleep interval to 0, zero, but watch out for the heavy CPU load. Even with a 1-second interval the load can get to around 25 percent. An interval of about 3 to 10 seconds is not bad, if you can stand the wait.

Now let's study the loop. We initialize the return code variable, RC, to 1, one. Then we start an until loop that tests for the target process on each loop iteration. If the process is not running, the sleep is executed and then the loop is executed again. If the target process is found to be running, we give the user notification that the process has started, with the timestamp, and display to the user the process that actually started. We need to give the user this process information just in case the grep command got a pattern match on an unintended pattern. The entire script is on the book's web site (www.wiley.com/go/michael2e) with the name proc_wait.ksh. This is crude, but it works well. (See Listing 10-2.)

```
[root:yogi]@/scripts/WILEY/PROC_MON# ./proc_wait.ksh xcalc
WAITING for xcalc to start...Thu Sep 27 21:11:47 EDT 2007
xcalc has Started Execution...Thu Sep 27 21:11:55 EDT 2007
root 26772 17866 13 21:11:54 pts/6 0:00 xcalc
```

Listing 10-2 proc_wait.ksh script in action

Monitoring for a Process to End

Monitoring for a process to end is also a simple procedure because it is really the opposite of the previous shell script. In this new shell script we want to add some extra options. First, we set a trap and inform the user if an interrupt occurred — for example, Ctrl+C is pressed. It would be nice to give the user the option of *verbose* mode. The verbose mode enables the listing of the active process(es). We can use a -v switch as a command-line argument to the shell script to turn on the verbose mode. To parse through the command-line arguments, we could use the **getopts** command; but for only one or two arguments, we can easily use a nested case statement. We will show how to use getopts later in the chapter. Again, we will use the double parentheses for numeric tests wherever possible. For the proc_mon.ksh script we are going to list out the entire script and review the process at the end. (See Listing 10-3.)

```
#!/usr/bin/ksh
#
# SCRIPT: proc_mon.ksh
# AUTHOR: Randy Michael
# DATE: 02/14/2007
# REV: 1.1.P
# PLATFORM: Not Platform Dependent
# PURPOSE: This script is used to monitor a process to end
#
    specified by ARG1 if a single command-line argument is
#
   used. There is also a "verbose" mode where the monitored
#
   process is displayed and ARG2 is monitored.
# USAGE: proc_mon.ksh [-v] process-to-monitor
# EXIT STATUS:
    0 ==> Monitored process has terminated
#
    1 ==> Script usage error
```

Listing 10-3 proc_mon.ksh shell script

```
2 ==> Target process to monitor is not active
#
   3 ==> This script exits on a trapped signal
#
#
# REV. LIST:
#
#
  02/22/2007 - Added code for a "verbose" mode to output the
            results of the 'ps aux' command. The verbose
#
#
            mode is set using a "-v" switch.
# set -x # Uncomment to debug this script
# set -n # Uncomment to debug without any command execution
SCRIPT NAME='basename $0'
*********
****
function usage
{
   echo "\n\n"
   echo "USAGE: $SCRIPT_NAME [-v] {Process_to_monitor}"
   echo "\nEXAMPLE: $SCRIPT_NAME my_backup\n"
   echo "OR"
   echo "\nEXAMPLE: $SCRIPT_NAME -v my_backup\n"
   echo "Try again...EXITING...\n"
}
*****
function exit_trap
{
   echo "\n...EXITING on trapped signal...\n"
}
*****
******
#################
# Set a trap...#
##################
trap 'exit_trap; exit 3' 1 2 3 15
# First Check for the Correct Number of Arguments
# One or Two is acceptable
```

```
Listing 10-3 (continued)
```

```
if (( $# != 1 && $# != 2 ))
then
    usage
    exit 1
fi
# Parse through the command-line arguments and see if verbose
# mode has been specified. NOTICE that we assign the target
# process to the PROCESS variable!!!
# Embedded case statement...
case $# in
      1)
             case $1 in
              '-v') usage
                    exit 1
                    ;;
                 *) PROCESS=$1
        esac
        ;;
      2) case $1 in
                '-v') continue
                     ;;
           esac
           case $2 in
                '-v') usage
                      exit 1
                      ;;
                   *) PROCESS=$2
                      ;;
           esac
           ;;
      *) usage
         exit 1
         ;;
esac
# Check if the process is running or exit!
ps aux | grep "$PROCESS" | grep -v "grep $PROCESS" \
       grep -v $SCRIPT_NAME >/dev/null
if (( $? != 0 ))
then
     echo "\n\n$PROCESS is NOT an active process...EXITING...\n"
```

Listing 10-3 (continued)

```
exit 2
fi
# Show verbose mode if specified...
if (( $# == 2 )) && [[ $1 = "-v" ]]
then
     # Verbose mode has been specified!
    echo "\n"
     # Extract the columns heading from the ps aux output
    ps aux | head -n 1
    ps aux | grep "$PROCESS" | grep -v "grep $PROCESS" \
              | grep -v $SCRIPT_NAME
fi
##### 0.K. The process is running, start monitoring...
SLEEP_TIME="1" # Seconds between monitoring
RC="0"
                    # RC is the Return Code
echo "\n\n" # Give a couple of blank lines
echo "$PROCESS is currently RUNNING...`date`\n"
*****
# Loop UNTIL the $PROCESS stops...
while (( RC == 0 )) # Loop until the return code is not zero
do
    ps aux | grep $PROCESS | grep -v "grep $PROCESS" \
           grep -v $SCRIPT_NAME >/dev/null 2>&1
    if (( $? != 0 )) # Check the Return Code!!!!!
     then
         echo "\n...$PROCESS has COMPLETED...`date`\n"
         exit 0
     fi
     sleep $SLEEP_TIME # Needed to reduce CPU Load!!!
done
# End of Script
```

Did you catch all the extra hoops we had to jump through? Adding command switches can be problematic. We will see a much easier way to do this later using the getopts shell command.

In Listing 10-3 we first defined two functions, which are both used for abnormal operation. We always need a usage function, and in this shell script we added a

trap_exit function that is to be executed only when a trapped signal is captured. The trap definition specifies exit signals 1, 2, 3, and 15. Of course, you cannot trap exit signal 9. This trap_exit function will display ... EXITING on a trapped signal.... Then the trap will execute the second command, exit 3. In the next step we check for the correct number of command-line arguments, one or two, and use an embedded case statement to assign the target process to a variable, PROCESS. If -v is specified in the *first* argument, \$1, of two command-line arguments, then verbose mode is used. Verbose mode will display the ps aux output that the grep command did the pattern match on. Otherwise, this information is not displayed. This is the first time that we look to see if the target process is active. If the target process is not executing, we just notify the user and exit with a return code of 2. Next comes the use of verbose mode if the -v switch is specified on the command line. Notice how we pull out the ps command's output columns heading information before we display the process using ps aux | head -n 1. This helps the user confirm that this is the correct match with the column header. Now we know the process is currently running so we start a loop. This loop will continue until either the process ends or the program is interrupted — for example, Ctrl+C is pressed.

The proc_mon.ksh script did the job, but we have no logging and the monitoring stops when the process stops. It would be really nice to track the process as it starts and stops. If we can monitor the transition, we can keep a log file to review and see if we can find a trend. The proc_mon.ksh shell script is shown in action in Listing 10-4.

```
[root:yogi]@/scripts/WILEY/PROC_MON# ./proc_mon.ksh xcalc
xcalc is NOT an active process...EXITING...
[root:yogi]@/scripts/WILEY/PROC_MON# ./proc_mon.ksh xcalc
xcalc is currently RUNNING...Thu Sep 27 21:14:08 EDT 2007
...xcalc has COMPLETED...Thu Sep 27 21:14:26 EDT 2007
```

Listing 10-4 proc_mon.ksh shell script in action

Monitor and Log as a Process Starts and Stops

Catching process activity as it cycles on and off can be a useful tool in problem determination. In this section, we are going to expand on both of our previous scripts and monitor for both startup and end time for a target process. We are also going to log everything and timestamp each start and stop event. Because we are logging everything, we also want to see the same data as it happens on the screen. The log file

can be reviewed at any time; we want to see it in "real time" (at least close to real time). We are going to make the startup and end time monitoring into functions this time, and as a result we are going to need to capture the current tty device, which may be a pseudo-terminal (pty), to use within these functions. The **tty** command will show the current terminal, and we can save this in a variable. For concurrent display and logging within the script, we pipe our output to tee -a \$LOGFILE. This **tee** command sends the output to both standard output and to the file that \$LOGFILE points to. But inside the functions we will use the specific tty device to send our output to, which we assign to a variable called TTY. Enough with the fluff; here is the script (in Listing 10-5), followed by a short explanation.

```
#!/bin/ksh
#
# SCRIPT: proc_watch.ksh
# AUTHOR: Randy Michael
# DATE: 09-12-2007
# REV: 1.0.P
# PLATFORM: Not Platform Dependent
# PURPOSE" This script is used to monitor and log
     the status of a process as it starts and stops.
#
#
# REV LIST:
# set -x # Uncomment to debug this script
# set -n # Uncomment to check syntax without ANY execution
LOGFILE="/tmp/proc_status.log"
[[ ! -s $LOGFILE ]] && touch $LOGFILE
PROCESS="$1" # Process to Monitor
SCRIPT_NAME=$(basename $0) # Script Name w/o the PATH
TTY=$(tty) # Current tty or pty
**********
*********
usage ()
{
   echo "\nUSAGE: $SCRIPT_NAME process_to_monitor\n"
}
```

Listing 10-5 proc_watch.ksh shell script

```
****
trap_exit ()
{
   # Log an ending time for process monitoring
   TIMESTAMP=$(date +%D0%T) # Get a new timestamp...
   echo "MON_STOP: Monitoring for $PROCESS ended ==> $TIMESTAMP" \
        | tee -a $LOGFILE
   # Kill all functions
   kill -9 $(jobs -p) 2>/dev/null
}
****
mon_proc_end ()
{
    END_RC="0"
    until (( END_RC != 0 ))
    do
        ps aux | grep -v "grep $PROCESS" | grep -v $SCRIPT_NAME \
               grep $PROCESS >/dev/null 2>&1
        END_RC=$? # Check the Return Code!!
        sleep 1 # Needed to reduce CPU load!
    done
    echo 'N' # Turn the RUN flag off
    # Grab a Timestamp
    TIMESTAMP=$(date +%D@%T)
    echo "END PROCESS: $PROCESS ended ==> $TIMESTAMP" >> $LOGFILE &
    echo "END PROCESS: $PROCESS ended ==> $TIMESTAMP" > $TTY
}
*****
mon_proc_start ()
{
    START RC="-1" # Initialize to -1
    until (( START_RC == 0 ))
    do
        ps aux | grep -v "grep $PROCESS" | grep -v $SCRIPT_NAME \
               grep $PROCESS >/dev/null 2>&1
        START_RC=$? # Check the Return Code!!!
                   # Needed to reduce CPU load!
        sleep 1
```

```
Listing 10-5 (continued)
```

```
done
    echo 'Y' # Turn the RUN flag on
    # Grab the timestamp
    TIMESTAMP=$(date +%D@%T)
    echo "START PROCESS: $PROCESS began ==> $TIMESTAMP" >> $LOGFILE &
    echo "START PROCESS: $PROCESS began ==> $TIMESTAMP" > $TTY
}
****
*****
### SET A TRAP ####
trap 'trap_exit; exit 0' 1 2 3 15
# Check for the Correct Command Line Argument - Only 1
if (( $# != 1 ))
then
    usage
    exit 1
fi
# Get an Initial Process State and Set the RUN Flag
ps aux | grep -v "grep $PROCESS" | grep -v $SCRIPT_NAME \
      grep $PROCESS >/dev/null
PROC_RC=$? # Check the Return Code!!
# Give some initial feedback before starting the loop
if (( PROC_RC == 0 ))
then
    echo "The $PROCESS process is currently running....Monitoring...."
    RUN="Y"
               # Set the RUN Flag to YES
else
   echo "The $PROCESS process is not currently running...Monitoring..."
               # Set the RUN Flag to NO
    RUN="N"
fi
TIMESTAMP=$(date +%D@%T) # Grab a timestamp for the log
# Use a "tee -a $#LOGFILE" to send output to both standard output
```

Listing 10-5 (continued)

```
# and to the file referenced by $LOGFILE
echo "MON_START: Monitoring for $PROCESS began ==> $TIMESTAMP" \
      tee -a $LOGFILE
# Loop Forever!!
while :
do
     case $RUN in
     'Y') # Loop Until the Process Ends
          RUN=$(mon_proc_end)
          ;;
     'N') # Loop Until the Process Starts
          RUN=$(mon_proc_start)
          ;;
     esac
done
# End of Script
```

The shell script in Listing 10-5 is a nice, modular shell script. The actual monitoring loop is the final while loop. The loop is short and tight, with all the work being done within the two functions, proc_mon_start and proc_mon_end. Notice that in both functions we remain in the loop until there is a transition from run to stop, or from not running to process startup. On each transition we return updated run status information back to the calling shell script with an **echo** command, as opposed to a return code. For the concurrent display to the screen and logging to the file, we use tee -a \$LOGFILE within the shell script body, and in the functions we redirect output to the tty device that we assigned to the \$TTY variable. We use the tty device to ensure that the screen output will go to the terminal, or pseudo-terminal, that we are currently looking at. We again did all numeric tests with the double-parentheses method. Notice that we do *not* use a \$ with a user-defined variable! For the while loop we are looping forever. The no-op character (while :) allows this to work (true would also work). The proc_watch.ksh shell script will continue to run until it is interrupted — for example, until Ctrl+C is pressed.

We have improved our script, but it does not let us know how many processes are active. There is no timing mechanism for the shell script; it just runs until interrupted. We are next going to expand on our script to do a few things differently. First, we want to be able to time the monitoring to execute for a specific period of time. We also want to let the user know how many processes are currently active and the PID of each process. In addition, we want to timestamp *each* process's startup and end time. To timestamp each process we can count the number of processes that are running during each loop iteration, and if the count changes we will grab a new timestamp and update the PID list for the currently running processes. We also will give the option to run some pre-startup, and/or post-event before the process starts, as the process

starts, or after the process has ended. We can see the proc_watch.ksh shell script in action in Listing 10-6.

```
[root:yogi]@/scripts/WILEY/PROC_MON# ./proc_watch.ksh xcalc
The xcalc process is currently running...Monitoring...
MON_START: Monitoring for xcalc began ==> 09/27/07@21:09:41
END PROCESS: xcalc ended ==> 09/27/07@21:09:56
START PROCESS: xcalc began ==> 09/27/07@21:10:06
END PROCESS: xcalc ended ==> 09/27/07@21:10:25
^C
MON_STOP: Monitoring for xcalc ended ==> 09/27/07@21:10:31
```

Listing 10-6 proc_watch.ksh shell script in action

Timed Execution for Process Monitoring, Showing Each PID, and Timestamp with Event and Timing Capability

Sound like a lot? After we get through this section, each step will be intuitively obvious. In the previous three scripts, we had no ability to monitor *each* process that matched the grepped pattern or to execute the monitoring for a specific amount of time. Because we are using the grep command, we might get multiple matches to a pattern. In case of multiple matches, we need to know (1) how many matches we have and (2) *each* process that was matched. This information can be very beneficial if we are monitoring a specific user's activities or anything where we are interested in the exact process IDs that are running.

We also want a good timing mechanism that will allow for easy, flexible timing of the duration of the monitoring activity. Because we have no way of knowing what user requirements may be, we want to allow for as much flexibility as possible. Let's go to the far side and allow timing from seconds to days, and anything in between. The easiest way to handle timing, but not the most accurate, is to add up all the seconds and count down from the total seconds to zero while sleeping for one second between counts. We could continuously check the date/time using the **date** command for a very accurate time, or — even better — we can kick off an at job to kill the script at some specific time in the future. The shell variable SECONDS is also useful. For this script we are going to use getopts to parse the command line for seconds, minutes, hours, days, and the process to monitor. Then we add up the seconds and count down to zero and quit. Alternatively, if the total seconds and a process are the only arguments, the user will be able to enter these directly — for only a process and total seconds getopts will not be used. The usage function will list two ways to use our new script.

Another nice option is the capability to run pre-startup, and/or post-events. By pre-startup, and post-events we are talking about running some command, script, or function before the process starts, as the process starts, after the process stops, or in

any combination. As an example, we may want to reboot the machine after a backup program ends, or we may want to set up environment variables before some process starts up. For the event options we also need to be as flexible as possible. For flexibility we will just add a function for each event that contains only the no-op character, : (colon), as a placeholder. A colon does not execute anything; it does nothing and always has a return code of 0, zero. Anything that a user may want to run before startup, at startup, or after the process has ended can be added into the appropriate function. We will use flags, or variables, to enable and disable the pre-startup, and post-events individually.

In this section we are going to do two things that may be new, using getopts to process the command-line arguments and executing a function in the background as a *co-process*. The getopts functionality is an easy and efficient way to parse through mixed command-line arguments, and the command switches can be with or without switch arguments. A co-process is an easy way to set up a communication link with a *background* script or function and the foreground.

Let's first look at how to use getopts to parse the command line. The getopts command is built into the shell. We use getopts to parse the command line for valid options specified by a single character, following a – (minus sign) or a + (plus sign). To specify that a command switch requires an argument, the switch character definition must be followed by a : (colon). If the switch does not require any argument, the : should be omitted. All the switch options put together are called the OptionString, and this is followed by some variable name that we define. The argument for each switch is stored in a variable called OPTARG as the arguments are parsed in a loop one at a time. If the *entire* OptionString is *preceded* by a : (colon), then any unmatched switch option causes a ? (question mark) to be loaded into the variable that we defined in the getopts command string. The form of the command follows:

getopts OptionString Name [Argument ...]

The easiest way to explain the getopts command is with an example. For our script we need seconds, minutes, hours, days, and a process to monitor. For each one we want to supply an argument — for example, -s 5 -m10 -p my_backup. In this example we are specifying 5 seconds, 10 minutes, and the process is my_backup. Notice that there does not have to be a space between the switch and the argument, and the arguments can be in any order on the command line. This is what makes getopts so great! The code to set up our example looks like Listing 10-7.

```
SECS=0  # Initialize all to zero
MINUTES=0
HOURS=0
DAYS=0
PROCESS=  # Initialize to null
while getopts ":s:m:h:d:p:" TIMED 2>/dev/null
do
```

Listing 10-7 Example getopts command usage

```
case $TIMED in
       s) SECS=$OPTARG
          ;;
       m) (( MINUTES = $OPTARG * 60 ))
          ;;
       h) (( HOURS = $OPTARG * 3600 ))
          ;;
       d) (( DAYS = $OPTARG * 86400 ))
          ;;
       p) PROCESS=$OPTARG
          ;;
       \?) usage
           exit 1
           ;;
    esac
done
(( TOTAL_SECONDS = SECS + MINUTES + HOURS + DAYS ))
```

There are a few things to note in Listing 10-7. The getopts command needs to be part of a while loop with a case statement within the loop. On each option we specified, -s, -m, -h, -d, and -p, we added a : (colon) after each switch character. This tells getopts that an argument is required for that particular switch character. The : (colon) before the OptionString list tells getopts that if an unspecified option is given on the command line, then getopts should set the TIMED variable to the ? character. The ? allows us to call the usage function and exit with a return code of 1 for an incorrect command-line option. The only thing to be careful of is that getopts does not care what arguments it receives, so it is our responsibility to check each argument to ensure that it meets our expectations; then we have to take action if we want to exit. The last thing to note in Listing 10-7 is that the first line of the while loop has redirection of the standard error (file descriptor 2) to the bit bucket. Anytime an unexpected argument is encountered, getopts sends a message to standard error, but it is not considered an error, just informational. Because we expect that incorrect command-line arguments may be entered, we can just ignore the messages and discard them with redirection to /dev/null, a.k.a. the bit bucket.

We also need to cover setting up a co-process. A *co-process* is a communications link between a foreground and a background process. The most common question is, "*Why* is this needed?" In our next script we are going to call a function that will handle all the monitoring for us while we do the timing control in the main script. The problem arises because *we need to run this function in the background*. Within the background process-monitoring function there are two loops, of which one loop is always executing. Without the ability to tell the loop to break out of the internal loop, it will continue to execute on its own after the main script and function have exited due to an interrupt. We know what this causes — *one or more defunct processes!* From the main script we need a way to communicate with the loop in the background function

to tell it to break out of the loop or exit the function cleanly when the countdown is complete and if the script is interrupted — for example, with Ctrl+C. To solve this little problem we kick off our background proc_watch function as a co-process. "How do we do this?" you ask. "*Pipe it to the background*" is the simplest way to put it, and that is also what it looks like. Look at the example in Listing 10-8.

```
function proc_watch
# This function is started as a co-process!!!
   while : # Loop forever
   do
         Some Code Here
         read BREAK_OUT # Do NOT need a "-p" to read!
         if [[ $BREAK_OUT = 'Y' ]]
         then
             return 0
         fi
   done
}
##### Start of Main #######
### Set a Trap ###
trap 'BREAK='Y'; print -p $BREAK; exit 2' 1 2 3 15
TOTAL_SECONDS=300
BREAK_OUT='N'
proc_watch &
                 # Start proc_watch as a co-process!!!!
until (( TOTAL_SECONDS == 0 ))
do
    (( TOTAL_SECONDS = TOTAL_SECONDS - 1 ))
    sleep 1
done
BREAK OUT='Y'
# Use "print -p" to communicate with the co-process variable
print -p $BREAK_OUT
exit 0
```

Listing 10-8 Example of using a co-process

In the code block in Listing 10-8 we defined the proc_watch function, which is the function that we want to start as a *background* process. As you can see, the proc_watch function has an infinite loop. If the main script is interrupted, then without a means to exit the loop within the proc_watch background function, the loop alone will continue to execute! To solve this we start the proc_watch as a co-process by "piping it to the background" using *pipe ampersand*, |&, as a suffix. Now when we want to communicate with the function from the main script, we use print -p \$BREAK_OUT. Inside the function, we just use the standard read command, read BREAK_OUT. The co-process is the mechanism that we are going to use to break out of the loop if the main script is interrupted on a trapped signal, and for normal countdown termination at the end of the script. Of course, we can never catch kill -9 with a trap.

Try setting up the scenario just described, without a co-process, with a background function that has an infinite loop. Then press the Ctrl+C key sequence to kill the main script and do a ps aux | more. You will see that the background *loop* is still executing! Get the PID, and do a kill -9 to kill it. Of course, if the loop's exit criteria are ever met, the loop will exit on its own.

Now take a look at the entire script, and see how we handled all these extra requirements. Pay close attention to the highlighted code in Listing 10-9.

```
#!/bin/ksh
#
# SCRIPT: proc_watch_timed.ksh
# AUTHOR: Randy Michael
# DATE: 09-14-2007
# REV: 1.0.P
# PLATFORM: Not Platform Dependent
# PURPOSE: This script is used to monitor and log
#
          the status of a process as it starts and stops.
#
         Command line options are used to identify the target
          process to monitor and the length of time to monitor.
#
#
          Each event is logged to the file defined by the
          $LOGFILE variable. This script also has the ability
#
          to execute pre, startup, and post events. These are
#
          controlled by the $RUN_PRE_EVENT, $RUN_STARTUP_EVENT,
#
          and $RUN_POST_EVENT variables. These variables control
#
#
          execution individually. Whatever is to be executed is to
          be placed in either the "pre_event_script",
#
#
          startup_event_script, or the
#
          "post_event_script" functions, or in any combination.
          Timing is controlled on the command line.
          USAGE: $SCRIPT_NAME total_seconds target_process
#
#
          Will monitor the specified process for the
#
           specified number of seconds.
```

Listing 10-9 proc_watch_timed.ksh shell script

```
USAGE: $SCRIPT_NAME [-s|-S seconds] [-m|-M minutes]
#
#
                           [-h|-H hours] [-d|-D days]
#
                           [-p|-P process]
#
#
        Will monitor the specified process for number of
        seconds specified within -s seconds, -m minutes,
#
        -h hours, and -d days. Any combination of command
#
        switches can be used.
# REV LIST:
# set -x # Uncomment to debug this script
# set -n # Uncomment to check syntax without ANY execution
******
****
typeset -u RUN_PRE_EVENT # Force to UPPERCASE
typeset -u RUN_STARTUP_EVENT # Force to UPPERCASE
typeset -u RUN_POST_EVENT # force to UPPERCASE
RUN_PRE_EVENT='N' # A 'Y' will execute, anything else will not
RUN_STARTUP_EVENT='Y' # A 'Y' will execute, anything else will not
RUN POST EVENT='Y' # A 'Y' will execute, anything else will not
LOGFILE="/tmp/proc status.log"
[[ ! -s $LOGFILE ]] && touch $LOGFILE
SCRIPT_NAME=$(basename $0)
TTY=$(tty)
INTERVAL="1" # Seconds between sampling
JOBS=
*****
*****
usage ()
{
echo "\n\n\t****USAGE ERROR*****"
echo "\n\nUSAGE: $SCRIPT NAME seconds process"
echo "\nWill monitor the specified process for the"
echo "specified number of seconds."
echo "\nUSAGE: $SCRIPT_NAME [-s|-S seconds] [-m|-M minutes]"
          [-h|-H hours] [-d|-D days] [-p|-P process]\n"
echo "
echo "\nWill monitor the specified process for number of"
echo "seconds specified within -s seconds, -m minutes,"
```

```
echo "-h hours and -d days. Any combination of command"
echo "switches can be used.\n"
echo "\nEXAMPLE: $SCRIPT_NAME 300 dtcalc"
echo "\n\nEXAMPLE: $SCRIPT_NAME -m 5 -p dtcalc"
echo "\nBoth examples will monitor the dtcalc process"
echo "for 5 minutes. Can specify days, hours, minutes"
echo "and seconds, using -d, -h, -m and -s \n'"
}
*******
trap_exit ()
# set -x # Uncommant to debug this function
# Log an ending time for process monitoring
echo "INTERRUPT: Program Received an Interrupt...EXITING..." > $TTY
echo "INTERRUPT: Program Received an Interrupt...EXITING..." >> $LOGFILE
TIMESTAMP=$(date +%D@%T) # Get a new timestamp...
echo "MON_STOPPED: Monitoring for $PROCESS ended ==> $TIMESTAMP\n" \
     >> $TTY
echo "MON_STOPPED: Monitoring for $PROCESS ended ==> $TIMESTAMP\n" \
     >> $LOGFILE
echo "LOGFILE: All Events are Logged ==> $LOGFILE \n" > $TTY
# Kill all functions
JOBS=$(jobs -p)
if [[ ! -z $JOBS && $JOBS != '' && $JOBS != '0' ]]
then
     kill $(jobs -p) 2>/dev/null 1>&2
fi
return 2
l
*****
pre_event_script ()
# Put anything that you want to execute BEFORE the
# monitored process STARTS in this function
: # No-OP - Needed as a placeholder for an empty function
# Comment Out the Above colon, ':'
PRE RC=$?
return $PRE_RC
*****
startup_event_script ()
# Put anything that you want to execute WHEN, or AS, the
# monitored process STARTS in this function
: # No-OP - Needed as a placeholder for an empty function
```

```
# Comment Out the Above colon, ':'
STARTUP RC=$?
return $STARTUP_RC
}
*****
post_event_script ()
{
# Put anything that you want to execute AFTER the
# monitored process ENDS in this function
: # No-OP - Need as a placeholder for an empty function
# Comment Out the Above colon, ':'
POST RC=$?
return $POST_RC
}
*****
# This function is used to test character strings
test_string ()
{
if (( $# != 1 ))
then
  echo 'ERROR'
   return
fi
C STRING=$1
# Test the character string for its composition
case $C_STRING in
    +([0-9])) echo 'POS_INT' # Integer >= 0
             ;;
    +([-0-9])) echo 'NEG_INT' # Integer < 0
             ;;
    +([a-z])) echo 'LOW_CASE' # lower case text
             ;;
    +([A-Z])) echo 'UP_CASE' # UPPER case text
             ;;
    +([a-z] [A-Z])) echo 'MIX_CASE' # MIxed CAse text
             ;;
            *) echo 'UNKNOWN' # Anything else
esac
}
*****
proc_watch ()
```

```
\# set -x \# Uncomment to debug this function
# This function does all of the process monitoring!
while :
            # Loop Forever!!
do
    case $RUN in
    'Y')
          # This will run the startup_event_script, which is a function
          if [[ $RUN_STARTUP_EVENT = 'Y' ]]
          then
             echo "STARTUP EVENT: Executing Startup Event Script..."\
                  > $TTY
             echo "STARTUP EVENT: Executing Startup Event Script..."\
                  >> $LOGFILE
             startup_event_script # USER DEFINED FUNCTION!!!
             RC=$? # Check the Return Code!!
             if (( "RC" == 0 ))
             then
                 echo "SUCCESS: Startup Event Script Completed RC -\
 ${RC}" > $TTY
                 echo "SUCCESS: Startup Event Script Completed RC -\
 ${RC}" >> $LOGFILE
             else
                 echo "FAILURE: Startup Event Script FAILED RC -\
 ${RC}" > $TTY
                 echo "FAILURE: Startup Event Script FAILED RC -\
 ${RC}" >> $LOGFILE
             fi
          fi
          integer PROC_COUNT='-1' # Reset the Counters
          integer LAST_COUNT='-1'
          # Loop until the process(es) end(s)
          until (( "PROC_COUNT" == 0 ))
          do
               # This function is a Co-Process. $BREAK checks to see if
               # "Program Interrupt" has taken place. If so BREAK will
               # be 'Y' and we exit both the loop and function.
               read BREAK
               if [[ $BREAK = 'Y' ]]
               then
                     return 3
               fi
```

```
PROC_COUNT=$(ps aux | grep -v "grep $PROCESS" \
                          | grep -v $SCRIPT_NAME \
                          grep $PROCESS | wc -1) >/dev/null 2>&1
              if (( "LAST_COUNT" > 0 && "LAST_COUNT" != "PROC_COUNT" ))
              then
                   # The Process Count has Changed ...
                   TIMESTAMP=$(date +%D@%T)
                   # Get a list of the PID of all of the processes
                   PID_LIST=$(ps aux | grep -v "grep $PROCESS" \
                          grep -v $SCRIPT_NAME \
                          grep $PROCESS | awk '{print $2}')
                   echo "PROCESS COUNT: $PROC_COUNT $PROCESS\
Processes Running ==> $TIMESTAMP" >> $LOGFILE &
                   echo "PROCESS COUNT: $PROC_COUNT $PROCESS\
Processes Running ==> $TIMESTAMP" > $TTY
                   echo ACTIVE PIDS: $PID_LIST >> $LOGFILE &
                   echo ACTIVE PIDS: $PID_LIST > $TTY
              fi
              LAST_COUNT=$PROC_COUNT
              sleep $INTERVAL # Needed to reduce CPU load!
         done
         RUN='N' # Turn the RUN Flag Off
         TIMESTAMP=$(date +%D@%T)
         echo "ENDING PROCESS: $PROCESS END time ==>\
$TIMESTAMP" >> $LOGFILE &
         echo "ENDING PROCESS: $PROCESS END time ==>\
$TIMESTAMP" > $TTY
         # This will run the post_event_script, which is a function
         if [[ $RUN_POST_EVENT = 'Y' ]]
         then
             echo "POST EVENT: Executing Post Event Script..."\
                   > $TTY
             echo "POST EVENT: Executing Post Event Script..."\
                   >> $LOGFILE &
             post_event_script # USER DEFINED FUNCTION!!!
             integer RC=$?
             if (( "RC" == 0 ))
             then
                 echo "SUCCESS: Post Event Script Completed RC -\
${RC}" > $TTY
```

```
echo "SUCCESS: Post Event Script Completed RC -\
${RC}" >> $LOGFILE
             else
                 echo "FAILURE: Post Event Script FAILED RC - ${RC}"\
                       > $TTY
                 echo "FAILURE: Post Event Script FAILED RC - ${RC}"\
                       >> $LOGFILE
             fi
         fi
    ;;
    'N')
         # This will run the pre_event_script, which is a function
         if [[ $RUN_PRE_EVENT = 'Y' ]]
         then
            echo "PRE EVENT: Executing Pre Event Script..." > $TTY
            echo "PRE EVENT: Executing Pre Event Script..." >> $LOGFILE
            pre_event_script # USER DEFINED FUNCTION!!!
            RC=$?
                    # Check the Return Code!!!
            if (( "RC" == 0 ))
            then
                 echo "SUCCESS: Pre Event Script Completed RC - ${RC}"\
                       > $TTY
                 echo "SUCCESS: Pre Event Script Completed RC - ${RC}"\
                       >> $LOGFILE
            else
                 echo "FAILURE: Pre Event Script FAILED RC - ${RC}"\
                       > $TTY
                 echo "FAILURE: Pre Event Script FAILED RC - ${RC}"\
                       >> $LOGFILE
            fi
         fi
         echo "WAITING: Waiting for $PROCESS to
startup...Monitoring..."
         integer PROC_COUNT='-1' # Initialize to a fake value
         # Loop until at least one process starts
         until (( "PROC_COUNT" > 0 ))
         do
              # This is a Co-Process. This checks to see if a "Program
              # Interrupt" has taken place. If so BREAK will be 'Y' and
              # we exit both the loop and function
              read BREAK
```

```
if [[ $BREAK = 'Y' ]]
             then
                  return 3
             fi
             PROC_COUNT=$(ps aux | grep -v "grep $PROCESS" \
                   | grep -v $SCRIPT_NAME | grep $PROCESS | wc -1) \
                    >/dev/null 2>&1
             sleep $INTERVAL # Needed to reduce CPU load!
        done
        RUN='Y' # Turn the RUN Flag On
        TIMESTAMP=$(date +%D0%T)
        PID_LIST=$(ps aux | grep -v "grep $PROCESS" \
                   grep -v $SCRIPT_NAME \
                  grep $PROCESS | awk '{print $2}')
        if (( "PROC_COUNT" == 1 ))
        then
             echo "START PROCESS: $PROCESS START time ==>\
$TIMESTAMP" >> $LOGFILE &
             echo ACTIVE PIDS: $PID_LIST >> $LOGFILE &
             echo "START PROCESS: $PROCESS START time ==>\
$TIMESTAMP" > $TTY
             echo ACTIVE PIDS: $PID_LIST > $TTY
        elif (( "PROC COUNT" > 1 ))
        then
             echo "START PROCESS: $PROC COUNT $PROCESS\
Processes Started: START time ==> $TIMESTAMP" >> $LOGFILE &
             echo ACTIVE PIDS: $PID_LIST >> $LOGFILE &
             echo "START PROCESS: $PROC_COUNT $PROCESS\
Processes Started: START time ==> $TIMESTAMP" > $TTY
             echo ACTIVE PIDS: $PID LIST > $TTY
        fi
    ; ;
  esac
done
}
**********
*****
### SET A TRAP ####
trap 'BREAK='Y';print -p $BREAK 2>/dev/null;trap_exit\
```

Listing 10-9 (continued)

```
2>/dev/null;exit 0' 1 2 3 15
BREAK='N' # The BREAK variable is used in the co-process proc_watch
PROCESS= # Initialize to null
integer TOTAL_SECONDS=0
# Check commnand line arguments
if (( $# > 10 || $# < 2 ))
then
     usage
     exit 1
fi
# Check to see if only the seconds and a process are
# the only arguments
if [[ ($# -eq 2) && ($1 != -*) && ($2 != -*) ]]
then
     NUM_TEST=$(test_string $1) # Is this an Integer?
     if [[ "$NUM_TEST" = 'POS_INT' ]]
     then
          TOTAL_SECONDS=$1 # Yep - It's an Integer
          PROCESS=$2
                           # Can be anything
     else
          usage
          exit 1
     fi
else
     # Since getopts does not care what arguments it gets, let's
     # do a quick sanity check to make sure that we only have
     # between 2 and 10 arguments and the first one must start
     # with a -* (hyphen and anything), else usage error
     case "$#" in
     [2-10]) if [[ $1 != -* ]]; then
                 usage; exit 1
             fi
          ;;
     esac
     HOURS=0
               # Initialize all to zero
     MINUTES=0
     SECS=0
     DAYS=0
     # Use getopts to parse the command line arguments
```

```
# For each $OPTARG for DAYS, HOURS, MINUTES and DAYS check to see
     # that each one is an integer by using the check_string function
     while getopts ":h:H:m:M:s:S:d:D:P:p:" OPT_LIST 2>/dev/null
     do
       case $OPT_LIST in
       h|H) [[ $(test_string $OPTARG) != 'POS_INT' ]] && usage && exit 1
            (( HOURS = $OPTARG * 3600 )) # 3600 seconds per hour
            ;;
       m|H) [[ $(test_string $OPTARG) != 'POS_INT' ]] && usage && exit 1
            (( MINUTES = $OPTARG * 60 )) # 60 seconds per minute
            ;;
       s|S) [[ $(test_string $OPTARG) != 'POS_INT' ]] && usage && exit 1
            SECS="$OPTARG"
                                      # seconds are seconds
            ;;
       d|D) [[ $(test_string $OPTARG) != 'POS_INT' ]] && usage && exit 1
            (( DAYS = $OPTARG * 86400 )) # 86400 seconds per day
            ;;
       p P) PROCESS=$OPTARG
                                         # process can be anything
            ;;
                                         # USAGE ERROR
        \?) usage
            exit 1
            ;;
         :) usage
            exit 1
            ;;
         *) usage
            exit 1
            ;;
       esac
     done
fi
# We need to make sure that we have a process that
# is NOT null or empty! - sanity check - The double quotes are required!
if [[ -z "$PROCESS" || "$PROCESS" = '' ]]
then
     usage
     exit 1
fi
# Check to see that TOTAL_SECONDS was not previously set
if (( TOTAL_SECONDS == 0 ))
then
     # Add everything together if anything is > 0
```

```
if [[ $SECS -gt 0 || $MINUTES -gt 0 || $HOURS -gt 0 \
          || $DAYS -gt 0 ]]
    then
         (( TOTAL_SECONDS = SECS + MINUTES + HOURS + DAYS ))
    fi
fi
# Last Sanity Check!
if (( TOTAL_SECONDS <= 0 )) || [ -z $PROCESS ]
then
    # Either There are No Seconds to Count or the
    # $PROCESS Variable is Null...USAGE ERROR...
    usage
    exit 1
fi
echo "\nCurrently running $PROCESS processes:\n" > $TTY
ps aux | grep -v "grep $PROCESS" | grep -v $SCRIPT_NAME \
      | grep $PROCESS > $TTY
PROC_RC=$? # Get the initial state of the monitored function
echo >$TTY # Send a blank line to the screen
(( PROC_RC != 0 )) && echo "\nThere are no $PROCESS processes running\n"
if (( PROC_RC == 0 )) # The Target Process(es) is/are running...
then
    RUN='Y' # Set the RUN flag to true, or yes.
    integer PROC_COUNT # Strips out the "padding" for display
    PROC_COUNT=$(ps aux | grep -v "grep $PROCESS" | grep -v \
                 $SCRIPT_NAME | grep $PROCESS | wc -1) >/dev/null 2>&1
    if (( PROC_COUNT == 1 ))
    then
         echo "The $PROCESS process is currently\
 running...\n"
    elif (( PROC_COUNT > 1 ))
    then
         print "There are $PROC_COUNT $PROCESS processes currently\
 running...\n"
    fi
else
```

```
echo "The $PROCESS process is not currently running...monitoring..."
    RUN='N' # Set the RUN flag to false, or no.
fi
TIMESTAMP=$(date +%D0%T) # Time that this script started monitoring
# Get a list of the currently active process IDs
PID_LIST=$(ps aux | grep -v "grep $PROCESS" \
                  grep -v $SCRIPT_NAME \
                  grep $PROCESS | awk '{print $2}')
echo "MON_STARTED: Monitoring for $PROCESS began ==> $TIMESTAMP" \
      tee -a $LOGFILE
echo ACTIVE PIDS: $PID_LIST | tee -a $LOGFILE
##### NOTICE ####
# We kick off the "proc_watch" function below as a "Co-Process"
# This sets up a two way communications link between the
# "proc_watch" background function and this "MAIN BODY" of
# the script. This is needed because the function has two
# "infinite loops", with one always executing at any given time.
# Therefore we need a way to break out of the loop in case of
# an interrupt, i.e. CTRL+C, and when the countdown is complete.
# The "pipe appersand", |_{\&}, creates the background Co-Process
# and we use "print -p $VARIABLE" to transfer the variable's
# value back to the background co-process.
*****
proc_watch |& # Create a Background Co-Process!!
WATCH_PID=$! # Get the process ID of the last background job!
# Start the Count Down!
integer SECONDS_LEFT=$TOTAL_SECONDS
while (( SECONDS_LEFT > 0 ))
do
     # Next send the current value of $BREAK to the Co-Process
     # proc_watch, which was piped to the background...
    print -p $BREAK 2>/dev/null
     (( SECONDS_LEFT = SECONDS_LEFT - 1 ))
     sleep 1 # 1 Second Between Counts
done
```

Listing 10-9 (continued)

The most important things to note in Listing 10-9 are the communication links used between the foreground main script and the background co-process function, proc_watch, and the use of getopts to parse the command-line arguments. Some other things to look at are the integer tests using the string_test function and the way that the user is notified of a new process either starting or stopping by timestamp. The updated process count, and the listing of all of the PIDs, and how text is sent to the previously captured tty display within the function. As usual, we use the double-parentheses numerical test method in the control structures. (Notice again that the \$ is not used to reference the user-defined variables!) This shell script is also full of good practices for using different control structures and the use of the logical AND and logical OR (&& and ||), which reduces the need for if...then...else... and case structures. One very important test needs to be pointed out — the "null/empty" test for the PROCESS variable just after getopts parses the command line. This test is so important because the getopts command does not care what arguments it is parsing; nothing will "error out." For this reason, we need to verify all the variables ourselves. The only thing getopts is doing is matching the command switches to the appropriate arguments, not the validity of the command-line argument! If this test is left out and invalid command-line arguments are present, grep command errors will cover the screen during the script's execution — bad; very bad!

A good review of Listing 10-9 is needed to point out some other interesting aspects. Let's start at the top.

In the definitions of the files and variables there are three variables that control the execution of the pre-startup, and post-events. The variables are RUN_PRE_EVENT, RUN_STARTUP_EVENT, and RUN_POST_EVENT, and for ease of testing, the variables are typeset to UPPERCASE. A "Y" will enable the execution of the function, in which a user can put anything that he or she wants to run. The *functions* are called pre_event_script, startup_event_script, and post_event_script, but don't

let the names fool you. We also identify the LOGFILE variable and test to see if a log file exists. If the file does not exist, we touch the \$LOGFILE variable, which creates an empty file with the filename that the \$LOGFILE variable points to. This script section also grabs the SCRIPT_NAME using the basename \$0 command, and we define the current tty device for display purposes. An important variable is INTERVAL. This variable defines the number of seconds between sampling the process list. It is *very* important that this value is greater than 0, zero! If the INTERVAL value is set to 0, zero, the CPU load will be extreme and will produce a noticeable load, to say the least.

The next section in Listing 10-9 defines all the functions used in this script. We have a usage function that is displayed for usage errors. Then there is the trap_exit function. The trap_exit function will execute on exit codes 1, 2, 3, and 15, which we will see in the trap statement later at START OF MAIN in the script. Next are the pre_event_script, startup_event_script, and post_event_script functions. You may ask why a function would have a name indicating it is a script. It is done this way to encourage the use of an external script, or program, for any pre-startup, or post-event activity, rather than editing this script and debugging an internal function. The next function is used to test character strings, thus the name test_string. If you have ever wondered how to test a string (the entire string!) for its composition, test_string will do the trick. We just use a regular expression test for a range of characters. The preceding + (plus sign) is required in this regular expression to specify that all characters are of the specified type.

NOTE Bash shell does not support the regular expression notation + (reg_expression) to test strings.

Then comes the main function in the script that does all the work, proc_watch. This function is also the one that is executed as the co-process that we have been talking so much about. The proc_watch function is an infinite loop that contains two internal loops, where one internal loop is always executing at any given time. During both of these internal loops we check the variable BREAK to see if the value is 'Y'. The 'Y' value indicates that the function should exit immediately. The BREAK variable is updated, or changed, from the main script and is "transferred" to this co-process background function using the print -p \$BREAK command within the main script. This variable is reread in the function, during each loop iteration, using the standard read BREAK command. This is what enables the clean exit from the background function's loop. The word background is key to understanding the need for the co-process. If the main script is interrupted, the *innermost* loop will continue to execute even after both the function and script end execution. It will exit on its own when the loop's exit parameters are met, but if they are never met we end up with a defunct process. To get around this problem we start the proc_watch function as a background co-process using | & as a suffix to the function — for example, proc_watch &. An easy way to think of a co-process is as a *pipe to the background*, and through this pipe we have a communications link.

For the main part of the shell script, at the START OF MAIN, we first set a trap. In the **trap** command, we set the BREAK variable to 'Y' to indicate that the proc_watch

co-process should exit, and we make the new BREAK value known to the co-process with the print -p \$BREAK 2 > /dev/null command. This command sometimes sends error notification to the standard error, file descriptor 2, but we want all error notification suppressed. Otherwise, the error messages would go to the screen during the script's execution, which is highly undesirable.

Next are the standard things of initializing a few variables and checking for the correct number of arguments. There are two ways to run this script: (1) only specifying the total seconds and the process to monitor or (2) using the command-line switches to specify the seconds, minutes, hours, days, and process to monitor. The latter method will use the getopts command to parse the arguments, but we do not need getopts for the first method. We first check to see if we are given only seconds and a process. We use the test_string function to ensure that the \$1 argument is a positive integer. The second argument could be anything except a null string or a string that begins with a - (hyphen). Otherwise, we will use the getopts command to parse the command to parse the command line.

The getopts command makes life much easier when we need to process commandline arguments; however, getopts does have its limitations. We just need to remember that getopts is parsing the command-line arguments, but it really does not care what the arguments are. Therefore, we need to do a sanity check on each and every argument to ensure that it meets the criteria that are expected. If the argument fails, we just run the usage function and exit with a return code of 1, one. Two tests are conducted on each argument. We test the PROCESS variable to make sure that it is not null, or empty, and we check all the numeric variables used for timing to make sure they are positive integers or 0, zero. The positive integer test is to ensure that at least one of the numeric variables, SECS, MINUTES, HOURS, and DAYS, has an integer value greater than 0, zero. If we get past this stage we assume we have creditable data to start monitoring.

The monitoring starts by getting an initial state of the process, either currently running or not running. With this information we initialize the RUN variable, which is used as a control mechanism for the rest of the script. Once the initialization text is both logged and sent to the screen, the proc_watch function is started as a background co-process, again using proc_watch |&. The main script just does a countdown to 0, zero, and exits. To make the proc_watch function exit cleanly we assign 'Y' to the BREAK variable and make this new value known to the co-process with the print -p \$BREAK command. Then we kill the background PID that we saved in the WATCH_PID variable and then exit the script with a return code of 0, zero. If the script is interrupted, the trap will handle stopping the co-process and exiting. See Listing 10-10.

```
[root:yogi]@/scripts/WILEY/PROC_MON# ./proc_watch_timed.ksh -m 5 -pxcalc
Currently running xcalc processes:
There are no xcalc processes running
The xcalc process is not currently running...monitoring...
MON_STARTED: Monitoring for xcalc began ==> 09/27/07@21:15:02
```

Listing 10-10 proc_watch_times.ksh shell script in action

ACTIVE PIDS: START PROCESS: xcalc START time ==> 09/27/07@21:15:19 ACTIVE PIDS: 26190 STARTUP EVENT: Executing Startup Event Script... SUCCESS: Startup Event Script Completed RC - 0 PROCESS COUNT: 2 xcalc Processes Running ==> 09/27/07@21:15:46 ACTIVE PIDS: 13060 26190 PROCESS COUNT: 3 xcalc Processes Running ==> 09/27/07@21:16:04 ACTIVE PIDS: 13060 18462 26190 PROCESS COUNT: 4 xcalc Processes Running ==> 09/27/07@21:16:27 ACTIVE PIDS: 13060 18462 22996 26190 PROCESS COUNT: 3 xcalc Processes Running ==> 09/27/07@21:16:39 ACTIVE PIDS: 18462 22996 26190 PROCESS COUNT: 4 xcalc Processes Running ==> 09/27/07@21:16:56 ACTIVE PIDS: 18462 22996 24134 26190 PROCESS COUNT: 3 xcalc Processes Running ==> 09/27/07@21:17:31 ACTIVE PIDS: 22996 24134 26190 PROCESS COUNT: 2 xcalc Processes Running ==> 09/27/07@21:17:41 ACTIVE PIDS: 22996 24134 PROCESS COUNT: 3 xcalc Processes Running ==> 09/27/07@21:18:39 ACTIVE PIDS: 21622 22996 24134 PROCESS COUNT: 2 xcalc Processes Running ==> 09/27/07@21:18:58 ACTIVE PIDS: 21622 22996 PROCESS COUNT: 3 xcalc Processes Running ==> 09/27/07@21:19:04 ACTIVE PIDS: 18180 21622 22996 PROCESS COUNT: 4 xcalc Processes Running ==> 09/27/07@21:19:10 ACTIVE PIDS: 18180 21622 22758 22996 PROCESS COUNT: 6 xcalc Processes Running ==> 09/27/07@21:19:17 ACTIVE PIDS: 18180 21622 22758 22996 23164 26244 PROCESS COUNT: 5 xcalc Processes Running ==> 09/27/07@21:19:37 ACTIVE PIDS: 18180 22758 22996 23164 26244 PROCESS COUNT: 4 xcalc Processes Running ==> 09/27/07@21:19:47 ACTIVE PIDS: 18180 22996 23164 26244 PROCESS COUNT: 3 xcalc Processes Running ==> 09/27/07@21:19:53 ACTIVE PIDS: 18180 22996 26244 PROCESS COUNT: 2 xcalc Processes Running ==> 09/27/07@21:19:55 ACTIVE PIDS: 18180 26244 PROCESS COUNT: 1 xcalc Processes Running ==> 09/27/07@21:20:05 ACTIVE PIDS: 18180 PROCESS COUNT: 0 xcalc Processes Running ==> 09/27/07@21:20:09 ACTIVE PIDS: ENDING PROCESS: xcalc END time ==> 09/27/07@21:20:11 POST EVENT: Executing Post Event Script ... SUCCESS: Post Event Script Completed RC - 0 MON_STOPPED: Monitoring for xcalc ended ==> 09/27/07@21:20:23 LOGFILE: All Events are Logged ==> /tmp/proc_status.log

Listing 10-10 (continued)

Other Options to Consider

The proc_watch_timed.ksh shell script is thorough, but it may need to be tailored to a more specific need. Some additional considerations are listed next.

Common Uses

These scripts are suited for things like monitoring how long a process runs, logging a process as it starts and stops, restarting a process that has terminated prematurely, and monitoring a problem user or contractor. We can also monitor activity on a particular tty port and send an email as a process starts execution. Use your imagination.

We can start the monitoring script on the command line, or as a cron or at job, and run it during the work day. A cron table entry might look like the following:

0 7 * * 1-5 /usr/local/bin/proc_watch_timed.ksh -h9 -p fred >/dev/null

This cron table entry would monitor any process in the process table that contained "fred" from 7:00 a.m. Monday through Friday for nine hours. Anything in the system's process list can be monitored from seconds to days.

Modifications to Consider

These scripts are generic, and you may want to make modifications. One option to consider is to list the actual lines in the process list instead of only the PID and a process count with a timestamp. For a more accurate timing you may want to check the date/time at longer intervals (as opposed to counting down). Another good idea is to get the timing data and run an at command to kill the script at the specified time. Also, consider using the shell built-in variable SECONDS. First initialize the SECONDS variable to 0, zero, and it will automatically increment each second as long as the parent process is executing. The pre-startup, and post-events are something else to look at, the startup in particular. The startup_event_script currently executes only when (1) the monitoring starts and the target process is running and (2) when the very first, if more than one, process starts, not as each process starts. You may want to modify this function's execution to run only as each individual process starts, and not to execute when monitoring starts and the target process is already running. Additionally, depending on what is to be executed for any of these events, some sleep time might be needed to allow for things to settle down. As we can see, there are many ways to do all this, and everyone has different expectations and requirements. Just remember that we never have a *final script*; we just try to be flexible!

Summary

In this chapter we started with a very basic idea of monitoring for a process to start or stop. We quickly built on user options to monitor the process state for a specified period of time and added timestamps. We also allowed the user to specify pre-startup, and post-events to execute as an option. Never try to do everything at once. Build

a short shell script that does the basic steps of your target goal and expand on the base shell script to build in the nice-to-have things. I use the proc_mon.ksh and proc_wait.ksh shell scripts almost daily for monitoring system events, and they sure do save a lot of time otherwise spent reentering the same command over and over again.

In the next chapter we are going to play around with random numbers — well, that would actually be pseudo-random numbers, because the computer is going to generate them. Stick around; the next chapter is very interesting.

Lab Assignments

- 1. Adapt the pre-event, startup, and post-event strategy to a generic shell script. Create a generic shell script that controls a program or application, currently unknown, and has pre-processing, startup, and post-processing capability. Just use a no-op as a placeholder in each function.
- 2. Modify the proc_watch_timed.ksh shell script in Listing 10-9 to show each process as well as the number of currently active processes.
- 3. Modify the shell script that resulted in Lab Assignment 2 to use the built-in shell variable SECONDS. To start timing, initialize the SECONDS variable to 0, zero. As the SECONDS variable automatically increments, your script should exit at the correct time.
- 4. Modify the proc_watch_timed.ksh shell script in Listing 10-9 to exit at the specified time, using the at command to kill the script.

CHAPTER 11

Pseudo-Random Number and Data Generation

When we enter the realm of randomness, we head straight for the highest level of mathematical complexity and outright conflict between researchers. This conflict revolves around the following question:

Can a computer create a true random number?

If you believe some experts, you are living in dreamland if you think a computer can create a true random number. However, we are going to study several ways to produce pseudo-random numbers. Some of these techniques are able to create numbers that are suitable for encryption keys and for cryptographic secure communication links, and some are not.

What Makes a Random Number?

It is very difficult, if not impossible, to create a *true* random number in a computer system. The problem is, at some level, repeatability and predictability of the number. When you start researching random numbers you quickly enter the realm of heavy mathematical theory, and many of the researchers have varying opinions of randomness. I will leave this discussion to university professors and professional statisticians. The only *true* random numbers that I know of, in my limited research, are the frequency variations of radioactive decay events and the frequency variations of white noise. Radioactive decay events would have to be detected in some way, and because we do not want to have any radioactive material hanging around we can use built-in computer programs called pseudo-random number generators, and even query the UNIX kernel's own random number generator by reading the /dev/random character special file.

A popular UNIX technique is to use a UNIX character special file called /dev/ random. If you search the Internet for /dev/random, you will find more information

than you could imagine on the topic of randomness. Randomness is a discussion topic with many experts in the field having widely varying viewpoints. I am not an expert on randomness, and an in-depth study of this topic is beyond the scope of this book.

Some of the numbers we will create in this chapter are sufficiently random for one-time encryption keys. Other numbers should not be used because they can be repeatable and cyclical in nature, at some level, but they are suitable to create unique filenames and pseudo-random data for a file.

The Bash, Bourne, and Korn shells provide a shell variable called — you guessed it — RANDOM. This pseudo-random number generator uses a *seed* as a starting point to create all future numbers in the sequence. After the initial seed is used to create a pseudo-random number, this resulting number is used to create the next random number, and so on. As you would expect, if you always start generating your numbers with the same seed each time, you will get the exact same number sequence each time. To change the repeatability we need to have a mechanism to vary the initial seed each time we start generating numbers. I like to use two different methods. The first is to use the current process ID (PID) because this number will vary widely and is an easy way to change the seed value each time we start generating numbers. The second method is to read the /dev/random character special file to create the initial seed. The /dev/urandom character special file is a better option than using the shell's RANDOM variable.

The Methods

In this chapter we are going to look at four techniques to generate pseudo-random numbers, and one method of using random data:

- Create numbers between zero and the maximum number allowed by the shell (32,767)
- Create numbers between one and a user-defined maximum
- Create fixed-length numbers between one and a user-defined maximum, with leading zeros added if needed
- Create a large 64-bit random number, and smaller random numbers, by reading the /dev/random character special file, in concert with the dd and od commands
- Create a specific-MB-size file filled with random alphanumeric characters

Each method is valid for a filename extension, but the /dev/random special character file will produce the highest level of randomness, followed by the /dev/urandom character special file. Of course, you may have other uses that require either a range of numbers or a random number of a fixed length, with leading zeros to fill the extra spaces. For example, you might need to create 64 random numbers. In any case, the basic concept is the same. We are going to look at all these methods throughout this chapter.

Method 1: Creating a Pseudo-Random Number Utilizing the PID and the RANDOM Shell Variable

To begin with the basic command-line syntax, we start out by initializing the RANDOM shell variable to the current PID:

```
RANDOM=$$
```

The double dollar signs (\$\$) specify the PID of the current system process. The PID will vary, so this is a good way to initialize the RANDOM variable so that we do not always repeat the same number sequence. Once the RANDOM environment variable is initialized we can use RANDOM just like any other variable. Most of the time we will use the echo command to print the next pseudo-random number. An example of using the RANDOM environment variable is shown in Listing 11-1.

```
# RANDOM=$$
# echo $RANDOM
23775
# echo $RANDOM
3431
# echo $RANDOM
12127
# echo $RANDOM $RANDOM
2087 21108
```

Listing 11-1 Using the RANDOM environment variable

By default the RANDOM variable will produce numbers between 0 and 32,767. Notice the last entry in Listing 11-1. We can produce more than one number at a time if we need to by adding more \$RANDOM entries to our echo statement. In showing our four methods of creating random numbers in this chapter, we are going to create four functions and then create a shell script that will use one of the four methods, depending on the user-supplied input. The next step is to write a shell script that will create unique filenames using a date/time stamp and a random number, and a file full of random characters.

Method 2: Creating Numbers between 0 and 32,767

Creating pseudo-random numbers using this default method is the simplest way to use the RANDOM environment variable. The only thing that we need to do is to initialize the RANDOM environment variable to an initial seed value and use the echo command to display the new number. The numbers will range from 0 to 32,767, which is the maximum for the RANDOM shell variable. You do not have control over the number of digits, except that the number of digits will not exceed five, and you cannot specify a maximum value for the number in this first method. The function get_random_number is shown in Listing 11-2.

```
function get_random_number
{
    # This function gets the next random number from the
    # $RANDOM variable. The range is 0 to 32767.
echo "$RANDOM"
}
```

Listing 11-2 get_random_number function

As you can see, the function is just one line, and we are assuming that the RANDOM environment variable is initialized in the main body of the calling shell script.

Method 3: Creating Numbers between 1 and a User-Defined Maximum

We often want to limit the range of numbers to not exceed a user-defined maximum. An example is creating lottery numbers between 1 and the maximum number, which might be 36 or 52. We are going to use the modulo arithmetic operator to reduce all numbers to a fixed set of numbers between [0..N-1], which is called *modulo N arithmetic*.

For our number range we need a user-supplied maximum value, which we will assign to a variable called UPPER_LIMIT. The modulo operator is the percent sign (%), and we use this operator the same way that we use the forward slash (/) in division. The modulo operation returns the remainder of the division. We still use the RANDOM environment variable to get a new pseudo-random number. This time, though, we are going to use the following equation to limit the number to not exceed the user-defined maximum using *modulo N arithmetic*.

```
RANDOM_NUMBER=$(($RANDOM % $UPPER_LIMIT + 1))
```

Notice that we added 1 to the equation.

Using the preceding equation will produce a pseudo-random number between 1 and the user-defined \$upper_LIMIT. The function using this equation is in_range_ random_number and is shown in Listing 11-3.

```
function in_range_random_number
{
    # Create a pseudo-random number less than or equal
    # to the $UPPER_LIMIT value, which is user defined
RANDOM_NUMBER=$(($RANDOM % $UPPER_LIMIT + 1))
echo "$RANDOM_NUMBER"
}
```

Listing 11-3 in_range_random_number function

The function in Listing 11-3 assumes that the RANDOM variable seed has been initialized in the main body of the calling shell script and that a user-defined UPPER_LIMIT variable has been set. This function will produce numbers between 1 and the user-defined maximum value, but the number of digits will vary as the numbers are produced.

Method 4: Creating Fixed-Length Numbers between 1 and a User-Defined Maximum

In both of the previous two examples we had no way of knowing how many digits the new number would contain. When we are creating unique filenames, it would be nice to have filenames that are consistent in length. We can produce fixed-length numbers by padding the number with leading zeros for each missing digit. As an example we want all of our numbers to have four digits. Now let's assume that the number that is produced is 24. Because we want 24 to have four digits, we need to pad the number with two leading zeros, which will make the number 0024. To pad the number we need to know the length of the character string that makes up the number. The Korn and Bash shells use the hash mark (#) preceding the variable enclosed within curly braces ({}), as shown here:

```
RN_LENGTH=$(echo ${#RANDOM_NUMBER})
```

If the RANDOM_NUMBER variable has the value 24 assigned, the result of the previous command is 2 (this RN_LENGTH variable points to the value 2), indicating a string length of two digits. We will also need the length of the UPPER_LIMIT value, and we will use the difference to know how many zeros to use to pad the pseudo-random number output. Take a close look at the code in Listing 11-4, where you will find the function in_range_fixed_length_random_number.

```
function in_range_fixed_length_random_number
{
    # Create a pseudo-random number less than or equal
    # to the $UPPER_LIMIT value, which is user defined.
    # This function will also pad the output with leading
    # zeros to keep the number of digits consistent.
    RANDOM_NUMBER=$(($RANDOM % $UPPER_LIMIT + 1))
    # Find the length of each character string
    RN_LENGTH=$(echo ${#RANDOM_NUMBER})
    UL_LENGTH=$(echo ${#UPPER_LIMIT})
    # Calculate the difference in string length
    (( LENGTH_DIFF = UL_LENGTH - RN_LENGTH ))
```

Listing 11-4 in_range_fixed_length_random_number function

```
# Pad the $RANDOM_NUMBER value with leading zeros
# to keep the number of digits consistent.
case $LENGTH_DIFF in
0) echo "$RANDOM_NUMBER"
   ;;
1) echo "0$RANDOM_NUMBER"
   ;;
2) echo "00$RANDOM_NUMBER"
   ;;
3) echo "000$RANDOM_NUMBER"
   ;;
4) echo "0000$RANDOM NUMBER"
   ;;
5) echo "00000$RANDOM_NUMBER"
   ;;
*) echo "$RANDOM_NUMBER"
   ;;
esac
}
```

In Listing 11-4 we use the same technique from Listing 11-3 to set an upper limit to create our numbers, but we add in code to find the string length of both the UPPER_LIMIT and RANDOM_NUMBER values, specified by the \${#UPPER_LIMIT} and echo \${#RANDOM_NUMBER} statements, respectively. The #, hash mark, tells the echo command to return the length of the string assigned to the variable. By knowing the length of both strings we subtract the random-number length from the upper-limit length and use the difference in a case statement to add the correct number of zeros to the output.

Because this is a function, we again need to assume that the UPPER_LIMIT is defined and the RANDOM environment variable is initialized in the main body of the calling shell script. The resulting output is a fixed-length pseudo-random number padded with leading zeros if the output string length is less than the upper-limit string length. Example output is shown in Listing 11-5 for an UPPER_LIMIT value of 9999.

Listing 11-5 Sample output for fixed-length random numbers

Why Pad the Number with Zeros the Hard Way?

An easier, and much cleaner, way to pad a number with leading zeros is to **typeset** the variable to a fixed length. The command in Listing 11-6 works for any length of number.

```
typeset -Z5 FIXED_LENGTH
FIXED_LENGTH=25
echo $FIXED_LENGTH
00025
```

Listing 11-6 Using the typeset command to fix the length of a variable

The example in Listing 11-6 used the typeset command to set the length of the FIXED_LENGTH variable to five digits. Then we assigned the value 25 to it. When we use the echo command to show the value assigned to the variable the result is 00025, which is fixed to five digits. Let's modify the function in Listing 11-4 to use this technique, as shown in Listing 11-7.

```
function in_range_fixed_length_random_number_typeset
# Create a pseudo-random number less than or equal
# to the $UPPER_LIMIT value, which is user defined.
# This function will also pad the output with leading
# zeros to keep the number of digits consistent using
# the typeset command.
# Find the length of each character string
UL_LENGTH=$(echo ${#UPPER_LIMIT})
# Fix the length of the RANDOM_NUMBER variable to
# the length of the UPPER_LIMIT variable, specified
# by the $UL_LENGTH variable.
typeset -Z$UL_LENGTH RANDOM_NUMBER
# Create a fixed length pseudo-random number
RANDOM_NUMBER=$(($RANDOM % $UPPER_LIMIT + 1))
# Return the value of the fixed length $RANDOM_NUMBER
echo $RANDOM_NUMBER
}
```

Listing 11-7 Using the typeset command in a random-number function

As you can see in Listing 11-7, we took all the complexity out of fixing the length of a number. The only value we need to know is the length of the UPPER_LIMIT variable

assignment. For example, if the upper limit is 9999, the length is 4. We use 4 to typeset the RANDOM_NUMBER variable to four digits.

Now that we have four functions that will create pseudo-random numbers, but before we proceed with a shell script that will use these functions, let's study the /dev/random and /dev/urandom character special files to produce random numbers.

Method 5: Using the /dev/random and /dev/urandom Character Special Files

Other methods to create pseudo-random numbers include using the character special files /dev/random and /dev/urandom. These character special files are an interface to the kernel's random number generator. The kernel random number generator uses *system noise* from device drivers, and other environmental noise a computer generates to create an *entropy pool* of random noise data. The kernel's random number generator keeps track of the number of bits of noise remaining in the entropy pool. When we read the /dev/random character special file, using the dd command, random bytes of noise are returned within the range of available noise in the entropy pool. When the entropy pool is empty, /dev/random will *block* while the kernel gathers more environmental system noise. During this blocking period, /dev/random will not return data and we can see a pause in execution of our shell scripts.

The /dev/urandom character special file does not block when the entropy pool is empty; therefore, the randomness is to a lesser degree. Using /dev/urandom in a shell script will not show the possible delays in execution as /dev/random will, so /dev/urandom is often the preferred method.

All of the techniques studied in this chapter using the /dev/random will produce numbers with the highest degree of randomness.

To *read* the /dev/random character special file, we will use the dd command. The dd command is used to convert and copy files. If you are not familiar with using the dd command, study the dd manual page, **man dd**.

Trying to read the /dev/random and /dev/urandom character special files directly with dd returns non-printable binary data. To get some usable random numbers, we need to pipe the dd command output to the **od**, octal dump, command. Specifically, we use od to dump the data to an unsigned integer. The code shown in Listing 11-8 assigns an unsigned random integer to the RN variable.

Listing 11-8 Using /dev/random to return a random number

Notice in Listing 11-8 that the dd command uses /dev/random as the input file. We set the count equal to 1 to return one byte of data. Then, and this is important, we send all the standard error output, specified by file descriptor 2, to the bit bucket. If we omit the 2>/dev/null redirection, we get unwanted data. The remaining standard output is piped to the od command to convert the binary data to an unsigned integer, specified by the -t u4 command switch. By changing the value assigned to u, we

change the length of the random number returned. To create a 64-bit, not 64-character, random number, we just change the -t u4 to -t u8. An example of using different u values is shown in Listing 11-9.

```
[root@booboo ~]# dd if=/dev/urandom count=1 2>/dev/null |
od -t u1 | awk '{print $2}'| head -n 1
121
[root@booboo ~]# dd if=/dev/urandom count=1 2>/dev/null |
od -t u2 | awk '{print $2}'| head -n1
10694
[root@booboo ~]# dd if=/dev/urandom count=1 2>/dev/null |
od -t u4 | awk '{print $2}'| head -n 1
1891340326
[root@booboo ~]# dd if=/dev/urandom count=1 2>/dev/null |
od -t u8 | awk '{print $2}'| head -n 1
2438974545704940532
[root@booboo ~]#
```

Listing 11-9 Creating different-size random numbers utilizing od

Notice in Listing 11-9 that as we increment the value assigned to the -t u#, our random number increases in length. A u value of 1 produces an 8-bit number, in this case 121. Increasing the u value to 2 produces a 16-bit random number, 10694. Going to a u value of 4 produces the 32-bit number 1891340326. The longest number using this method is the 64-bit number 2438974545704940532 we produced with a u value of 8. These are the only valid u values for unsigned integers.

Also notice in Listing 11-9 that we used /dev/urandom in place of /dev/random. By using /dev/urandom, we avoid blocking if the entropy pool becomes empty. No big deal here, but /dev/random produces a higher level of randomness.

There are many other things we can do when creating random numbers in this manner. In Listing 11-10 we use the /dev/urandom character special file to create an output with a specific number of random numbers, with one number on each line. Pay special attention to the last dd command.

```
#!/bin/Bash
#
# SCRIPT: random_number_testing.Bash
# AUTHOR: Randy Michael
# DATE: 8/8/2007
```

Listing 11-10 random_number_testing.Bash shell script

Listing 11-10 (continued)

The results of executing the random_number_testing.Bash shell script in Listing 11-10 are shown in Listing 11-11.

```
[root@booboo scripts]# ls -ltr
-rwxr-xr-- 1 root root 582 07:17 random_number_testing.Bash
-rw-r--r-- 1 root root 370 07:30 64random_numbers.txt
-rw-r--r-- 1 root root 382031 07:30 65536random_numbers.txt
-rw-r--r-- 1 root root 1048576 07:30 1MBemptyfile
[root@booboo scripts]# wc -1 64random_numbers.txt
64 64random_numbers.txt
[root@booboo scripts]# wc -1 65536random_numbers.txt
65536 65536random_numbers.txt
[root@booboo scripts]# head 64random_numbers.txt
51890
17416
58012
4220
48095
18080
6847
4963
39421
44931
[root@booboo scripts]#
```

Listing 11-11 random_number_testing.Bash script in action

Notice the three files created in Listing 11-11. The first file, 64random_numbers.txt, contains 64 random numbers, one per line. The second file, 65536random_numbers.txt, contains 65,536 random numbers, one per line.

The third file is named 1Mbemptyfile. Notice in Listing 11-10 that the third da command uses the /dev/zero character special file. This file produces NULL data. So, by replacing the /dev/random with /dev/zero, we create a specific size file of NULL data. This is kind of like a sparse file.

NOTE Be very careful using this method so that you do not wipe out a critical system file. I'm not really sure of an application where you would need a 1 MB file filled with NULL data, but if you need one, here you go!

Shell Script to Create Pseudo-Random Numbers

Using the three functions from Listings 11-2, 11-3, and 11-7, we are going to create a shell script that, depending on the command-line arguments, will use one of these three functions. We first need to define how we are going to use each function.

With the usage definitions from Table 11-1 let's create a shell script. We already have the functions to create the numbers, so we will start with BEGINNING OF MAIN in the shell script.

For the usage function we will need the name of the shell script. We never want to hard-code the name of a shell script because someone may rename the shell script for one reason or another. To query the system for the actual name of the shell script we use the basename \$0 command. This command will return the name of the shell script, specified by the \$0 argument, with the directory path stripped out. I like to use either of the following command-substitution methods to create a SCRIPT_NAME variable:

SCRIPT_NAME=`basename \$0`

or

SCRIPT_NAME=\$(basename \$0)

The result of both command-substitution commands is the same. Next we need to initialize the RANDOM environment variable. As previously described, we are going to use the current process ID as the initial seed for the RANDOM variable:

RANDOM=\$\$

SHELL SCRIPT USAGE	FUNCTION USED TO CREATE THE NUMBER
random_number.ksh	Without argument will use get_random_number
random_number.ksh 9999	With one numeric argument will use in_range_ random_number
random_number.ksh -f 9999	With -f as the first argument followed by a numeric argument will use in_range_fixed_length_ random_number_typeset

Table 11-1 random_number.ksh Shell Script Usage

The SCRIPT_NAME and the RANDOM variables are the only initialization needed for this shell script. The rest of the script is a case statement that uses the number of command-line arguments (\$#) as a value to decide which random number function we will use. We also do some numeric tests to ensure that "numbers" are actually numeric values. For the numeric tests we use the regular expression +([0-9]) in a case statement. If the value is a number, then we do nothing, which is specified by the no-op character, colon (:).

NOTE The Bash shell does not support the regular expression notation +([0-9]) to test strings.

The entire shell script is shown in Listing 11-12.

```
#!/usr/bin/ksh
# AUTHOR: Randy Michael
# SCRIPT: random number.ksh
# DATE: 11/12/2007
# REV: 1.2.P
# PLATFORM: Not Platform Dependent
#
# EXIT CODES:
     0 - Normal script execution
#
        1 - Usage error
#
#
# REV LIST:
#
# set -x # Uncomment to debug
# set -n # Uncomment to check syntax without any command execution
*****
**********
function usage
{
echo "\nUSAGE: $SCRIPT_NAME [-f] [upper_number_range]"
echo "\nEXAMPLE: $SCRIPT_NAME"
echo "Will return a random number between 0 and 32767"
echo "\nEXAMPLE: $SCRIPT_NAME 1000"
echo "Will return a random number between 1 and 1000"
echo "\nEXAMPLE: $SCRIPT_NAME -f 1000"
echo "Will add leading zeros to a random number from"
echo "1 to 1000, which keeps the number of digits consistent\n"
}
```

Listing 11-12 random_number.ksh shell script

```
****
function get_random_number
{
# This function gets the next random number from the
# $RANDOM variable. The range is 0 to 32767.
echo "$RANDOM"
}
****
function in_range_random_number
# Create a pseudo-random number less than or equal
# to the $UPPER_LIMIT value, which is user defined
RANDOM_NUMBER=$(($RANDOM % $UPPER_LIMIT + 1))
echo "$RANDOM_NUMBER"
}
*****
function in_range_fixed_length_random_number_typeset
{
# Create a pseudo-random number less than or equal
# to the $UPPER LIMIT value, which is user defined.
# This function will also pad the output with leading
# zeros to keep the number of digits consistent using
# the typeset command.
# Find the length of each character string
UL_LENGTH=$(echo ${#UPPER_LIMIT})
# Fix the length of the RANDOM_NUMBER variable to
# the length of the UPPER_LIMIT variable, specified
# by the $UL_LENGTH variable.
typeset -Z$UL_LENGTH RANDOM_NUMBER
# Create a fixed length pseudo-random number
RANDOM_NUMBER=$(($RANDOM % $UPPER_LIMIT + 1))
# Return the value of the fixed length $RANDOM_NUMBER
```

```
echo $RANDOM_NUMBER
}
*****
*****
SCRIPT_NAME=`basename $0`
RANDOM=$$ # Initialize the RANDOM environment variable
         # using the PID as the initial seed
case $# in
   get_random_number
0)
;;
1) UPPER_LIMIT="$1"
   # Test to see if $UPPER_LIMIT is a number
   case $UPPER_LIMIT in
   +([0-9])) : # Do Nothing...It's a number
                # NOTE: A colon (:) is a no-op in Korn shell
          ;;
           echo "\nERROR: $UPPER_LIMIT is not a number..."
   *)
           usage
           exit 1
          ;;
   esac
   # We have a valid UPPER_LIMIT. Get the number.
   in_range_random_number
;;
2) # Check for the -f switch to fix the length.
   if [[ $1 = '-f' ]] || [[ $1 = '-F' ]]
   then
       UPPER_LIMIT="$2"
       # Test to see if $UPPER_LIMIT is a number
       case $UPPER_LIMIT in
       +([0-9])) : # Do nothing...It's a number
                    # NOTE: A colon (:) is a no-op in Korn shell
```

Listing 11-12 (continued)

```
;;
         *)
                   echo "\nERROR: $UPPER_LIMIT is not a number..."
                   usage
                   exit 1
               ;;
         esac
         in_range_fixed_length_random_number_typeset
    else
         echo "\nInvalid argument $1, see usage below..."
         usage
         exit 1
    fi
;;
*) usage
    exit 1
;;
esac
# End of random_number.ksh shell script
```

Notice in Listing 11-12 that we will allow only zero, one, or two command-line arguments. More than three arguments produces an error, and nonnumeric values, other than -f or -F in argument one, will produce a usage error. Output using the random_number.ksh shell script is shown in Listing 11-13.

```
yogi@/scripts# random_number.ksh 32000
10859
yogi@/scripts# random_number.ksh -f 32000
14493
yogi@/scripts# ./random_number.ksh -f 32000
05402
yogi@/scripts# ./random_number.ksh -f
ERROR: -f is not a number...
USAGE: random_number.ksh [-f] [upper_number_range]
EXAMPLE: random_number.ksh
Will return a random number between 0 and 32767
EXAMPLE: random_number.ksh 1000
```

Listing 11-13 random_number.ksh shell script in action

Will return a random number between 1 and 1000 EXAMPLE: random_number.ksh -f 1000 Will add leading zeros to a random number from 1 to 1000, which keeps the number of digits consistent

Listing 11-13 (continued)

The last part of Listing 11-13 is a usage error. Notice that we give an example of each of the three uses for the random_number.ksh shell script as well as state why the usage error occurred.

Now that we have the shell script to produce pseudo-random numbers, we need to move on to creating unique filenames.

Creating Unique Filenames

In writing shell scripts we sometimes run into a situation where we are creating files faster than we can make the filenames unique. Most of the time a date/time stamp can be added as a suffix to the filename to make the filename unique, but if we are creating more than one file per second, we end up overwriting the same file during a single second. To get around this problem, we can create pseudo-random numbers to append to the filename after the date/time stamp. In the next section and in Chapter 12, "Creating Pseudo-Random Passwords," we will study creating *pseudo-random characters* by using the computer-generated numbers as pointers to array elements that contain keyboard characters. We will use a similar method to create a *random file*.

The goal of this section is to write a shell script that will produce unique filenames using a date/time stamp with a pseudo-random number as an extended suffix. When I create these unique filenames I like to keep the length of the filenames consistent, so we are going to use only one of the random number functions, in_range_fixed_length_random_number_typeset.

We have a few new pieces to put into this new shell script. First we have to assume that there is some program or shell script that will be putting data into each of the unique files. To take care of executing the program or shell script we can add a function that will call the external program, and we will redirect our output to the new unique filename on each loop iteration. The second piece is that we need to ensure that we never use the same number during the same second. Otherwise, the filename is not unique and the data will be overwritten. We need to keep a list of each number that we use during each second and reset the USED_NUMBERS list to null on each new second. In addition we need to grep the list each time we create a new number to see if it has already been used. If the number has been used we just create a new number and check for previous usage again.

The procedure to step through our new requirements is not difficult to understand once you look at the code. The full shell script is shown in Listing 11-14, and an example of using the shell script is shown in Listing 11-15. Please study the script carefully, and we will go through the details at the end.

```
#!/usr/bin/ksh
#
# AUTHOR: Randy Michael
# SCRIPT: mk_unique_filename.ksh
# DATE: 11/12/2007
# REV: 1.2.P
#
# PLATFORM: Not Platform Dependent
#
# EXIT CODES:
         0 - Normal script execution
#
#
          1 - Usage error
#
# REV LIST:
#
#
# set -x # Uncomment to debug
# set -n # Uncomment to debug without any execution
******
function usage
{
echo "\nUSAGE: $SCRIPT NAME base file name\n"
exit 1
}
****
function get_date_time_stamp
DATE_STAMP=$(date +'%m%d%y.%H%M%S')
echo $DATE_STAMP
}
*******
function get_second
{
THIS SECOND=$(date +%S)
echo $THIS_SECOND
}
****
function in_range_fixed_length_random_number_typeset
```

Listing 11-14 mk_unique_filename.ksh shell script

Create a pseudo-random number less than or equal # to the \$UPPER_LIMIT value, which is user defined. # This function will also pad the output with leading # zeros to keep the number of digits consistent using # the typeset command. # Find the length of each character string UL_LENGTH=\$(echo \${#UPPER_LIMIT}) # Fix the length of the RANDOM_NUMBER variable to # the length of the UPPER LIMIT variable, specified # by the \$UL_LENGTH variable. typeset -Z\$UL_LENGTH RANDOM_NUMBER # Create a fixed length pseudo-random number RANDOM_NUMBER=\$((\$RANDOM % \$UPPER_LIMIT + 1)) # Return the value of the fixed length \$RANDOM_NUMBER echo \$RANDOM_NUMBER } ***** function my_program # Put anything you want to process in this function. I # recommend that you specify an external program of shell # script to execute. echo "HELLO WORLD - \$DATE_ST" > \$UNIQUE_FN & : # No-Op - Does nothing but has a return code of zero # 3 ****** ********** SCRIPT_NAME=\$(basename \$0) # Query the system for this script name # Check for the correct number of arguments - exactly 1

Listing 11-14 (continued)

```
if (( $# != 1 ))
then
      echo "\nERROR: Usage error...EXITING..."
      usage
fi
# What filename do we need to make unique?
BASE_FN=$1
               # Get the BASE filename to make unique
RANDOM=$$
               # Initialize the RANDOM environment variable
               # with the current process ID (PID)
UPPER_LIMIT=32767 # Set the UPPER_LIMIT
CURRENT_SECOND=99 # Initialize to a nonsecond
LAST_SECOND=98
                  # Initialize to a nonsecond
USED_NUMBERS=
                  # Initialize to null
PROCESSING="TRUE" # Initialize to run mode
while [[ $PROCESSING = "TRUE" ]]
do
    DATE_ST=$(get_date_time_stamp) # Get the current date/time
    CURRENT_SECOND=$(get_second) # Get the current second
    RN=$(in_range_fixed_length_random_number_typeset) #
Get a new number
    # Check to see if we have already used this number this second
    if (( CURRENT_SECOND == LAST_SECOND ))
    then
         UNIQUE=FALSE # Initialize to FALSE
         while [[ "$UNIQUE" != "TRUE" ]] && [[ ! -z "$UNIQUE" ]]
         do
            # Has this number already been used this second?
            echo $USED_NUMBERS | grep $RN >/dev/null 2>&1
            if (( $? == 0 ))
            then
                # Has been used...Get another number
                RN=$(in_range_fixed_length_random_number)
            else
                # Number is unique this second ...
                UNIQUE=TRUE
                # Add this number to the used number list
                USED_NUMBERS="$USED_NUMBERS $RN"
```

```
fi
         done
    else
         USED NUMBERS=
                          # New second...Reinitialize to null
    fi
    # Assign the unique filename to the UNIQUE_FN variable
    UNIQUE FN=${BASE FN}.${DATE ST}.$RN
    # echo $UNIQUE_FN # Comment out this line!!
    LAST_SECOND=$CURRENT_SECOND # Save the last second value
    # We have a unique filename...
    #
    # Process something here and redirect output to $UNIQUE FN
    #
    my_program
    #
    # If processing is finished assign "FALSE" to the
    # PROCESSING VARIABLE
    # if [[ $MY_PROCESS = "done" ]]
    #
      then
    #
           PROCESSING="FALSE"
    # fi
done
```

We use five functions in the shell script in Listing 11-14. As usual, we need a function for correct usage. We are expecting exactly one argument to this shell script, the *base filename* to make into a unique filename. The second function is used to get a date/time stamp. The **date** command has a lot of command switches that allow for flexible date/time stamps. We are using two digits for month, day, year, hour, minute, and second, with a period (.) between the date and time portions of the output. This structure is the first part that is appended to the base filename. The date command has the following syntax: date+'%m%d%y.%H%M%S'.

We also need the current second of the current minute. The current second is used to ensure that the pseudo-random number that is created is unique to each second, thus a unique filename. The date command is used again using the following syntax: date+%S.

The in_range_fixed_length_random_number_typeset function is used to create our pseudo-random numbers in this shell script. This function keeps the number of digits consistent for each number that is created. With the base filename, date/time stamp, and the unique number put together, we are assured that every filename has the same number of characters.

One more function is added to this shell script. The my_program function is used to point to the program or shell script that needs all of these unique filenames. It is better to point to an external program or shell script than to try to put everything in the internal my_program function and debug the internal function on an already working shell script. Of course, I am making an assumption that you will execute the external program once during each loop iteration, which may not be the case. At any rate, this script will show the concept of creating unique filenames while remaining in a tight loop. Or just write the functions into your application.

At the BEGINNING OF MAIN in the main body of the shell script we first query the system for the name of the shell script. The script name is needed for the usage function. Next we check for exactly one command-line argument. This single command-line argument is the base filename that we use to create further unique filenames. The next step is to assign our base filename to the variable BASE_FN for later use.

The RANDOM shell variable is initialized with an initial seed, which we decided to be the current process ID (PID), specified by the \$\$ shell variable. This technique helps to ensure that the initial seed changes each time the shell script is executed. For this shell script we want to use the maximum value as the UPPER_LIMIT, which is 32,767. If you need a longer or shorter pseudo-random number, you can change this value to anything you want. If you make this number longer than five digits the extra preceding digits will be zeros. There are four more variables that need to be initialized. We initialize both CURRENT_SECOND and LAST_SECOND to non-second values 99 and 98, respectively. The USED_NUMBERS list is initialized to NULL, and the PROCESSING variable is initialized to TRUE. The PROCESSING variable allows the loop to continue creating unique filenames and to keep calling the my_process function. Any non-TRUE value stops the loop and thus ends execution of the shell script.

A while loop is next in our shell script, and this loop is where all of the work is done. We start out by getting a new date/time stamp and the current second on each loop iteration. Next a new pseudo-random number is created and is assigned to the RN variable. If the current second is the same as the last second, then we start another loop to ensure that the number that we created has not been previously used during the current second. It is highly unlikely that a duplicate number would be produced in such a short amount of time, but to be safe we need to do a sanity check for any duplicate numbers.

When we get a unique number we are ready to put the new filename together. We have three variables that together make up the filename: *\$BASE_FN, \$DATE_ST,* and *\$RN*. The next command puts the pieces together and assigns the newly constructed filename to the variable to the UNIQUE_FN variable:

UNIQUE_FN=\${BASE_FN}.\${DATE_ST}.\$RN

Notice the use of the curly braces ({}) around the first two variables, BASE_FN and DATE_ST. The curly braces are needed because there is a character that is not part of the variable name without a space. The curly braces separate the variable from the character to ensure that we do not get unpredictable output. Because the last variable, \$RN, does not have any character next to its name, the curly braces are not needed, but it is *not* a mistake to add them.

The only thing left is to assign the \$CURRENT_SECOND value to the LAST_SECOND variable and to execute the my_program function, which actually uses the newly created filename. I have commented out the code that would stop the script's execution. You will need to edit this script and make it suitable for your particular purpose. The mk_unique_filename.ksh shell script is shown in action in Listing 11-15.

```
yogi@/scripts# ./mk_unique_filename.ksh /tmp/myfilename
/tmp/myfilename.120607.131507.03038
/tmp/myfilename.120607.131507.15593
/tmp/myfilename.120607.131507.11760
/tmp/myfilename.120607.131508.08374
/tmp/myfilename.120607.131508.01926
/tmp/myfilename.120607.131508.07238
/tmp/myfilename.120607.131509.07554
/tmp/myfilename.120607.131509.12343
/tmp/myfilename.120607.131510.08496
/tmp/myfilename.120607.131510.18285
/tmp/myfilename.120607.131510.18895
/tmp/myfilename.120607.131511.16618
/tmp/myfilename.120607.131511.30612
/tmp/myfilename.120607.131511.16865
/tmp/myfilename.120607.131512.01134
/tmp/myfilename.120607.131512.19362
/tmp/myfilename.120607.131512.04287
/tmp/myfilename.120607.131513.10616
/tmp/myfilename.120607.131513.08707
/tmp/myfilename.120607.131513.27006
/tmp/myfilename.120607.131514.15899
/tmp/myfilename.120607.131514.18913
/tmp/myfilename.120607.131515.27120
/tmp/myfilename.120607.131515.23639
/tmp/myfilename.120607.131515.13096
/tmp/myfilename.120607.131516.19111
/tmp/myfilename.120607.131516.05964
/tmp/myfilename.120607.131516.07809
/tmp/myfilename.120607.131524.03831
/tmp/myfilename.120607.131524.21628
/tmp/myfilename.120607.131524.19801
/tmp/myfilename.120607.131518.13556
/tmp/myfilename.120607.131518.24618
/tmp/myfilename.120607.131518.12763
# Listing of newly created files
yogi@/tmp# ls -ltr /tmp/myfilename.*
-rw-r--r- root system Dec 06 13:15
/tmp/myfilename.120607.131507.15593
```

Listing 11-15 mk_unique_filename.ksh shell script in action

```
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131507.03038
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131508.08374
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131508.01926
-rw-r--r- root system Dec 06 13:15
/tmp/myfilename.120607.131507.11760
-rw-r--r- root system Dec 06 13:15
/tmp/myfilename.120607.131509.12343
-rw-r--r-- root system Dec 06 13:15
/tmp/mvfilename.120607.131509.07554
-rw-r--r- root system Dec 06 13:15
/tmp/myfilename.120607.131508.07238
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131510.18285
-rw-r--r- root system Dec 06 13:15
/tmp/myfilename.120607.131510.08496
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131511.30612
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131511.16618
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131510.18895
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131512.19362
-rw-r--r- root system Dec 06 13:15
/tmp/myfilename.120607.131512.01134
-rw-r--r- root system Dec 06 13:15
/tmp/myfilename.120607.131511.16865
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131513.10616
-rw-r--r- root system Dec 06 13:15
/tmp/myfilename.120607.131513.08707
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131512.04287
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131514.18913
-rw-r--r- root system Dec 06 13:15
/tmp/myfilename.120607.131514.15899
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131513.27006
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131515.27120
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131515.23639
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131515.13096
```

Listing 11-15 (continued)

```
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131516.19111
-rw-r--r- root system Dec 06 13:15
/tmp/myfilename.120607.131516.05964
-rw-r--r- root system Dec 06 13:15
/tmp/myfilename.120607.131524.21628
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131524.03831
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131516.07809
-rw-r--r- root system Dec 06 13:15
/tmp/myfilename.120607.131518.24618
-rw-r--r- root system Dec 06 13:15
/tmp/myfilename.120607.131518.13556
-rw-r--r-- root system Dec 06 13:15
/tmp/myfilename.120607.131524.19801
-rw-r--r- root system Dec 06 13:15
/tmp/myfilename.120607.131518.12763
```

Listing 11-15 (continued)

Creating a File Filled with Random Characters

Ever need a file filled with random character data? Well, here's your miracle! The shell script in Listing 11-16 will create a user-defined-MB-size file that contains random alphanumeric characters.

For this one, study the random_file.Bash shell script in Listing 11-16 first. At the end we will cover this script in detail.

```
#!/bin/Bash
# SCRIPT: random_file.Bash
# AUTHOR: Randy Michael
# DATE: 8/3/2007
# REV: 1.0
# PLATFORM: Not Platform Dependent
#
# PURPOSE: This script is used to create a
#
    specific size file of random characters.
      The methods used in this script include
#
      loading an array with alphanumeric
#
      characters, then using the /dev/random
#
       character special file to seed the RANDOM
#
       shell variable, which in turn is used to
```

Listing 11-16 random_file.Bash shell script

```
www.it-ebooks.info
```

```
extract a random set of characters from the
#
     KEYS array to build the OUTFILE of a
#
#
      specified MB size.
# set -x # Uncomment to debug this script
#
# set -n # Uncomment to check script syntax
       # without any execution. Do not forget
#
       # to put the comment back in or the
#
       # script will never execute.
#
****
# DEFINE FILES AND VARIABLES HERE
*****
typeset -i MB_SIZE=$1
typeset -i RN
typeset -i i=1
typeset -i X=0
WORKDIR=/scripts
OUTFILE=${WORKDIR}/largefile.random.txt
>$OUTFILE
THIS_SCRIPT=$(basename $0)
CHAR_FILE=${WORKDIR}/char_file.txt
****
# DEFINE FUNCTIONS HERE
*****
build_random_line ()
# This function extracts random characters
# from the KEYS array by using the RANDOM
# shell variable
C=1
LINE=
until (( C > 79 ))
do
  LINE="${LINE}${KEYS[$(($RANDOM % X + 1))]}"
  ((C = C + 1))
done
# Return the line of random characters
echo "$LINE"
}
```

```
*****
elapsed_time ()
{
SEC=$1
((SEC < 60)) & echo -e "[Elapsed time: \
$SEC seconds]\c"
(( SEC >= 60 && SEC < 3600 )) && echo -e \
"[Elapsed time: $(( SEC / 60 )) min $(( SEC % 60 )) sec]\c"
(( SEC > 3600 )) && echo -e "[Elapsed time: \
$(( SEC / 3600 )) hr $(( (SEC % 3600) / 60 )) min \
$(( (SEC % 3600) % 60 )) sec]\c"
3
******
load_default_keyboard ()
{
# Loop through each character in the following list and
# append each character to the $CHAR FILE file. This
# produces a file with one character on each line.
for CHAR in 1 2 3 4 5 6 7 8 9 0 q w e r t y u i o \
          pasdfghjkl zxcvbnm \
          QWERTYUIOPASDFGHJKL \
          Z X C V B N M 0 1 2 3 4 5 6 7 8 9
do
    echo "$CHAR" >> $CHAR_FILE
done
}
*****
usage ()
{
echo -e "\nUSAGE: STHIS SCRIPT Mb size"
echo -e "Where Mb size is the size of the file to build\n"
}
*****
# BEGINNING OF MAIN
*****
if (( $# != 1 ))
then
    usage
```

```
exit 1
fi
# Test for an integer value
case $MB_SIZE in
[0-9]) : # Do nothing
          ;;
       *) usage
         ;;
esac
# Test for the $CHAR_FILE
if [ ! -s "$CHAR_FILE" ]
then
    echo -e "\nNOTE: $CHAR_FILE does not esist"
    echo "Loading default keyboard data."
    echo -e "Creating $CHAR_FILE...\c"
    load_default_keyboard
    echo "Done"
fi
# Load Character Array
echo -e "\nLoading array with alphanumeric character elements"
while read ARRAY_ELEMENT
do
    ((X = X + 1))
   KEYS [$X] = $ARRAY_ELEMENT
done < $CHAR_FILE
echo "Total Array Character Elements: $X"
# Use /dev/random to seed the shell variable RANDOM
echo "Querying the kernel random number generator for a random seed"
RN=$(dd if=/dev/random count=1 2>/dev/null \
    | od -t u4 | awk '{print $2}'| head -n 1)
# The shell variable RANDOM is limited to 32767
echo "Reducing the random seed value to between 1 and 32767"
RN=$(( RN % 32767 + 1 ))
```

```
# Initialize RANDOM with a new seed
echo "Assigning a new seed to the RANDOM shell variable"
RANDOM=$RN
echo "Building a $MB_SIZE MB random character file ==> $OUTFILE"
echo "Please be patient, this may take some time to complete..."
echo -e "Executing: .\c"
# Reset the shell SECONDS variable to zero seconds.
SECONDS=0
TOT_LINES=$(( MB_SIZE * 12800 ))
until (( i > TOT_LINES ))
do
    build random line >> $OUTFILE
    (($((i % 100)) == 0)) && echo -e ".\c"
    ((i = i + 1))
done
# Capture the total seconds
TOT_SEC=$SECONDS
echo -e "\n\nSUCCESS: $OUTFILE created at $MB_SIZE MB\n"
elapsed_time $TOT_SEC
# Calculate the bytes/second file creation rate
(( MB_SEC = ( MB_SIZE * 1024000 ) / TOT_SEC ))
echo -e "\n\nFile Creation Rate: $MB_SEC bytes/second\n"
echo -e "File size:\n"
1s -1 $OUTFILE
echo
*****
# END OF random_file.Bash SCRIPT
*****
```

For this script to work as written, you will need to create a /scripts directory. Before we go through the code in the random_file.Bash script in Listing 11-16, let's look at the resulting output in Listing 11-17.

```
[root@booboo scripts]# ./random_ le.Bash 5
Loading array with alphanumeric character elements
Total Array Character Elements: 72
Querying the kernel random number generator for a random seed
Reducing the random seed value to between 1 and 32767
Assigning a new seed to the RANDOM shell variable
Building a 5 MB random character le ==> /scripts/large le.random.txt
Please be patient, this may take some time to complete ...
Executing:
SUCCESS: /scripts/large le.random.txt created at 5 MB
[Elapsed time: 5 min 42 sec]
File Creation Rate: 14970 bytes/second
File size:
-rwxr-xr-x 1 root root 5120000 2007-08-07 09:04
/scripts/large le.random.txt
[root@booboo scripts] # head /scripts/large le.random.txt
zo0ZT4y0K4OAacvW712S0Xi18262xQV6b4Zt671FzzfDL9j487F620YHC1nN830PDCj4aR0R8y8Mn26
AYWUDcHP74E1NWtBQBomvA0KpXAE4Q7k5oFCLaQ23HJ9eWb0UMimE6Rx50LkpIEF1HwRbGaInVIFz4y
vW98z9cJhkJ0B66kdGT67fI4t1BQF7G1c1Lgd0q3Fde49oh1W3pUZ3v4Sutu176SYs8qdMRI9SPM0g9
CjaWtjnTD01Rp6Go8qgtbiQjgwN8eieGT8D8YGzQMj19AnHPbZmVkJ3ngYu574t1tNI0GwvAtx6ZQDr
5RK66tJzIw168Q2wHQHTn5oRZA22Ab3v3GHLB0a16KU7dhco2X968MQNKIuzMP3pnE3vlptj1RrF1MC
6b8bqw3kc4UB3dqKs2hKAjVJkvk9VNPhzPaA0j2U4Q2hzZOrUdRem2Q5ypqa6H0I8m8NZr7vwMSB39C
yyIJZdbb65HECXD13JXZys8sGeJ88T17fZmCvhb06Q5G4Wx5Sr4Vc2Y2Ft5y3kfTzaK0ifHN9kBW1sX
K9tgpgpWvLld7A0h53S6uvneBrT21crJXMD8NDEJ9CGS9y1eMPmLfuH7prMME0Bn451adD36MJRR1E6
BcaBG087NZslC38X7aaj44NW3fn8yz7sWD4TwBtUpJvm6w1x32P9tcviM1Q6YyWm3DXrP31hNTvcZfX
KqtILEaLBjv21NJU8bpk9f3zwAy92h8XE1I5wxpKw3Qqvbmc8V00iTdU8zqHj9P8h9651Qaqh9qrtr9
[root@booboo scripts]#
```

Listing 11-17 random_file.Bash shell script in action

Now let's go through the random_file.Bash shell script in Listing 11-16 from the top.

We first define some files and variables. The MB_SIZE variable is a user-defined command-line argument, \$1. We typeset the MB_SIZE variable to an integer value, and assign \$1 to this variable all in one statement. We also define our \$OUTFILE and \$CHAR_FILE files. The \$OUTFILE is the random file we create and the \$CHAR_FILE is the character file we use to load alphanumeric characters into the KEYS array.

Next we define some functions. The build_random_line utilizes the following while loop to create one line of random alphanumeric characters:

```
until (( C > 79 ))
do
    LINE="${LINE}${KEYS[$(($RANDOM % X + 1))]}"
    (( C = C + 1 ))
done
```

You may wonder why we produce 79 and not 80 characters per line. The new-line character takes up the last character in the line.

The next function is elapsed_time. This function evaluates the seconds, passed as ARG1 to the function, and outputs the hours, minutes, and seconds.

We added the load_default_keyboard function to build a \$CHAR_FILE if the user has not created a custom file.

The BEGINNING OF MAIN is intuitively obvious when you look closely at the code. We first check for a single command-line argument. Next we test the value to ensure \$MB_SIZE is an integer. Then we test to ensure the \$CHAR_FILE exists and its size is greater than zero bytes. If this test fails, we execute the build_default_keyboard function to build our required \$CHAR_FILE. The next loop loads the array KEYS with all the characters in the \$CHAR_FILE. By creating random numbers between 1 and the total number of array elements, we can point to random characters stored in the array.

The next step is to use the /dev/random character special file to create a seed for the shell variable RANDOM. The trick for this is to use modulo N arithmetic to reduce the number to less than 32,767, and assign this value to the RANDOM variable.

Now, just before we start producing random characters, we reset the shell SECONDS variable to 0, zero. The shell SECONDS variable continuously counts seconds. It starts counting seconds as soon as the shell is started. If you type echo \$SECONDS on the command line, you will see the seconds that elapsed since you logged in, or at least since you started this particular shell. We can set the SECONDS shell variable to any integer values we want, and it will continue counting from that point as the seconds tick by. We need the second of execution for the elapsed_time function.

For each 1 MB in desired file size, we need to create 12,700 lines of random characters. So, we just multiply \$MB_SIZE by 12,700 and that is how many times we loop to create the random file.

Once this loop completes, we save the SECONDS by assigning this value to the TOT_SEC variable. We execute the elapsed_time function and give the user feedback of the execution time, then we calculate the bytes per second, and finally do a long listing of the \$OUTFILE using ls -1 \$OUTFILE.

Other Things to Consider

As with any chapter, we can add to the shell scripts. To find the fastest method, we sometimes need to use the shell's built-in **time** command. To time the execution of a script, just precede the program or shell script to execute with time:

```
time my_shell_script
```

The shell's time command is different from the UNIX /usr/bin/time command. The shell's time command produces a more granular result, as shown here:

Command:

/usr/bin/time ls -l /usr Timing result: Real 0.32 User 0.07 System 0.08 Command:

time ls -l /usr

Timing result:

real	0m0.15s
user	0m0.08s
sys	0m0.07s

Summary

In this chapter we stepped through some different techniques of creating pseudorandom numbers. We then used this knowledge to create unique filenames, and then created a file, of a specified MB size, filled with random character data.

Using the /dev/random character special file produces numbers suitable for some security-related projects; however, the RANDOM shell variable will produce predictable and cyclical numbers. Play around with these shell scripts and functions and modify them for your needs. In Chapter 12 we will use pseudo-random numbers to create pseudo-random passwords. Just remember, you can make mathematics as difficult as you wish. I hope you gained a lot of knowledge in this chapter and that you stick around for the next chapter!

Lab Assignments

- 1. Write a shell script that contains three functions. Function 1 will create 1,000 random numbers using the RANDOM shell variable. Function 2 will create 1,000 random numbers using the /dev/urandom character special file. Function 3 will create 1,000 random numbers using the /dev/random character special file. In the shell script, use the shell time utility to record the execution times of each method. Explain any differences you observe.
- 2. Write a shell script to create a 128-character encryption key. Use the highest level of randomness for this script.
- 3. Rewrite the random_file.Bash shell script in Listing 11-16 but this time use the /dev/urandom character special file to create all the random numbers. Is there a difference in execution time? Explain any differences observed.

CHAPTER 12

Creating Pseudo-Random Passwords

Got security? Most of the user community does not know how to create secure passwords that are not easy to guess. Users tend to have several passwords that they rotate. The problem with these "rotating" passwords is that they are usually easy to guess. For example, users find that birth dates, children's names, Social Security numbers, addresses, department names/numbers, and so on make good passwords that are easy to remember. Sometimes they even use words found in any dictionary, which is a starting point for any cracker. In this chapter we are going to create a shell script that creates *pseudo-random passwords*.

Randomness

If you look back at Chapter 11, "Pseudo-Random Number and Data Generation," you can see the exercise that we used to create pseudo-random numbers. If you have not studied Chapter 11, it would help a lot if you did.

The numbers we created in Chapter 11 are not true random numbers, but some are secure enough for encryption applications and software keys. Others, however, are not as secure because of the cyclical nature of how "random numbers" are created if the same seed is used as a starting point. For example, if you are using the RANDOM shell variable, and you always start a random number sequence with the same *seed*, or first number, you will always have the same sequence of numbers. In Chapter 11, one method used is to use the process ID (PID) of the current process, which is the executing shell script, as the seed for creating pseudo-random numbers. This use of the PID is one good choice because PIDs are created by the system in a somewhat random nature.

The other methods we studied include using the character special files /dev/random and /dev/urandom. These character special files are an interface to the kernel's random number generator. The kernel's random number generator uses *system noise* from device

drivers, and other environmental noise a computer generates, to create an *entropy pool*. The kernel's random number generator keeps track of the number of bits of noise remaining in the entropy pool. When we read the /dev/random character special file, using the **dd** command, random bytes of noise are returned within the range of available noise in the entropy pool. When the entropy pool is empty, /dev/random will *block* while the kernel gathers more environmental system noise. During this blocking period, /dev/random will not return data and we can see a pause in execution of our shell scripts.

The /dev/urandom character special file does not block when the entropy pool is empty; therefore, the randomness is to a lesser degree. Using /dev/urandom in a shell script will not show the possible delays in execution as /dev/random and therefore is the preferred method for many shell scripts.

Now that I have lost you in random numbers, you are probably asking, "What does a random number have to do with a password?" As we proceed, the answer will be intuitively obvious.

Creating Pseudo-Random Passwords

We started this chapter with a discussion on randomness because we are going to use computer-generated pseudo-random numbers, and then use these generated numbers as pointers to specific array elements of keyboard characters, which are stored in the array KEYS. In this chapter you get a practical use for generating random numbers. And you thought Chapter 11 was a waste of time.

The script idea goes like this: We use an external file that contains keyboard characters, one character per line. You can put any keyboard characters that you want in this file. I just went down the rows on the keyboard from left to right, starting on the top row of keys with numbers. As I went through all the keyboard keys, I then added a second set of numbers from the number keypad, as well as all the uppercase and lowercase characters. The nice thing about this strategy is that you have the ability to specify the exact group of characters that make a valid password in your shop. Country-specific keyboards, which use characters other than those of the U.S. keyboards, also benefit from this strategy. Beware that characters with a special meaning should be delimited with a backslash (\) — for example, $\langle . | .$

Once we have the keyboard file created, we load the keyboard data into an *array*. Don't panic! Bash, Bourne, and Korn shell arrays are easy to work with, as you will see in the scripting section as well as in the array introduction section. When we have all array elements loaded, we know how many total elements we have to work with. Using techniques described in Chapter 11, we create pseudo-random numbers between one and the total number of array elements, *n*. With an array *pointer*, which is nothing more than our pseudo-random number, pointing to an individual array element, which is the next random character, we add this character as we build a text string. The default length of this character string, which is the password we are creating, is eight characters; however, this can be changed on the command line to make the password longer or shorter by adding an integer value specifying the new password length.

The final step is to print the password to the screen. We also add two command-line switch options, -n and -m. The -n switch specifies that the user wants to create a

new default keyboard data file. The -m switch specifies that the user wants to print a *password page*. In our shop we are required to put some passwords, such as root, in multiple security envelopes to be locked in a safe, just in case. To remove the risk of typos, I print the password page, which has three copies of the password data on the same page, and then I cut the sheet into three pieces. After folding each of the three slips of paper and sealing each one in a security envelope, I give them to my Admin Manager.

As you can see, creating passwords is not something that I take lightly! Weak passwords make for a real security risk, and any user, especially a Systems Administrator, needs to take a proactive approach to create secure passwords that are as random as you can make them. This chapter is a valuable asset to any security team as well as for the common user.

Syntax

As with any of our scripting sessions we first need the correct syntax for the primary commands that we are going to use in the shell script. In this case we need to introduce arrays and the commands that are used to work with the array and the array elements. There is a lot more than loading an array to creating this shell script. We will get to the random numbers later. When we get to the scripting section, you will see the other tasks that I have in mind, and you can pick up a pointer or two from the chapter.

Arrays

In Bash, Bourne, and Korn shells, we can create one-dimensional *arrays*. A onedimensional array contains a sequence of *array elements*, which are like the boxcars connected together on a train track. An array element can be just about anything, except for another array. I know you're thinking that you can use an array to access an array to create two- and three-dimensional arrays. This sounds like a great Lab Assignment, but otherwise, it is beyond the scope of this book, so we will stick to one-dimensional arrays.

For our task, we are going to load our array with single-character array elements that are loaded into the array from an external file. However, an array element can be a text string, number, line of text, print queue name, web address, or just about anything you can list. For this application of an array we only need a single character.

Loading an Array

An array can be loaded in two ways. You can define and load the array in one step with the set -A command, or you can load the array one element at a time. Both techniques are shown here.

Defining and Loading the KEYS Array in One Step

We can load the KEYS array in one step using the set command with the -A (defining an array) switch option, as shown here.

set -A KEYS q w e r t y u i o p \[\] a s d f g h j k l \\$

Notice in this list that the characters [,], and \$ have been *escaped* to remove their special function by adding a *backslash* character. If we do not escape these characters, errors and strange behavior may occur as you tried to load or display the array elements. You will see this on a larger scale in the shell script. Also remember that if you enclose a list in double quotes or single tic marks, it is treated as a single array element for the assignment, not as individual array elements.

Loading the KEYS Array One Array Element at a Time

The second option for loading the array KEYS is to use a while read loop and use a file as input to the while loop. In this example we load the array elements one at a time using a counter to index the KEYS array:

```
X=0
while read ARRAY_ELEMENT
do
   (( X = X + 1 ))
    KEYS[$X]="$ARRAY_ELEMENT"
done < $ARRAY_ELEMENT_FILE</pre>
```

The first loading option, which uses the set -A command, requires that you hard-code the keyboard layout into the shell script, which removes much of the flexibility that you want when restricting or expanding password content. Using the while loop method we can use an external file and load this file with any characters that we want, and we can have as many or as few characters defined for passwords as we like. We can also duplicate characters and change the order of the characters any way we wish.

As the counter is incremented on each while loop iteration, we load the array elements in sequential order from the top of the file, and starting with array element 1, KEYS[1]. When we get to the end of the file, we know how many elements we have loaded in the array by the value of the array counter, \$X. To see the specific value of array element 22, you can use the following syntax:

```
# echo ${KEYS[22]}
;
```

As you can see from the response, the 22nd array element that was loaded is a semicolon character (;). We can also display the number of array elements using either of the following two options:

```
# echo ${#KEYS[*])
# echo ${#KEYS[@])
```

Notice we started with array element 1, one. The UNIX shells also support array element 0, zero, but the pseudo-random numbers we create start at array element one, not zero. We will look at arrays more closely as we write our shell script.

Building the Password-Creation Script

I want to explain this shell script one step at a time, and we have a lot to cover, so let's get started. First, you need to understand the order of execution and each task that is involved in this script.

Order of Appearance

We need to start out by defining the files and variables that are required for the script. The following section shows the variables that are defined for this shell script.

Default password length. LENGTH=8 # Persons to notify if the password is revealed or the "glass has been broken." NOTIFICATION_LIST=<Manager notification list> # Default printer to print the password report. DEFAULT PRINTER=<printer or queue name> # The name of this shell script with the directory path removed. SCRIPT=\$ (basename \$0) # Temporary hold file for the printer report. OUTFILE=/tmp/tmppwd.out # File containing keyboard characters. KEYBOARD_FILE=/scripts/keyboard.keys # Print report flag. PRINT_PASSWORD_MANAGER_REPORT=<TRUE or Anything else> # This next test ensures we use the correct # echo command based on the executing shell ECHO="echo -e" [[\$(basename \$SHELL) == ksh]] && ECHO=echo # Ensure we use the correct echo command syntax for each executing shell.

The purpose of each variable is shown after the pound sign (#) on each line. Notice in particular the last entry where we test for the shell we are executing in. Because we use the echo command, and we sometimes use the *backslash operators*, (for example, \n), the executing shell is important. If the \$SHELL variable is sh or Bash, we need to add the -e switch to the echo command. If we are executing in ksh, the -e is not required to recognize these special operators. For the purpose of reviewing the following functions, only a single echo will be used. The actual mk_passwd.Bash shell script does test and use this shell test method, and thus \$ECHO executes the proper echo command.

Define Functions

We have six functions to go through in this section. The functions described here are listed in their order of appearance in the shell script, mk_passwd.Bash. In each of the function descriptions there is a function listing for you to follow through. The shell declaration on the first line of the script can be Bourne, Bash, or Korn; we are working shell-neutral.

in_range_random_number Function Description

As we studied in Chapter 11, the /dev/random and /dev/urandom character special files use system noise to create random numbers. Both methods offer an easy way to change the seed value each time we start generating numbers. However, the /dev/urandom will not block should the entropy pool become empty. So, we will use this option for our random numbers.

We often want to limit the range of numbers not to exceed a user-defined maximum. An example is creating lottery numbers between 1 and the maximum number, which might be 36. We are going to use the modulo arithmetic operator (%) to reduce all numbers to a fixed set of numbers between [0..N-1], which is called *modulo N arithmetic*. We are going to use this pseudo-random number to index array elements in the KEYS array.

For our number range we need a script-defined maximum value, which we will assign to a variable called UPPER_LIMIT. This UPPER_LIMIT variable is defined when the KEYS array has been loaded because it represents the total number of elements that are contained in the KEYS array. The modulo operator is the percent sign (%), and we use this operator the same way that we use the forward slash (/) in division. However, the result is the *remainder* of the division, where the / returns the integer part of the division result. On each loop iteration we assign our new pseudo-random number to the RN variable. This time, though, we are going to use the following equation to limit the number to not exceed the script-defined maximum:

```
RANDOM_NUMBER=$(( RN % UPPER_LIMIT + 1))
```

Notice that we added one to the result. Using the preceding equation will produce a pseudo-random number between 1 and the script-defined \$UPPER_LIMIT, which is the total number of elements in the KEYS array. We use this function to point to a random array element to produce a password character. The function using this equation is in_range_random_number and is shown in Listing 12-1.

```
function in_range_random_number
{
    # Create a pseudo-random number less than or equal
    # to the $UPPER_LIMIT value, which is defined in the
    # main body of the shell script.

RN=$(dd if=/dev/random count=1 2>/dev/null \
```

Listing 12-1 in_range_random_number function

```
| od -t u2 | awk '{print $2}'| head -n 1)
RANDOM_NUMBER=$(( RN % UPPER_LIMIT + 1))
echo "$RANDOM_NUMBER"
}
```

Notice in the function in Listing 12-1 that we use the /dev/urandom character special file in concert with dd and od. We assume here that there is a script-defined UPPER_LIMIT variable set. This function will produce numbers between 1 and the script-defined maximum value.

load_default_keyboard Function Description

As it turns out, the \$KEYBOARD_FILE can have as many or as few characters as you wish to add. However, many users want a quick startup and an easy way to create this required file? This is the reason why I added this function to themk_passwd.kshshell script.

There are two mechanisms for loading a *default* keyboard layout. The first way is when the shell script is unable to locate the \$KEYBOARD_FILE on the system. In this case, the user is prompted to load the default keyboard layout. The second option is to add -n as a command-line switch. We will get to parsing command-line switches later in this chapter. In either of the two situations the user is still prompted before the \$KEYBOARD_FILE is loaded with default keyboard layout.

Other than prompting the user to load the default keyboard layout, we need to supply a list of keyboard characters to load into the file. At this point let's look at the function code in Listing 12-2 and cover the details at the end.

```
function load_default_keyboard
{
    # If a keyboard data file does not exist then the user
    # is prompted to load the standard keyboard data into the
    # $KEYBOARD_FILE, which is defined in the main body of
    # the shell script.
    clear    # Clear the screen
    echo "\nLoad the default keyboard data file? (Y/N): \c"
    read REPLY
    case $REPLY in
    y|Y):
        ;;
    *) echo "\nSkipping the load of the default keyboard file...\n"
```

Listing 12-2 load_default_keyboard function

```
return
    ;;
esac
cat /dev/null > $KEYBOARD_FILE
echo "\nLoading the Standard U.S. QWERTY Keyboard File...\c"
# Loop through each character in the following list and
# append each character to the $KEYBOARD_FILE file. This
# produces a file with one character on each line.
for CHAR in \' 1 2 3 4 5 6 7 8 9 0 \- \= \\ gwertyuio \
            p \[ \] asdfghjkl \; \' zxcvbnm \, \
            \. \/ \\ \~ \! \@ \# \$ \% \^ \& \* \( \) _ \+ \| \
            QWERTYUIOP \{ \} ASDFGHJKL \: \" \
            Z X C V B N M \< \> \? \| \. 0 1 2 3 4 5 6 7 8 9 \/ \
            \* \- \+
do
    echo "$CHAR" >> $KEYBOARD_FILE
done
echo "\n\\"...\n"
sleep 1
```

Now I want to direct your attention to the for loop in Listing 12-2, which is in boldface text. The idea is to loop through each character one at a time and append the character to the *\$KEYBOARD_FILE*. The result is a file that contains the keyboard layout, listed one character per line. The file shows one character per line to make it easier to load the file and the *KEYS* array.

In the list of characters please notice that most of the non-alphanumeric characters are preceded by a backslash (\), not just the shell special characters. As we discussed previously, this backslash is used to *escape* the special meaning of these characters. When you precede a special character with the backslash, you are able to use the character as a literal character, just like the alphanumeric characters, and if a backslash precedes the other non-alphanumeric characters, it is ignored. The list of characters that are escaped is shown here:

' ! @ # \$ % ^ & * () _ - = + [] { }

On each loop iteration one character is appended to the \$KEYBOARD_FILE using the following command:

echo "\$CHAR" >> \$KEYBOARD_FILE

When the file is loaded, which happens extremely fast, we notify the user that the load is complete and then sleep for one second. I added this sleep 1 at the end of

this function because the load happened so fast that the user needed a second to see the message.

check_for_and_create_keyboard_file Function Description

Is this function name descriptive enough? I like to know exactly what a function is used for by the name of the function.

The purpose of this function is to check for the existence of the <code>\$KEYBOARD_FILE</code> and to prompt the user to load the default keyboard layout into the <code>\$KEYBOARD_FILE</code>. The user has the option to load the default data or not to load it. If the user declines to load the keyboard data file, this script will not work. To get around this little problem, we just notify the user of this error and exit the shell script.

When the user gets the error message, he or she is also informed of the name of the missing file and a description of what the script expects in the file — specifically, one keyboard character per line. The full function is shown in Listing 12-3.

```
function check_for_and_create_keyboard_file
{
# If the $KEYBOARD_FILE does not exist then
# ask the user to load the "standard" keyboard
# layout, which is done with the load_default_keyboard
# function.
if [ ! -s $KEYBOARD_FILE ]
then
     echo "\n\nERROR: Missing Keyboard File"
     echo "\n\nWould You Like to Load the"
     echo "Default Keyboard Layout?"
     echo "\n\t(Y/N): \c"
     typeset -u REPLY=FALSE
     read REPLY
     if [[ $REPLY != Y ]]
     then
          echo "\n\nERROR: This shell script cannot operate"
          echo "without a keyboard data file located in"
          echo "\n==> $KEYBOARD_FILE\n"
          echo "\nThis file expects one character per line."
          echo "\n\t...EXITING...\n"
          exit 3
else
          load default keyboard
          echo "\nPress ENTER when you are you ready to continue: \c"
          read REPLY
          clear
     fi
fi
}
```

Listing 12-3 check_for_and_create_keyboard_file function

To check for the existence of the \$KEYBOARD_FILE, we use the -s test in an if statement, as shown here:

Notice that we negated the test by adding an exclamation point (! -s). This is actually a test to see if the file is *not* greater than zero bytes in size *or* that the \$KEYBOARD_FILE does not exist. If either of these conditions is met, we display some messages to the user and ask the user if the default keyboard layout should be loaded.

If the user acknowledges the question with a "Y" or a "y," we execute the load_default_keyboard function, which we studied in the previous section, "load_default_keyboard Function Description." After the keyboard data is loaded into the \$KEYBOARD_FILE, we stop and ask the user to press ENTER to continue. Once the user presses ENTER, the script creates a pseudo-random password, which we will cover in a later section.

build_manager_password_report Function Description

You may be asking, "Why do you want to *print* a password?" There are a lot of reasons to print a password, but only *one* of the answers is valid! *For security reasons*. Now I really lost you! How can a printed password be good for security? It's simple: The root password needs to be protected at all costs. Our machines do not have direct login access to root, but we use an auditing script that captures every keystroke of the root user. If a machine has failed and you need to log on to the system on the console, you are definitely going to need access to the root password. For this reason we keep three copies of the root password in secure envelopes, and they get locked up for safekeeping.

The build_manager_password_report function creates a file, pointed to by the \$OUTFILE variable, that has three copies of the same information on a single page. Look at the function shown in Listing 12-4 to see the message.

```
function build_manager_password_report
{
    # Build a file to print for the secure envelope
(
    echo "\n RESTRICTED USE!!!"
echo "\n\timmediately send an e-mail to:\n"
echo " $NOTIFICATION_LIST"
echo "\n\tif this password is revealed!"
echo "\n\tAIX root password: $PW\n"
echo "\n\n"
```

Listing 12-4 build_manager_password_report function

```
echo "\n
                       RESTRICTED USE!!!"
echo "\n\n\tImmediately send an e-mail to:\n"
        $NOTIFICATION_LIST"
echo "
echo "\n\tif this password is revealed!"
echo "\n\tAIX root password: $PW\n"
echo "\n\n"
echo "\n
                     RESTRICTED USE!!!"
echo "\n\n\tImmediately send an e-mail to:\n"
echo " $NOTIFICATION_LIST"
echo "\n\tif this password is revealed!"
echo "\n\tAIX root password: $PW\n"
   ) > $OUTFILE
}
```

Notice that the entire message is enclosed in parentheses, with the final output redirected to the *\$OUTFILE* file using a single output redirection statement at the end of the function, as shown in the following syntax:

```
( echo statements.... ) > $OUTFILE
```

This method runs all the echo commands as a separate shell and sends the resulting output to the *\$OUTFILE* using output redirection.

Also notice the \$NOTIFICATION_LIST variable. This variable is set in the main body of the script. This variable contains the list of people who must be notified if the password is ever released, as stated in the message in the function. This list can also be accomplished by using a sendmail alias. To create a sendmail alias, edit the aliases file, which is usually located in the /etc/mail or /etc/sendmail directory. Edit the aliases file in that directory and include the following syntax to create a new alias:

```
pwd_alert: eddie@mycomp.com, susan@mycorp.com, admin_support
```

Adding this line to the aliases file creates a new mail alias named pwd_alert. Sending mail to this alias will send email to everyone listed after the colon: Eddie, Susan, and everyone listed in the admin_support mail alias.

To make the new alias work, though, do not forget to run the **newaliases** command. The newaliases command rereads the aliases file to pick up the changes. For more details, see the sendmail man page, **man sendmail**.

When I want to send one of these printouts to the printer, I always run to get it as soon as the page comes out of the printer. This is an extremely important piece of

paper! I take it to my desk and cut the page into three pieces and seal each one in a secure envelope and have it locked up for safekeeping.

A sample manager's password report is shown in Listing 12-5.

```
RESTRICTED USE!!!
    Immediately send an e-mail to:
Donald Duck, Yogi Bear, and Mr. Ranger
   if this password is revealed!
   UNIX root password: E-,6Kc11
            RESTRICTED USE!!!
    Immediately send an e-mail to:
Donald Duck, Yogi Bear, and Mr. Ranger
   if this root password is revealed!
   UNIX root password: E-,6Kc11
            RESTRICTED USE!!!
    Immediately send an e-mail to:
Donald Duck, Yogi Bear, and Mr. Ranger
    if this root password is revealed!
   UNIX root password: E-,6Kc11
```

Listing 12-5 Password report printout

You need to edit this function and change the message to suit your environment. If you do not need this functionality, then never use the -m switch, or reply "No" when

asked to confirm the printing. This sounds like a good Lab Assignment. Edit the script and remove the functionality completely.

usage Function Description

It is always a good idea to show the user a USAGE: statement when incorrect or insufficient input is detected. (We will get to detecting input errors later in this chapter.) For our mk_passwd.ksh shell script we have four options and several combinations.

We can execute the mk_passwd.ksh script with no arguments, and we can execute the mk_passwd.ksh shell script with the -n and -m command-line switches. The -n switch automatically loads the default keyboard layout into the \$KEYBOARD_FILE file. We can also change the length of the password, which is defined as eight characters by default. Any combination of these command options can be executed. Please look closely at the Backus-Naur form (BNF) of the USAGE: statement shown in Listing 12-6.

```
function usage
{
echo "\nUSAGE: $SCRIPT [-m] [-n] [password_length]\n"
echo " Where:
    -m Creates a password printout for Security
    -n Loads the default keyboard data keys file
    password_length Integer value that overrides
    the default 8 character
    password length.\n"
}
```

Listing 12-6 usage function

The Backus-Naur form is a standard for expressing the proper order and usage of a statement. First introduced by John Backus in 1959, it was later simplified by Peter Naur, thus its current name. John Backus died in 2007. Study the next usage statement for John, please.

When a usage error is detected, the script executes the usage function that displays the following message:

```
USAGE: mk_passwd.ksh [-m] [-n] [password_length]
Where:
    -m Creates a password printout for Security
    -n Loads the default keyboard data keys file
    password_length Integer value that overrides
        the default 8 character
        password length.\n"
```

trap'exit Function Description

This function, trap_exit, is executed only when an exit signal is *trapped*. You will see how to set a trap a little later. The purpose of this function is to execute any commands that are listed in the function. In our case, we want to remove the *SOUTFILE* before exiting the shell script. Additionally, we do not want to see any messages sent to stderr if the file does not exist. The statement is shown in the following code:

```
function trap_exit
{
rm -f $OUTFILE >/dev/null 2>&1
}
```

Notice that we redirect the stderr output to stdout, which is specified by the file descriptor notation 2>&1, but not before we send everything to the bit bucket, specified by >/dev/null.

That is it for the functions. The next section covers the testing and parsing required for the command arguments.

Testing and Parsing Command-Line Arguments

Because this shell script has command-line options to control execution, we need to test the validity of each command-line argument and then parse through each one to set up how the script is to be executed. We have four tests that need to be performed to validate each argument.

Validating the Number of Command-Line Arguments

The first step is to ensure that the number of command-line arguments is within the limit of arguments we are expecting. For this script we are expecting no more than three arguments. To test the number of arguments, we use the \$# shell variable to test if the number of arguments is greater than 3. This test code is shown here:

Notice that we used the mathematical test, denoted by the double parentheses expression ((expression)). One thing to note about the syntax of this test is that for user- or script-defined variables it is not required to use the dollar sign (\$) in front of the variable name. For shell variables you must use the shell notation here, too. If the number of arguments on the command line exceeds 3, we display the usage function and exit the shell script with a return code of 1.

Test for Valid Command-Line Arguments

We really have only three valid command-line arguments. Because -n and -m are lowercase alphabetic characters, we may as well add their uppercase counterparts for people who love to type uppercase characters. Now we have only five valid command-line arguments:

- Any integer
- -n and -N to indicate creating a new \$KEYBOARD_FILE
- -m and -M to indicate that the manager's password report is to be printed

This seems easy enough to test for using a **case** statement to parse through the command-line arguments using the \$@ values, which is a list of the command-line arguments separated by a single space. Look at the block of code in Listing 12-7 for details.

```
# Test for valid command line arguments -
# Valid arguments are "-n, -N, -m, -M, and any integer
if (( $# != 0 ))
then
   for CMD_ARG in $@
   do
         case $CMD_ARG in
         +([-0-9]))
               # The '+([-0-9]))' test notation is looking for
               # an integer. Any integer is assigned to the
               # length of password variable, LENGTH
               LENGTH=$CMD_ARG
               ;;
                  : # The colon (:) is a no-op, which does nothing
       -n | -N)
               ;;
       -m | -M)
                  : # The colon (:) is a no-op, which does nothing
               ;;
          *)
                    # Invalid command-line argument, show usage and exit
               usage
               exit 1
               ;;
         esac
   done
fi
```

Listing 12-7 Code for testing for command-line arguments

Before we test the validity of each argument, we ensure that there is at least one command-line argument to test. If we have some arguments to test, we start a case

statement to parse through each argument on the command line. As the arguments are parsed, the value is assigned to the CMD_ARG variable.

Notice the very first test, + ([0-9]). This regular expression is testing for an integer value. When we add this integer test to the case statement, we need to add the last close parenthesis,), for the case statement. If the test is true, we know that an integer has been supplied that overrides the default eight-character password length, specified by the LENGTH variable.

NOTE Bash shell does not support the regular expression + ([0-9]) to test for an integer.

The tests for -n, -N, -m, and -M are do-nothings, or no-ops in this case. A no-op is specified by the colon character (:). The no-op does not do anything, but it always has a 0, zero, return code. When our valid command options are found, the case statement goes to the next argument on the command line.

When an invalid command-line option is detected, the function displays the usage message and exits the script with a return code of 1, one, which is defined as a usage error.

Ensuring the \$LENGTH Variable Is an Integer

As a final sanity check of the \$LENGTH variable, I added this extra step to ensure that it is assigned an integer value. This test is similar to the test in the previous section, but it is restricted to testing the LENGTH variable assignment. This test code is shown in Listing 12-8.

Listing 12-8 Testing \$LENGTH for an integer value

If the LENGTH variable does not have an integer assignment, the usage message function is shown and the script exits with a return code of 1, which is defined as a usage error.

Parsing Command-Line Arguments with getopts

The getopts shell function is the best tool for parsing through command-line arguments. With the getopts function we can take direct action or set variables as a valid command-line arguments are found. We can also find invalid command-line arguments, if they are preceded with a minus sign (–).

The getopts function is used with a while loop that contains a case statement. The basic syntax is shown in Listing 12-9.

```
while getopts ":n N V: m M" ARGUMENT 2>/dev/null 2>&1
do
    case $ARGUMENT in
    n N) # Do stuff for -n and -N
          ;;
     m M) # Do stuff for -m and -M
          ;;
      V) # The colon (:) after the V, V:, specifies
          # that -V must have an option attached on the command line.
          ;;
      \?) # The very first colon (:n) specifies that any unknown
          # argument (-A, for example) produces a question mark (?) as
          # output. For these unknown arguments we show the usage
          # message and exit with a return code of 1, one.
          ::
     esac
done
```

Listing 12-9 Basic syntax for using the shell getopts function

As you can see, using getopts to parse command-line arguments is an easy way to catch invalid command-line arguments and also to assign values or tasks to specific arguments. The nice thing about this method is that we do not have to worry about the order of the arguments, or command-line spacing between the command-line argument and the argument to the command-line switch.

Let's look at the code for parsing the command line for this shell script, as shown in Listing 12-10.

Listing 12-10 getopts command-line parsing

```
echo "\nPress ENTER when you are you ready to continue: \c"
read REPLY
clear
;;
m|M)
# Print the Manager Password Report
PRINT_PASSWORD_MANAGER_REPORT=TRUE
;;
\?) # Show the usage message
usage
exit 1
esac
done
```

In our getopts statement, located on the line with the while loop, we provide for four command-line options. These include -n and -N options to execute the load_default_keyboard function. For the -m and -M options the printer variable is set to TRUE. Any other options result in the script exiting with a return code of 1. Notice the $\?$ in the case statement. This specifies that any invalid option is assigned the question mark (?), specifying an unknown option. We do not have any colons after any options, so we are not expecting any values to be assigned to any arguments.

Beginning of Main

Now that we have defined all the variables and functions and verified all the command-line arguments, we are ready to start the main part of the mk_passwd.ksh shell script.

Setting a Trap

The first thing to do is to set a trap. A trap allows us to take action before the shell script or function exits, if an exit signal is trappable and defined. We can *never* trap a kill -9 exit. This kill option does not do anything graceful; it just removes the process from the system process table, and it no longer exists. The more common exit signals are 1, 2, 3, and 15. For a complete list of exit signals, see Chapter 1, "Scripting Quick Start and Review," or enter **kill -l** (that's a lowercase *L*) on the command line.

Our trap is shown here:

```
trap 'trap_exit; exit 2' 1 2 3 15
```

When a trapped exit signal is detected, in this case signals 1, 2, 3, or 15, the trap executes the two commands enclosed within the single tic marks, (' commands '). The commands include running the trap_exit function that removes the \$OUTFILE file; then the script exits with a return code of 2, which has been defined as a trap exit for this shell script.

Checking for the Keyboard File

This shell script is useless without a keyboard data file and cannot execute anything. To check for the existence of the <code>\$KEYBOARD_FILE</code>, we execute the <code>check_for_and_create_keyboard_file</code> function. As we saw previously, this function checks to see if a keyboard data file is on the system. If the file is not found, the user is prompted to automatically load the default keyboard layout, which is a standard 109-key QWERTY keyboard. This functionality allows for a quick start for new users and an easy recovery if the file is deleted. When we want to load a custom keyboard layout, all that is needed is to replace the default keyboard file with a new keyboard layout file.

Loading the KEYS Array

Once we have a \$KEYBOARD_FILE we are ready to load the KEYS array with the keyboard characters. This shell script segment loads the KEYS array with file data. The easiest way to do this is to use a while loop to read each line of the file, which in this case is a single character, while feeding the loop from the bottom, as shown in Listing 12-11.

Listing 12-11 Code to load the KEYS array

In Listing 12-11 we initialize a loop counter, X, to zero. This counter is used to index each array element in sequential order. Next we start the while loop to read each line of data, a single character, and assign the value to the ARRAY_ELEMENT variable on each loop iteration.

Inside of the while loop the counter is incremented as the loop progresses, and the KEYS array is assigned a new array element on each loop iteration until all the file data is loaded into the KEYS array. Notice the command syntax we use to load an array element:

```
KEYS[$X]=$ARRAY_ELEMENT
```

At the bottom of the while loop after done, notice the *input* redirection into the loop. This is one of the fastest ways to parse a file line-by-line. For more information on this and other file-parsing methods, see Chapter 2, "24 Ways to Process a File Line-by-Line." The last task is to define the UPPER_LIMIT variable. This variable is used to create the pseudo-random numbers that are used to point to the KEYS array elements when creating a new pseudo-random password.

Building a New Pseudo-Random Password

The code to build a new password is short and relatively easy to understand. The code is shown in Listing 12-12. After the code listing, we will cover the details.

```
# Create the pseudo-random password in this section
clear
      # Clear the screen
        # Initialize the password to NULL
PW=
# Build the password using random numbers to grab array
# elements from the KEYS array.
X=0
while ((X < LENGTH))
do
    # Increment the password length counter
    ((X = X + 1))
    # Build the password one char at a time in this loop
    PW=${PW}${KEYS[$(in_range_random_number $UPPER_LIMIT)]}
done
# Done building the password
```

Listing 12-12 Building a new pseudo-random password code

We first initialize the password variable (PW) to a null value, specified by PW=. When we make a variable assigned to nothing, the variable is set to NULL. Next we use a while loop to count through the password length, and on each loop iteration create a new random alphanumeric character until the password is built.

Inside the while loop we use a single command to build the password by adding a new pseudo-random character as we go through each loop iteration. Building the password works like this: We start with a NULL variable, PW. Then, on each loop iteration, we assign the PW variable the previous \$PW assignment, which it had from the last loop iteration. Then we add to this current character string a new character, which we generate using the in_range_random_number function inside the KEYS array element assignment using command substitution. The in_range_random_number function expects as input the \$UPPER_LIMIT value, which is 109 keys for the default keyboard layout in this script. Using this method we use the function directly in the KEY array element assignment. This is one method to build a list.

Printing the Manager's Password Report for Safekeeping

This last section of code will create a temporary report file for printing purposes. The only time this section of code is executed is when the -m or -M command-line argument is present. In the getopts command-line parsing section, the PRINT_PASSWORD_ MANAGER_REPORT variable is assigned the value TRUE. Any other value disables the printing option.

This section of code, shown in Listing 12-13, tests the printing variable and if TRUE, executes the build_manager_password_report function. The user is then prompted to print to the default printer, which is listed in the text. The user has a chance to change the printer/queue at this point or to cancel the printing completely. If the \$OUTFILE is printed, the lp command adds the -c switch to make a copy of the file in the spooler. This method allows us to immediately delete the password report file from the system. We just do not want this report file sitting on the system for very long.

```
# Print the Manager's password report, if specified
# on the command with the -m command switch.
if [ $PRINT_PASSWORD_MANAGER_REPORT = TRUE ]
then
  typeset -u REPLY=N
  echo "\nPrint Password Sheet for the Secure Envelope? (Y/N)? \c"
 read REPLY
 if [[ $REPLY = 'Y' ]]
  then
    build_manager_password_report
    REPLY= # Set REPLY to NULL
    echo "\nPrint to the Default Printer ${DEFAULT_PRINTER} (Y/N)? \c"
    read REPLY
    if [[ $REPLY = 'Y' ]]
     then
         echo "\nPrinting to $DEFAULT_PRINTER\n"
          lp -c -d $DEFAULT_PRINTER $OUTFILE
     else
         echo "\nNEW PRINT QUEUE: \c"
         read DEFAULT_PRINTER
          echo "\nPrinting to $DEFAULT_PRINTER\n"
         lp -c -d $DEFAULT_PRINTER $OUTFILE
     fi
  else
     echo "\n\n\tO.K. - Printing Skipped..."
```

Listing 12-13 Code to create and print the password report

The last two things that are done at the end of this shell script are to remove the \$OUTFILE, if it exists, and then prompt the user to press Enter to clear the screen and exit. We do not want to leave a password on the screen for anyone to read.

That is it for the steps involved to create the mk_passwd.ksh shell script. The entire shell script is shown in Listing 12-14. Pay particular attention to the boldface text throughout the mk_passwd.ksh shell script.

```
#!/bin/ksh
#
# AUTHOR: Randy Michael
# SCRIPT: mk_passwd.ksh
# DATE: 11/12/2007
# REV: 2.5.P
# PLATFORM: Not Platform Dependent
#
# PURPOSE: This script is used to create pseudo-random passwords.
#
          An external keyboard data file is utilized, which is
#
          defined by the KEYBOARD_FILE variable. This keyboard
#
          file is expected to have one character on each line.
          These characters are loaded into an array, and using
#
#
          pseudo-random numbers generated, the characters are
#
          "randomly" put together to form a string of characters.
#
          By default, this script produces 8 character passwords,
#
          but this length can be changed on the command line by
#
          adding an integer value after the script name. There are
#
          two command line options, -n, which creates the default
#
          KEYBOARD_FILE, and -m, which prints the manager's
          password report. This password report is intended
          to be locked in a safe for safe keeping.
# EXIT CODES:
#
                0 - Normal script execution
#
                1 - Usage error
```

```
Listing 12-14 mk_passwd.ksh shell script
```

```
2 - Trap exit
            3 - Missing Keyboard data file
#
#
            4 - $DEFAULT_PRINTER is NULL
#
# REV LIST:
#
        6/26/2007: Added two command line options, -n, which
        creates a new $KEYBOARD_FILE, and -m, which prints
#
#
        the manager's password report.
#
        8/8/2007: Changed the random number method from the
#
        shell RANDOM variable to now use the /dev/urandom
        character special file in concert with dd and od.
# set -x # Uncomment to debug
# set -n # Uncomment to check syntax without any command execution
****
*****
LENGTH=8 # Default Password Length
# Notification List for Printing the Manager's
# Password Report for Locking Away Passwords
# Just in Case You are Unavailable.
NOTIFICATION_LIST="Donald Duck, Yogi Bear, and Mr. Ranger"
# Define the Default Printer for Printing the Manager's
# Password Report. The user has a chance to change this
# printer at execution time.
DEFAULT_PRINTER="hp4@yogi"
SCRIPT=$(basename $0)
OUTFILE="/tmp/tmppdw.file"
KEYBOARD_FILE=/scripts/keyboard.keys
PRINT_PASSWORD_MANAGER_REPORT="TO_BE_SET"
# This next test ensures we use the correct
# echo command based on the executing shell
ECHO="echo -e"
[[ $(basename $SHELL) = ksh ]] && ECHO=echo
#
```

Listing 12-14 (continued)

```
function in_range_random_number
{
# Create a pseudo-random number less than or equal
# to the $UPPER_LIMIT value, which is defined in the
# main body of the shell script.
RN=$(dd if=/dev/urandom count=1 2>/dev/null \
    | od -t u2 | awk '{print $2}'| head -n 1)
RANDOM_NUMBER=$(( RN % UPPER_LIMIT + 1))
echo "$RANDOM_NUMBER"
3
******
function load_default_keyboard
# If a keyboard data file does not exist then the user is
# prompted to load the standard keyboard data into the
# $KEYBOARD_FILE, which is defined in the main body of
# the shell script.
clear # Clear the screen
$ECHO "\nLoad the default keyboard data file? (Y/N): \c"
read REPLY
case $REPLY in
y Y) :
    ;;
 *) $ECHO "\nSkipping the load of the default keyboard file...\n"
    return
    ;;
esac
cat /dev/null > $KEYBOARD FILE
$ECHO "\nLoading the Standard Keyboard File...\c"
# Loop through each character in the following list and
# append each character to the $KEYBOARD_FILE file. This
# produces a file with one character on each line.
for CHAR in \' 1 2 3 4 5 6 7 8 9 0 - = \\ q w e r t y u i o \
            p \[ \] asdfghjkl \; \'zxcvbnm \, \
            \. \/ \\ \~ \! \@ \# \$ \% \^ \& \* \( \) _ \+ \| \
            QWERTYUIOP \{ \} ASDFGHJKL \: \" \
```

Listing 12-14 (continued)

```
Z X C V B N M \< \> \? \| \. 0 1 2 3 4 5 6 7 8 9 \/ \
            \* \- \+
do
    $ECHO "$CHAR" >> $KEYBOARD_FILE
done
$ECHO "\n\t...Done...\n"
sleep 1
}
#
*****
function check_for_and_create_keyboard_file
{
# If the $KEYBOARD_FILE does not exist then
# ask the user to load the "standard" keyboard
# layout, which is done with the load_default_keyboard
# function.
if [ ! -s $KEYBOARD_FILE ]
then
    $ECHO "\n\nERROR: Missing Keyboard File"
    $ECHO "\n\nWould You Like to Load the"
    $ECHO "Default Keyboard Layout?"
    $ECHO "\n\t(Y/N): \c"
    typeset -u REPLY=FALSE
    read REPLY
    if [ $REPLY != Y ]
    then
        $ECHO "\n\nERROR: This shell script cannot operate"
        $ECHO "without a keyboard data file located in"
        $ECHO "\n==> $KEYBOARD_FILE\n"
        $ECHO "\nThis file expects one character per line."
        $ECHO "\n\t...EXITING...\n"
        exit 3
    else
        load_default_keyboard
        $ECHO "\nPress ENTER when you are ready to continue: \c"
        read REPLY
        clear
    fi
fi
}
#
****
function build_manager_password_report
{
```

```
# Build a file to print for the secure envelope
(
$ECHO "\n
                       RESTRICTED USE!!!"
$ECHO "\n\n\tImmediately send an e-mail to:\n"
$ECHO "
         $NOTIFICATION_LIST"
$ECHO "\n\tif this password is revealed!"
$ECHO "\n\tAIX root password: $PW\n"
$ECHO "\n\n"
$ECHO "\n
                       RESTRICTED USE!!!"
$ECHO "\n\n\tImmediately send an e-mail to:\n"
$ECHO " $NOTIFICATION_LIST"
$ECHO "\n\tif this password is revealed!"
$ECHO "\n\tAIX root password: $PW\n"
$ECHO "\n\n"
$ECHO "\n
                      RESTRICTED USE!!!"
$ECHO "\n\n\tImmediately send an e-mail to:\n"
$ECHO " $NOTIFICATION_LIST"
$ECHO "\n\tif this password is revealed!"
$ECHO "\n\tAIX root password: $PW\n"
   ) > $OUTFILE
}
*****
function usage
{
$ECHO "\nUSAGE: $SCRIPT [-m] [-n] [password_length]\n"
$ECHO " Where:
    -m Creates a password printout for Security
    -n Loads the default keyboard data keys file
    password_length - Integer value that overrides
                     the default 8 character
                    password length.\n"
```

```
}
#
*****
function trap_exit
{
rm -f $OUTFILE >/dev/null 2>&1
}
*****
*****
####### VALIDATE EACH COMMAND LINE ARGUMENT ########
*****
# Check command line arguments - $# < 3</pre>
if (($# > 3))
then
  usage
  exit 1
fi
*****
#
# Test for valid command line arguments -
# Valid arguments are "-n, -N, -m, -M, and any integer
if (($# != 0))
then
 for CMD_ARG in $@
 do
     case $CMD_ARG in
     +([-0-9]))
         # The '+([-0-9]))' test notation is looking for
         # an integer. Any integer is assigned to the
         # length of password variable, LENGTH
         LENGTH=$CMD_ARG
         ;;
     -n) :
         ;;
     -N) :
         ;;
     -m) :
         ;;
```

```
-M) :
             ;;
        *)
             usage
             exit 1
              ;;
        esac
   done
fi
****
#
# Ensure that the $LENGTH variable is an integer
case $LENGTH in
+([0-9])) : # The '+([-0]))' test notation is looking for
           # an integer. If it is an integer then the
           # No-Op, specified by a colon, (Do Nothing)
           # command is executed, otherwise this script
           # exits with a return code of 1, one.
         ;;
*) usage
  exit 1
   ;;
esac
*****
#
# Use the getopts function to parse the command
# line arguments.
while getopts ":nNmM" ARGUMENT 2>/dev/null
do
    case $ARGUMENT in
    n N)
       # Create a new Keyboard Data file
       load_default_keyboard
       $ECHO "\nPress ENTER when you are ready to continue: \c"
       read REPLY
       clear
       ;;
    m | M)
       # Print the Manager Password Report
       PRINT_PASSWORD_MANAGER_REPORT=TRUE
       ;;
    \?)
       # Show the usage message
       usage
```

Listing 12-14 (continued)

```
exit 1
   esac
done
*****
****
# Set a trap
trap 'trap_exit; exit 2' 1 2 3 15
****
# Check for a keyboard data file
check_for_and_create_keyboard_file
*****
X=0 # Initialize the array counter to zero
# Load the array called "KEYS" with keyboard elements
# located in the $KEYBOARD_FILE.
while read ARRAY_ELEMENT
do
   ((X = X + 1)) # Increment the counter by 1
   # Load an array element in the array
   KEYS [$X] =$ARRAY_ELEMENT
done < $KEYBOARD_FILE
UPPER LIMIT=$X # Random Number Upper Limit
*****
# Create the pseudo-random password in this section
    # Clear the screen
clear
```

```
PW=
       # Initialize the password to NULL
# Build the password using random numbers to grab array
# elements from the KEYS array.
X=0
while ((X < LENGTH))
do
   # Increment the password length counter
   ((X = X + 1))
   # Build the password one char at a time in this loop
   PW=${PW}${KEYS[$(in_range_random_number $UPPER_LIMIT)]}
done
# Done building the password
# Display the new pseudo-random password to the screen
$ECHO "\n\n
              The new $LENGTH character password is:\n"
$ECHO "\n
                 ${PW}\n"
*****
# Print the Manager's password report, if specified
# on the command with the -m command switch.
if [ $PRINT_PASSWORD_MANAGER_REPORT = TRUE ]
then
 typeset -u REPLY=N
 $ECHO "\nPrint Password Sheet for the Secure Envelope? (Y/N)? \c"
 read REPLY
 if [[ $REPLY = 'Y' ]]
 then
    build_manager_password_report
    REPLY= # Set REPLY to NULL
    $ECHO "\nSend to Default Printer ${DEFAULT_PRINTER} (Y/N)? \c"
    read REPLY
    if [[ $REPLY = 'Y' ]]
    then
         $ECHO "\nPrinting to $DEFAULT_PRINTER\n"
```

Listing 12-14 (continued)

```
lp -c -d $DEFAULT_PRINTER $OUTFILE
    else
        $ECHO "\nNEW PRINT QUEUE: \c"
        read DEFAULT_PRINTER
        if [ -z "$DEFAULT_PRINTER" ]
        then
            echo "ERROR: Default printer cannot be NULL...Exiting...."
            exit 5
        fi
        $ECHO "\nPrinting to $DEFAULT_PRINTER\n"
        lp -c -d $DEFAULT_PRINTER $OUTFILE
    fi
 else
    $ECHO "\n\tO.K. - Printing Skipped..."
 fi
fi
**********
#
# Remove the $OUTFILE, if it exists and has a size
# greater than zero bytes.
[ -s $OUTFILE ] && rm $OUTFILE
# Clear the screen and exit
$ECHO "\n\nPress ENTER to Clear the Screen and EXIT: \c"
read X
clear
# End of mk_passwd.ksh shell script
```

This was an interesting shell script to create. I hope you picked up some pointers in this chapter. I tried to add a lot of script options but not make the script too difficult to understand.

Other Options to Consider

As with any script, improvements can be made. Remember the proper usage of the echo command. For Bourne and Bash shells, add the -e switch when using the backslash operators, n, and so on (for example, echo -e nHello Worldn"), but not Korn shell, where the -e is not needed to enable these operators.

Password Reports?

Do you need to create password reports for your manager and directors? If not, you should disable the ability to create any file that contains any password and disable printing any passwords. This is easy to disable by commenting out the getopts parsing for the -m and -M command-line options.

Which Password?

You certainly do not have to accept the first password that is produced by this script. It usually takes me 5 to 10 tries to get a password that I may be able to remember. Don't stop at the first one — keep going until you get a password that you like but is not guessable.

Other Uses?

Sure, there are other uses for this shell script. Any time that you need a pseudo-random list of keyboard characters, you can use this shell script to create the list. License and encryption keys come to mind. If you are selling software and you need to create some unguessable keys, run the script and specify the length of the key as an integer value.

Summary

This was an excellent exercise in creating pseudo-random numbers and using a function directly in a command assignment. We used arrays to store our keyboard data so that any element is directly accessible. This chapter goes a long way in making any task intuitively obvious to solve. We love a good challenge.

In the next chapter we are going to look at floating-point math in shell scripts and the bc utility. I hope you stick around for some more tips and tricks.

Lab Assignments

- 1. Time the execution of the following three shell scripts, and explain any difference you observe:
 - a. Write a shell script to create a 32-character license key using the /dev/random character special file.
 - b. Write a shell script to create a 32-character license key using the /dev/urandom character special file.
 - c. Write a shell script to create a 32-character license key using the /dev/random character special file to seed the RANDOM shell variable, and then create the encryption key using the RANDOM shell variable,
- 2. Create a 5,000-character password. How long did that task take?

CHAPTER 13

Floating-Point Math and the bc Utility

Have you ever needed to do some floating-point math in a shell script? If the answer is yes, then you're in luck. On UNIX machines there is an interactive program called **bc** that is an interpreter for the arbitrary-precision arithmetic language. You can start an interactive session by typing bc on the command line. Once in the session you can enter most complex arithmetic expressions as you would in a calculator. The bc utility can handle more than I can cover in this chapter, so we are going to keep the scope limited to simple floating-point math in shell scripts.

In this chapter we are going to create shell scripts that add, subtract, multiply, divide, and average a list of numbers. With each of these shell scripts the user has the option of specifying a *scale*, which is the number of significant digits to the right of the decimal point. If no scale is specified, an integer value is given in the result. Because the bc utility is an interactive program, we are going to use a *here document* to supply input to the interactive bc program. We cover using a here document in detail throughout this chapter. Of course, we can write an expect script to handle the automation. Sounds like a good Lab Assignment.

Syntax

By now you know the routine: We need to know the syntax before we can create a shell script. Depending on what we are doing we need to create a mathematical statement to present to bc for a here document to work. A here document works kind of like a *label* in other programming languages. The syntax that we are going to use in this chapter will have the following form:

```
VARIABLE=$(bc <<LABEL
scale=$SCALE
$(MATH_STATEMENT)
LABEL)</pre>
```

The way a here document works is some label name, in this case LABEL, is added just after the bc command. This LABEL has double redirection for input into the interactive program, bc <<LABEL. From this starting label until the same label is encountered again, everything in between is used as input to the bc program. By doing this we are automating an interactive program. We can also do this automation using another technique. We can use echo, print, and printf to print all of the data for the math statement and pipe the output to bc. It works like the following commands:

```
VARIABLE=$(print 'scale = 10; 104348/33215' | bc)
```

or

```
VARIABLE=$(print 'scale=$SCALE; ($MATH_STATEMENT)' | bc)
```

In either case we are automating an interactive program. This is the purpose of a here document. It is called a here document because the required input is *here*, as opposed to somewhere else, such as user input from the keyboard. When all of the required input is supplied *here*, it is a here document.

Creating Some Shell Scripts Using bc

We have the basic syntax, so let's start with a simple shell script to add numbers together. The script is expecting a list of numbers as command-line arguments. Additionally, the user may specify a *scale* if the user wants the result calculated as a floating-point number to a set precision. If a floating-point number is *not* specified, the result is presented as an integer value.

Creating the float_add.ksh Shell Script

The first shell script that we are going to create is float_add.ksh. The idea of this shell script is to add together a list of numbers that the user provides as command-line arguments. The user also has the option of setting a scale for the precision of floating-point numbers. Let's take a look at the float_add.ksh shell script in Listing 13-1, and we will go through the details at the end.

```
#!/usr/bin/ksh
#
# SCRIPT: float_add.ksh
# AUTHOR: Randy Michael
# DATE: 03/01/2007
# REV: 1.1.A
#
#
# PURPOSE: This shell script is used to add a list of numbers
# together. The numbers can be either integers or floating-
# point numbers. For floating-point numbers the user has
```

Listing 13-1 float_add.ksh shell script

```
the option of specifying a scale of the number of digits to
        the right of the decimal point. The scale is set by adding
#
#
        a -s or -S followed by an integer number.
# EXIT CODES:
#
     0 ==> This script completed without error
     1 ==> Usage error
#
      2 ==> This script exited on a trapped signal
#
# REV. LIST:
# set -x # Uncomment to debug this script
# set -n # Uncomment to debug without any command execution
**********
****
SCRIPT_NAME=$(basename $0) # The name of this shell script
SCALE="0"
           # Initialize the scale value to zero
           # Initialize the NUM_LIST variable to NULL
NUM_LIST=
COUNT=0
           # Initialize the counter to zero
           # Set MAX_COUNT to the total number of
MAX_COUNT=$#
           # command-line arguments.
****
*****
function usage
{
echo "\nPURPOSE: Adds a list of numbers together\n"
echo "USAGE: $SCRIPT_NAME [-s scale_value] N1 N2...Nn"
echo "\nFor an integer result without any significant decimal places..."
echo "\nEXAMPLE: $SCRIPT_NAME 2048.221 65536 \n"
echo "OR for 4 significant decimal places"
echo "\nEXAMPLE: $SCRIPT_NAME -s 4 8.09838 2048 65536 42.632"
echo "\n\t...EXITING...\n"
}
*****
function exit trap
echo "\n...EXITING on trapped signal...\n"
}
```

```
**********
*****
###### Set a Trap ######
trap 'exit_trap; exit 2' 1 2 3 15
*****
# Check for at least two command-line arguments
if [ $# -1t 2 ]
then
    echo "\nERROR: Please provide a list of numbers to add"
   usage
    exit 1
fi
# Parse the command-line arguments to find the scale value, if present.
while getopts ":s:S:" ARGUMENT
do
   case $ARGUMENT in
       s S) SCALE=$OPTARG
           ;;
        \?) # Because we may have negative numbers we need
           # to test to see if the ARGUMENT that begins with a
           # hyphen (-) is a number, and not an invalid switch !!!
           for TST_ARG in $*
           do
             if [[ $(echo $TST_ARG | cut -c1) = '-' ]] \
                && [ $TST_ARG != '-s' -a $TST_ARG != '-S' ]
              then
                case $TST_ARG in
                +([-0-9])) : # No-op, do nothing
                    ;;
                +([-0-9].[0-9]))
                   : # No-op, do nothing
                    ;;
                +([-.0-9])) : # No-op, do nothing
                    ;;
                *) echo "\nERROR: Invalid argument \
on the command line"
                  usage
                  exit 1
                  ;;
```

Listing 13-1 (continued)

```
esac
               fi
           done
            ;;
   esac
done
*****
# Parse through the command-line arguments and gather a list
# of numbers to add together and test each value.
while ((COUNT < MAX_COUNT))
do
    ((COUNT = COUNT + 1))
    TOKEN=$1
             # Grab a command-line argument on each loop iteration
    case $TOKEN in
                 # Test each value and look for a scale value.
            -s|-S) shift 2
                  ((COUNT = COUNT + 1))
                  ;;
        -s${SCALE}) shift
                  ;;
        -S${SCALE}) shift
                  : :
                *) # Add the number ($TOKEN) to the list
                  NUM_LIST="${NUM_LIST} $TOKEN"
                  ((COUNT < MAX_COUNT)) && shift
                  ;;
    esac
done
******
# Ensure that the scale is an integer value
case $SCALE in
               # Test for an integer
    +([0-9])) : # No-Op - Do Nothing
            ;;
          *) echo "\nERROR: Invalid scale - $SCALE - \
Must be an integer"
            usage
            exit 1
            ;;
esac
*****
```

Listing 13-1 (continued)

```
# Check each number supplied to ensure that the "numbers"
# are either integers or floating-point numbers.
for NUM in $NUM_LIST
do
    case $NUM in
     +([0-9])) # Check for an integer
               : # No-op, do nothing.
               ;;
     +([-0-9])) # Check for a negative whole number
               : # No-op, do nothing
               ;;
     +([0-9] | [.] [0-9]))
               # Check for a positive floating-point number
               : # No-op, do nothing
               ;;
     +(+[0-9][.][0-9]))
               # Check for a positive floating-point number
               # with a + prefix
               : # No-op, do nothing
               ;;
     +(-[0-9][.][0-9]))
               # Check for a negative floating-point number
               : # No-op, do nothing
               ;;
     +([-.0-9]))
               # Check for a negative floating-point number
               : # No-op, do nothing
               ;;
     +([+.0-9]))
               # Check for a positive floating-point number
               : # No-op, do nothing
               : :
     *) echo "\nERROR: $NUM is NOT a valid number"
       usage
        exit 1
        : :
     esac
done
**********
# Build the list of numbers to add
ADD=
        # Initialize the ADD variable to NULL
        # Initialize the PLUS variable to NULL
PLUS=
# Loop through each number and build a math statement that
# will add all of the numbers together.
```

Listing 13-1 (continued)

```
for X in $NUM_LIST
do
     # If the number has a + prefix, remove it!
     if [[ $(echo $X | cut -c1) = '+' ]]
     then
         X=$(echo $X | cut -c2-)
     fi
     ADD="$ADD $PLUS $X"
     PLUS="+"
done
********
# Do the math here by using a here document to supply
# input to the bc command. The sum of the numbers is
# assigned to the SUM variable.
SUM=$(bc <<EOF
scale = $SCALE
(${ADD})
EOF)
****************
# Present the result of the addition to the user.
echo "\nThe sum of: $ADD"
echo "\nis: ${SUM}\n"
```

Listing 13-1 (continued)

Let's take it from the top. We start the shell script in Listing 13-1 by defining some variables. These five variables, SCRIPT_NAME, SCALE, NUM_LIST, COUNT, and MAX_COUNT, are predefined for later use. The SCRIPT_NAME variable assignment extracts the filename of the script from the system using the basename \$0 command, and SCALE is used to define the precision of floating-point numbers that are calculated. The NUM_LIST variable is used to hold valid numbers that are to be calculated, where the command switch and the switch-argument are removed from the list. The COUNT and MAX_COUNT variables are used to scan all the command-line arguments to find the numbers.

In the next section we define the functions. This shell script has two functions, usage and exit_trap. The usage function shows the user how to use the script, and the exit_trap function is executed only when a trapped exit signal is captured. Of course, you cannot trap a kill -9. At the START OF MAIN the first thing that we do is to set a trap. A trap allows us to take some action when the trapped signal is captured. For example, if the user presses Ctrl+C, we may want to clean up some temporary files before the script exits. A trap allows us to do this.

A trap has the form of trap 'command; command; ...; exit 2' 1 2 3 15. We first enclose the commands that we want to execute within tic marks (single

quotes) and then give a list of exit signals that we want to capture. As I said before, it is not possible to capture a kill -9 signal because the system really just yanks the process out of the process table and it ceases to exist.

After setting the trap we move on to verifying that each of the command-line arguments is valid. To do this verification we do five tests. These five tests consist of checking for at least two command-line arguments, using getopts to parse the command-line switches, test for invalid switches, and assign switch-arguments to variables for use in the shell script. The next step is to scan each argument on the command line and extract the numbers that we need to do our calculations. Then the \$SCALE value is checked to ensure that it points to an integer value, and the final test is to check the "numbers" that we gathered from the command-line scan and ensure that each one is either an integer or a floating-point number.

Testing for Integers and Floating-Point Numbers

I want to go over the integer and floating-point test before we move on. At this point in the script we have a list of "numbers" — at least they are supposed to be numbers — and this list is assigned to the NUM_LIST variable. Our job is to verify that each value in the list is either an integer or a floating-pointing number. Look at the code segment shown in Listing 13-2.

```
# Check each number supplied to ensure that the "numbers"
# are either integers or floating-point numbers.
for NUM in $NUM LIST
do
    case $NUM in
    +([0-9])) # Check for an integer
               : # No-op, do nothing.
              ;;
     +(-[0-9])) # Check for a negative whole number
               : # No-op, do nothing
               ;;
     +([0-9] [.][0-9]))
               # Check for a positive floating-point number
               : # No-op, do nothing
               ;;
     +(+[0-9]|[.][0-9]))
               # Check for a positive floating-point number
               # with a + prefix
               : # No-op, do nothing
               ;;
     +(-[0-9][.][0-9]))
               # Check for a negative floating-point number
               : # No-op, do nothing
               ;;
```

Listing 13-2 Testing for integers and floating-point numbers

```
+([-.0-9]))
    # Check for a negative floating-point number
    : # No-op, do nothing
    ;;
+([+.0-9]))
    # Check for a positive floating-point number
    : # No-op, do nothing
    ;;
*) echo "\nERROR: $NUM is NOT a valid number"
    usage
    exit 1
    ;;
    esac
done
```

Listing 13-2 (continued)

We use a for loop to test each value in the NUM_LIST. On each loop iteration the current value in the \$NUM_LIST is assigned to the NUM variable. Within the for loop we have set up a case statement. For the tests we use regular expressions to indicate a range, or type of value, that we are expecting. If the value does not meet the criteria that we defined, the * is matched and we execute the usage function before exiting the shell script.

The Korn shell regular expressions for testing for integers and floating-point numbers include +([0-9]), +(-[0-9]), +([0-9]], +(-[0-9]), +(-[0-9]), +(-[0-9]), +(-[0-9]), +(-[0-9]), +(-[0-9]), +(-[0-9]), and +([+,0-9])). The first two tests are for integers and negative whole numbers. The last five tests are for positive and negative floating-point numbers. Notice the use of the plus sign (+), minus sign (-), and the decimal point (.). The placement of the plus sign, minus sign, and the decimal point are important when testing the string. Because a floating-point number, both positive and negative, can be represented in many forms, we need to test for all combinations. Floating-point numbers are one of the more difficult tests to make, as you can see by the number of tests that are required.

NOTE The Bash and Bourne shells do not support regular expressions using the +([0-9]) type notations as shown in the above Korn shell examples.

Building a Math Statement for the bc Command

Once we are sure that all the data is valid, we proceed to building the actual math statement that we are sending to the bc utility. To build this statement, we are going to loop through our newly confirmed \$NUM_LIST of numbers and build a string with a plus sign (+) between each of the numbers in the \$NUM_LIST. This is a neat trick. We first initialize two variables to NULL, as shown here:

```
ADD=
PLUS=
```

As we build the math statement, the ADD variable will hold the entire statement as it is added to. The PLUS variable will be assigned the + character inside the for loop on the first loop iteration. This action prevents the + sign showing up as the first character in the string we are building. Let's look at this code segment here:

On the first loop iteration only the first number in the \$NUM_LIST is assigned to the ADD variable. On each of the following loop iterations a plus sign (+) is added, followed by the next number in the \$NUM_LIST, specified by the X variable on each loop iteration, until all of the numbers and plus signs have been assigned to the ADD variable. As an example, we have the following list of numbers:

12 453.766 223.6 3.145927 22

Also notice that we added a test for the number beginning with a + sign. We need to strip out this character so that we do not have two plus signs together when we present the equation to the bc program, or an error will occur. As we build the math statement the following assignments are made to the ADD variable on each loop iteration:

```
ADD="12"
ADD="12 + 453.766"
ADD="12 + 453.766 + 223.6"
ADD="12 + 453.766 + 223.6 + 3.145927"
ADD="12 + 453.766 + 223.6 + 3.145927 + 22"
```

Using a Here Document

When the looping finishes we have built the entire math statement and have it assigned to the ADD variable. Now we are ready to create the here document to add all of the numbers together with the bc utility. Take a look at the here document shown here:

Do the math here by using a here document to supply # input to the bc command. The sum of the numbers is # assigned to the SUM variable. SUM=\$(bc <<EOF</pre>

```
scale=$SCALE
(${ADD})
EOF)
```

For this here document the label is the EOF character string (you will see this used a lot in shell scripts). The bc command has its input between the first EOF and the ending EOF. The first EOF label starts the here document, and the second EOF label ends the here document. Each line between the two labels is used as input to the bc command. There are a couple of requirements for a here document. The first requirement is that the starting label *must* be preceded by double input redirection (<<EOF). The second requirement is that there are *never* any blank spaces at the beginning of *any* line in the here document. If even one blank space is placed in column one, strange things may begin to happen. Depending on what you are doing, and the interactive program you are using, the here document may work, but it may not! This is one of the most difficult programming errors to find when you are testing, or using, a shell script with a here document. To be safe, just leave out any beginning spaces.

The final step is to display the result to the user. Listing 13-3 shows the float_add.ksh shell script in action.

```
[root:yogi]@/scripts# ./float_add.ksh -s 8 2 223.545 332.009976553
The sum of: 2 + 223.545 + 332.009976553
to a scale of 8 is 557.554976553
```

Listing 13-3 float_add.ksh shell script in action

Notice that the scale is set to 8, but the output has 9 decimal places because we used 9 significant digits to the right of the decimal point with 332.009976553. The 9 decimal places take precedence over the scale value. However, if we had not specified a scale value our answer would be an integer instead of a floating-point number. This is just how the bc program works. It is not an error to add in a scale, but the result does not use it in this case. The man page for the bc program can provide you with more details on this effect. We will see how the scale works in some of the other shell scripts later in this chapter.

That is it for the addition shell script, but we still have four more shell scripts to go in this chapter. Each of the following shell scripts is very similar to the script in Listing 13-1. With this being the case I am going to cover different aspects of each of the following scripts and also show where the differences lie. Please keep reading to catch a few more shell programming tips.

Creating the float_subtract.ksh Shell Script

As the float_add.ksh shell script performed addition on a series of numbers, this section studies the technique of subtraction. Because this shell script is very similar to the shell script in Listing 13-1, we are going to show the shell script and study the details at the end. The float_subtract.ksh shell script is shown in Listing 13-4.

```
#!/usr/bin/ksh
# SCRIPT: float_subtract.ksh
# AUTHOR: Randy Michael
# DATE: 02/23/2007
# REV: 1.1.A
# PURPOSE: This shell script is used to subtract a list of numbers.
        The numbers can be either integers or floating-point
#
        numbers. For floating-point numbers the user has the
#
        option to specify a scale of the number of digits to
        the right of the decimal point. The scale is set by
        adding a -s or -S followed by an integer number.
# EXIT STATUS:
      0 ==> This script completed without error
#
      1 ==> Usage error
      2 ==> This script exited on a trapped signal
#
# REV. LIST:
# set -x # Uncomment to debug this script
# set -n # Uncomment to debug without any command execution
*****
*****
SCRIPT_NAME=`basename $0` # The name of this shell script
            # Initialize the scale value to zero
SCALE="0"
            # Initialize the NUM_LIST to NULL
NUM_LIST=
COUNT=0
            # Initialize the counter to zero
MAX COUNT=$# # Set MAX COUNT to the total number of
            # command-line arguments
*****
*********
function usage
{
echo "\nPURPOSE: Subtracts a list of numbers\n"
echo "USAGE: $SCRIPT NAME [-s scale value] N1 N2...Nn"
echo "\nFor an integer result without any significant decimal places..."
echo "\nEXAMPLE: $SCRIPT_NAME 2048.221 65536 \n"
echo "OR for 4 significant decimal places"
echo "\nEXAMPLE: $SCRIPT_NAME -s 4 8.09838 2048 65536 42.632"
```

```
Listing 13-4 float_subtract.ksh shell script
```

```
echo "\n\t...EXITING...\n"
}
****
function exit_trap
{
      echo "\n...EXITING on trapped signal...\n"
}
****
*****
###### Set a Trap ######
trap 'exit_trap; exit 2' 1 2 3 15
# Check for at least two command-line arguments
if (($# < 2))
then
      echo "\nERROR: Please provide a list of numbers to subtract"
      usage
      exit 1
fi
# Parse the command-line arguments to find the scale value, if present.
while getopts ":s:S:" ARGUMENT
do
   case $ARGUMENT in
       s S) SCALE=$OPTARG
           ::
        \?) # Because we may have negative numbers we need
           # to test to see if the ARGUMENT that begins with a
           # hyphen (-) is a number, and not an invalid switch!!!
           for TST_ARG in $*
           do
             if [[ $(echo $TST_ARG | cut -c1) = '-' ]] \
               && [ $TST_ARG != '-s' -a $TST_ARG != '-S' ]
             then
               case $TST_ARG in
               +([-0-9])) : # No-op, do nothing
                        ;;
               +([-0-9].[0-9]))
```

Listing 13-4 (continued)

```
: # No-op, do nothing
                            ;;
                 +([-.0-9])) : # No-op, do nothing
                            ;;
                *) echo "\nERROR: Invalid argument on the command line"
                    usage
                    exit 1
                    ;;
                  esac
                fi
            done
            ;;
   esac
done
*****
# Parse through the command-line arguments and gather a list
# of numbers to subtract.
while ((COUNT < MAX_COUNT))
do
    ((COUNT = COUNT + 1))
    TOKEN=$1
    case $TOKEN in
             -s|-S) shift 2
                   ((COUNT = COUNT + 1))
                   ;;
         -s${SCALE}) shift
                   ;;
         -S${SCALE}) shift
                   ;;
                 *) NUM_LIST="${NUM_LIST} $TOKEN"
                   ((COUNT < MAX_COUNT)) && shift
                   ;;
    esac
done
*****
# Ensure that the scale is an integer value
case $SCALE in
    +([0-9])) : # No-Op - Do Nothing
             ;;
         *) echo "\nERROR: Invalid scale - $SCALE - Must be an integer"
             usage
             exit 1
```

Listing 13-4 (continued)

```
;;
esac
******
# Check each number supplied to ensure that the "numbers"
# are either integers or floating-point numbers.
for NUM in $NUM_LIST
do
    case $NUM in
    +([0-9])) # Check for an integer
             : # No-op, do nothing.
             ;;
    +([-0-9])) # Check for a negative whole number
             : # No-op, do nothing
             ;;
    +([0-9] | [.] [0-9]))
             # Check for a positive floating-point number
              : # No-op, do nothing
             ;;
    +(+[0-9] [.][0-9]))
             # Check for a positive floating-point number
             # with a + prefix
              : # No-op, do nothing
             ;;
    +([-0-9]|.[0-9]))
             # Check for a negative floating-point number
              : # No-op, do nothing
             ;;
    +(-[.][0-9]))
             # Check for a negative floating-point number
              : # No-op, do nothing
             ;;
    +([+.0-9]))
             # Check for a positive floating-point number
             : # No-op, do nothing
             ;;
    *) echo "\nERROR: $NUM is NOT a valid number"
       usage
       exit 1
       ;;
    esac
done
*****
```

Listing 13-4 (continued)

```
# Build the list of numbers to subtract
SUBTRACT=
           # Initialize the SUBTRACT variable to NULL
MINUS=
           # Initialize the MINUS variable to NULL
# Loop through each number and build a math statement that
# will subtract the numbers in the list.
for X in $NUM_LIST
do
     # If the number has a + prefix, remove it!
     if [[ $(echo $X | cut -c1) = '+' ]]
     then
         X=$(echo $X | cut -c2-)
     fi
     SUBTRACT="$SUBTRACT $MINUS $X"
     MINUS='-'
done
**********
# Do the math here by using a here document to supply
# input to the bc command. The difference of the numbers is
# assigned to the DIFFERENCE variable.
DIFFERENCE=$(bc <<EOF
scale=$SCALE
(${SUBTRACT})
EOF)
*****
# Present the result of the subtraction to the user.
echo "\nThe difference of: $SUBTRACT"
echo "\nis: ${DIFFERENCE}\n"
```

Listing 13-4 (continued)

The parts of the float_subtract.ksh shell script, shown in Listing 13-4, that remain unchanged from Listing 13-1 include the following sections: variable definitions and the usage function, which is unchanged except that the references to addition are changed to subtraction. Additionally, all the same tests are performed on the user-provided data to ensure the data integrity. When we get to the end of the shell script where the math statement is created and the here document performs the calculation, we get into some changes.

Using getopts to Parse the Command Line

Let's first cover parsing the command line for the -s and -S switches and these switch-arguments that we use to define the floating-point precision with the getopts command. Using getopts for command-line parsing is the simplest method. It sure beats trying to program all of the possibilities inside the shell script. The first thing to note about getopts is that this command does not care *what* is on the command line! getopts is interested in only command switches, which must begin with a hyphen (-), such as -s and -S for this shell script. Let's look at the getopts code segment and see how it works:

```
# Parse the command-line arguments to find the scale value, if present.
while getopts ":s:S:" ARGUMENT
do
    case $ARGUMENT in
         s S) SCALE=$OPTARG
              ;;
          \?) # Because we may have negative numbers we need
              # to test to see if the ARGUMENT that begins with a
              # hyphen (-) is a number, and not an invalid switch!!!
              for TST_ARG in $*
              do
                 if [[ $(echo $TST_ARG | cut -c1) = '-' ]] \
                    && [ $TST_ARG != '-s' -a $TST_ARG != '-S' ]
                 then
                    case $TST_ARG in
                    +([-0-9])) : # No-op, do nothing
                               ;;
                    +([-0-9].[0-9]))
                                 : # No-op, do nothing
                                 ;;
                    +([-.0-9])) : # No-op, do nothing
                                 : :
                    *) echo "\nERROR: $TST_ARG is an invalid argument\n"
                       usage
                       exit 1
                        ;;
                    esac
                  fi
              done
    esac
done
```

A getopts statement starts with a while loop. To define valid command-line switches for a shell script, you add the list of characters that you want to use for command-line switches just after the while getopts part of the while statement.

It is a good practice to enclose the list of command-line switches in double quotes ("list"). The next thing that you need to notice is the use of the colons (:) in the list of valid switches. The placement and usage of the colons is important. Specifically, if the list *starts* with a colon, then any undefined command-line switch that is located will be assigned the question mark (?) character. The question mark character is then assigned to the ARGUMENT variable (which can actually be any variable name). Whenever the ? is matched it is a good idea to exit the script or send an error message to the user, and show the user the correct usage of the shell script before exiting. This ability to catch usage errors is what makes getopts a very nice and powerful tool to use.

But in our case when we encounter the ?, we may just have a negative number! Therefore, any time we encounter a hyphen (-) on the command line, we need to test for a negative number before we tell the user that the input is invalid. This piece of code is in the case statement after the ?.

The other colon (:) used in the list specifies that the switch character that appears immediately *before* the colon requires a switch-argument. Looking at the following getopts example statement may help to clarify the colon usage:

while getopts ":s:S:rtg:" ARGUMENT

In this getopts statement the list begins with a colon, so any command-line switch other than -s, -s, -r, -t, and -g will cause the ARGUMENT variable to be assigned the ? character, indicating a usage error. When any *defined* command-line argument is located on the command line it is assigned to the ARGUMENT variable (you can use any variable name here). When any *undefined* command-line switch is located, and the valid switch list *begins* with a colon, the question mark character is assigned to the ARGUMENT variable. If the switch list does *not* begin with a colon, the undefined switch is ignored. In our shell script we do not want to ignore any invalid command-line argument but we also do not want a negative number to be considered invalid input. This is where we do the extra test on the command line.

Looking at each of the individually defined switches in the previous example, -s and -S each require a switch-argument. The -r, -t, and -g switches do *not* have an argument because they do not have a colon after them in the definition list. When a switch is encountered that requires a switch-argument, the switch-argument is assigned to a variable called OPTARG. In our case the switch-argument to -s or -S is the value of the scale for precision floating-point arithmetic, so we make the following assignment: SCALE=\$OPTARG in the case statement inside the while loop. As with the float_add.ksh shell script, the scale does not give the results that you expect. The use of the scale is in this shell script as a learning experience and you will see expected results in the following shell scripts in this chapter.

Just remember when using getopts to parse command-line arguments for valid switches that getopts could not care less what is on the command line. It is up to you to verify that all of the data that you use is valid for your particular purpose. This is why we make so many tests of the data that the user inputs on the command line.

Building a Math Statement String for bc

Next we move on to the end of the shell script where we build the math statement for the bc command. In building the math statement that we use in the here document,

we now use the SUBTRACT and MINUS variables in the for loop. Take a look at the code segment listed here to build the math statement:

```
# Build the list of numbers to subtract
            # Initialize the SUBTRACT variable to NULL
SUBTRACT=
MTNUS=
            # Initialize the MINUS variable to NULL
# Loop through each number and build a math statement that
# will subtract the numbers in the list.
for X in $NUM LIST
do
      # If the number has a + prefix, remove it!
      if [[ $(echo $X | cut -c1) = '+' ]]
      then
           X=$(echo $X | cut -c2-)
      fi
      SUBTRACT="$SUBTRACT $MINUS $X"
      MINUS='-'
done
```

Notice that we initialize the SUBTRACT and MINUS variables to NULL. We do this because on the first loop iteration we do not want a minus sign (-) included. The minus sign is defined within the for loop. The SUBTRACT variable is initialized to NULL because we want to begin with an empty statement string. As we start the for loop, using the valid list of numbers that we so painstakingly verified, we add only the first number in the NUM_LIST variable. On the second loop iteration, and continuing until all of the numbers in the NUM_LIST have been exhausted, we add a minus sign to the math statement, followed by the next number in the list. Additionally, we took the extra step of removing any plus sign (+) that may be a prefix to any positive number. This step is required because we do not want the + in the equation, or an error will occur because there will be a *- and* a + between two numbers. During this for loop the entire math statement is assigned to the SUBTRACT variable. The statement is built in the following manner, assuming that we have the following numbers to work with:

12 453.766 -223.6 3.145927 22

As we build the math statement the following assignments are made to the SUBTRACT variable:

```
SUBTRACT="12"

SUBTRACT="12 - 453.766"

SUBTRACT="12 - 453.766 - -223.6"

SUBTRACT="12 - 453.766 - -223.6 - 3.145927"

SUBTRACT="12 - 453.766 - -223.6 - 3.145927 - 22"
```

Here Document and Presenting the Result

I want to cover a here document one more time because it is important to know what you can and cannot do with this technique. With the math statement created we are

ready to create the here document to add all of the numbers together with the bc utility. Take a look at the here document shown here:

```
# Do the math here by using a here document to supply
# input to the bc command. The difference of the numbers is
# assigned to the DIFFERENCE variable.
DIFFERENCE=$(bc <<EOF
scale=$SCALE
(${SUBTRACT})
EOF)
```

Just like the here document in Listing 13-1, float_add.ksh, this here document label is the EOF character string. The bc command has its input between the starting EOF label and the ending EOF label. The first label starts the here document, and the second EOF label ends the here document. Each line between the two labels is used as input to the bc command. As mentioned earlier, there are a couple of requirements for a here document. The first requirement is that the starting label *must* be preceded by double input redirection (<<EOF). The second requirement is that there are *never* any blank spaces at the beginning of *any* line in the here document. If even one blank space is placed in column one, strange things may begin to happen with the calculation. This is the cause of a lot of frustration when programming here documents. This blank-space problem is one of the most difficult programming errors to find when you are testing, or using, a shell script with a here document.

The final step is to display the result to the user. Listing 13-5 shows the float_ subtract.ksh shell script in action.

```
[root:yogi]@/scripts# float_subtract.ksh -s 4 8.09838 2048 65536 42.632
The difference of: 8.09838 - 2048 - 65536 - 42.632
to a scale of 4 is -67618.53362
```

Listing 13-5 float_subtract.ksh shell script in action

The float_subtract.ksh shell script is very similar to the float_add.ksh shell script. Again, notice that the scale had no effect on the result of this calculation because we used 5 significant digits to the right of the decimal point with 8.09838. The 5 decimal places take precedence over the scale value. However, if we had not specified a scale value our answer would be an integer instead of a floating-point number. The man page for bc has more information on using scale. The next three shell scripts have some variations also. With this commonality I am going to deviate and cover some of the different aspects of each of the following scripts and show where the differences lie.

Creating the float_multiply.ksh Shell Script

This time we are going to multiply a list of numbers. Using the same front end, for the most part, this shell script changes the building of the math statement and has a new

here document. I want to cover the technique that we use to scan the command-line arguments to find the nonswitch arguments and their associated switch arguments. What remains after the command-line argument scan should be only a list of numbers, which is assigned to the NUM_LIST variable. Of course, we do test each number with regular expressions just as before. Let's look at the float_multiply.ksh shell script shown in Listing 13-6 and study the details at the end.

```
#!/usr/bin/ksh
#
# SCRIPT: float_multiply.ksh
# AUTHOR: Randy Michael
# DATE: 02/23/2007
# REV: 1.1.P
#
# PURPOSE: This shell script is used to multiply a list of numbers
       together. The numbers can be either integers or floating-
       point numbers. For floating-point numbers the user has
#
#
       the option of specifying a scale of the number of digits to
       the right of the decimal point. The scale is set by adding
#
        a -s or -S followed by an integer number.
#
#
# EXIT STATUS:
     0 ==> This script/function exited normally
#
      1 ==> Usage or syntax error
#
#
     2 ==> This script/function exited on a trapped signal
# REV. LIST:
#
# set -x # Uncomment to debug this script
# set -n # Uncomment to debug without any command execution
******
******
SCRIPT_NAME=$(basename $0) # The name of this shell script
SCALE="0"
           # Initialize the scale value to zero
NUM LIST=
           # Initialize the NUM LIST to NULL
COUNT=0
           # Initialize the counter to zero
MAX COUNT=$#
           # Set MAX COUNT to the total number of
           # command-line arguments
***********
******
```

Listing 13-6 float_multiply.ksh shell script

```
function usage
{
echo "\nPURPOSE: Multiplies a list of numbers together\n"
echo "USAGE: $SCRIPT_NAME [-s scale_value] N1 N2...Nn"
echo "\nFor an integer result without any significant decimal places..."
echo "\nEXAMPLE: $SCRIPT_NAME 2048.221 65536 \n"
echo "OR for 4 significant decimal places"
echo "\nEXAMPLE: $SCRIPT_NAME -s 4 8.09838 2048 65536 42.632"
echo "\n\t...EXITING...\n"
3
**********
function exit_trap
{
echo "\n...EXITING on trapped signal...\n"
}
*****
*********
###### Set a Trap ######
trap 'exit_trap; exit 2' 1 2 3 15
****
# Check for at least two command-line arguments
if (($# < 2))
then
   echo "\nERROR: Please provide a list of numbers to multiply"
   usage
   exit 1
fi
*****
# Parse the command-line arguments to find the scale value, if present.
while getopts ":s:S:" ARGUMENT
do
   case $ARGUMENT in
       s S) SCALE=$OPTARG
           ;;
       \?) # Because we may have negative numbers we need
          # to test to see if the ARGUMENT that begins with a
```

```
Listing 13-6 (continued)
```

```
# hyphen (-) is a number, and not an invalid switch!!!
             for TST_ARG in $*
             do
                if [[ $(echo $TST_ARG | cut -c1) = '-' ]] \
                   && [ $TST_ARG != '-s' -a $TST_ARG != '-S' ]
                then
                   case $TST ARG in
                   +([-0-9])) : # No-op, do nothing
                             ;;
                   +([-0-9].[0-9]))
                              : # No-op, do nothing
                              ;;
                   +([-.0-9])) : # No-op, do nothing
                              ;;
                   *) echo "\nERROR: $TST_ARG is an invalid argument\n"
                      usage
                      exit 1
                      ;;
                   esac
                 fi
             done
             ;;
   esac
done
****
# Parse through the command-line arguments and gather a list
# of numbers to multiply together.
while ((COUNT < MAX_COUNT))
do
    ((COUNT = COUNT + 1))
    TOKEN=$1
    case $TOKEN in
              -s|-S) shift 2
                     ((COUNT = COUNT + 1))
                     ;;
         -s${SCALE}) shift
                     ;;
         -S${SCALE}) shift
                     ;;
                  *) NUM_LIST="${NUM_LIST} $TOKEN"
                     ((COUNT < MAX_COUNT)) && shift
                     ;;
    esac
done
```

Listing 13-6 (continued)

```
*****
# Ensure that the scale is an integer value
case $SCALE in
    +([0-9])) : # No-Op - Do Nothing
             ;;
          *) echo "\nERROR: Invalid scale - $SCALE - Must be an integer"
              usage
              exit 1
              ;;
esac
*****
# Check each number supplied to ensure that the "numbers"
# are either integers or floating-point numbers.
for NUM in $NUM_LIST
do
case $NUM in
    +([0-9])) # Check for an integer
              : # No-op, do nothing.
              : :
    +([-0-9])) # Check for a negative whole number
              : # No-op, do nothing
              ;;
    +([0-9] | [.] [0-9]))
              # Check for a positive floating-point number
              : # No-op, do nothing
              ;;
    +(+[0-9] |[.][0-9]))
              # Check for a positive floating-point number
              # with a + prefix
              : # No-op, do nothing
              ;;
    +([-0-9]|.[0-9]))
              # Check for a negative floating-point number
              : # No-op, do nothing
              ::
    +(-.[0-9]))
              # Check for a negative floating-point number
              : # No-op, do nothing
              ;;
    +([+.0-9]))
              # Check for a positive floating-point number
              : # No-op, do nothing
              ;;
```

Listing 13-6 (continued)

```
*) echo "\nERROR: $NUM is NOT a valid number"
      usage
      exit 1
       ;;
    esac
done
*****
# Build the list of numbers to multiply
MULTIPLY= # Initialize the MULTIPLY variable to NULL
         # Initialize the TIMES variable to NULL
TIMES=
# Loop through each number and build a math statement that
# will multiply all of the numbers together.
for X in $NUM_LIST
do
     # If the number has a + prefix, remove it!
     if [[ $(echo $X | cut -c1) = '+' ]]
     then
         X=$(echo $X | cut -c2-)
     fi
     MULTIPLY="$MULTIPLY $TIMES $X"
     TIMES='*'
done
****
# Do the math here by using a here document to supply
# input to the bc command. The product of the multiplication
# of the numbers is assigned to the PRODUCT variable.
PRODUCT=$(bc <<EOF
scale=$SCALE
SMULTIPLY
EOF)
**********
# Present the result of the multiplication to the user.
echo "\nThe product of: $MULTIPLY"
echo "\nto a scale of $SCALE is ${PRODUCT}\n"
```

Listing 13-6 (continued)

As you can see in Listing 13-6, most of the previous two shell scripts have been carried over for use here. Now I want to cover in a little more detail how the scanning of the command-line arguments works when we extract the command switches, and the associated switch-arguments, from the entire list of arguments.

Parsing the Command Line for Valid Numbers

To start the extraction process we use the two previously initialized variables, COUNT and MAX_COUNT. The COUNT variable is incremented during the processing of the while loop, and the MAX_COUNT variable has been initialized to the value of \$#, which specifies the total number of command-line arguments given by the user. The while loop runs until the COUNT variable is greater than or equal to the MAX_COUNT variable.

Inside the while loop the COUNT variable is incremented by one, so on the first loop iteration the COUNT equals 1, one, because it was initialized to 0, zero. Next is the TOKEN variable. The TOKEN variable always points to the \$1 positional parameter throughout the while loop execution. Using the current value of the \$1 positional parameter, which is pointed to by the TOKEN variable, as the case statement argument we test to see if \$TOKEN points to a known value. The current known values on the command line are the -s and -S switches that are used to define the scale for floating-point arithmetic, if a scale was given, and the integer value of the SCALE. There are only two options for the value of the scale:

-s {Scale Integer}

Because these are the only possible scale values (we also allow an uppercase –S) for the command line, we can test for this condition easily in a case statement. Remember that I said the \$TOKEN variable always points to the \$1 positional parameter? To move the other positional parameters to the \$1 position we use the **shift** command. The shift command alone will shift the \$2 positional parameter to the \$1 position. What if you want to move the \$3 positional parameter to the \$1 position? We have two options: Use two shift commands in series, or add an integer as an argument to the shift command. Both of the following commands move the \$3 positional parameter to the \$1 position:

```
shift; shift
```

or

shift 2

Now you may be wondering what happens to the previous \$1, and in this case \$2, positional parameter values. Well, anything that is shifted from the \$1 position goes to the bit bucket! But this is the result that we want here.

If the value of the positional parameter in the \$1 position is the -s or -S switch alone, we shift two positions. We do this double shift because we know that there should

be an integer value after the -s or -S switch, which is the integer switch-argument that defines the scale. On the other hand, if the user did *not* place a space between the -s or -S switch and the switch-argument, we shift only once. Let's say that the user entered either of the following command statements on the command line:

```
[root:yogi]@/scripts# float_multiply.ksh -s 4 8.09838 2048 65536 42.632
```

or

```
[root:yogi]@/scripts# float_multiply.ksh -s4 8.09838 2048 65536 42.632
```

Notice in the first command the user added a space between the switch and the switch-argument (-s 4). In this situation our test will see the -s as a single argument, so we need to shift two places to move past the switch argument, which is 4. In the second command statement the user did *not* add a space between the switch and the switch argument (-s4). This time we shift only one position because the switch and the switch argument are together in the \$1 positional parameter, which is what \$TOKEN points to.

There is one more thing that I want to point out. On each loop iteration the COUNT is incremented by 1, one, as you would expect. But if we shift two times, we need to increment the COUNT by 1, one, a second time so we do not count past the number of arguments on the command line. This is very important! If you leave out this extra counter incrementation, the shell script errors out. Every little piece of this loop has a reason for being there. Speaking of the loop, please study the while loop in the code segment shown in Listing 13-7.

```
# Parse through the command-line arguments and gather a list
# of numbers to multiply together.
while ((COUNT < MAX_COUNT))
do
     ((COUNT = COUNT + 1))
     TOKEN=$1
     case $TOKEN in
               -s -S) shift 2
                      ((COUNT = COUNT + 1))
                       ;;
          -s${SCALE}) shift
                       ;;
          -S${SCALE}) shift
                      ; ;
                    *) NUM LIST="${NUM LIST} $TOKEN"
                      ((COUNT < MAX_COUNT)) && shift
                       ;;
     esac
done
```

Listing 13-7 Code to parse numbers from the command line

The techniques to build the math statement and to do the calculations with a here document using the bc command are changed only slightly. Of course, because we are multiplying a string of numbers instead of adding or subtracting, we changed the build code to add a *, instead of a + or -. The here document is exactly the same except that the result is assigned to the PRODUCT variable. Please look closely at the float_multiply.ksh shell script shown in Listing 13-6 and study the subtle changes from the previous two shell scripts in Listing 13-1 and Listing 13-3.

The float_multiply.ksh shell script is shown in action in Listing 13-8. Notice in this output that the scale setting still has no effect on the output.

```
[root:yogi]@/scripts# float_multiply.ksh -s 4 8.09838 2048 65536 42.632
The product of: 8.09838 * 2048 * 65536 * 42.632
is 46338688867.08584
```

Listing 13-8 float_multiply.ksh shell script in action

In the next section we move on to study division. We had to do some creative engineering to change the previous shell script to work with only two numbers. Keep reading — I think you will pick up a few more pointers.

Creating the float_divide.ksh Shell Script

For the division script we had to do some changes because we are dealing with only two numbers, as opposed to an unknown string of numbers. The float_divide.ksh shell script starts out the same as the previous three scripts, with the same variables and a modified usage function. The first test is for the correct number of command-line arguments. In this shell script we can handle from two to four arguments, with the option to specify a scale value for precision of floating-point numbers.

In the getopts statement we perform the same test to parse out the scale switch, -s or -S, and the switch-argument. When, however, we get to parsing the entire list of command-line arguments to gather the numbers for the division, we do things a little differently. The while loop is the same, with the counter and the TOKEN variable always pointing to the \$1 positional parameter, which we use as we shift command-line arguments to the \$1 position. It is in the case statement that we do our modification. For division we need a *dividend* and a *divisor*, which has the form in a division statement of ((QUOTIENT=\$DIVIDEND / \$DIVISOR)). As we parse the command-line arguments we assign the first number to the DIVIDEND variable and the second number to the DIVISOR. Look at the code segment in Listing 13-9, and we will go into the details at the end.

```
# Parse through the command-line arguments and gather a list
# of numbers to divide.
TOTAL_NUMBERS=0
while ((COUNT < MAX_COUNT))
do
    ((COUNT = COUNT + 1))
    TOKEN=$1
    case $TOKEN in
               -s|-S) shift 2
                      ((COUNT = COUNT + 1))
                      ;;
          -s${SCALE}) shift
                      : :
          -S${SCALE}) shift
                      ;;
                   *) ((TOTAL_NUMBERS = TOTAL_NUMBERS + 1))
                      if ((TOTAL_NUMBERS == 1))
                      then
                           DIVIDEND=$TOKEN
                      elif ((TOTAL_NUMBERS == 2))
                      then
                           DIVISOR=$TOKEN
                      else
                           echo "ERROR: Too many numbers to divide"
                           usage
                           exit 1
                      fi
                      NUM_LIST="$NUM_LIST $TOKEN"
                      ((COUNT < MAX_COUNT)) && shift
                      ;;
     esac
done
```

Listing 13-9 Code to extract the dividend and divisor

Notice the boldface text in the case statement in Listing 13-9. When a number is encountered we use a variable called TOTAL_NUMBERS to keep track of how many numbers are on the command line. If \$TOTAL_NUMBERS is equal to 1, one, we assign the value of the \$TOKEN variable to the DIVIDEND variable, the number on the top in a division math statement. When \$TOTAL_NUMBERS is equal to 2, we assign the value of the \$TOKEN variable to the DIVISOR variable. If the \$TOTAL_NUMBERS counter variable exceeds 2, we print an error message to the screen, execute the usage function, and exit the script with a return code of 1, which is a normal usage error for this shell script.

Notice that we are also keeping the NUM_LIST variable. We use the \$NUM_LIST to verify that each "number" is actually an integer or a floating-point number by using the regular expressions that we covered previously in this chapter.

Notice in the shell script in Listing 13-10 that we omitted the step of building the math statement. In this script it is not necessary because we have the dividend and divisor captured in the code segment in Listing 13-9. Check out the shell script in Listing 13-10, and pay close attention to the boldface text.

```
#!/usr/bin/ksh
# SCRIPT: float_divide.ksh
# AUTHOR: Randy Michael
# DATE: 02/23/2007
# REV: 1.1.A
# PURPOSE: This shell script is used to divide two numbers.
       The numbers can be either integers or floating-point
#
#
       numbers. For floating-point numbers the user has the
#
       option to specify a scale of the number of digits to
       the right of the decimal point. The scale is set by
#
       adding a -s or -S followed by an integer number.
#
#
# EXIT STATUS:
    0 ==> This script exited normally
#
    1 ==> Usage or syntax error
#
     2 ==> This script exited on a trapped signal
# REV. LIST:
# set -x # Uncomment to debug this script
# set -n # Uncomment to debug without any command execution
*****
******
SCRIPT_NAME=`basename $0`
SCALE="0"
         # Initialize the scale value to zero
NUM_LIST=
          # Initialize the NUM_LIST to NULL
COUNT=0
           # Initialize the counter to zero
MAX COUNT=$# # Set MAX COUNT to the total number of
           # command-line arguments
**********
*****
```

Listing 13-10 float_divide.ksh shell script

```
function usage
{
echo "\nPURPOSE: Divides two numbers\n"
echo "USAGE: $SCRIPT_NAME [-s scale_value] N1 N2"
echo "\nFor an integer result without any significant decimal places..."
echo "\nEXAMPLE: $SCRIPT_NAME 2048.221 65536 \n"
echo "OR for 4 significant decimal places"
echo "\nEXAMPLE: $SCRIPT_NAME -s 4 2048.221 65536"
echo "\n\t...EXITING...\n"
}
**********
function exit_trap
{
      echo "\n...EXITING on trapped signal...\n"
}
******
*********
###### Set a Trap ######
trap 'exit_trap; exit 2' 1 2 3 15
# Check for at least two command-line arguments
# and not more than four
if (($# < 2))
then
      echo "\nERROR: Too few command line arguments"
      usage
      exit 1
elif (($# > 4))
then
      echo "\nERROR: Too many command line arguments"
      usage
      exit 1
fi
# Parse the command-line arguments to find the scale value, if present.
while getopts ":s:S:" ARGUMENT
do
```

Listing 13-10 (continued)

```
case $ARGUMENT in
        s|S) SCALE=$OPTARG
             ;;
         \?) # Because we may have negative numbers we need
             # to test to see if the ARGUMENT that begins with a
             # hyphen (-) is a number, and not an invalid switch!!!
             for TST ARG in $*
             do
                if [[ $(echo $TST_ARG | cut -c1) = '-' ]] \
                   && [ $TST_ARG != '-s' -a $TST_ARG != '-S' ]
                then
                   case $TST_ARG in
                   +([-0-9])) : # No-op, do nothing
                              ;;
                   +([-0-9].[0-9]))
                               : # No-op, do nothing
                               ;;
                   +([-.0-9])) : # No-op, do nothing
                               ;;
                   *) echo "\nERROR: $TST_ARG is an invalid argument\n"
                      usage
                      exit 1
                      ;;
                   esac
                 fi
             done
             ;;
   esac
done
******
# Parse through the command-line arguments and gather a list
# of numbers to divide.
TOTAL NUMBERS=0
while ((COUNT < MAX_COUNT))
do
     ((COUNT = COUNT + 1))
     TOKEN=$1
     case $TOKEN in
              -s|-S) shift 2
                     ((COUNT = COUNT + 1))
                     ;;
         -s${SCALE}) shift
                     ;;
```

Listing 13-10 (continued)

```
-S${SCALE}) shift
                    ;;
                 *) ((TOTAL_NUMBERS = TOTAL_NUMBERS + 1))
                    if ((TOTAL_NUMBERS == 1))
                    then
                        DIVIDEND=$TOKEN
                    elif ((TOTAL_NUMBERS == 2))
                    then
                        DIVISOR=$TOKEN
                    else
                        echo "ERROR: Too many numbers to divide"
                        usage
                        exit 1
                    fi
                   NUM_LIST="$NUM_LIST $TOKEN"
                    ((COUNT < MAX_COUNT)) && shift
                    ;;
    esac
done
*****
# Ensure that the scale is an integer value
case $SCALE in
    +([0-9])) : # No-op - Do Nothing
             ;;
    *) echo "\nERROR: Invalid scale - $SCALE - Must be an integer"
             usage
             exit 1
             ;;
esac
*****
# Check each number supplied to ensure that the "numbers"
# are either integers or floating-point numbers.
for NUM in $NUM LIST
do
    case $NUM in
    +([0-9])) # Check for an integer
             : # No-op, do nothing.
             ;;
    +([-0-9])) # Check for a negative whole number
             : # No-op, do nothing
             ;;
    +([0-9] | [.] [0-9]))
```

Listing 13-10 (continued)

```
# Check for a positive floating-point number
              : # No-op, do nothing
              ;;
    +(+[0-9] | [.] [0-9]))
              # Check for a positive floating-point number
              # with a + prefix
              : # No-op, do nothing
              ;;
    +([-0-9]|.[0-9]))
              # Check for a negative floating-point number
              : # No-op, do nothing
              ;;
    +(-.[0-9]))
              # Check for a negative floating-point number
              : # No-op, do nothing
              ;;
    +([+.0-9]))
              # Check for a positive floating-point number
              : # No-op, do nothing
              ;;
    *) echo "\nERROR: $NUM is NOT a valid number"
       usage
       exit 1
       : :
    esac
done
*****
# Do the math here by using a here document to supply
# input to the bc command. The quotient of the division is
# assigned to the QUOTIENT variable.
QUOTIENT=$(bc <<EOF
scale=$SCALE
$DIVIDEND / $DIVISOR
EOF)
*****
# Present the result of the division to the user.
echo "\nThe quotient of: $DIVIDEND / $DIVISOR"
echo "\nto a scale of $SCALE is ${QUOTIENT}\n"
```

Listing 13-10 (continued)

Let's look at the here document we use to calculate the QUOTIENT using the bc utility at the end of Listing 13-10. We already have extracted the dividend and divisor directly from the command line, so we skipped building the math statement. Using command

substitution we use double input redirection with a label (<<EOF), which defines the beginning of a here document, to set the scale of the precision of floating-point numbers and to divide the two numbers. If no scale was given on the command line, the scale is 0, zero, indicating an integer value. The here document ends with the final label (EOF) to end the here document and exit the bc utility, which is an interactive program. The final step is to present the result to the user. In Listing 13-11 you can see the float_divide.ksh shell script in action.

```
[root:yogi]@/scripts# float_divide.ksh -s 6 .3321 -332.889
The quotient of: .3321 / -332.889
to a scale of 6 is -.000997
```

Listing 13-11 float_divide.ksh shell script in action

Notice that the scale worked with the division script because we never used a floating-point number greater than 6 significant digits in our division, therefore we never overrode the scale value. We have completed shell scripts for addition, subtraction, multiplication, and division. I want to present one more variation in the next section.

Creating the float_average.ksh Shell Script

Using the addition shell script from Listing 13-1 we can make a couple of minor modifications and take the average of a series of numbers.

The first addition to Listing 13-1 is the addition of the variable TOTAL_NUMBERS. To average a list of numbers, we need to know how many numbers are in the list so that we can divide the SUM by the total number of numbers. The counter is added in the sanity check of the \$NUM_LIST numbers, where we are ensuring that the numbers are either integers or floating-point numbers. This modification is shown in Listing 13-12.

```
TOTAL_NUMBERS=0
```

Listing 13-12 Code segment to keep a running total of numbers

```
: # No-op, do nothing
               ;;
     +(+[0-9]|[.][0-9]))
               # Check for a positive floating-point number
               # with a + prefix
               : # No-op, do nothing
               ;;
     +([-0-9]|.[0-9]))
               # Check for a negative floating-point number
               : # No-op, do nothing
               ;;
     +(-.[0-9]))
               # Check for a negative floating-point number
               : # No-op, do nothing
               ;;
     +([+.0-9]))
               # Check for a positive floating-point number
               : # No-op, do nothing
               ;;
     *) echo "\nERROR: $NUM is NOT a valid number"
        usage
        exit 1
        ;;
     esac
done
```

Listing 13-12 (continued)

The two lines of modification are highlighted in boldface text in Listing 13-12. The only other modifications are with the here document, where we added a division to the \$ADD by the \$TOTAL_NUMBERS, and the code to present the result to the user. This code modification is shown in Listing 13-13.

```
# Do the math with a here document for the bc command
AVERAGE=$(bc <<EOF
scale=$SCALE
(${ADD}) / $TOTAL_NUMBERS
EOF)
# Present the result to the user
echo "\nThe average of: $(echo $ADD | sed s/+//g)"
echo "\nto a scale of $SCALE is ${AVERAGE}\n"
```

Listing 13-13 Code segment to average a list of numbers

In Listing 13-13 notice how the averaging of the numbers is done. In the float_add . ksh script in listing 13.1 an addition math statement was created and assigned to the ADD variable. Now we use this ADD variable as input to the bc command in the here document and divide the result of the addition by the total number of numbers given on the command line, \$TOTAL_NUMBERS. The result is an average of the numbers.

In the next step we present the result to the user. Notice the sed statement that is in boldface text. This sed statement is replacing every occurrence of the plus sign (+) with a blank space. The result is a list of the numbers only. We could have just as easily used the \$NUM_LIST variable, but I wanted to slip a sed statement into this chapter somewhere. The float_average.ksh shell script is shown in action in Listing 13-14.

```
[root:yogi]@/scripts# float_average.ksh -s 8 .22389 65 -32.778 -.221
The average of: .22389 65 -32.778 -.221
to a scale of 8 is 8.05622250
```

Listing 13-14 float_average.ksh shell script in action

The float_average.ksh shell script listing is shown in Listing 13-15.

```
#!/usr/bin/ksh
#
# SCRIPT: float_aaverage.ksh
# AUTHOR: Randy Michael
# DATE: 03/01/2007
# REV: 1.1.A
# PURPOSE: This script is used to average a list of
  floating-point numbers.
#
# EXIT STATUS:
#
      0 ==> This script completed without error
       1 ==> Usage error
#
      2 ==> This script exited on a trapped signal
#
# REV. LIST:
#
# set -x # Uncomment to debug this script
# set -n # Uncomment to debug without any command execution
SCRIPT_NAME=`basename $0`
ARG1="$1"
ARG2="$2"
ARG LIST="$*"
```

Listing 13-15 float_average.ksh shell script

```
TOTAL_TOKENS="$#"
SCALE="0"
**********
**********
function usage
{
 echo "\n\n"
 echo "Numbers to average...\n"
 echo "USAGE: $SCRIPT_NAME [-s scale_value] N1 N2...N#"
 echo "\nFor an integer result without any significant decimal
places..."
 echo "\nEXAMPLE: $SCRIPT_NAME 2048.221 65536 \n"
 echo "OR for 4 significant decimal places"
 echo "\nEXAMPLE: $SCRIPT_NAME -s 4 8.09838 2048 65536 42.632\n"
 echo "Try again...EXITING...\n"
}
*****
function exit_trap
{
echo "\n...EXITING on trapped signal...\n"
3
*********
*****
###### Set a Trap ######
trap 'exit_trap; exit 2' 1 2 3 15
if [ $# -1t 2 ]
then
  echo "\nIncorrect number of command auguments...Nothing to
average..."
  usage
  exit 1
fi
if [ $ARG1 = "-s" ] || [ $ARG1 = "-S" ]
then
  echo $ARG2 | grep [[:digit:]] >/dev/null 2>&1
  if [ $? -ne "0" ]
  then
```

Listing 13-15 (continued)

```
echo "\nERROR: Invalid argument - $ARG2 ... Must be
    an integer...\n"
        usage
        exit 1
    fi
    echo $ARG2 | grep "\." >/dev/null 2>&1
    if [ $? -ne "0" ]
    then
        SCALE="$ARG2"
        NUM_LIST='echo $ARG_LIST | cut -f 3- -d " "'
        (( TOTAL_TOKENS = $TOTAL_TOKENS - 2 ))
    else
        echo "\nERROR: Invalid scale - $ARG2 ... Scale must \
    be an integer...\n"
       usage
        exit 1
    fi
else
   NUM_LIST=$ARG_LIST
fi
for TOKEN in $NUM_LIST
do
    echo $TOKEN | grep [[:digit:]] >/dev/null 2>&1
   RC=$?
    case $RC in
    0) cat /dev/null
       ;;
    *) echo "\n$TOKEN is not a number...Invalid argument list"
      usage
       exit 1
       ;;
    esac
done
# Build the list of numbers to add for averaging...
ADD=" "
PLUS=""
for X in $NUM_LIST
do
   ADD="$ADD $PLUS $X"
   PLUS="+"
done
# Do the math here
```

Listing 13-15 (continued)

```
AVERAGE=`bc <<EOF
scale=$SCALE
(${ADD}) / $TOTAL_TOKENS
EOF`
echo "\n${AVERAGE}\n"</pre>
```

Listing 13-15 (continued)

Well, that about does it for our bc utility fun. Play around with bc; it is a pretty powerful tool that does some very complex arithmetic.

Other Options to Consider

As always, these scripts can be improved, just as any shell script can be improved. As you saw in each of the shell scripts in this chapter, we did a lot of tests to verify the integrity of the data the user entered on the command line. You may be able to combine some of these tests, but I still like to separate each piece so that whoever comes along in the future can follow the shell script easily. Sure, some of these can be done in three lines of code, but this does not allow for data verification, and the user would have to rely on the cryptic system error messages that do not always tell *where* the data error is located.

Creating More Functions

As an exercise for this chapter, replace each of the data tests with functions. This is easy to do! All that is required is that the function must be defined before it can be used. So, put these new functions at the top of the shell scripts in the DEFINE FUNCTIONS HERE section. When you extract a code segment from the main body of the shell script, make a comment to yourself that "XYZ Function goes here." Then use one of the following techniques to make the code segment into a function:

```
function my_new_function_name
{
    Place Code Segment Here
    }
or
    my_new_function_name ()
    {
        Place Code Segment Here
    }
```

Both techniques produce the same result, but I prefer the first method because it is more intuitive to new shell programmers. Remember where the scope of your variables can be seen. A variable in the main body of the shell script is a *global* variable, which can be seen by all functions. A variable inside of a function has limited scope and can be seen in the function and any function that the current function calls, but not in the calling shell script. There are techniques that we have covered in this book to get around these scope limitations, so I hope you have read the whole book. Experiment! That is how you learn.

I hope you enjoyed studying this chapter. Please explore the other options that are available in the bc command; you will be surprised by what you can do.

Summary

We have covered a lot of material in this chapter. I hope that you will now find that math is not difficult in a shell script and that it can be done to the precision required. The bc command is very powerful, and we only touched the surface of the ability of bc here. For more information on bc look at the man page, **man bc**.

In the next chapter we are moving on to changing numbers between number bases. We start with the basics and move to a shell script that converts any number in any number base to any other number base. See you in the next chapter!

Lab Assignments

- 1. Utilize an expect script to interact with the bc utility to add, subtract, multiply, divide, and average floating-point numbers.
- 2. Rewrite all five shell scripts in this chapter to utilize functions in the testing of the user-inputted numbers.

снартек 14

Number Base Conversions

On many occasions in computer science you need to convert numbers between different number bases. For example, you may need to translate a hexadecimal number into an octal representation, or if you are a software developer you may want to license the software you create for a specific machine. One way of creating a machine-specific license key is to use the IP address of the machine to create a hexadecimal character string, which will allow the software to execute only on that specific machine. The first example here is a common occurrence, but the latter one is a little more obscure.

In this chapter we are going to present some number base conversion techniques and also show how to create a shell script that produces a license key, as in our second example. Converting between number bases is very straightforward, and we are going to go through each step. Before we can write a shell script we need the correct command syntax. In this case we add setting up the proper environment for the system to do all of the hard work automatically.

Syntax

By far, the easiest way to convert a number from one base to another is to use the **typeset** command with the -ibase Korn shell option, specifying a valid shell-supported base between base 2 and base 36. Bash and Bourne shells do not support this particular typeset command notation. The typeset command is used a lot in this book, mostly to force a character string to uppercase or lowercase and to classify a variable as an integer value. This time we are adding to the integer setting, specified by typeset -i VAR_NAME, by adding the number base that the variable is to maintain. For example, if the variable BASE_16_NUM is to always contain a hexadecimal number, the next command will set the variable's environment:

typeset -i16 BASE_16_NUM

After the BASE_16_NUM variable is typeset to base 16, any value assigned to this variable is automatically converted to hexadecimal. We can also typeset a variable *after* a number has been assigned. This applies not only to base-10 numbers, but also to *any* base number up to the system and shell limit, which is at least base 36. Let's look at some examples of converting between bases.

To quickly convert any supported base number (base 2 through base 36) to base 10, use the following mathematical notation:

```
echo $((base#number))
```

For example, the following command converts the base-16 number fel8a4 to its base-10 equivalent:

```
[root@yogi:/scripts]> echo $((16#fe18a4))
16652452
```

Using both of these techniques, we can convert between any number bases.

Example 1: Converting from Base 10 to Base 16

```
[root@yogi:/scripts]> typeset -i16 BASE_16_NUM
[root@yogi:/scripts]> BASE_16_NUM=47295
[root@yogi:/scripts]> echo $BASE_16_NUM
```

16#b8bf

Notice the output in Example 1. The output starts out by setting the number base that is represented, which is base 16 here. The string after the pound sign (#) is the hexadecimal number, b8bf. Next we want to convert from base 8, octal, to base 16, hexadecimal. We use the same technique, except this time we must specify the number base of the octal number in the assignment, as shown in Example 1.

Example 2: Converting from Base 8 to Base 16

```
[root@yogi:/scripts]> typeset -i16 BASE_16_NUM
[root@yogi:/scripts]> BASE_16_NUM=8#472521
[root@yogi:/scripts]> echo $BASE_16_NUM
```

16#735c9

In Example 2 notice that we assigned the octal number 472521 to the BASE_16_NUM variable by specifying the number base followed by the base-8 number, BASE_16_NUM=8#472521. When this base-8 number is assigned to the BASE_16_NUM variable it is automatically converted to base 16. As you can see, the system can do the hard work for us.

In UNIX, there is never just one way to accomplish a task, and number base conversions are no exception. We can also use the **printf** command to convert between

number bases. The printf command accepts base-10 integer values and converts the number to the specified number base. The following options are available:

- o Accepts a base-10 integer and prints the number in octal
- x Accepts a base-10 integer and prints the number in hexadecimal

Let's look at two examples of using the printf command.

NOTE By default, the printf command does not produce a line-feed on some operating systems. Notice the trailing echo command that produces the final line-feed.

Example 3: Converting Base 10 to Octal

[root@yogi:/scripts]> printf %o 20398; echo

47656

In Example 3 notice the added percent sign (%) before the printf command option. This % tells the printf command that the following lowercase o is a number base conversion to octal.

Example 4: Converting Base 10 to Hexadecimal

[root@yogi:/scripts]> printf %x 20398; echo

4fae

Although not as flexible as the typeset command, the printf command allows you to do base conversions from base 10 to base 8 and base 16. I like the extra flexibility of the typeset command, so this is the conversion method that we are going to use most of the time in this chapter.

Scripting the Solution

The most common number base conversion that computer-science people use is conversions between base 2, 8, 10, and 16. We want to be able to convert back and forth between these, and other, bases in this chapter. To do this conversion we are going to create five shell scripts to show the flexibility, and use, of number base conversions. The following shell scripts are covered:

- Base 2 (binary) to base 16 (hexadecimal) shell script
- Base 10 (decimal) to base 16 (hexadecimal) shell script
- Script to create a software key based on the hexadecimal representation of an IP address

- Command-line script to translate between *any* number base between 2 and 36
- Interactive script to translate between any number base between 2 and 36

We have a lot to cover in this chapter, but these shell scripts are not too difficult to follow. I hope you pick up a few tips and techniques in this chapter, as well as the whole book.

Base 2 (Binary) to Base 16 (Hexadecimal) Shell Script

This is the first conversion that most computer-science students learn in school. It is easy enough to do this conversion with a pencil and paper, but, hey, we want *automation*! This shell script to convert from binary to hexadecimal uses the typeset technique, as all of these scripts use. You know the basic principle of the conversion, so let's present the shell script and cover the details at the end. The equate_base_2_to_16.ksh shell script is shown in Listing 14-1.

```
#!/usr/bin/ksh
#
# SCRIPT: equate_base_2_to_16.ksh
# AUTHOR: Randy Michael
# DATE: 07/07/2007
# REV: 1.2.P
# PURPOSE: This script is used to convert a base 2 number
#
        to a base 16 hexadecimal representation.
        This script expects that a base 2 number
#
        is supplied as a single argument.
#
#
# EXIT CODES:
            0 - Normal script execution
             1 - Usage error
#
#
# REV LIST:
#
# set -x # Uncomment to debug this script
# set -n # Uncomment to check command syntax without any execution
*****
# DEFINE FILES AND VARIABLES HERE
*****
SCRIPT_NAME=`basename $0`
# Set up the correct echo command usage. Many Linux
# distributions will execute in Bash even if the
```

Listing 14-1 equate_base_2_to_16.ksh shell script listing

```
# script specifies Korn shell. Bash shell requires
# we use echo -e when we use \n, \c, etc.
case $SHELL in
*/bin/Bash) alias echo="echo -e"
         ;;
esac
******
# DEFINE FUNCTIONS HERE
*****
function usage
echo "\nUSAGE: $SCRIPT_NAME {base 2 number}"
echo "\nEXAMPLE: $SCRIPT_NAME 1100101101"
echo "\nWill return the hexadecimal base 16 number 32d"
echo "\n\t ...EXITING...\n"
}
******
# BEGINNING OF MAIN
*****
# Check for a single command-line argument
if (($# != 1))
then
      echo "\nERROR: A base 2 number must be supplied..."
      usage
      exit 1
fi
# Check that this single command-line argument is a binary number!
case $1 in
+([0-1])) BASE_2_NUM=$1
        ;;
     *) echo "\nERROR: $1 is NOT a base 2 number"
        usage
        exit 1
        ;;
esac
# Assign the base 2 number to the BASE_16_NUM variable
BASE_16_NUM=2#$BASE_2_NUM
```

Listing 14-1 (continued)

```
# Now typeset the BASE_16_NUM variable to base 16.
# This step converts the base 2 number to a base 16 number.
typeset -i16 BASE_16_NUM
# Display the resulting base 16 representation
echo $BASE_16_NUM
```

Listing 14-1 (continued)

In Listing 14-1 all of the real work is done with three commands. The rest of the code is for testing the user input and providing the correct usage message when an error is detected. Two tests are performed on the user input. First, the number of command-line arguments is checked to ensure that exactly one argument is supplied on the command line. The second test is to ensure that the single command-line argument is a binary number. Let's look at these two tests.

The \$# shell variable shows the total number of command-line arguments, with the executing command/shell-script's filename in the \$0 position, and the single command-line argument represented by the positional parameter \$1. For this shell script the value \$# shell variable must be equal to 1, one. This test is done using the mathematical test shown here:

```
if (($# != 1))
then
            echo "\nERROR: A base 2 number must be supplied..."
            usage
            exit 1
fi
```

The second test is to ensure that a base 2 number is given on the command line. For this test we use a good ole regular expression. You have to love the simplicity of making this type of test. Because a binary number can consist only of 0, zero, or 1, one, it is an easy test with a regular expression. The idea is to specify a valid range of *characters* that can make up a binary number. The tests for decimal and hexadecimal are similar. The regular expression that we use is used in the case statement shown here:

```
case $1 in
+([0-1])) BASE_2_NUM=$1
    ;;
    *) echo "\nERROR: $1 is NOT a base 2 number"
    usage
    exit 1
    ;;
esac
```

The regular expression shown here has the form +([0-1]) and is used as a test for the specified valid range of numbers 0 through 1. If the range is valid, we assign the

binary number to the BASE_2_NUM variable. We will look at more regular expressions later in this chapter.

When we are satisfied that we have valid data we are ready to do the number base conversion. The first step is to assign the binary number that was supplied on the command line to the BASE_16_NUM variable. Notice that thus far we have not typeset any of the variables, so the variable can contain *any character string*. It is *how* we assign the binary number to the BASE_16_NUM variable that is important. When the binary value is assigned to the variable the current number base is specified, as shown here:

```
BASE_16_NUM=2#$BASE_2_NUM
```

Notice in this assignment that the BASE_2_NUM variable is preceded by the number base, which is base 2 in this case. This allows for the number to be assigned to the BASE_16_NUM as a base 2 number. The base translation takes place in the next step where we typeset the variable to base 16, as shown here:

```
typeset -i16 BASE_16_NUM
```

With the BASE_16_NUM variable typeset to base 16, specified by the -i16 argument, the binary number is translated to hexadecimal. We could just as easily typeset the BASE_16_NUM variable at the top of the shell script, but it really does not matter.

Base 10 (Decimal) to Base 16 (Hexadecimal) Shell Script

This shell script is very similar to the shell script in the previous section. We are really changing just the tests and the conversion values. Other than these few changes the two shell scripts are identical. Again, I want to present the shell script and cover the details at the end. The equate_base_10_to_16.ksh shell script is shown in Listing 14-2.

```
#!/usr/bin/ksh
#
# SCRIPT: equate_base_10_to_16.ksh
# AUTHOR: Randy Michael
# DATE: 07/07/2007
# REV: 1.2.P
#
# PURPOSE: This script is used to convert a base 10 number
         to a base 16 hexadecimal representation.
#
         This script expects that a base 10 number
#
#
         is supplied as a single argument.
# EXIT CODES:
#
              0 - Normal script execution
               1 - Usage error
#
#
# REV LIST:
```

Listing 14-2 equate_base_10_to_16.ksh shell script listing

```
#
#
# set -x # Uncomment to debug this script
# set -n # Uncomment to check command syntax without any execution
******
# DEFINE FILES AND VARIABLES HERE
*****
SCRIPT_NAME=`basename $0`
# Set up the correct echo command usage. Many Linux
# distributions will execute in Bash even if the
# script specifies Korn shell. Bash shell requires
# we use echo -e when we use n, c, etc.
case $SHELL in
*/bin/Bash) alias echo="echo -e"
         ;;
esac
*****
# DEFINE FUNCTIONS HERE
******
function usage
{
echo "\nUSAGE: $SCRIPT NAME {base 10 number}"
echo "\nEXAMPLE: $SCRIPT NAME 694"
echo "\nWill return the hexadecimal number 2b6"
echo "\n\t...EXITING...\n"
}
*****
# BEGINNING OF MAIN
*****
# Check for a single command-line argument
if (($# != 1))
then
      echo "\nERROR: A base 10 number must be supplied..."
     usage
      exit 1
fi
# Check that this single command-line argument is a base 10 number!
case $1 in
```

Listing 14-2 (continued)

```
+([0-9])) BASE_10_NUM=$1
          ;;
       *) echo "\nERROR: $1 is NOT a base 10 number"
          usage
          exit 1
          ;;
esac
# Assign the base 10 number to the BASE_16_NUM variable
BASE_16_NUM=10#$BASE_10_NUM
# NOTE: Since base 10 is implied by default,
# the 10# may be omitted as a prefix. Both notations
# are supported
# Now typeset the BASE_16_NUM variable to base 16.
# This step converts the base 10 number to a base 16 number.
typeset -i16 BASE_16_NUM
# Display the resulting base 16 number representation
echo $BASE_16_NUM
# This following code is optional. It removes the number base
# prefix. This may be helpful if using this script with
# other programs and scripts.
# Set up the correct awk usage. Solaris
needs to
# use nawk instead of awk.
#
# case $(uname) in
# SunOS) AWK="nawk"
# ;;
# *) AWK="awk"
# ;;
# esac
# Strip out the base prefix and the pound sign (#). (Optional)
#
# echo $BASE_16_NUM | grep -q "#"
# if (($? == 0))
# then
         echo $BASE_16_NUM | $AWK -F '#' '{print $2}'
#
# else
#
         echo $BASE_16_NUM
# fi
```

Listing 14-2 (continued)

In Listing 14-2 we have a few things to point out. First, notice the usage function and how we use the extracted name of the shell script directly from the system, specified by SCRIPT_NAME='basename \$0'. This is command substitution using *back tics*, which are located in the upper-left corner of a standard keyboard under the ESC key. Using this technique is equivalent to using the *dollar parentheses* method, specified by \$(command), as we use in most chapters in this book. Notice in the assignment that the basename \$0 command holds the name of the shell script. We always want to query the system for a script name in the main body of the shell script, before it is used in a usage function. If we use the basename \$0 command in a function, the response would be the name of the *function*, not the name of the shell script. We never want to hard-code the script name, because someone may change the name of the shell script in the future.

The next thing that I want to point out is the change made to the regular expression. Before, we were testing for a binary number, which can consist of only 0 and 1. This time we are testing for a decimal number, which can consist of only numbers 0 through 9. The new regular expression is shown here:

```
case $1 in
+([0-9])) BASE_10_NUM=$1
    ;;
    *) echo "\nERROR: $1 is NOT a base 10 number"
    usage
    exit 1
    ;;
esac
```

In this case statement we are testing the ARG[1] variable, represented by the \$1 positional parameter. This regular expression will assign \$1 to the BASE_10_NUM variable only if the characters are numbers between 0 and 9. If any other character is found in this character string, an ERROR message is displayed and the usage function is called before the script exits with a return code of 1, one.

In the assignment of the base 10 number to the BASE_16_NUM variable, notice the change in the variable assignment as shown here:

```
BASE_16_NUM=10#$BASE_10_NUM
```

Notice that the \$BASE_10_NUM variable is preceded by number base representation, 10#. Because this is a decimal number we really did not need to do this because base 10 is implied, but to be consistent it was added.

The last thing that I want to point out is the optional code at the end of the shell script in Listing 14-2. If you are using this shell script with other programs or shell scripts to produce number base conversions, this optional code strips out the number base prefix. Look at the code segment shown here:

```
# Strip out the base prefix and the pound sign (#). (Optional)
#
# Set up the correct awk usage. Solaris needs to
# use nawk instead of awk.
#
# case $(uname) in
# SunOS) AWK="nawk"
```

```
#
       ;;
#
    *) AWK="awk"
#
       ;;
#esac
#
# echo $BASE_16_NUM | grep -q "#"
#
# if (($? == 0))
# then
# echo $BASE_16_NUM | $AWK -F '#' '{print $2}'
# else
#
       echo $BASE_16_NUM
# fi
```

This code is commented out, but let's look at what it does. The purpose is to remove the base number prefix and leave only the number alone, which implies that you must have some built-in logic to know the number base in which the number is represented. The first step is to test for the existence of a pound sign (#). We do this by printing the variable with the echo command and piping the output to a grep statement, using the quiet option -q. This command option does not produce any output to the screen, but we test the return code to see if a pattern match was made. If the return code is 0, zero, then a match was made and there is a pound sign in the string. Because we want to display everything *after* the pound sign (#) we use this pound sign as a field separator. To split the string and leave only the number, which will be in the second field now, we can use either **cut** or **awk**. Let's use awk for a change of pace. To do field separation with awk, we use the -F switch, followed by the character(s) that represents a separation of the fields, which is the # here. Then we just print the second field, specified by the \$2 positional parameter, and we are left with the number alone. If you use this optional code segment, always remember to keep track of the current number base and comment out the echo statement that precedes this code block.

Script to Create a Software Key Based on the Hexadecimal Representation of an IP Address

With the techniques learned in the last two shell scripts let's actually do something that is useful. In this section we are going to create a shell script that will create a software license key based on the IP address of the machine. To tie the license key to the machine and the software we are going to convert each set of numbers in the machine's IP address from decimal to hexadecimal. Then we are going to combine all of the hexadecimal numbers together to make a license key string. This is pretty primitive, but it is a good example for using base conversions. Again, let's look at the code and go through the details at the end. The mk_swkey.ksh shell script is shown in Listing 14-3.

```
#!/usr/bin/ksh
#
# SCRIPT: mk_swkey.ksh
# AUTHOR: Randy Michael
```

Listing 14-3 mk_swkey.ksh shell script

```
# DATE: 07/07/2007
# REV: 1.2.P
#
# PURPOSE: This script is used to create a software
        license key based on the IP address of the
#
#
         system that this shell script is executed on.
         The system is queried for the system's IP
#
#
         address. The IP address is stripped of the
#
         dots (.), and each number is converted to
         hexadecimal. Then each hex string is combined
#
         into a single hex string, which is the software
#
#
         license key.
#
# REV LIST:
#
# set -x # Uncomment to debug this script
# set -n # Uncomment to check command syntax without any execution
*****
# DEFINE FILES AND VARIABLES HERE
******
# Set up the correct echo command usage. Many Linux
# distributions will execute in Bash even if the
# script specifies Korn shell. Bash shell requires
# we use echo -e when we use n, c, etc.
case $SHELL in
*/bin/Bash) alias echo="echo -e"
          ;;
esac
# Set up the correct awk usage. Solaris needs to
# use nawk instead of awk.
case $(uname) in
SunOS) AWK="nawk"
     ;;
   *) AWK="awk"
      ;;
esac
*****
# DEFINE FUNCTIONS HERE
*****
function convert_base_10_to_16
{
```

Listing 14-3 (continued)

```
# set -x # Uncomment to debug this function
typeset -i16 BASE_16_NUM
BASE_10_NUM=$1
BASE_16_NUM=$((10#${BASE_10_NUM}))
# Strip the number base prefix from the hexadecimal
# number. This prefix is not needed here.
echo $BASE_16_NUM | grep -q '#'
if (($? == 0))
then
       echo $BASE_16_NUM | awk -F '#' '{print $2}'
else
       echo $BASE_16_NUM
fi
}
******
# BEGINNING OF MAIN
****
# Query the system for the IP address using the "host $(hostname)"
# command substitution.
IP=$(host $(hostname) | awk '{print $3}' | awk -F ',' '{print $1}')
# Field delimit the IP address on the dots (.) and assign each
# number to a separate variable in a "while read" loop.
echo $IP | $AWK -F '.' '{print $1, $2, $3, $4}'\
| while read a b c d junk
do
    # Convert each of the numbers in the IP address
    # into hexadecimal by calling the "convert_base_10_to 16"
    # function.
    FIRST=$(convert base 10 to 16 $a)
    SECOND=$(convert_base_10_to_16 $b)
    THIRD=$(convert_base_10_to_16 $c)
    FORTH=$(convert_base_10_to_16 $d)
done
# Combine all of the hexadecimal strings into a single
# hexadecimal string, which represents the software key.
echo "${FIRST}${SECOND}${THIRD}${FORTH}"
```

Listing 14-3 (continued)

In the script in Listing 14-3 we are actually doing something useful — at least, if you license your own software this script is useful. To start this shell script off we converted the base-10-to-base-16 code into a function called convert_base_10_to_16. This allows us to call the function four times, one for each piece of the IP address. In this function I want you to notice that we typeset the BASE_16_NUM variable to base 16 at the top of the function, as opposed to the bottom in the previous shell script. It does not make any difference *where* it is set as long as it is set before the value is returned to the main body of the shell script, or displayed.

Also, the optional code segment that was commented out on Listing 14-2 is now used in this shell script. In this case we know that we are converting to a hexadecimal number, and we do not want the number base prefix to appear in the software license key. We use the following code segment to remove the prefix from the output:

Notice the silent execution of the grep command using the -q command switch. Then, if a pound sign is found, the awk statement uses the # as a field delimiter by specifying the -F '#' switch, and then the second field is extracted. This is the value that is returned back to the main body of the shell script from the conversion function.

At the BEGINNING_OF_MAIN we start out by querying the system for the system's IP address using the following command:

IP=\$(host \$(hostname) | awk '{print \$3}' | awk -F ',' '{print \$1}')

Let's step through each part of this command. On my system I pulled an IP address out of the air for this demonstration. On the yogi machine the command substitution host \$(hostname) results in the following output:

```
[root:yogi]@/scripts# host $(hostname)
yogi is 163.155.204.42,
```

From this output you can see that the IP address is located in the third field, 163.155.204.42,. Notice that I have an extra comma (,) tacked on to the end of the IP address, which we do not want included. After we extract the third field from the command output we pipe this result to an awk statement (the cut command will do the same thing). In the awk part of the statement we set the field delimiter to the comma (,) that we want to get rid of, using the -F ', ' notation. Now that the string

is field-delimited on the comma we just extract the first field, which is the IP address alone. This result is then assigned to the IP variable using command substitution.

Now that we have the whole IP address we can chop it up into a series of four numbers. Once we have four individual numbers we can convert each of the decimal numbers into their hexadecimal equivalent. To separate the IP address into separate individual numbers we can use cut, sed, or awk. For consistency let's keep using awk. This time we field-delimit the string, which is an IP address, using the dots (.) and then use the print argument for the awk command to print each of the four fields. At this point we are left with the following four numbers:

163 155 204 42

Now we have some numbers to work with. Because we want to work on each number individually, we pipe this output to a while read loop and assign each of the four numbers to a separate variable. Then, inside of the loop, we convert each decimal number into hexadecimal. The entire command statement is shown here:

echo \$IP | \$AWK -F '.' '{print \$1, \$2, \$3, \$4}' | while read a b c d junk

Notice the final variable at the end of the while statement, junk. I added this as a catchall for anything that may be tacked on to the previous pipe outputs. It does not matter if there is anything to capture, but if there are "extra" fields the junk variable will catch everything remaining in the output. Other than this, each of the four fields is stored in the variables a, b, c, and d.

Inside of the while read loop we call the conversion function four times, once for each variable, as shown here:

```
FIRST=$(convert_base_10_to_16 $a)
SECOND=$(convert_base_10_to_16 $b)
THIRD=$(convert_base_10_to_16 $c)
FOURTH=$(convert_base_10_to_16 $d)
```

The result of these four function calls is the assignment of the hexadecimal values to four new variables, FIRST, SECOND, THIRD, and FOURTH. Now that we have the hexadecimal values all we need to do is combine the hex strings into a single string. The combination is shown here:

```
echo "${FIRST}${SECOND}${THIRD}${FOURTH}"
```

The resulting output from the IP address that I pulled out of the air for temporary use (163.155.204.42) is shown next:

```
[root:yogi]@/scripts# ./mk_swkey.ksh
a39bcc2a
```

This hexadecimal string is the software license that is tied to the IP address.

Script to Translate between Any Number Base

So far we have been working in a restricted environment with limited ability to switch between number bases. This script will convert *any* number to *any* number base within the limits of the system. The base conversion's availability is to base 36 at least, and some systems may go higher. I am not sure what you would do with a base-36 number, but you can make one if you want to.

In this script we rely on two command-line switches, each requiring an argument, and the "number" to convert. The two switches are -f {Starting Number Base, or From:} and -t {Ending Number Base, or To:}. These two parameters tell the shell script what number base we are converting from and what number base we want to convert the number *to*, which is where the -f and -t command-line switches came from. This is another shell script that needs to be presented first, and we will cover the details at the end. The equate_any_base.ksh shell script is shown in Listing 14-4.

```
#!/usr/bin/ksh
#
# SCRIPT: equate_any_base.ksh
# AUTHOR: Randy Michael
# DATE: 07/07/2007
# REV: 1.2.P
# PURPOSE: This script is used to convert a number to any
         supported number base, which is at least base 36.
#
          This script requires that two command-line
#
         arguments and the "number" to be converted
#
          are present on the command line. An example
#
          number base conversion is shown here:
#
#
          equate_any_base.ksh -f16 -t2 e245c
#
          2#1110001001001011100
#
#
          This example converts the base 16 number, e245c, to
#
          the base 2 equivalent, 2#1110001001001011100.
#
          The 2#, which precedes the binary number, shows
#
          the base of the number represented.
#
# EXIT CODES:
#
                0 - Normal script execution
#
                1 - Usage error
# set -x # Uncomment to debug this shell script
# set -n # Uncomment to check syntax without any execution
```

Listing 14-4 equate_any_base.ksh shell script

```
******
# DEFINE FILES AND VARIABLES HERE
*****
SCRIPT_NAME=$(basename $0)
COUNT=0
MAX_COUNT=$#
# Set up the correct echo command usage. Many Linux
# distributions will execute in Bash even if the
# script specifies Korn shell. Bash shell requires
# we use echo -e when we use n, c, etc.
case $SHELL in
*/bin/Bash) alias echo="echo -e"
         ;;
esac
*****
# DEFINE FUNCTIONS HERE
*****
function usage
{
echo "\n\t***USAGE ERROR***"
echo "\nPURPOSE: This script converts between number bases"
echo "\nUSAGE: $SCRIPT_NAME -f{From base#} -t{To base#} NUMBER"
echo "\nEXAMPLE: $SCRIPT NAME -f16 -t10 fc23"
echo "\nWill convert the base 16 number fc23 to its"
echo "decimal equivalent base 10 number 64547"
echo "\n\t ...EXITING...\n"
}
****
# CHECK COMMAND-LINE ARGUMENTS HERE
****
# The maximum number of command-line arguments is five
# and the minimum number is three.
if (($# > 5))
then
      echo "\nERROR: Too many command-line arguments\n"
      usage
      exit 1
elif (($# < 3))
then
      echo "\nERROR: Too few command-line arguments\n"
```

Listing 14-4 (continued)

```
usage
        exit 1
fi
# Check to see if the command-line switches are present
echo $* | grep -q '\-f' || (usage; exit 1)
echo $* | grep -q '\-t' || (usage; exit 1)
# Use getopts to parse the command-line arguments
while getopts ":f:t:" ARGUMENT
do
  case $ARGUMENT in
    f) START_BASE="$OPTARG"
        ;;
    t) END_BASE="$OPTARG"
        ;;
    \?) usage
       exit 1
        ;;
  esac
done
# Ensure that the START_BASE and END_BASE variables
# are not NULL.
if [ -z "$START_BASE" ] || [ "$START_BASE" = '' ] \
   || [ -z "$END_BASE" ] || [ "$END_BASE" = '' ]
then
     echo "\nERROR: Base number conversion fields are empty\n"
    usage
    exit 1
fi
# Ensure that the START_BASE and END_BASE variables
# have integer values for the number base conversion.
case $START_BASE in
+([0-9])) : # Do nothing - Colon is a no-op.
           ;;
        *) echo "\nERROR: $START_BASE is not an integer value"
           usage
           exit 1
           ;;
esac
case $END_BASE in
```

Listing 14-4 (continued)

```
+([0-9])) : # Do nothing - Colon is a no-op.
          ;;
       *) echo "\nERROR: $END_BASE is not an integer value"
          usage
          exit 1
          ;;
esac
****
# BEGINNING OF MAIN
****
# Begin by finding the BASE_NUM to be converted.
# Count from 1 to the max number of command-line arguments
while ((COUNT < MAX_COUNT))
do
   ((COUNT == COUNT + 1))
  TOKEN=$1
  case $TOKEN in
     -f) shift; shift
         ((COUNT == COUNT + 1))
         ;;
     -f${START_BASE}) shift
         ;;
     -t) shift; shift
         ((COUNT == COUNT + 1))
         ;;
     -t${END_BASE}) shift
           ;;
     *) BASE_NUM=$TOKEN
        break
        ;;
  esac
done
# Typeset the RESULT variable to the target number base
typeset -i$END_BASE RESULT
# Assign the BASE_NUM variable to the RESULT variable
# and add the starting number base with a pound sign (#)
# as a prefix for the conversion to take place.
# NOTE: If an invalid number is entered a system error
# will be displayed. An example is inputting 1114400 as
# a binary number, which is invalid for a binary number.
```

Listing 14-4 (continued)

RESULT="\${START_BASE}#\${BASE_NUM}"
Display the result to the user or calling program.
echo "\$RESULT"
End of script...

Listing 14-4 (continued)

Please stay with me here! This script in Listing 14-4 is really not as difficult as it looks. Because we are requiring the user to provide command-line arguments we need to do a lot of testing to ensure that we have good data to work with. We also need to give the user good and informative feedback if a usage error is detected. Remember: always let the user know what is going on. Keeping the user informed is just good script writing!

Let's start at the top of the equate_any_base.ksh shell script and work our way through the details. The first thing we do is to define three variables. The \$SCRIPT_NAME variable points to the name of this shell script. We need to query the system for this script name for two reasons. First, the name of the script may change in the future; second, the SCRIPT_NAME variable is used in the usage function. If we had executed the basename \$0 command inside of the usage function we would get usage instead of the name of the shell script. This is an important point to make. We need to know where the scope lies when referring to positional parameters, \$0 in this case. When we refer to positional parameters in the main body of a shell script, the positional parameters are command-line arguments, including the name of the shell script. When we refer to positional parameters inside of a function, the scope of the positional parameters lies with the arguments supplied to the function, not the shell script. In either case, the name of the shell script, or function, can be referenced by the basename \$0 command.

The next two variable definitions, COUNT=0 and MAX_COUNT=\$#, are to be used to parse through each of the shell script's command-line arguments, where \$# represents the total number of command-line arguments of the shell script. We will go into more detail on these two variables a little later.

In the next section we define any functions that we need for this shell script. For this shell script we need just a usage function. If you look at this usage function, though, we have a good deal of information to describe how to use the shell script in Listing 14-4. We state the purpose of the shell script followed by the USAGE statement. Then we supply an example of using the shell script. This really helps users who are not familiar with running this script.

As I stated before, we need to do a lot of checking because we are relying on the user to supply command-line arguments for defining the execution behavior. We are going to do seven independent tests to ensure that the data we receive is good data that we can work with.

The first two tests are to ensure that we have the correct number of command-line arguments. For the equate_any_base.ksh shell script the user may supply as few

as three arguments and as many as five arguments. This variation may sound a little strange, but when we go to the **getopts** command it will be intuitively obvious. For testing the number of arguments we just use an if...then...elif...fi structure where we test the \$# shell parameter to make sure that the value is not greater than five and is not less than three, as shown here:

```
if (($# > 5))
then
            echo "\nERROR: Too many command-line arguments\n"
            usage
            exit 1
elif (($# < 3))
then
            echo "\nERROR: Too few command-line arguments\n"
            usage
            exit 1
fi</pre>
```

Using getopts to Parse the Command Line

Now we get to use getopts to parse through each command-line switch and its arguments. The getopts command recognizes a command switch as any character that is preceded by a hyphen (-) — for example, -f and -t. The getopts command really does not care *what* is on the command line, unless it is a command switch or its argument. Let's look at a couple of examples of command-line arguments so I can clear the mud.

Example 5: Correct Usage of the equate_any_base.ksh Shell Script

[root:yogi]@/scripts# ./equate_any_base.ksh -f 2 -t16 10110011110101

Notice in Example 5 the use of the two command switches, -f 2 and -t16. Both of these are valid because getopts does not care if there is a space or no space between the switch and the switch-argument, and the order of appearance does not matter either. As you can see in Example 5, we can have as few as three command-line arguments if no spaces are used, or as many as five if both command switches have a space between the command switch and the switch-argument.

Example 6: Incorrect Usage of the equate_any_base.ksh Shell Script

[root:yogi]@/scripts# ./equate_any_base.ksh -i -f 2 -t 16 10110011110101

In Example 6 we have an error condition in two different ways. The first error is that there are six command-line arguments given to the equate_any_base.ksh shell script. The second error is that there is an undefined command switch, -i, given on the command line. This is a good place to go through using getopts to parse a defined set of command-line switches and arguments.

The purpose of the getopts command is to process command-line arguments and check for valid options. The getopts command is used with a while loop and has an enclosed case statement to let you take action for each correct and incorrect argument found on the command line. We can define command-line switches to require an argument, or the switch can be defined as a standalone command switch. The order of the switch does not matter, but if the switch is defined to require an argument, the switch-argument must follow the switch, either with or without a space. When getopts finds a switch that requires an argument, the argument is always assigned to a variable called OPTARG. This variable allows you to assign the switch-argument value to a useful variable name to use in the shell script. Let's look at the getopts definition that is used in this shell script:

```
while getopts ':f:t:' ARGUMENT
do
    case $ARGUMENT in
    f) START_BASE="$OPTARG"
    ;;
    t) END_BASE="$OPTARG"
    ;;
    (?) usage
    exit 1
    ;;
    esac
done
```

There are two parts to the getopts definition. The first is the while loop that contains the getopts statement, and the second is the case statement that allows you to do something when a valid or invalid switch is found. In the while loop we have defined two valid command switches, -f and -t. When you define these you do not add the hyphen (-) in the case statement, but it is required on the command line. Notice the colons (:) in the definitions. The beginning colon specifies that when an undefined switch is found — for example, -i — then the invalid switch is matched with the question mark (?) in the case statement. In our case we always run the usage function and immediately exit the shell script with a return code of 1, one. Also notice that we *escaped* the ? with a backslash (\?). By escaping the ? character (\?), we can use the ? as a regular character without any special meaning or function.

When a colon (:) is present *after* a switch definition, it means that the switch must have an argument associated with it. If the switch definition does *not* have a colon after it, the switch has no argument. For example, the statement getopts ":t:f:i" defines -t and -f as command-line switches that require an argument and -i as a switch that has no argument associated with it.

When a switch is found, either defined or undefined, it is assigned to the ARGUMENT variable (you can use any variable name here), which is used by the case statement. For defined variables we need a match in the case statement, but for undefined switches the ARGUMENT is assigned ? *if* the getopts definition begins with a colon (:). Additionally, when a defined switch is found that requires an argument, the argument to the switch is assigned to the OPTARG variable (you cannot change this

variable name) during the current loop iteration. This is the mechanism that we use to get our *from* and *to* number base definitions, START_BASE and END_BASE, for the equate_any_base.ksh shell script.

Continuing with the Script

As I stated before, getopts does not care what is on the command line if it is not a command switch or a switch argument. So, we need more sanity checks. The next test is to ensure that both -f and -t command-line switches are present on the command line as arguments. We must also check to ensure that the START_BASE and END_BASE variables are not empty and also make sure that the values are integers. We can do all of these sanity checks with the code segment in Listing 14-5.

```
# Check to see if the command-line switches are present
echo $* | grep -q '\-f' || (usage; exit 1)
echo $* | grep -q '\-t' || (usage; exit 1)
# Use getopts to parse the command-line arguments
while getopts ":f:t:" ARGUMENT
do
  case $ARGUMENT in
    f) START_BASE="$OPTARG"
        ;;
     t) END_BASE="$OPTARG"
        ;;
    \?) usage
        exit 1
        ;;
  esac
done
# Ensure that the START_BASE and END_BASE variables
# are not NULL.
if [ -z "$START_BASE" ] || [ "$START_BASE" = '' ] \
   || [ -z "$END_BASE" ] || [ "$END_BASE" = '' ]
then
     echo "\nERROR: Base number conversion fields are empty\n"
    usage
     exit 1
fi
# Ensure that the START_BASE and END_BASE variables
# have integer values for the number base conversion.
```

Listing 14-5 Code segment to verify number base variables

```
case $START_BASE in
+([0-9])) : # Do nothing - Colon is a no-op.
           ;;
        *) echo "\nERROR: $START_BASE is not an integer value"
           usage
           exit 1
           ;;
esac
case $END BASE in
+([0-9])) : # Do nothing - Colon is a no-op.
           ;;
        *) echo "\nERROR: $END BASE is not an integer value"
           usage
           exit 1
           ;;
esac
```

Listing 14-5 (continued)

Starting at the top in Listing 14-5, we first check to ensure that both -f and -t are present as command-line arguments. Next the getopts statement parses the command line and populates the START_BASE and END_BASE variables. After getopts we test the START_BASE and END_BASE variables to ensure that they are not NULL. When you do have NULL value tests always remember to use double quotes ("\$VAR_NAME") around the variable names, or you will get an error if they are actually empty, or NULL. This is one of those hard-to-find errors that can take a long time to track down.

In the next two case statements we use a regular expression to ensure that the \$START_BASE and \$END_BASE variables are pointing to integer values. If either one of these variables is not an integer, we give the user an informative error message, show the correct usage by running the usage function, and exit the shell script with a return code of 1, one.

Beginning of Main

At this point we have confirmed that the data that was entered on the command line is valid, so let's do our number base conversion. Because we have all of the command switches and switch-arguments on the command line, we actually need to *find* the "number" that is to be converted between bases. To find our number to convert we need to scan all of the command-line arguments, starting with the argument at \$1 and continuing until the number is found, or until the last argument, which is pointed to by the \$# shell variable.

Scanning the command-line arguments and trying to find the "number" is a little tricky. First, the "number" may be in any valid number base that the system supports, so we may have alphanumeric characters. But we do have one thing going for us: we know the command switches and the integer values of the \$START_BASE and \$END_BASE variables. We still need to consider that there may or may not be spaces between the command switches and the switch-arguments. Let's think about this a

minute. If a single command-line argument is one of the command switches, we know that the user placed a space between the command switch and the switch-argument. On the other hand, if a single command-line argument is a command-line switch *and* its switch-argument, we know that the user does *not* place a space between the command switch and the switch-argument. By using this logic we can use a simple case statement to test for these conditions. When we get to a command-line argument that does not fit this logic test, we have found the "number" that we are looking for.

Look at the code segment in Listing 14-6, and we will go into a little more detail at the end.

```
# Count from 1 to the max number of command-line arguments
while ((COUNT < MAX COUNT))
do
   ((COUNT == COUNT + 1))
  TOKEN=$1
   case $TOKEN in
     -f) shift; shift
         ((COUNT == COUNT + 1))
            ;;
      -f${START_BASE}) shift
            ;;
      -t) shift; shift
         ((COUNT == COUNT + 1))
            ;;
      -t${END_BASE}) shift
           ;;
      *) BASE NUM=$TOKEN
        break
         ;;
   esac
done
```

Listing 14-6 Code segment to parse the command line

Remember that at the beginning of the shell script we defined the variables COUNT=0 and MAX_COUNT=\$#. Now we get a chance to use them. I also want to introduce the **shift** command. This Korn shell built-in allows us to always reference the \$1 command-line argument to access *any* argument on the command line. To go to the next command-line argument we use the shift command to make the next argument, which is \$2 here, shift over to the \$1 position parameter. If we want to shift more than one position we can either execute multiple shift commands or just add an integer value to the shift command to indicate how many positions that we want to shift to the \$1 position. Both of the following commands shift positional parameters two positions to the \$1 argument:

shift; shift shift 2

The idea in our case statement is to do *one* shift if a command-line switch *with* its switch-argument is found at \$1 and to shift *two* positions if a command-line switch is found alone. We start with a while loop and increment a counter by one. Then we use the TOKEN variable to always grab the value in the \$1 position. We make the test to check for a command-line switch alone or a command-line switch plus its switch argument. If the \$1 positional parameter contains either of these, we shift accordingly. If the test is not matched, we have found the number that we are looking for. So, this is really not that difficult a test when you know what the goal is.

When we have found the "number," which is assigned to the BASE_NUM variable, we are ready to do the conversion between number bases. We do the conversion as we did in the previous shell scripts in this chapter, except that this time we use the variable assignments of the START_BASE and END_BASE variables as number bases to start at and to end with, as shown in the next command statement:

```
RESULT="${START_BASE}#${BASE_NUM}"
```

Let's assume that the \$START_BASE variable points to the integer 2, and the \$BASE_NUM variable points to the binary number 1101101011. Then the following command statement is equivalent to the preceding statement:

```
RESULT="2#1101101011"
```

The next step is to typeset the BASE_TO variable to the target number base. This is also accomplished using the previously defined variable END_BASE, as shown here:

typeset -i\$END_BASE RESULT

Now let's assume that the target number base, \$END_BASE, is 16. The following command statement is equivalent to the preceding variable statement:

typeset -i16 RESULT

The only thing left to do is print the result to the screen. You can use echo, print, or printf to display the result. I still like to use echo, so this is the final line of the shell script:

echo \$RESULT

An Easy, Interactive Script to Convert Between Bases

The script is Listing 14-7 is a simple, interactive shell script that prompts the user for the input number in any base between base 2 and base 36. It then prompts the user for the number base to convert the number to. Study the chg_base.ksh shell script in Listing 14-7 and we will step through the script at the end.

```
#!/bin/ksh
#
# SCRIPT: chg_base.ksh
# AUTHOR: Randy Michael
# DATE: 10/4/2007
# REV: 1.1.A
#
# PURPOSE: This script converts numbers between base
# 2 through base 36. The user is prompted for input.
#
# NOTE: Numbers are in the following format:
#
#
   base#number
#
# EXAMPLE: 16#264bf
#
*****
# DEFINE FILES AND VARIABLES HERE
****
# Set up the correct awk usage. Solaris needs to
# use nawk instead of awk.
case $(uname) in
SunOS) AWK="nawk"
     ;;
   *) AWK="awk"
     ;;
esac
# Set up the correct echo command usage. Many Linux
# distributions will execute in Bash even if the
# script specifies Korn shell. Bash shell requires
# we use echo -e when we use n, c, etc.
case $SHELL in
*/bin/Bash) alias echo="echo -e"
         ;;
esac
*****
# BEGINNING OF MAIN
****
# Prompt the user for an input number with base
```

Listing 14-7 chg_base.ksh shell script

```
echo "\nInput the number to convert in the following format:
Format: base#number
Example: 16#264BF \n"
read ibase_num
# Extract the base from the ibase_num variable
ibase=$(echo $ibase_num | $AWK -F '#' '{print $1}')
# Test to ensure the input base is an integer
case $ibase in
  [0-9]*) : # do nothing
          ;;
       *) echo "\nERROR: $ibase is not a valid number for input base\n"
          exit 1
          ;;
esac
# Test to ensure the input base is between 2 and 36
if (( ibase < 2 || ibase > 36 ))
then
    echo "\nERROR: Input base must be between 2 and 36\n"
    exit 1
fi
# Ask the user for the output base
echo "\nWhat base do you want to convert $ibase_num to?
NOTE: base 2 through 36 are valid\n"
echo "Output base: \c"
read obase
# Test to ensure the output base is an integer
case $obase in
  [0-9]*) : # do nothing
          ;;
       *) echo "\nERROR: $obase is not a valid number\n"
          exit 1
          ;;
esac
# Test to ensure the output base is between 2 and 36
```

```
Listing 14-7 (continued)
```

```
if (( obase < 2 || obase > 36 ))
then
    echo "\nERROR: Output base must be between 2 and 36\n"
    exit 1
fi

# Save the input number before changing the base
in_base_num=$ibase_num
# Convert the input number to the desired output base
typeset -i$obase ibase_num
# Assign the output base number to an appropriate variable name
obase_num=$ibase_num
# Display the result
echo "\nInput number $in_base_num is equivalent to $obase_num\n"
```

Listing 14-7 (continued)

The key to the chg_base.ksh shell script in Listing 14-7 is that both the input and output numbers are represented in *base#number* notation as we have previously done, unless, of course, it is a base-10 number. Base 10 is implied by default, so the *base#* part is not displayed, nor is it required as input. The script accepts both notations.

Starting from the top of the chg_base.ksh shell script in Listing 14-7, we first set up the correct awk command usage. If we use awk -F to define a field delimiter on Solaris, this command will fail. For Solaris, we must use nawk in place of awk. The following code segment sets up the correct command usage:

```
# Set up the correct awk usage. Solaris needs to
# use nawk instead of awk.
case $(uname) in
SunOS) AWK="nawk"
    ;;
    *) AWK="awk"
    ;;
esac
```

Next we need to query the system for the executing \$SHELL, so we use the correct syntax when using the echo command with backslash operators, such as n, c, b, and so on when executing in different shells. Many Linux distributions will execute in a Bash shell even though we specify Korn shell on the very first line of the script. Bash shell requires the use of the echo – e switch to enable the backslash operators. Korn shell, by default, recognizes the backslash operators. Listing 14-8 shows the case statement

to alias the echo command to echo –e if the executing shell is */bin/Bash. Now, when we need to use the echo command, we are assured it will display text correctly.

```
# Set up the correct echo command usage. Many Linux
# distributions will execute in Bash even if the
# script specifies Korn shell. Bash shell requires
# we use echo -e when we use \n, \c, etc.
case $SHELL in
*/bin/Bash) alias echo="echo -e"
;;
esac
```

Listing 14-8 case statement to set proper echo usage

With the correct usage set for the echo command, we now just prompt the user for the input number, specifying the base#number format. Next we extract the number base from the input number and test for an integer with the regular expression +([0-9]) in a case statement.

NOTE Bash shell does not support the regular expression + ([0-9]).

Next we ensure that the number base value is between the shell-supported values base 2 through base 36.

We next prompt the user for the base to convert the input number to using the read OBASE statement. Then we perform tests to ensure \$OBASE is an integer and the value is greater than 1 and less than 37.

With these tests complete, we are ready to do the number base conversion, but first we save the input number to the in_base_num variable. Next, using the typeset command, the number base conversion is done, as shown here:

```
typeset -i$obase ibase_num
```

We then assign the new base to a more appropriate variable:

obase_num=\$ibase_num

Then display the result to the user:

\$ECHO "\nInput number \$in_base_num is equivalent to \$obase_num\n"

To see the chg_base.ksh shell script in action, check out Listing 14-9.

```
[root:yogi]@/scripts# ./chg_base.ksh
Input the number to convert in the following format:
Format: base#number
Example: 16#264BF
8#72544100474
What base do you want to convert 8#72544100474 to?
NOTE: base 2 through 36 are valid
Output base: 2
Input number 8#72544100474 is equivalent to
2#11101010110010000100000100111100
[root:yogi]@/scripts# ./chg_base.ksh
Input the number to convert in the following format:
```

Listing 14-9 chg_base.ksh shell script in action

base#number
Example: 16#264BF
22#1ahk3df3b
What base do you want to convert 22#1ahk3df3b to?
NOTE: base 2 through 36 are valid
Output base: 10
Input number 22#1ahk3df3b is equivalent to 81850831953
[root:yogi]@/scripts#

Listing 14-9 (continued)

Listing 14-9 gives two examples of using the interactive chg_base.ksh shell script. Notice that an output in base 10, by default, does not show the base in base#number notation because base 10 is implied. If you enter a base 10 number as an input number, you can omit the 10# part of the number, because base 10 is, again, implied by default. The script supports both methods of notation for base 10.

Using the bc Utility for Number Base Conversions

So far in this chapter we have been doing everything in Korn shell. The reason is simple: Bash does not support the typeset -ibase VAR notation for number base conversions. Another reason is that Bash does not support the regular expression +([0-9]) that we use to test for an integer. We can still convert number bases in Bash shell, however, by using the bc utility — the calculator program we studied in Chapter 13, "Floating-Point Math and the bc Utility." Testing for the integer? We will use a different technique.

The bc utility uses two internal variables to define input and output number bases: ibase and obase. In both cases bc converts the number to decimal. So, to convert between bases, we must first convert to decimal and then convert to the desired output base. Additionally, bc only supports number bases between base 2 and base 16.

The chg_base_bc.Bash shell script in Listing 14-10 uses the same framework as the chg_base.ksh shell script in Listing 14-7. Study the chg_base_bc.Bash shell script in Listing 14-10 and we will cover the details at the end.

```
#!/bin/Bash
#
# SCRIPT: chg_base_bc.Bash
# AUTHOR: Randy Michael
# DATE: 10/4/2007
# REV: 1.1.A
#
# PURPOSE: This script converts numbers between base
# 2 through base 16 using the bc utility. The user
# is prompted for input.
#
# NOTE: Numbers are in the following format:
#
   base#number
#
#
# EXAMPLE: 16#264bf
*****
# DEFINE FILES AND VARIABLES HERE
*****
# Set up the correct awk usage. Solaris needs to
# use nawk instead of awk.
case $(uname) in
SunOS) AWK="nawk"
     ;;
   *) AWK="awk"
     ;;
esac
****
# BEGINNING OF MAIN
*****
# Prompt the user for an input number with base
echo -e "\nInput the number to convert in the following format:
Format: base#number
Example: 16#264BF \n"
read IBASE_NUM
```

Listing 14-10 chg_base_bc.Bash shell script

```
# Extract the base and the number from the ibase_num variable
IBASE=$(echo $IBASE_NUM | $AWK -F '#' '{print $1}')
INUMBER=$(echo $IBASE_NUM | $AWK -F '#' '{print $2}')
# Test to ensure the input base is between 2 and 16
if (( IBASE < 2 || IBASE > 16 ))
then
    echo -e "\nERROR: Input base must be between 2 and 16\n"
    exit 1
fi
# The bc utility requires all number bases greater
# than 10 use uppercase characters for all
# non-numeric character numbers, i.e. hex numbers.
# We use the tr command to uppercase all lowercase
# characters.
if (( IBASE > 10 ))
then
   INUMBER=$(echo $INUMBER | tr '[a-z]' '[A-Z]')
fi
# Ask the user for the output base
echo -e "\nWhat base do you want to convert $IBASE_NUM to?
NOTE: base 2 through 16 are valid\n"
echo -e "Output base: \c"
read OBASE
# Test to ensure the output base is an integer
case $OBASE in
  [0-9]*) : # do nothing
          ;;
       *) echo -e "\nERROR: $obase is not a valid number\n"
          exit 1
          ;;
esac
# Test to ensure the output base is between 2 and 16
if (( OBASE < 2 || OBASE > 16 ))
then
    echo -e "\nERROR: Output base must be between 2 and 16\n"
    exit 1
fi
```

```
# Save the input number before changing the base
IN_BASE_NUM=$IBASE_NUM
# Convert the input number to decimal
if (( IBASE != 10 ))
then
    DEC_EQUIV=$(echo "ibase=$IBASE; $INUMBER" | bc)
fi
# Convert the number to the desired output base
RESULT=$(echo "obase=$OBASE; $DEC_EQUIV" | bc)
# Display the result
echo -e "\nInput number $IN_BASE_NUM is equivalent to \
$(OBASE)#${RESULT}\n"
```

We start the chg_base_bc.Bash shell script in Listing 14-10 by defining the correct awk command usage. If the UNIX flavor is Solaris, we need to use nawk instead of awk because awk in Solaris does not support the following awk -F syntax, denoted here in boldface text:

```
awk -F : '{print $1}'
```

To get around this little problem, we use gawk instead of awk. The following case statement sets up the correct awk usage:

```
# Set up the correct awk usage. Solaris needs to
# use nawk instead of awk.
case $(uname) in
SunOS) AWK="nawk"
    ;;
    *) AWK="awk"
    ;;
esac
```

Now when we use the \$AWK variable, the correct awk command will execute.

At BEGINNING OF MAIN, we ask the user for an input number using the specific format base#number. The base is the number base of the input number and the number part is the number specified in the number base — for example, 16#264BF. However, for this script we are expecting the base#number notation even for base 10 input numbers.

The next step is to split the number the user entered into two parts: the base part and the number part. This data extraction is shown here:

Extract the base and the number from the ibase_num variable IBASE=\$(echo \$IBASE_NUM | \$AWK -F '#' '{print \$1}') INUMBER=\$(echo \$IBASE_NUM | \$AWK -F '#' '{print \$2}')

Notice here that we use the variable \$AWK to execute the correct awk command based on the UNIX flavor we discovered earlier.

With the two parts of the input number we next verify that the input base is an integer. Remember that Bash shell does not support the regular expression +([0-9)) to test for an integer. Instead, we can test for the input base to be between base 2 and base 16. If we find a base value out of this range, we show the user a usage error and exit — but we still test for a valid value. The test for a valid input number base is shown here:

```
# Test to ensure the input base is between 2 and 16
if (( IBASE < 2 || IBASE > 16 ))
then
        echo -e "\nERROR: Input base must be between 2 and 16\n"
        exit 1
fi
```

After verifying that the input base is in a valid range, we need to uppercase all non-numeric characters in input numbers that are in a base greater than 10, such as hexadecimal numbers. We do this using the **tr** command. We can use the tr command to both uppercase and lowercase text. The uppercase conversion is shown here:

This tr command changes all lowercase characters in INUMBER to uppercase characters. If we had wanted to make all uppercase characters lowercase, we would have swapped the '[a-z]' '[A-Z]' with '[A-Z]' '[a-z]', as shown here:

```
INUMBER=$(echo $INUMBER | tr '[A-Z]' '[a-z]')
```

NOTE The single quotes around the square brackets are required.

Now that we have all the tests complete and we have ensured that any lowercase characters are converted to uppercase characters, we ask the user for the desired output base. Again, we must test the user input to ensure the base is between 2 and 16. This test of the output base is shown here:

```
# Test to ensure the output base is between 2 and 16
if (( OBASE < 2 || OBASE > 16 ))
then
    echo -e "\nERROR: Output base must be between 2 and 16\n"
    exit 1
fi
```

If the base is out of range, we show the user a usage error and exit.

Now come the actual conversions. We first must convert the number to decimal before we convert the number to the desired output base. The following bc command statement converts the input base number to base-10 conversion:

```
# Convert the input number to decimal
if (( IBASE != 10 ))
then
    DEC_EQUIV=$(echo "ibase=$IBASE; $INUMBER" | bc)
fi
```

Of course, if the input base is already base 10, this conversion is not necessary.

With the input number now represented in base 10, we can use bc to convert the number to the desired output base, as shown here:

```
# Convert the number to the desired output base
RESULT=$(echo "obase=$OBASE; $DEC_EQUIV" | bc)
```

The only thing we are doing is to echo the data that bc requires and then piping this output into the bc utility.

The chg_base_bc.Bash shell script is shown in action in Listing 14-11.

```
[root:yogi.tampabay.rr.com]@/scripts# ./chg_base_bc.Bash
Input the number to convert in the following format:
Format: base#number
Example: 16#264BF
16#264bf
```

Listing 14-11 chg_base_bc.Bash shell script in action

```
What base do you want to convert 16#264bf to?
NOTE: base 2 through 16 are valid
Output base: 8
Input number 16#264bf is equivalent to 8#462277
[root:yogi.tampabay.rr.com]@/scripts#
```

The bc utility is a very powerful tool and can handle some very complex arithmetic. For more information, see the bc manual page, man bc, and look back at Chapter 13.

Other Options to Consider

As with all of the scripts in this book, we can always make some changes to any shell script to improve it or to customize the script to fit a particular need.

Software Key Shell Script

To make a software key more complicated, you can hide the hexadecimal representation of the IP address within some pseudo-random numbers, which we studied in Chapters 10, "Process Monitoring and Enabling Pre-Processing, Startup, and Post-Processing Events," and 11, "Pseudo-Random Number and Data Generation." As an example, add five computer-generated pseudo-random numbers as both a prefix and a suffix to the hexadecimal IP address representation. Then to verify the license key in your software program, you can extract the hex IP address from the string. There are several techniques to do this verification, and I am going to leave the details up to you as a little project.

Summary

We went through a lot of variations in this chapter, but we did hit the scripts from different angles. Number base conversion can be used for many purposes, and we wrote one script that takes advantage of the translation. Software keys are usually more complicated than this script example, but I think you get the basic idea.

In the next chapter we are going to create a shell script I like to call hgrep, or *highlighted* grep. This script uses the same command syntax as grep but displays the entire text with the matching pattern highlighted in reverse video. See you in the next chapter!

Lab Assignments

- 1. Write an Expect script to automate the chg_base.ksh shell script in Listing 14-7. The expect script should use two command-line arguments: one for the input number, and the second for the output number base for conversion.
- 2. Write an Expect script to automate the chg_base_bc.Bash shell script in Listing 14-10. The Expect script should use two command-line arguments: one for the input number, and the second for the output number base for conversion.

CHAPTER 15

hgrep: Highlighted grep Script

Ever want to find text in a large file easily? The larger the text file, the more you will appreciate this shell script. We can use reverse video in shell scripts for more than just making pretty menus. What about highlighting text in a file or in a command's output? In this chapter we are going to show an example of using reverse video in a shell script that works similarly to the **grep** command. Instead of displaying the line(s) that match the pattern, we are going to display the entire file, or command output, with the matched pattern highlighted in reverse video. I like to call this **hgrep**.

In the process of creating this shell script, an initial test script was developed that ended up being very complicated. It started by grepping each line for the specified pattern. If the pattern was found in the line, then a scan of the line, character by character, was started to locate the exact pattern in the line for highlighting, then we grepped again for the pattern in remaining lines of text, and so on. This initial code had quite a few problems, other than the complicated nature of the script, caused by UNIX special characters making the output do some very interesting things when scanning shell script code. Regular text files worked fine, but the script was *very* slow to execute.

Then there was the revelation that the **sed** command should somehow be able to handle the pattern matching — and do so a lot faster than parsing the file with a shell script. A shell script is really not meant to work on a file line-by-line and *character by character*; it can be done, but this is what Perl is for! The problem to resolve using sed was how to add in the highlighting control within a sed command statement. After thinking about using sed and command substitution for a while, I had a working script in about 15 minutes (we might have a record!), and the following is what I came up with.

NOTE Since the time I wrote this script for the first edition of *Mastering Unix Shell Scripting* in 2001, the GNU more command now has this capability. Type more filename, then search by pressing the / key and typing the search

pattern followed by Enter. The search will take you to the first occurrence of the pattern and highlight in reverse video, this, and every, pattern match in the file. To shell script a solution, follow along.

Reverse Video Control

There are two commands that control reverse video: **tput smso** turns *soft* reverse video on, and **tput rmso** turns highlighting back off. The tput command has many other options to control the terminal (see Table 15-1 later in this chapter), but **tput sgr0** (sgr-zero) will turn every tput option off. To highlight text, we turn reverse video on, print whatever we want highlighted, and then turn reverse video off. We can also save this output, with the highlighted text, in a file. To display the file with highlighted text, we can use **pg**, or **page**, and on some operating systems **more** will work. The more command diaplayed the characters that make up the *escape sequence* for the highlighted text, not the highlighted text itself. You would see the same result using the vi editor. On Solaris both commands displayed the highlighted text, but not all operating systems have the pg and page commands.

There is one common mistake that will prevent this shell script from working: not double quoting the variables — for example, "\$STRING". The double quotes have no effect on a single-word pattern match, but for multiword string patterns, where there is white space, the variables *must* be double quoted or standard error will produce command usage errors within the script. The errors are due to the fact that each word that makes up the string pattern will be interpreted as a separate argument instead of one entity when the double quotes are present. The double quotes are very important when working with string variables containing white space. Forgetting the double quotes is a very hard error to find when troubleshooting code!

The sed command is next. Remember the basic sed syntax that we use in this book:

sed s/current_string/new_string/g \$FILENAME

In our script, we want to use the sed command statement to redirect output to a file, and then use pg, page, or more to display the file:

```
sed s/current_string/new_string/g $FILENAME > $OUTPUT_FILE
pg $OUTPUT_FILE
    --verse--
more $OUTPUT_FILE
```

To add in the reverse video piece, we have to do some command substitution within the sed statement itself using the tput commands — this is the part that had to be worked out. Where we specify the new_string we will add in the control for reverse video using command substitution, one to turn highlighting on and one to turn it back

off. When the command substitution is added, our sed statement will look like the following:

sed s/current_string/\$(tput smso)new_string\$(tput rmso)/g

In our case, the current_string and new_string will be the same because we only want to highlight existing text without changing it. We also want the string to be assigned to a variable, as in the next command:

```
sed s/"$STRING"/$(tput smso)"$STRING"$(tput rmso)/g
```

Notice the double quotes around the string variable, "\$STRING". Just another reminder: do not forget to add the double quotes around variables with white space!

As an experiment using command substitution, try this next command statement on any UNIX machine:

```
sed s/`hostname`/$(tput smso)`hostname`$(tput rmso)/g /etc/hosts
```

In the preceding command statement, notice that we used both types of command substitution, enclosing the command within back tics, `command`, and the dollar parentheses method, \$(command). The statement will **cat** the /etc/hosts file and highlight the machine's hostname in reverse video each time it appears in the file. Now try the same command, but this time pipe the command to more. Try the same command again using pg and page instead of more, if your machine supports the page commands. If your machine does not have the pg command, the more command should work. If your operating system has both pg and more, notice that using more may not display the string pattern in reverse video — it will display the characters that make up the *escape sequence* that the tput commands create, but Solaris is an exception. We will need to consider this when we display the result on different operating systems.

To make this script have the same look and feel as the grep command, we want to be able to supply input via a file, as a command-line argument, or as standard input from a command pipe. When supplying a filename to the script as a command-line argument, we need to ensure that the file exists, its size is greater than zero bytes, it is readable by this script, and the string pattern is matched in the file. We could leave out the last step, but if the pattern is not in the file it would be nice to let the user know. If we are getting input from standard input instead of a file specified as an argument — for example, cat /etc/hosts | hgrep.Bash `hostname` — we need to check for the string pattern in the output file instead of the input file. Then we can still inform the user if the pattern is not found.

Building the hgrep.Bash Shell Script

Now that we have the basic command syntax, let's build the hgrep. Bash shell script. There are two types of input for this script: file input and standard input. For the file input we need to do some sanity checks so that we don't get standard error

messages from the system. We also want to give the user some feedback if there is something that will cause an error using the specified file as input — for example, the file does not exist or is not readable by the script because of file permissions. The command syntax using the hgrep.Bash script should be the same as the grep command, which is:

grep pattern [filename]

By looking at this we can determine that we will sanity-check the file only when we have two command-line arguments; otherwise, we are using piped-in standard input, which implies that we check the file only when \$# is equal to 2. We begin with checking the command-line arguments and making assignments of the arguments to variables:

```
if (( $# != 1 ))
then
    # Input coming from standard input
    PATTERN="$1" # Pattern to highlight
    FILENAME= # Assign NULL to FILENAME
elif (( $# != 2 ))
then
    # Input coming from $FILENAME file
    PATTERN="$1" # Pattern to highlight
    FILENAME="$2" # File to use as input
else
    # Incorrect number of command-line arguments
    usage
    exit 1
fi
```

We should now have enough to get us started. If we have a single command-line argument, we assign \$1 to PATTERN and assign the FILENAME variable a NULL value. If there are two command-line arguments, we assign \$1 to PATTERN and \$2 to FILENAME. If we have zero or more than two arguments, we display the usage message and exit with a return code of 1, one. The function for correct usage is listed here. Notice we add the -e switch to the echo command, enabling the backslash cursor controls, since this is a Bash shell script.

```
function usage
{
  echo -e "\nUSAGE: $SCRIPT_NAME pattern [filename]\n"
}
```

Follow through the hgrep.Bash script in Listing 15-1, and the process will be explained at the end of the shell script. Note that this is a Bash script, but it will run in Bourne or Korn shells. Just declare your favorite shell.

```
#!/bin/Bash
#
# SCRIPT: hgrep.Bash
# AUTHOR: Randy Michael
# DATE: 07/09/2007
# REV 3.0
# PLATFORM: Not Platform Dependent
# PURPOSE: This script is used to highlight text in a file.
    Given a text string and a file the script will search
#
#
    the file for the specified string pattern and highlight
    each occurrence. For standard input the script pages a
#
#
    temporary file which has the string text highlighted.
#
# REV LIST:
# set -x # Uncomment to debug
# set -n # Uncomment to check script syntax without execution
# EXIT CODES:
       0 ==> Script exited normally
#
       1 ==> Usage error
#
       2 ==> Input File Error
#
#
       3 ==> Pattern not found in the file
# REV LIST:
#
    03/12/2001 - Randy Michael - Sr. Sys. Admin.
#
    Added code to just exit if the string is not in
#
    the target file.
#
#
    03/13/2001 - Randy Michael - Sr. Sys. Admin.
#
    Added code to ensure the target file is a readable
#
    "regular" non-zero file.
#
#
    03/13/2001 - Randy Michael - Sr. Sys. Admin.
    Added code to highlight the text string and filename
#
    in the error and information messages.
#
#
#
    08-22-2001 - Randy Michael - Sr. Sys. Admin
#
    Changed the code to allow this script to accept standard
#
    input from a pipe. This makes the script work more like the
#
    grep command
#
#
    7/02/2007 - Randy Michael - Sr. Sys. Admin
#
    Converted this script to Bash shell.
#
```

```
******
    DEFINE FILES AND VARIABLES HERE
#
****
SCRIPT NAME=$(basename $0)
OUTPUT_FILE="/tmp/highlightfile.out"
>$OUTPUT_FILE
*****
    DEFINE FUNCTIONS HERE
#
****
function usage
{
echo -e "\nUSAGE: $SCRIPT_NAME pattern [filename]\n"
}
*****
±
      CHECK COMMAND SYNTAX
*****
# Input coming from standard input
if (( $# == 1 ))
then
    # Input coming from standard input
    PATTERN="$1" # Pattern to highlight
    FILENAME = # Assign NULL to FILENAME
elif (( $# == 2 ))
then
    # Input coming from $FILENAME file
    PATTERN="$1" # Pattern to highlight
    FILENAME="$2" # File to use as input
    # Does the file exist as a "regular" file?
    if [ ! -f "$FILENAME" ]
    then
      echo -e "\nERROR: $FILENAME does not exist...\n"
     usage
      exit 2
    fi
    # Is the file empty?
```

```
if [ ! -s "$FILENAME" ]
    then
       echo -e "\nERROR: \c"
       tput smso
       echo -e "$FILENAME\c"
       tput sgr0
       echo -e " file size is zero...nothing to search\n"
       usage
       exit 2
    fi
    # Is the file readable by this script?
    if [ ! -r "$FILENAME" ]
    then
       echo -e "\nERROR: \c"
       tput smso
       echo -e "${FILENAME}\c"
       tput sgr0
       echo -e " is not readable to this program...\n"
       usage
       exit 2
    fi
    # Is the pattern anywhere in the file?
    grep -q "$PATTERN" "$FILENAME"
    if (( $? != 0 ))
    then
       echo -e "\nSORRY: The string \c"
       tput smso
       echo -e "${PATTERN}\c"
       tput sgr0
       echo -e " was not found in \\c"
       tput smso
       echo -e "${FILENAME}\c"
       tput sgr0
       echo -e "\n\n....EXITING...\n"
       exit 3
    fi
else
    # Incorrect number of command-line arguments
    usage
    exit 1
fi
*****
#
      BEGINNING OF MAIN
```

```
*****
# There is no $FILENAME if we get input from a pipe...
if [[ ! -z "$FILENAME" && "$FILENAME" != '' ]]
then
    # Using $FILENAME as input
    case $(uname) in
   AIX | HP-UX)
      # This is a fancy "pg" command. It acts similar to the
      # "more" command but instead of showing the percentage
      # displayed it shows the page number of the file
     sed s/"${PATTERN}"/$(tput smso)"${PATTERN}"$(tput sgr0)/g \
     "$FILENAME" | pg -csn -p"Page %d:"
     exit 0
     ;;
    *)
     sed s/"${PATTERN}"/$(tput smso)"${PATTERN}"$(tput sqr0)/g \
     "$FILENAME" | more
     exit 0
     ;;
    esac
else
     # Input is from standard input...
    sed s/"${PATTERN}"/$(tput smso)"${PATTERN}"$(tput sgr0)/g \
        > $OUTPUT_FILE
    # Is the pattern anywhere in the file?
    grep -q "$PATTERN" $OUTPUT_FILE
    if (($? != 0))
    then
       echo -e "\nERROR: The string \c"
       tput smso
       echo -e "${PATTERN}\c"
       tput sgr0
       echo -e " was not found in standard input\c"
       echo -e "\n\n....EXITING...\n"
        exit 3
    fi
fi
# Check the operating system; on AIX and HP-UX we need to
# use the "pg", or "page" command. The "more" command does
# not work to highlight the text; it will only show the
# characters that make up the escape sequence. All
# other operating system usr the "more" command.
```

```
case $(uname) in
AIX |HP-UX)
    # This is a fancy "pg" command. It acts similar to the
    # "more" command but instead of showing the percentage
    # displayed it shows the page number of the file
    cat $OUTPUT_FILE | pg -csn -p"Page %d:"
    ;;
*)
    cat $OUTPUT_FILE | more
    ;;
esac
```

In the shell script in Listing 15-1 we first check for the correct number of command-line arguments; either one or two arguments are valid. Otherwise, the script usage message is displayed, and the script will exit with a return code of 1. If we have the correct number of arguments, we assign the arguments to variables. If we have two command-line arguments, an input file is specified in \$2 — at least it is supposed to be a file. We need to do some sanity checking on this second command-line argument by first checking to see that the file exists as a regular file. We do not want to do anything with the file if it is a block or character special file, a directory, or any other *nonregular* file. Next we make sure that the file is not empty. Then we ensure that the script can read the file, and finally, we grep for the pattern in the file to see if we have anything to highlight. If all the tests are passed, we can proceed.

By checking if the \$FILENAME variable is null, or empty, we know which type of input we are dealing with. A null or empty \$FILENAME variable means we use standard input, which is input from a pipe in this case. If \$FILENAME is not null, we have a file specified as input to the script on the command line.

The difference in handling an input file versus standard input is that we will directly read an input file, perform the sed substitution, and display the file one page at a time. Otherwise, the input is already coming in from a pipe directly into the sed statement and we capture this sed output in a temporary file specified by the *\$OUTPUT_FILE* variable. It's that simple!

We have one more check before displaying the output. We need to grep for "\$PATTERN" in the \$FILENAME or \$OUTPUT_FILE to see if it exists. If not, we display a *string not found* message and exit.

The output display is interesting because the more command will not work on HP-UX or AIX to display the highlighted text. For HP-UX and AIX we use pg instead of more. To determine which flavor of UNIX we are running, we use the **uname** command in a case statement. If the OS is either AIX or HP-UX, we use a *fancy* pg command, which has output that appears similar to the more output. Using pg -csn -p"Page %d:" will display the page number of the file, whereas more displays the percentage of the file. All other UNIX flavors will use more to display the output file.

The script in Listing 15-1 is a good example of how a little ingenuity can greatly simplify a challenge. We sometimes make things more complicated than they need to be, as in my initial test script that parsed through the file line-by-line and character by character, searching for the pattern. We live and learn!

Other Options to Consider

As with every script there is room for improvement or customization, however you want to look at it.

Other Options for the tput Command

The only tput command option that we worked with was the tput smso command, which is used to turn on highlighting. The tput command has many other options to control terminal display. In our example we did a highlight of not only the text but also the surrounding *block* for *each* character. We could also highlight only the text piece, double video the entire text block, or underline with other options — for example, we could have underlined bold text. The tput command is fun to play with. The short list of command options is shown in Table 15-1.

TPUT BELL	RINGS THE BELL
tput blink	Starts blinking mode
tput bold	Starts double intensity (much brighter than reverse video)
tput civis	Turns off the cursor
tput cnorm	Makes the cursor normal again
tput cr	Sends a carriage to the terminal
tput cvvis	Makes the cursor very bright
tput dim	Starts one-half intensity mode
tput ed	Clears to the end of the display
tput el	Clears to the end of the line
tput flash	Sends a visible bell (good for sending a flash to someone's screen)
tput invis	Starts invisible text mode
tput prot	Starts protected mode
tput rc	Restores the last saved cursor position (paved by tput $\ {\tt sc}$)

Table 15-1 Options for the tput Command

(continued)

TPUT BELL	RINGS THE BELL
tput rev	Begins reverse video mode (bright!)
tput rmso	Ends the standout mode (reverses tput smso)
tput rmul	Ends the underline (underscore) mode
tput sc	Saves the cursor position
tput sgr0	Turns off all video modes
tput smso	Starts the standout mode (soft reverse video we used in this chapter)
tput smul	Starts the underline (underscore) mode

Table 15-1 (continued)

Table 15-1 is only an abbreviated listing of the tput command options. As you can see, we can do a lot with the text on the screen. Use your imagination, and play around with the commands.

Summary

In this chapter we introduced using reverse video to highlight text within our output. We also showed how to do command substitution inside a sed command statement. There are many more options for the tput command to control the terminal; for example, we could have underlined the matching pattern. The nice thing about the tput command is that it will let you mix things up, too.

In the next chapter, we are going to study some techniques to monitor one or more processes, with options to wait for processing to start or stop execution, or both. We also allow for pre- and post-events to be defined for the process. I hope you gained some knowledge in this chapter, and every chapter! Please review the lab assignments.

Lab Assignments

- 1. Modify the hgrep.Bash shell script in Listing 15-1 to blink/flash matching patterns on the screen.
- Modify the hgrep.Bash shell script in Listing 15-1 to process standard input directly, without relying on a temporary file for internal script tests and processing. (Hint: Include required tests in one statement.)
- 3. Count the occurrences of pattern matches and give the end user this feedback. How you give this feedback is up to you.

CHAPTER 16

Monitoring Processes and Applications

The most critical part of any business is ensuring that applications continue to run without error. In this chapter, we are going to look at several techniques for monitoring applications and critical processes that the applications rely on. The problem with trying to write this chapter is that there are so many applications in the corporate world that the techniques to monitor them vary widely.

From the lowest level, we can **ping** the machine to see if it is up. A ping, though, is not an operating system response, but rather a machine response that only confirms that the network adapter is configured. Then again, if the machine is in the DMZ, ping is probably disabled at the router. At a higher level, we can look at the processes that are required for the application to run properly, but this too does not completely confirm, 100 percent, that the application is working properly. The only way to ensure the application is working properly is to interact with the application. As an example, if we have a database that the application requires we can do a simple SQL query to ensure that the database is working properly. For interactive applications, we can try to use a *here* document or an expect script to log in to the application, or server, and maybe even perform a small task. Applications work differently, so solutions to ensure that the application is up and running properly will vary widely.

We are going to look at monitoring local processes, remote monitoring using OpenSSH (ssh) or Remote Shell (rsh), checking for active Oracle databases, and using an **expect** script to log in remotely and execute one or more commands. We will check an application URL and HTTP server status, and finally use **egrep** to monitor a group of processes to give notification when they have all completed execution in this chapter.

Monitoring Local Processes

Above pinging a host machine, the most common application monitoring technique is to look for the critical processes that are required for the application to work properly.

This is also a good practice when we have a flaky application that has a process that dies intermittently.

The basic technique is to use the ps -ef | grep target_process | grep -v grep command syntax. If you have more than one required process, this command statement needs to be executed for each of the processes individually for an accurate result. We can also build an egrep statement of PIDs and monitor all of the processes at once. However, we do not want to use egrep in place of **grep** in some cases. If egrep is used, we can get a positive result if *any* of the processes are currently running. For example, if we lose a critical HTTP process, and any of the other "targeted" processes are still running, we will get a false positive that everything is okay if we use egrep to search all of the processes at once. We use egrep when we want to monitor for all processes to *end execution*, not to ensure all process are running.

The key to making this technique work is to find a *unique* string pattern that represents the target process. The PID is no good if you are not sure if your PID is the parent or child process, because the process may have a child or parent process that has the same PID somewhere in the ps -ef output. Finding a unique string pattern that works with the grep command is key. This command is easily tested using the following command syntax, from the command line:

ps -ef | grep Appserver | grep -v grep

This command statement assumes that we are looking for a process called Appserver. Notice that we always pipe (|) the last pipe's output to | grep -v grep. This last grep on grep is needed so that the system will not report on the grep Appserver process. In the process table each part of the command statement that has a pipe will have a separate PID.

Then there is another thing to consider if this command is executed in a shell script. The shell script name may show up in the grep output, depending on how the shell script is written. To get around this little problem, we need to query the system to capture the shell script's filename and add a third grep to the ps -ef command statement using the following syntax:

```
SCRIPT_NAME=$(basename $0)
```

ps -ef | grep target_process | grep -v grep | grep -v \$SCRIPT_NAME

Now we have a command that will work if, *and only if*, a unique character string can be found that separates the target process from all other processes. This usually takes a few tries for each application that we want to monitor.

In Listing 16-1 we have a code segment from a shell script that monitors an application service, using a unique character string. This particular application service is defined by the APPSVC variable. If this service is not currently running, there is an attempt to restart the application service and an email is sent to my text pager and my regular email account. Follow the code segment in Listing 16-1.

```
******
   DEFINE FILES AND VARIABLES HERE
*****
APPSVC="/usr/local/sbin/appstrt_u1"
MAILLIST="1234567890@mypage.provider.abc randy@my.domain.com"
MAILFILE="/tmp/mailfile.out"
TIMESTAMP=$(date +%m%d%y%H%M%S)
APPS_LOG="/usr/local/log/appsvc.log"
[ -s $APP_LOG ] || touch $APP_LOG
****
  BEGINNING OF MAIN
****
# Check to see if the APPSVC process(es) is/are running
APPSVC_COUNT=$(ps -ef | grep "$APPSVC" | grep -v grep \
                  grep -v $SCRIPT_NAME | wc -1)
# If the count is zero then we need to attempt to restart the service
if (( $APPSVC_COUNT == 0 ))
then
    # Need to attempt an Application server restart.
   echo "Attempting Restart: $APPSVC" > $MAILFILE
    # Send email notification
   mailx -r rmichael@my.domain.com -s "App Down" $MAILLIST < $MAILFILE
    # Make a log entry
   echo "ERROR: $TIMESTAMP - Appsvc DOWN - Attempting Restart $APPSVC" \
         >> $APP_LOG
    # Make another log entry
    echo "STARTING APPLICATION SERVER - $TIMESTAMP" >>$APPS_LOG
   # Attempt the restart !!!
    su - appsvc -c '/usr/local/sbin/appsvc start 2>&1' >> $APP_LOG
fi
```

Listing 16-1 Code segment to monitor an application process

In the code segment in Listing 16-1, notice that we defined a unique string for the process, which in this case is the fully qualified pathname to the APPSVC variable. Because this application server can have multiple instances running at the same time, we need to get a count of how many of these processes are running. If the process count is 0, zero, a restart of the application server is attempted.

During the restart effort an email notification is sent to reflect that the application service is down and the script is attempting a restart. This information is also logged in the \$APP_LOG file *before* the restart command. Notice the restart command at the end of the script segment. This monitoring script is executed from the root cron table every 10 minutes. Because the script is running as root it is easy to use the **su** (switch user) command to execute a single command as the appsvc user for the restart. If you are not familiar with this technique, study the syntax in Listing 16-1 and the man page for the su command.

Remote Monitoring with Secure Shell and Remote Shell

In the previous section we studied a "local" shell script. No one said, though, that you could not run this same script from a remote machine. This is where Remote Shell and Open Secure Shell (OpenSSH) come into play.

Open Secure Shell is a freeware encryption replacement for telnet, ftp, and rsh, for the most part. Every place I have worked over the past four years has disabled rsh, telnet, and ftpas a result of SOX audits. These commands are a security risk because communication is not encrypted and, with a sniffer or a tcpdump, usernames and passwords can be seen in plain text. When we use the ssh command, we establish a connection between two machines, and a secure *tunnel* allows encrypted communication between two *trusted* machines. Using ssh we can log in to another trusted machine in the network, we can copy files between the machines with scp and sftp in an encrypted state, and we can run commands on a remote trusted machine. You can download OpenSSH at http://www.openssh.com.

To establish *password-free* encrypted connections, an encryption key pair must be created on both machines. This encryption key is located on both machines in the user's \$HOME/.ssh directory. All of the details to set up the password-free encrypted connections are shown in great detail in the ssh manual page (**man ssh**).

Of course, you can script it, too! The keyit script, shown in Listing 16-2, takes care of the task of adding the public key to the authorized_keys file.

```
#!/usr/bin/ksh
#
# SCRIPT: keyit
# AUTHOR: Randy Michael
# DATE: 7/31/2007
# REV: 1.0
# PLATFORM: Not platform dependent
# REQUIREMENTS: OpenSSH
#
#
# PURPOSE: This script is used to set up
# encryption keypairs between two hosts.
```

Listing 16-2 keyit shell script

```
# set -x # Uncomment to debug this script
#
# set -n # Uncomment to check script syntax
       # without any execution. Do not
#
#
       # forget to add the comment back,
       # or the script will never execute.
#
#
# USAGE: keyit remote_host username
****
# DEFINE FILES AND VARIABLES HERE
*****
RHOST=$1
THIS_USER=$2
THIS_SCRIPT=$(basename $0)
THIS_HOST=$(hostname)
# DEFINE FUNCTIONS HERE
*******
usage ()
{
echo "\nUSAGE: $THIS SCRIPT \
remote_host username\n"
}
success_message ()
{
KTYPE=$1
echo "\nSUCCESS: $KTYPE key pairs configured for $THIS_USER on $RHOST"
echo "\n$THIS_USER should no longer require an SSH password on $RHOST"
echo "when logging in directly, however, using the ssh commands:\n"
echo "\tssh -1 $THIS_USER $RHOST\nAND\n\tssh ${THIS_USER}@${RHOST}"
echo "\nWHILE LOGGED IN LOCALLY AS ANOTHER USER will still not work"
echo "without a valid user password\n"
}
failure_message ()
echo "\nERROR: Setting up the $KEYTYPE key pairs failed"
echo "Ensure that OpenSSH is installed and running"
```

```
echo "on both hosts. Then ensure that the user has"
echo "a .ssh directory in their \$HOME directory."
echo "See the man page on ssh and ssh-keygen"
echo "for more details and manual setup\n"
}
keyit_dsa ()
{
# Append the local public key to the same user's
# authorized_users file
cat ~${THIS_USER}/.ssh/id_dsa.pub | ssh ${THIS_USER}@$RHOST \
"cat >> ~${THIS_USER}/.ssh/authorized_keys"
if (( $? == 0 ))
then
   success_message dsa
else
   failure_message
fi
}
******
keyit_rsa ()
{
# Append the local public key to the same user's
# authorized_users file
cat ~${THIS_USER}/.ssh/id_rsa.pub | ssh ${THIS_USER}@$RHOST \
"cat >> ~${THIS_USER}/.ssh/authorized_keys"
if (($? == 0))
then
   success_message rsa
else
   failure_message
fi
}
# BEGINNING OF MAIN
# Ensure the user $THIS_USER exists on the local system
```

Listing 16-2 (continued)

```
if ! $(/usr/bin/id $THIS_USER >/dev/null 2>&1)
then
    echo "\nERROR: $THIS_USER is not a valid user on $THIS_HOST\n"
   usage
    exit 1
fi
# Ensure ssh is installed locally
if [[ ! -x /usr/bin/ssh && ! -x /usr/local/bin/ssh ]]
then
   echo "\nERROR: SSH does not appear to be installed on this machine"
    echo "This script requires SSH...Exiting...\n"
   usage
   exit 2
fi
# Check for proper usage
if ! [ $2 ]
then
  usage
  exit 1
fi
# Ping the remote host 1 ping
if ! $(ping -c1 $RHOST >/dev/null 2>&1)
then
   echo "\nERROR: $RHOST is not pingable...Exiting...\n"
   exit 2
fi
# Set up the key pairs for the configured key(s)
SET=0
if [ -s ~$THIS_USER/.ssh/id_dsa.pub ]
then
  keyit_dsa
  SET=1
fi
if [ -s ~$THIS_USER/.ssh/id_rsa.pub ]
then
   keyit_rsa
   SET=2
fi
```

```
if (( SET == 0 ))
then
  echo "\nERROR: SSH public key is not set for $THIS_USER..."
  echo "\nTo Configure Run: ssh-keygen -t type"
  echo "Where type is rsa or dsa encryption\n"
  echo "Would you like to set up the keys now? (y/n): \c"
  read REPLY
  case $REPLY in
  y Y) if $(id $THIS_USER >/dev/null 2>&1)
       then
          echo "\nEncryption Type: (dsa or rsa?): \c"
          read KEYTYPE
          case "$KEYTYPE" in
          +([d|D][s|S][a|A])) KEYTYPE=dsa
             ;;
          +([r|R][s|S][a|A])) KEYTYPE=rsa
             ;;
          *) echo "\nERROR: Invalid entry...Exiting..."
             exit 1
             ;;
          esac
         echo "\nAccept the defaults and do not enter a passphrase...\n"
          su - $THIS USER "-c ssh-keygen -t $KEYTYPE"
          if (($? == 0))
          then
             echo "\nSuccess, keying $THIS_USER on $RHOST\n"
             keyit_${KEYTYPE} $KEYTYPE
          fi
       else
          echo "\nERROR: $THIS_USER username does not exist\n"
       fi
      ;;
    *) # Do nothing
       : # A colon, :, is a "no-op"
       ;;
  esac
fi
*****
# END OF KEYIT SCRIPT
*****
```

Listing 16-2 (continued)

Let's look at a couple of examples of using ssh. The first example, shown in Listing 16-3, shows a simple login without the key pair created.

```
# ssh randy@dino
The authenticity of host 'dino (10.10.10.6)' can't be established.
RSA key fingerprint is c5:19:37:b9:59:ad:3a:18:6b:45:57:2d:ab:b8:df:bb.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'dino (10.10.10.6)' (RSA) to
the list of known hosts.
randy@dino's password:
Last unsuccessful login: Tue Jul 2 13:58:18 EDT 2006 on
/dev/pts/24 from bambam
Last login: Wed Aug 7 10:28:00 EDT 2006 on /dev/pts/18 from bambam
                       Welcome to dino!
  Please see the README file in /usr/lpp/bos for information
pertinent to this release of the operating system
                *********[YOU HAVE NEW MAIL]
[randy@dino] $
```

Listing 16-3 Sample Secure Shell login

Notice in Listing 16-3 that the login to dino required a password, which indicates that the systems do not have the encryption key pairs set up. This does get a bit annoying when you are trying to run a command on a remote machine using an ssh tunnel. With the key pairs created on both machines we can monitor remote machines using encryption, and no password is required. As an example, suppose we need to check the filesystem usage on dino and we are logged into yogi. By adding the command that we want to execute on dino to the end of the ssh login statement, we establish a trusted connection between the two machines, and the command executes on the remote machine with standard output going to the local machine. Of course, this is equivalent to a remote shell, rsh, except that the information is encrypted using ssh in place of rsh. A simple example of this technique is shown in Listing 16-4.

```
[randy@yogi] ssh randy@dino df -k
Filesystem 1024-blocks Free %Used Iused %Iused Mounted on
/dev/hd4 196608 66180 67% 2330 3% /
```



/dev/hd2	1441792	488152	67%	29024	98	/usr
/dev/hd9var	2162688	1508508	31%	868	1%	/var
/dev/hd3	131072	106508	19%	361	2%	/tmp
/dev/hd1	589824	235556	61%	15123	11%	/home
/dev/local_lv	393216	81384	80%	2971	4%	/usr/local
/dev/oracle_lvx	1507328	307388	80%	5008	2%	/oracle
/dev/arch_lvx	13631488	8983464	35%	44	1%	/oradata

Notice in the output in Listing 16-4 that there was no prompt for a password, and that the result was presented back to the local terminal. Once the key pairs are set up, you can do remote monitoring with ease. The keyit script is a handy tool to keep around. Let's move on to Oracle now.

Checking for Active Oracle Databases

I wanted to add a couple of examples of interacting with an application in this chapter, and I picked an Oracle database as the example using a SQL+database query, and using **autoexpect** to automate logging in to a remote machine. First the Oracle database tests. We will look at three steps to check the Oracle database status. The first step is to list all of the Oracle instances defined in the /etc/oratab file. This file is colon-separated (:) with the Oracle instance name(s) in the first field, \$1. The function shown in Listing 16-4 first checks to see if a /etc/oratab file exists. If the file is not found, a notification message is displayed on the user's terminal and the function returns a 3 for a return code. Otherwise, the /etc/oratab file is parsed to find the Oracle instance name(s). Removing all of the lines in the file that begin with comments, specified by beginning with a hash mark (#), is done using a **sed** statement in combination with the ^# notation. Removing the comment lines is easy using the sed statement, as shown here with a /etc/hosts file as an example:

```
cat /etc/hosts | sed /^#/d > /etc/hosts.without_beginning_comments
```

The output of the preceding command shows all the IP address and hostname entries, except that the commented-out lines have been removed. The $^{#}$ is the key to finding the commented lines, which translates to *begins with a* #.

We could have used grep -v ^# just as easily as the previous sed statement. I wanted to mix it up a little bit.

Check out the function in Listing 16-5 to see how we use this technique to parse the Oracle instances from the /etc/oratab file.

```
function show_oratab_instances
{
    if [ ! -f "$ORATAB" ]
    then
        echo "\nOracle instance file $ORATAB does not exist\n"
```

Listing 16-5 show_oratab_instances function

```
return 3
else
cat $ORATAB | sed /^#/d | awk -F: '{print $1}'
fi
}
```

The output of the show_oratab_instances function in Listing 16-5 is a list of all of the Oracle instances defined on the system. We have already removed the lines that are comments; next comes the **awk** statement that extracts the first field, specified by awk -F: '{print \$1}'. In this awk statement, the -F: specifies that the line is field separated by colons (:). Once we know the field separator, we just extract the first field (\$1), which is the Oracle instance name.

NOTE The Solaris implementation of awk does not support specifying a field delimiter using awk -F : type notation. For Solaris you must use nawk (new awk) instead of awk. If you add the following code segment at the top of a shell script that uses this notation the alias will take care of the problem:

```
case $(uname) in
SunOS) alias awk=nawk
;;
esac
```

Now we are going to use the same function shown in Listing 16-5 to get the status of all of the defined Oracle instances by checking for the process for each instance. This technique is shown in Listing 16-6.

Listing 16-6 show_all_instances_status function

Notice in Listing 16-6 that we use the function from Listing 16-5 to get the list of Oracle instances to query the system for. If you do this, remember the show_oratab_ instances function must be defined in the script before the show_all_instances_ status function is used. *Define it before you use it*. In this case, all we are doing is

using the ps -ef command again. This time we narrow down the list with a grep on the string ora. This output is piped (|) to another grep statement, where we are looking for the instance name for the current loop iteration, specified by \$INSTANCE. Of course, we need to strip out any grep processes from the output, so we add one more grep -v grep. If the return code of the entire ps -ef statement is 0, zero, then the instance is running; if the return is anything other than 0, zero, the instance is not running, specified by the if ((\$? ! = 0)) test.

We are still looking at the process level. I have seen cases when the instance processes are running, but I still could not log in to the database. For a final test we need to do an actual SQL query of the database to interact with Oracle. This just needs to be a very simple query to prove that we can interact with the database and get data back.

To actually query the Oracle database we can use a simple SQL+statement, as shown in Listing 16-7. This two-line SQL script is used in the function simple_SQL_query, shown in Listing 16-8 using the **sqlplus** command.

```
select * from user_users;
exit
```

Listing 16-7 my_sql_query.sql SQL script

As you can see in Listing 16-7, this is not much of a query, but it is all that we need. This SQL script, my_sql_query.sql, is used in the sqlplus function in Listing 16-8. Notice in this function, simple_SQL_query, that the sqlplus command statement requires a username, password, and an Oracle SID name to work. This file should have file permission to limit read access for security. See the function code in Listing 16-8.

```
function simple_SQL_query
{
USER=oracle
PASSWD=oracle
SID=yogidb
sqlplus ${USER}/${PASSWD}@$SID @my_sql_query.sql
}
```

Listing 16-8 simple_SQL_query function

The function shown in Listing 16-8 can be shortened further, *if* you are logged in to the system as the oracle user or executing a script as the oracle user. If these conditions are met, you can run a simpler version of the previous sqlplus, as shown in Listing 16-9, with the output of the query; however, the Oracle Listener is not tested as in the previous sqlplus statement in Listing 16-8. The sqlplus command in Listing 16-9 should be run on the local machine.

```
[oracle@yogi] sqlplus / @/usr/local/bin/mysql_query.sql
SQL*Plus: Release 8.1.7.0.0 - Production on Wed Aug 7 16:07:30 2007
(c) Copyright 2000 Oracle Corporation. All rights reserved.
Connected to:
Oracle8i Enterprise Edition Release 8.1.7.4.0 - Production
With the Partitioning option
JServer Release 8.1.7.4.0 - Production
USERNAME
                        USER_ID ACCOUNT_STATUS
_____ ____
LOCK_DATE EXPIRY_DATE DEFAULT_TABLESPACE
_____
                      CREATED
TEMPORARY_TABLESPACE
                               INITIAL_RSRC_CONSUMER_GROUP
_____ __ ___
EXTERNAL NAME
_____
OPSSORACLE
                           940 OPEN
                USERS
                  18-APR-2007
TEMP
Disconnected from Oracle8i Enterprise Edition Release
8.1.7.4.0 - Production
With the Partitioning option
JServer Release 8.1.7.4.0 - Production
```

Listing 16-9 Example of a SQL+Oracle query

This is about as simple as it gets! You can check the return code from the sqlplus command shown in Listing 16-9. If it is zero, the query worked. If the return code is nonzero, the query failed and the database should be considered down. In any case, the Database Administrator needs to be notified of this condition.

Using autoexpect to Create an expect Script

As we saw in Chapter 8, "Automating Interactive Programs with Expect and Autoexpect," an expect script enables us to automate interaction with machines and applications. Now here's your miracle: we can automate *creating* the expect script by using autoexpect. After starting an autoexpect session, all your interaction, and system responses, are saved in a file. The default filename is script.exp, so you need to change the filename. The trick is to *not make a typo*! You cannot even use the backspace key because this keystroke too is saved. To show how this works, follow along through Listing 16-10. The task is to log in to a remote machine and run the who and w commands, and then log out. To begin, enter **autoexpect**, then, *without making a mistake*, interact with the application or system. When you are finished, press the Ctrl+D key sequence to exit the autoexpect session, and save the file as script.exp.

```
#!/usr/bin/expect -f
# This Expect script was generated by autoexpect on Tue
Jul 31 04:46:35 2007
# Expect and autoexpect were both written by Don Libes, NIST.
# Note that autoexpect does not guarantee a working script. It
# necessarily has to guess about certain things. Two reasons a script
# might fail are as follows:
# 1) timing - A surprising number of programs (rn, ksh, zsh, telnet,
# etc.) and devices discard or ignore keystrokes that arrive "too
# quickly" after prompts. If you find your new script hanging up at
# one spot, try adding a short sleep just before the previous send.
# Setting "force_conservative" to 1 (see below) makes Expect do this
# automatically - pausing briefly before sending each character. This
# pacifies every program I know of. The -c flag makes the script do
# this in the first place. The -C flag allows you to define a
# character to toggle this mode off and on.
set force conservative 0 ;# set to 1 to force conservative mode even if
                      ;# script wasn't run conservatively originally
if {$force_conservative} {
    set send_slow {1 .1}
     proc send {ignore arg} {
            sleep .1
            exp_send -s -- $arg
     }
}
# 2) differing output - Some programs produce different output each time
# they run. The "date" command is an obvious example. Another is
# ftp, if it produces throughput statistics at the end of a file
# transfer. If this causes a problem, delete these patterns or replace
# them with wildcards. An alternative is to use the -p flag (for
# "prompt") which makes Expect only look for the last line of output
# (i.e., the prompt). The -P flag allows you to define a character to
# toggle this mode off and on.
```

Listing 16-10 script.exp

```
# Read the man page for more info.
#
# -Don
set timeout -1
spawn $env(SHELL)
match_max 100000
expect -exact " \]0;root@localhost:/scripts
\[?1034h\[root@localhost scripts\]# "
send -- "telnet yogi\r"
expect -exact "telnet yogi\r
Trying 192.168.1.101...\r\r
Connected to yogi.\r\r
Escape character is '^\]'.\r\r
١r
١r
telnet (yogi.tampabay.rr.com)\r
\r\r
\r\r
١r
\r
\r
\r
١r
١r
١r
١r
١r
١r
\r
١r
١r
١r
١r
١r
\r
١r
١r
١r
١r
\r
١r
١r
\rAIX Version 5\r
r(C) Copyrights by IBM and by others 1982, 2000, 2005.
\rlogin: "
send -- "randyr"
```

```
expect -exact "randy\r
randy's Password: "
send -- "abc123\r"
expect -exact "\r
***********************
*\r
*\r
* Welcome to AIX Version 5.3!
*\r
*
*\r
*\r
* Please see the README file in /usr/lpp/bos for
information pertinent to *\r
* this release of the AIX Operating System.
*\r
*
*\r
*
*\r
*****
Last unsuccessful login: Sun Jul 29 00:23:33 EDT 2007
on /dev/pts/1 from 192.168.1.102\r
Last login: Sun Jul 29 00:24:26 EDT 2007 on /dev/pts/1
from 192.168.1.102\r
١r
randy@/home/randy # "
send -- "who\r"
expect -exact "who\r
                Jul 26 05:19 (yogi)
        pts/0
                                        \r
root
randy
        pts/1
                  Jul 29 00:26
                                 (192.168.1.102) \r
randy@/home/randy # "
send -- "w\r"
expect -exact "w\r
 12:26AM up 2 days, 23:52, 2 users, load average: 0.03, 0.06, 0.05\r
                 login@ idle JCPU PCPU what\r
User
      tty
                Thu05AM 22:20
12:26AM 0
root
      pts/0
                                     25
                                              1 -ksh∖r
                12:26AM
                                      0
                                                0 w\r
randy
       pts/1
randy@/home/randy # "
send -- "exit\r"
expect -exact "exit\r
Connection closed by foreign host.\r\r"
expect eof
```

Listing 16-10 (continued)

When you first look at the result, it's a bit overwhelming. Not to worry, we are going to drop about 90 percent of this file in the bit bucket. The trick with expect is *not to expect too much*. The lines of code we are interested in are highlighted in bold text in Listing 16-10.

Now we just need to send commands and expect a known reply. Follow through the shortened version of this expect script, as shown in Listing 16-11.

```
#!/usr/bin/expect -f
#
set force_conservative 0 ;# set to 1 to force conservative mode even if
         ;# script wasn't run conservatively originally
if {$force_conservative} {
   set send_slow {1 .1}
   proc send {ignore arg} {
           sleep .1
           exp_send -s -- $arg
    }
}
set timeout -1
spawn $env(SHELL)
match_max 100000
send -- "telnet yogi\r"
expect "\rlogin: "
send -- "randy\r"
expect -exact "randy\r
randy's Password: "
send -- "abc123\r"
expect "randy@/home/randy # "
send -- "who\n"
expect "randy@/home/randy # "
send -- "w\r"
expect "randy@/home/randy # "
send -- "exit\r"
expect -exact "exit\r
Connection closed by foreign host.\r\r"
send -- "\r"
expect eof
```

Listing 16-11 expect script to remotely execute the who and w commands

The first thing to notice about Listing 16-11 is that we omitted the -exact switch on most of the expect statements. What we *expect* is just the command-line prompt. So, if you set the same PS1 environment variable on all your machines, you will have the same prompt, as long as the host name is not part of the prompt. You can also set the PS1 here in the expect script. You should use whatever prompt the server or application gives you if you do not set it in the expect script. I always have to

play around with expect a bit when I remove the unneeded replies. The output of executing the expect script in Listing 16-11 is shown in Listing 16-12.

```
[root@booboo scripts]# ./login_remotely.exp
spawn /bin/Bash
telnet yogi
[root@booboo scripts]# telnet yogi
Trying 192.168.1.101...
Connected to yogi.
Escape character is '^]'.
telnet (yogi.tampabay.rr.com)
AIX Version 5
(C) Copyrights by IBM and by others 1982, 2000, 2006.
login: randy
randy's Password:
       Welcome to AIX Version 5.1!
* Please see the README file in /usr/lpp/bos for
information pertinent to
* this release of the AIX Operating System.
Last unsuccessful login: Sun Jul 29 00:23:33 EDT 2007
on /dev/pts/1 from 192.168.1.102
Last login: Sun Jul 29 00:38:02 EDT 2007 on /dev/pts/1
from 192.168.1.102
randy@/home/randy # who
who
         pts/0
                   Jul 30 01:07 (192.168.1.100)
root
randy
         pts/1
                    Jul 30 04:15 (192.168.1.102)
randy@/home/randy # w
w
```

Listing 16-12 expect script to execute remote who and w commands in action

```
04:15AM up 4 days, 3:42, 2 users, load average: 0.16, 0.05, 0.04

User tty login@ idle JCPU PCPU what

root pts/0 01:07AM 40 17 1 -ksh

randy pts/1 04:15AM 0 0 0 w

randy@/home/randy # exit

exit

Connection closed by foreign host.

[root@booboo scripts]#

[root@booboo scripts]# exit

exit

Script done on Wed 01 Aug 2007 08:35:54 AM EDT
```

Listing 16-12 (continued)

Play around with expect and autoexpect by first trying it with FTP. For more details on expect, study Chapter 8.

Checking if the HTTP Server/Application Is Working

Some applications use a web browser interface. For this type of application we can use a command-line browser, such as **linx**, to attempt to reach a specific URL, which in turn should bring up the specified application web page. The function shown in Listing 16-13 utilizes the linx command-line browser to check both the HTTP server and the web page presented by the specified URL, which is passed to the function in the \$1 argument.

```
check_HTTP_server ()
{
LINX="/usr/local/bin/lynx" # Define the location of the linx program
URL=$1
                        # Capture the target URL in the $1 position
URLFILE=/tmp/HTTP.$$
                        # Define a file to hold the URL output
*****
$LINX "$URL" > $URLFILE # Attempt to reach the target URL
if (($? != 0))
                         # If the URL is unreachable - No Connection
then
    echo "\n$URL - Unable to connect\n"
    cat SURLFILE
                          # Else the URL was found
else
    while read VER RC STATUS # This while loop is fed from the bottom
                           # after the "done" using input redirection
    do
```

Listing 16-13 check_HTTP_server function

Listing 16-13 (continued)

This is a nice function for checking the status of a web server and also to see if an application URL is accessible. You should test this function against doing the same task manually using a graphical browser. This has been tested on an application front-end, and it works as expected; however, a good test is recommended before implementing this, or any other code, in this book. You know all about the disclaimer stuff. (I am really not even here writing this book, or so the disclaimer says.)

What about Waiting for Something to Complete Executing?

If you are waiting on some processes to finish execution, egrep is a good option — for example, if you are running several backups, or **rsync** sessions, and you want to be notified when the processing is complete. One method is to execute the commands in the background, and save the PID of the last background process using the shell variable \$!. The code snippet in Listing 16-14 builds an egrep statement using the saved PIDs.

Listing 16-14 Using egrep to monitor processes

```
$MACHINE:/dba/oradata/data_0${N} &
   THIS_PID=$! # Save the last PID
    # Build an egrep list of PIDs
    if [ -z "$EGREP_LIST" ]
    then
        EGREP_LIST="$THIS_PID"
    else
       EGREP_LIST="${EGREP_LIST}|${THIS_PID}"
    fi
done
sleep 10 # Sleep a few seconds to let the process complete.
echo "\nRsync is running: -- [Start time: $date].\c"
REMAINING_PROC=$(ps -ef | grep "$EGREP_LIST" | grep -v grep \
                 awk '{print $2}' | wc -1)
# Print a dot every 60 seconds for feedback to the user
until (( $REMAING_PROC == 0 ))
do
      REMAINING_PROC=$(ps -ef | grep "$EGREP_LIST" | grep -v grep \
                     | awk '{print $1}' | wc -1)
     echo ".\c"
     sleep 60
done
echo "\nCOMPLETE: Rsync copies completed $(date)\n"
```

Listing 16-14 (continued)

This code in Listing 16-14 starts seven rsync copy sessions in the background, specified by ending the rsync commands with an &, ampersand. Next we save the last background PID using the THIS_PID=\$! variable assignment. As we gather each background PID, we build an egrep statement. We sleep for 10 seconds to let the rsync sessions start up, and then we start monitoring all the processes at once using egrep. When the process ID count reaches zero, we display the completion to the end user.

Other Things to Consider

As with any code that is written, it can always be improved. Each of the functions and code segments presented in this chapter are just that, code segments. When you are monitoring applications, code like this is only one part of a much bigger shell script, or at least it should be. The monitoring should start at the lowest level, which is sending a ping to the application host to ensure that the machine is powered on and booted. Then we apply more layers as we try to build a script that will allow us to debug the

problem. I have presented only a few ideas; it is your job to work out the details for your environment.

Proper echo Usage

Also remember the proper shell usage of echo when we use the backslash operators:

echo "\nHello World\n"

- For Bourne and Bash shell echo -e "\nHello World\n"
- For Korn shell echo "\nHello World\n"

Application APIs and SNMP Traps

Most enterprise management tools come with application programming interfaces (APIs) for the more common commercial applications; however, we sometimes must write shell scripts to fill in the gaps. This is where SNMP traps come in. Because the enterprise management tool *should* support SNMP traps, the APIs allow the application to be monitored using the SNMP MIB definitions on both the management server and the client system.

When an enterprise management tool supports SNMP traps, you can usually write your own shell scripts that can use the tool's MIB and SNMP definitions to get the message out from your own shell scripts. As an example, the command shown here utilizes a well-known monitoring tool's SNMP and MIB data to allow a trap to be sent:

/usr/local/bin/trapclient \$MON_HOST \$MIB_NUM \$TRAP_NUM \$TRAP_TEXT

In this command the MON_HOST variable represents the enterprise management workstation. The MIB_NUM variable represents the specific code for the MIB parameter. The TRAP_NUM variable represents the specific trap code to send, and the TRAP_TEXT is the text that is sent with the trap. This type of usage varies depending on the monitoring tool that you are using. At any rate, there are techniques that allow you to write shell scripts to send traps. The methods vary, but the basic syntax remains the same for SNMP.

Summary

This is one of those chapters where it is useless to write a bunch of shell scripts. I tried to show some of the techniques of monitoring applications and application processes, but the details are too varied to cover in a single chapter. I have laid down a specific process that you can utilize to build a very nice tool to monitor your systems and applications. Always start with a ping! If the box is unpingable, your first job is to get the machine booted or to call hardware support.

In the next steps you have several options, including interacting with the application, as we did with an SQL+query of an Oracle database and the expect scripts. We also covered monitoring specific processes that are a little flaky and die every once in a while. I have two applications that I have to monitor this way, and I have not had even one phone call since I put this tool in place. The key is to keep the business in business, and the best way to do that is to be very proactive. This is where good monitoring and control shell scripts make you look like gold.

Remember: no one *ever* notices an application except when it is down!

In the next chapter we will cover monitoring a system for full filesystems. Methods covered include a typical percentage method to the number of megabytes free, for very large filesystems. Chapter 17, "Filesystem Monitoring," ends with a shell script that does auto detection using the filesystem size to set the monitoring method.

Lab Assignments

- 1. Write an expect script to log in to a remote machine and run the keyit script so that you have the ssh encryption key pairs configured in both directions. You should be able to ssh between machines without a password.
- 2. Rewrite the keyit shell script using expect.
- 3. If you have an application to monitor, write a shell script to tackle the task.

PART

Scripts for Systems Administrators

Chapter 18: Monitoring Paging and Swap Space Chapter 19: Monitoring System Load

Chapter 17: Filesystem Monitoring

Chapter 20: Monitoring for Stale Disk Partitions

Chapter 21: Turning On/Off SSA Identification Lights

Chapter 22: Automated Hosts Pinging with Notification of Failure

Chapter 23: Creating a System-Configuration Snapshot

Chapter 24: Compiling, Installing, Configuring, and Using sudo

Chapter 25: Print-Queue Hell: Keeping the Printers Printing

Chapter 26: Those Pesky Sarbanes-Oxley (SOX) Audits

Chapter 27: Using Dirvish with rsync to Create Snapshot-Type Backups

Chapter 28: Monitoring and Auditing User Keystrokes

CHAPTER 17

Filesystem Monitoring

The most common monitoring task is monitoring for full filesystems. On different flavors of UNIX the monitoring techniques are the same, but the commands and fields in the output vary slightly. This difference is due to the fact that command syntax and the output columns vary depending on the UNIX system.

We are going to step through the entire process of building a script to monitor filesystem usage and show the philosophy behind the techniques used. In scripting this solution we will cover six monitoring techniques, starting with the most basic monitoring — percentage of space used in each filesystem.

The next part will build on this original base code and add *exceptions* capability allowing an override of the script's set threshold for a filesystem to be considered full. The third part will deal with *large filesystems*, which are typically considered to be a filesystems larger than 2 gigabytes, 2 GB. This script modification will use the megabytes (MB)-of-free-space technique.

The fourth part will add exception capability to the MB-of-free-space method. The fifth part in this series combines both the percentage-of-used-space and MB-of-free-space techniques with an added *auto-detect* feature to decide how to monitor each filesystem. Regular filesystems will be monitored with percent used and large filesystems as MB of free space, and, of course, with the exception capability. The sixth and final script will allow the filesystem monitoring script to run on AIX, Linux, HP-UX, OpenBSD, or Solaris without any further modification.

Syntax

Our first task, as usual, is to get the required command syntax. For this initial example, we are going to monitor an AIX system. (HP-UX, Linux, OpenBSD, and Solaris will be covered later.) The command syntax to look at the filesystems in kilobytes, KB, or 1024-byte blocks, is df -k in AIX.

Filesystem	1024-blocks	Free	%Used	Iused	%Iused	Mounted on
/dev/hd4	32768	16376	51%	1663	11%	/
/dev/hd2	1212416	57592	96%	36386	13%	/usr
/dev/hd9var	53248	30824	43%	540	5%	/var
/dev/hd3	106496	99932	7%	135	1%	/tmp
/dev/hd1	4096	3916	5%	25	3%	/home
/proc	-	-	-	-	-	/proc
/dev/hd10opt	638976	24456	97%	15457	10%	/opt
/dev/scripts_	lv 102400	95264	7%	435	2%	/scripts
/dev/cd0	656756	0	100%	328378	100%	/cdrom

Let's take a look at the output of the df -k command on an AIX 5L machine:

The fields in the command output that we are concerned about are column 1, the Filesystem device; column 4, the %Used; and Mounted on in column 7. There are at least two reasons that we want both the filesystem device and the mount point. The first reason is to know if it is an NFS mounted filesystem. This first column will show the NFS server name as part of the device definition if it is NFS mounted. The second reason is that we will not want to monitor a mounted CD/DVD-ROM. A CD/DVD-ROM will always show that it is 100 percent used because it is mounted as read-only and you cannot write to it (I know, CD-RW/DVD-RW drives, but these are still not the norm in most business environments, and because we are not trying to write to this type of device, we will ignore it).

As you can see in the bottom row of the preceding output, the /cdrom mount point does indeed show that it is 100 percent utilized. We want to omit this from the output along with the column heading at the top line. The first step is to show everything except for the column headings. We can use the following syntax:

df -k | tail +2

This delivers the following output without the column headings:

/dev/hd4	32768	16376	51%	1663	11%	/
/dev/hd2	1212416	57592	96%	36386	13%	/usr
/dev/hd9var	53248	30824	43%	540	5%	/var
/dev/hd3	106496	99932	7%	135	1%	/tmp
/dev/hd1	4096	3916	5%	25	3%	/home
/proc	-	-	-	-	-	/proc
/dev/hd10opt	638976	24456	97%	15457	10%	/opt
/dev/scripts_lv	102400	95264	7%	435	2%	/scripts
/dev/cd0	656756	0	100%	328378	100%	/cdrom

This output looks a bit better, but we still have a couple of things we are not interested in. The /cdrom is at 100 percent all of the time, and the /proc mount point has no values, just hyphens. The /proc filesystem is new to AIX 5L (it used to be hidden), and because it has no values, we want to eliminate it from our output. Notice the device, in column 1, for the CD-ROM is /dev/cd0. This is what we want to use

as a tag to pattern match on instead of the mount point because it may at some point be mounted somewhere else, for example /mnt. We may also have devices /dev/cd1 and /dev/cd2, too, if not now perhaps in the future. This, too, is easy to take care of, though. We can expand on our command statement to exclude both lines from the output with one **egrep** statement, as in the following:

df -k | tail +2 | egrep -v '/dev/cd[0-9] /proc'

In this statement we used the egrep command with a -v switch. The -v switch means to show everything *except* what it patterned matched on. The egrep is used for extended regular expressions; in this case, we want to exclude two rows of output. To save an extra grep statement, we use egrep and enclose what we are pattern matching on within single tic marks, ' ', and separate each item in the list with a pipe symbol, |. The following two commands are equivalent:

df -k | tail +2 | grep -v '/dev/cd[0-9]' | grep -v '/proc'
df -k | tail +2 | egrep -v '/dev/cd[0-9] /proc'

Also notice in both statements the pattern match on the CD-ROM devices. The grep and egrep statements will match devices /dev/cd0 up through the last device, for example /dev/cd24, using /dev/cd[0-9] as the pattern match.

Using egrep saves a little bit of code, but both commands produce the same output, shown here:

/dev/hd4	32768	16376	51%	1663	11%	/
/dev/hd2	1212416	57592	96%	36386	13%	/usr
/dev/hd9var	53248	30864	43%	539	5%	/var
/dev/hd3	106496	99932	7%	134	1%	/tmp
/dev/hd1	4096	3916	5%	25	3%	/home
/dev/hd10opt	638976	24456	97%	15457	10%	/opt
/dev/scripts_lv	102400	95264	7%	435	2%	/scripts

In this output we have all of the rows of data we are looking for; however, we have some extra columns that we are not interested in. Now let's extract out the columns of interest, 1, 4, and 7. Extracting the columns is easy to do with an **awk** statement. The **cut** command will work, too. Using an awk statement is the cleanest method, and the columns are selected using the positional parameters, or columns, \$1, \$2, \$3, ..., \$n. As we keep building this command statement, we add in the awk part of the command:

First, notice that we extended our command onto the next line with the backslash character, \. This convention helps with the readability of the script. In the awk part of the statement, we placed a comma and a space after each field, or positional parameter.

The comma and space are needed to ensure that the fields remain separated by at least one space. This command statement leaves the following output:

```
/dev/hd4 51% /
/dev/hd2 96% /usr
/dev/hd9var 43% /var
/dev/hd3 7% /tmp
/dev/hd1 5% /home
/dev/hd10opt 97% /opt
/dev/scripts_lv 7% /scripts
```

For ease of working with our command output, we can write it to a file and work with the file. In our script we can define a file and point to the file with a variable. The following code will work:

WORKFILE="/tmp/df.work" # df output work file
>\$WORKFILE # Initialize the file to zero size

Before we go any further, we also need to decide on a trigger threshold for when a filesystem is considered full, and we want to define a variable for this, too. For our example, we will say that anything over 85 percent is considered a full filesystem, and we will assign this value to the variable FSMAX:

FSMAX="85"

From these definitions we are saying that any monitored filesystem that has used *more than* 85 percent of its capacity is considered *full*. Our next step is to loop through each row of data in our output file. Our working data file is /tmp/df.work, which is pointed to by the \$WORKFILE variable, and we want to compare the second column, the percentage used for each filesystem, to the \$FSMAX variable, which we initialized to 85. But we still have a problem; the \$WORKFILE entry still has a %, percent sign, and we need an integer value to compare to the \$FSMAX value. We will take care of this conversion with a **sed** statement. We use sed for character substitution and manipulation and, in this case, character removal. The sed statement is just before the numerical comparison in a loop that follows. Please study Listing 17-1, and pay close attention to the bold text.

```
#!/usr/bin/ksh
#
# SCRIPT: fs_mon_AIX.ksh
# AUTHOR: Randy Michael
# DATE: 08-22-2007
# REV: 1.1.P
# PURPOSE: This script is used to monitor for full filesystems,
# which are defined as "exceeding" the FSMAX value.
```

Listing 17-1 fs_mon_AIX.ksh shell script

```
A message is displayed for all "full" filesystems.
#
#
# REV LIST:
# set -n # Uncomment to check script syntax without any execution
# set -x # Uncomment to debug this script
##### DEFINE FILES AND VARIABLES HERE ####
                       # Max. FS percentage value
FSMAX="85"
WORKFILE="/tmp/df.work" # Holds filesystem data
>$WORKFILE
                       # Initialize to empty
OUTFILE="/tmp/df.outfile" # Output display file
>$OUTFILE
                       # Initialize to empty
THISHOST=`hostname` # Hostname of this machine
# Get the data of interest by stripping out /dev/cd#,
# /proc rows and keeping columns 1, 4 and 7
df -k | tail +2 | egrep -v '/dev/cd[0-9] /proc' \
      awk '{print $1, $4, $7}' > $WORKFILE
# Loop through each line of the file and compare column 2
while read FSDEVICE FSVALUE FSMOUNT
do
      FSVALUE=$(echo $FSVALUE | sed s/\%//g) # Remove the % sign
      typeset -i FSVALUE
      if [ $FSVALUE -gt $FSMAX ]
      then
          echo "$FSDEVICE mounted on $FSMOUNT is ${FSVALUE}%" \
               >> $OUTFILE
      fi
done < $WORKFILE # Feed the while loop from the bottom!!</pre>
if [[ -s $OUTFILE ]]
then
      echo "\nFull Filesystem(s) on $THISHOST\n"
     cat $OUTFILE
     print
fi
```

Listing 17-1 (continued)

The items highlighted in the script are all important to note. We start with getting the hostname of the machine. We want to know which machine the report is relating to. Next we load the **SWORKFILE** with the filesystem data. Just before the numerical test is made we remove the % sign and then typeset the variable, FSVALUE, to be an integer. Then we make the over-limit test, and if the filesystem in the current loop iteration has exceeded the threshold of 85 percent, we append a message to the SOUTFILE. Notice that the while loop is getting its data from the bottom of the loop, after done. This is the fastest technique to process a file line by line without using file descriptors. After processing the entire file we test to see if the SOUTFILE exists and is greater than zero bytes in size. If it has data, we print an output header, with a newline before and after, and display the SOUTFILE file followed by another blank line. In Listing 17-1 we used an assortment of commands to accomplish the same task in a different way — for example, using VARIABLE=\$ (command) and VARIABLE=`command` to execute a command and assign the command's output to a variable, and the use of the echo and print commands. In both instances the result is the same. We again see there is not just one way to accomplish the same task.

We also want to explain how we use sed for character substitution. The basic syntax of the sed statement is as follows:

```
sed s/current_string/new_string/g [filename]
```

The filename is optional, and we need to use the output from a command, so we are going to use the following sed command syntax:

command | sed s/current_string/new_string/g

When we extend our command and pipe the last pipe's output to the sed statement, we get the following:

The point to notice about the preceding sed part of the command statement is that we had to *escape* the %, percent sign, with a \, backslash. This is because % is a special character in UNIX. To remove the special meaning from, or to escape, the function, we use a backslash before the % sign. This lets us literally use % as a text character as opposed to its system/shell-defined value or function (% is the modulo math operator). See Listing 17-2.

```
Full Filesystem(s) on yogi
/dev/hd2 mounted on /usr is 96%
/dev/hd10opt mounted on /opt is 97%
```

Listing 17-2 Full filesystem script in action

This script is okay, but we really are not very concerned about these filesystems being at these current values. The reason is that /usr and /opt, on AIX, should remain

static in size. The /usr filesystem is where the OS and application code for the system reside, and /opt, new to AIX 5L version 5.1, as a mount point, is where Linux and many GNU programs' code resides. So how can we give an exception to these two filesystems?

Adding Exceptions Capability to Monitoring

The fs_mon_AIX.ksh script is great for what it is written for, but in the real world we always have to make exceptions and we always strive to cover all of the "gotchas" when writing shell scripts. Now we are going to add the capability to override the default FSMAX threshold. Because we are going to be able to override the default, it would be really nice to be able to either raise or lower the threshold for individual filesystems.

To accomplish this script tailoring, we need a data file to hold our exceptions. We want to use a data file so that people are not editing the shell script every time a filesystem threshold is to be changed. To make it simple, let's use the file /usr/local/bin/exceptions, and point to the file with the EXCEPTIONS variable. Now that we know the name of the file, we need a format for the data in the \$EXCEPTIONS file. A good format for this data file is the /mount_point and a NEW_MAX%. We will also want to ignore any entry that is commented out with a pound sign, #. This may sound like a lot, but it is really not too difficult to modify the script code and add a function to read the exceptions file. Now we can set it up.

The Exceptions File

To set up our exceptions file, we can always use /usr/local/bin, or your favorite place, as a *bin directory*. To keep things nice we can define a bin directory for the script to use. This is a good thing to do in case the files need to be moved for some reason. The declarations are shown here:

```
BINDIR="/usr/local/bin"
EXCEPTIONS="${BINDIR}/exceptions"
```

Notice the curly braces around the BINDIR variable when it is used to define the EXCEPTIONS file. This is always a good thing to do if the variable name will have a character that is not associated with the variable's name, next to the variable name without a space. Otherwise, an error may occur that could be very hard to find! Notice the difference in usage shown here.

EXCEPTIONS="\$BINDIR/exceptions"

versus

```
EXCEPTIONS="${BINDIR}/exceptions"
```

In all of the ways there are to set up exceptions capability, grep seems to come up the most. Please *avoid the grep mistake*! The two fields in the \$EXCEPTIONS file are the /mount_point and the NEW_MAX% value. The first instinct is to grep on the /mount_point, but what if /mount_point is root, /? If you grep on /, and the / entry is not the first entry in the exceptions file, you will get a pattern match on the wrong entry, and thus use the wrong \$NEW_MAX% in deciding if the / mount point is full. In fact, if you grep on / in the exceptions file, you will get a match on the first entry in the file every time. Listing 17-3 shows some *wrong code* that made this very grep mistake.

```
while read FSDEVICE FSVALUE FSMOUNT
do
     # Strip out the % sign if it exists
     FSVALUE=$(echo $FSVALUE | sed s/\%//g) # Remove the % sign
     if [[ -s $EXCEPTIONS ]] # Do we have a non-empty file?
     then # Found it!
        # Look for the current $FSMOUNT value in the file
              #WRONG CODE, DON'T MAKE THIS MISTAKE USING grep!!
        cat $EXCEPTIONS | grep -v "^#" | grep $FSMOUNT \
                        | read FSNAME NEW_MAX
        if [ $? -eq 0 ] # Found it!
        then
            if [[ $FSNAME = $FSMOUNT ]] # Sanity check
            then
                 NEW_MAX=$(echo $NEW_MAX | sed s/\%//g)
                 if [ $FSVALUE -gt $NEW_MAX ] # Use the new $NEW_MAX
                 then
                    echo "$FSDEVICE mount on $FSMOUNT is ${FSVALUE}%" \
                           >> $OUTFILE
                 fi
            elif [ $FSVALUE -gt $FSMAX ] # Not in $EXCEPTIONS file
            then
                 echo "$FSDEVICE mount on $FSMOUNT is ${FSVALUE}%" \
                      >> $OUTFILE
            fi
        fi
     else # No exceptions file...use script default
             if [ SFSVALUE - gt SFSMAX ]
             then
                  echo "$FSDEVICE mount on $FSMOUNT is ${FSVALUE}%" \
                      >> $OUTFILE
             fi
     fi
done < $WORKFILE
```

Listing 17-3 The wrong way to use grep

The code in Listing 17-3 really looks as if it should work, and it does *some of the time*! To get around the error that grep introduces, we need to just set up a function that will look for an exact match for each entry in the exceptions file.

Now let's look at this new technique. We want to write two functions: one to load the \$EXCEPTIONS file data without the comment lines (the lines beginning with a #) while omitting all blank lines into a data file, and one to search through the exceptions file data and perform the tests.

This is a simple one-line function to load the \$EXCEPTIONS file data into the \$DATA_EXCEPTIONS file:

```
function load_EXCEPTIONS_file
{
    # Ignore any line that begins with a pound sign, #
# and also remove all blank lines
cat $EXCEPTIONS | grep -v "^#" | sed /^$/d > $DATA_EXCEPTIONS
}
```

In the preceding function, we use the ^, caret character, along with grep -v to ignore any line beginning with a #, pound sign. We also use the ^\$ with the sed statement to remove any blank lines and then redirect output to a data file, which is pointed to by the \$DATA_EXCEPTIONS variable. After we have the exceptions file data loaded, we have the following check_exceptions function that will look in the \$DATA_EXCEPTIONS file for the current mount point and, if found, will check the \$NEW_MAX value to the system's reported percent-used value. The function will present back to the script a return code relating to the result of the test.

```
function check_exceptions
{
# set -x # Uncomment to debug this function
while read FSNAME NEW_MAX # Feeding data from Bottom of Loop !!!
do
        if [[ $FSNAME = $FSMOUNT ]] # Correct /mount_point?
        then
                # Get rid of the % sign, if it exists!
                NEW_MAX=$(echo $NEW_MAX | sed s/\%//g)
                if [ $FSVALUE -gt $NEW_MAX ]
                then # Over Limit...Return a "0", zero
                        return 0 # FOUND OVER LIMIT - Return 0
                else # Found in the file but is within limits
                        return 2 # Found OK
                fi
        fi
done < $DATA_EXCEPTIONS # Feed from the bottom of the loop !!
return 1 # Not found in File
}
```

This check_exceptions function is called during each loop iteration in the main script and returns a 0, zero, if the /mount_point is found to exceed the NEW_MAX%. It

will return a 2 if the mount point was found to be okay in the exceptions data file, and return a 1, one, if the mount point was not found in the *\$DATA_EXCEPTIONS* file. There are plenty of comments throughout this new script, so feel free to follow through and pick up a few pointers — pay particular attention to the bold text in Listing 17-4.

```
#!/usr/bin/ksh
# SCRIPT: fs_mon_AIX_except.ksh
# AUTHOR: Randy Michael
# DATE: 08-22-2007
# REV: 2.1.P
#
# PURPOSE: This script is used to monitor for full filesystems,
    which are defined as "exceeding" the FSMAX value.
#
     A message is displayed for all "full" filesystems.
# PLATFORM: AIX
#
# REV LIST:
#
         08-23-2007 - Randy Michael
        Added code to override the default FSMAX script threshold
#
         using an "exceptions" file, defined by the $EXCEPTIONS
#
         variable, that list /mount_point and NEW_MAX%
# set -n # Uncomment to check syntax without any execution
# set -x # Uncomment to debug this script
##### DEFINE FILES AND VARIABLES HERE ####
FSMAX="85"
                      # Max. FS percentage value
WORKFILE="/tmp/df.work" # Holds filesystem data
>$WORKFILE # Initialize to empty
OUTFILE="/tmp/df.outfile" # Output display file
>$OUTFILE
                     # Initialize to empty
BINDIR="/usr/local/bin" # Local bin directory
THISHOST=`hostname` # Hostname of this machine
EXCEPTIONS="${BINDIR}/exceptions" # Overrides $FSMAX
DATA_EXCEPTIONS="/tmp/dfdata.out" # Exceptions file w/o #, comments
####### DEFINE FUNCTIONS HERE #####
function load_EXCEPTIONS_file
```

```
Listing 17-4 fs_mon_AIX_except.ksh shell script
```

```
# Ignore any line that begins with a pound sign, #
# and omit all blank lines
cat $EXCEPTIONS | grep -v "^#" | sed /^$/d > $DATA_EXCEPTIONS
3
******
function check exceptions
£
# set -x # Uncomment to debug this function
while read FSNAME NEW_MAX # Feeding data from Bottom of Loop !!!
do
       if [[ $FSNAME = $FSMOUNT ]] # Correct /mount_point?
       then
             # Get rid of the % sign, if it exists!
              NEW_MAX=$(echo $NEW_MAX | sed s/\%//g)
              if [ $FSVALUE -gt $NEW_MAX ]
              then # Over Limit...Return a "0", zero
                     return 0 # FOUND OUT OF LIMITS - Return 0
              fi
       fi
done < $DATA EXCEPTIONS # Feed from the bottom of the loop!!
return 1 # Not found in File
}
*****
****
# If there is an exceptions file...load it...
[[ -s $EXCEPTIONS ]] && load_EXCEPTIONS_file
# Get the data of interest by stripping out /dev/cd#,
# /proc rows and keeping columns 1, 4, and 7
df -k | tail +2 | egrep -v '/dev/cd[0-9] /proc' \
  | awk '{print $1, $4, $7}' > $WORKFILE
# Loop through each line of the file and compare column 2
while read FSDEVICE FSVALUE FSMOUNT
```

Listing 17-4 (continued)

```
# Feeding the while loop from the BOTTOM !!
do
     # Strip out the % sign if it exists
     FSVALUE=$(echo $FSVALUE | sed s/\%//g) # Remove the % sign
     if [[ -s $EXCEPTIONS ]] # Do we have a non-empty file?
     then # Found it!
        # Look for the current $FSMOUNT value in the file
        # using the check_exceptions function defined above.
        check_exceptions
        RC=$? # Get the return code from the function
        if [ $RC -eq 0 ] # Found Exceeded in Exceptions File !!
        then
            echo "$FSDEVICE mount on $FSMOUNT is ${FSVALUE}%" \
                  >> $OUTFILE
        elif [ $RC -eq 1 ] # Not found in exceptions, use defaults
        then
             if [ $FSVALUE -gt $FSMAX ] # Use Script Default
             then
                  echo "$FSDEVICE mount on $FSMOUNT is ${FSVALUE}%" \
                        >> $OUTFILE
             fi
        fi
     else # No exceptions file use the script default
             if [ $FSVALUE -gt $FSMAX ] # Use Script Default
             then
                  echo "$FSDEVICE mount on $FSMOUNT is ${FSVALUE}%" \
                        >> $OUTFILE
             fi
     fi
done < $WORKFILE # Feed the while loop from the bottom...
# Display output if anything is exceeded...
if [[ -s $OUTFILE ]]
then
      echo "\nFull Filesystem(s) on ${THISHOST}\n"
      cat $OUTFILE
     print
fi
```

Listing 17-4 (continued)

Notice in the script that we never acted on the return code 2. Because the mount point is found to be okay, there is nothing to do except to check the next mount point. The /usr/local/bin/exceptions file will look something like the file shown in Listing 17-5.

```
# FILE: "exceptions"
# This file is used to override the $FSMAX
# value in the filesystem monitoring script
# fs_mon_except.ksh. The syntax to override
# is a /mount-point and a NEW_MAX%:
# EXAMPLE:
# /opt 97
# OR
# /usr 96%
# All lines beginning with a # are ignored as well as
# the % sign, if you want to use one...
/opt 96%
/usr 97
/ 50%
```

Listing 17-5 Example exceptions file

When we execute the fs_mon_AIX_except.ksh script, with the exceptions file entries from Listing 17-5, the output looks like the following on yogi (see Listing 17-6).

```
Full Filesystem(s) on yogi
/dev/hd4 mount on / is 51%
/dev/hd10opt mount on /opt is 97%
```

Listing 17-6 Full filesystem on yogi script in action

Notice that we added a limit for the root filesystem, /, and set it to 50 percent, and also that this root entry is not at the top of the list in the exceptions file shown in Listing 17-5 so we have solved the grep problem. You should be able to follow the logic through the preceding code to see that we met all of the goals we set out to accomplish in this section. There are plenty of comments to help you understand each step.

Are we finished? Not by a long shot! What about monitoring large filesystems? Using the percentage of filesystem space used is excellent for regular filesystems, but if you have a 10 GB or 100 GB filesystem and it is at 90 percent, you still have 1 GB or 10 GB of free space, respectively. Even at 99 percent you have 100 MB to 1 GB of space left. For large filesystems we need another monitoring method.

Using the MB-of-Free-Space Method

Sometimes a percentage is just not accurate enough to get the detailed notification that is desired. For these instances, and in the case of large filesystems, we can use awk on the df -k command output to extract the KB of Free Space field and compare this to a

threshold trigger value, specified in either KB or MB. We are going to modify both of the scripts we have already written to use the KB of Free Space field. Please understand that I could just use df -m or df -g to specify df command output in MB or GB of space. However, if I did that, you would not get a chance to do a little math in a shell script. Just keep these two commands in mind for the Lab Assignments at the end of this chapter.

Remember our previous df -k command output:

Filesystem	1024-blocks	Free	%Used	Iused	%Iused	Mounted on
/dev/hd4	32768	16376	51%	1663	11%	/
/dev/hd2	1212416	57592	96%	36386	13%	/usr
/dev/hd9var	53248	30824	43%	540	5%	/var
/dev/hd3	106496	99932	7%	135	1%	/tmp
/dev/hd1	4096	3916	5%	25	3%	/home
/proc	-	-	-	-	-	/proc
/dev/hd10opt	638976	24456	97%	15457	10%	/opt
/dev/scripts_1	lv 102400	95264	7%	435	2%	/scripts
/dev/cd0	656756	0	100%	328378	100%	/cdrom

Instead of the fourth field of the percentage used, we now want to extract the third field with the 1024-blocks, or KB of Free space. When someone is working with the script it is best that an easy and familiar measurement is used; the most common is MB of free space. To accomplish this we will need to do a little math, but this is just to have a more familiar measurement to work with. As before, we are going to load the command output into the \$WORKFILE, but this time we extract columns \$1, \$3, and \$7:

We also need a new threshold variable to use for this method. The MIN_MB_FREE variable sounds good. But what is an appropriate value to set the threshold? In this example, we are going to use 50 MB, although it could be any value:

MIN_MB_FREE="50MB"

Notice that we added MB to the value. We will remove this later, but it is a good idea to add the measurement type just so that the ones who follow will know that the threshold is in MB. Remember that the system is reporting in KB, so we have to multiply our 50 MB times 1024 to get the actual value that is equivalent to the system-reported measurement. We also want to strip out the MB letters and typeset the MIN_MB_FREE variable to be an integer. In the compound statement that follows, we take care of everything except typesetting the variable:

((MIN_MB_FREE = \$(echo \$MIN_MB_FREE | sed s/MB//g) * 1024))

The order of execution for this compound command is as follows: First, the innermost () command substitution is executed, which replaces the letters MB, if they exist, with

null characters. Next is the evaluation of the math equation and assignment of the result to the MIN_MB_FREE variable. Equating MIN_MB_FREE may seem a little confusing, but remember that the system is reporting in KB so we need to get to the same power of 2 to also report in 1024-byte blocks. Other than these small changes, the script is the same as the original, as shown in Listing 17-7.

```
#!/usr/bin/ksh
#
# SCRIPT: fs_mon_AIX_MBFREE.ksh
# AUTHOR: Randy Michael
# DATE: 08-22-2007
# REV: 1.5.P
# PURPOSE: This script is used to monitor for full filesystems,
     which are defined as "exceeding" the FSMAX value.
#
#
    A message is displayed for all "full" filesystems.
#
# REV LIST:
#
          Randy Michael - 08-27-2007
#
         Changed the code to use MB of free space instead of
#
          the %Used method.
# set -n # Uncomment to check syntax without any execution
# set -x # Uncomment to debug this script
##### DEFINE FILES AND VARIABLES HERE ####
MIN_MB_FREE="50MB" # Min. MB of Free FS Space
WORKFILE="/tmp/df.work" # Holds filesystem data
>$WORKFILE # Initialize to empty
OUTFILE="/tmp/df.outfile" # Output display file
>$OUTFILE # Initialize to empty
THISHOST=`hostname` # Hostname of this machine
# Get the data of interest by stripping out /dev/cd#,
# /proc rows and keeping columns 1, 4 and 7
df -k | tail +2 | egrep -v '/dev/cd[0-9] /proc' \
   | awk '{print $1, $3, $7}' > $WORKFILE
# Format Variables
(( MIN_MB_FREE = $(echo $MIN_MB_FREE | sed s/MB//g) * 1024 ))
# Loop through each line of the file and compare column 2
```

Listing 17-7 fs_mon_AIX_MBFREE.ksh shell script

```
while read FSDEVICE FSMB_FREE FSMOUNT
do
      FSMB_FREE=$(echo $FSMB_FREE | sed s/MB//g) # Remove the "MB"
      if (( FSMB_FREE < MIN_MB_FREE ))
      then
          (( FS_FREE_OUT = FSMB_FREE / 1000 ))
          echo "\$FSDEVICE mounted on \$FSMOUNT only has \
      ${FS FREE OUT}MB Free" >> $OUTFILE
      fi
done < $WORKFILE # Feed the while loop from the bottom !!
if [[ -s $OUTFILE ]]
then
      echo "\nFull Filesystem(s) on $THISHOST\n"
      cat $OUTFILE
     print
fi
```

Listing 17-7 (continued)

The fs_mon_AIX_MBFREE.ksh script in Listing 17-7 is a good start. Listing 17-8 shows this shell script in action.

```
/dev/hd4 mounted on /only has 16MB Free
/dev/hd9var mounted on /var only has 30MB Free
/dev/hd1 mounted on /home only has 3MB Free
/dev/hd10opt mounted on /opt only has 24MB Free
```

Listing 17-8 Shell script in action

Full Filesystem(s) on yogi

This output in Listing 17-8 is padded by less than 1 MB due to the fact that we divided the KB free column by 1,000 for the output, measured in MB. If the exact KB is needed, the division by 1,000 can be omitted. What about giving this script exception capability to raise or lower the threshold, as we did for the percentage technique? We already have the percentage script with the check_exception function so that we can modify this script and function to use the same technique of parsing through the \$EXCEPTIONS file.

Using MB of Free Space with Exceptions

To add exception capability to the fs_mon_AIX_MBFREE.ksh shell script, we will again need a function to perform the search of the \$EXCEPTIONS file, if it exists. This

time we will add some extras. We may have the characters MB in our data, so we need to allow for this. We also need to test for null characters, or no data, and remove all blank lines in the exception file. The easiest way to use the function is to supply an appropriate return code back to the calling script. We will set up the function to return 1, one, if the mount point is found to be out of limits in the \$DATA_EXCEPTIONS file. It will return 2 if the /mount_point is in the exceptions data file but is *not* out of limits. The function will return 3 if the mount point is not found in the exceptions data file. This will allow us to call the function to check the exception file, and based on the return code, we make a decision in the main body of the script.

We already have experience modifying the script to add exception capability, so this should be a breeze, right? When we finish, the exception modification will be intuitively obvious.

Because we are going to parse through the exceptions file, we need to run a sanity check to see if someone made an incorrect entry and placed a colon, :, in the file intending to override the limit on an NFS mounted filesystem. This error should never occur if the local mount point is used, but because a tester I know did so, I now check and correct the error, if possible. We just cut out the second field using the colon, :, as a delimiter. Listing 17-9 shows the modified check_exceptions function. Check out the highlighted parts in particular.

```
function check_exceptions
# set -x # Uncomment to debug this function
while read FSNAME FSLIMIT
do
    # Do an NFS sanity check
    echo $FSNAME | grep ":" >/dev/null \
         && FSNAME=$(echo $FSNAME | cut -d ":" -f2)
    # Make sure we do not have a null value
    if [[ ! -z "$FSLIMIT" && "$FSLIMIT" != '' ]]
    then
        ((FSLIMIT = \$(echo \$FSLIMIT | sed s/MB//g) * 1024))
        if [[ $FSNAME = $FSMOUNT ]]
        then
            # Get rid of the "MB" if it exists
            FSLIMIT=$(echo $FSLIMIT | sed s/MB//g)
            if (( FSMB_FREE < FSLIMIT ))
            then
                return 1 # Found out of limit
            else
                return 2 # Found OK
            fi
        fi
    fi
```

Listing 17-9 New check_exceptions function

```
done < $DATA_EXCEPTIONS # Feed the loop from the bottom!!!
return 3 # Not found in $EXCEPTIONS file
}</pre>
```

Listing 17-9 (continued)

A few things to notice in this function are the NFS and null value sanity checks, as well as the way that we feed the while loop from the bottom, after the done statement. First, the sanity checks are very important to guard against incorrect NFS entries and blank lines, or null data, in the exceptions file. For the NFS colon check we use the double ampersands, &&, as opposed to the if...then... statement. It works the same but is cleaner in this type of test. The other point is the null value check. We check for both a zero-length variable and null data. The double ampersands, &&, are called a logical AND function, and the double pipes, ||, are a logical OR function. In a logical AND, &&, all of the command statements must be true for the return code of the entire statement to be 0, zero. When a logical OR receives the first true statement in the test list it will immediately exit the test, or command statement, with a return code of 0, zero. Both are good to use, but some people find it hard to follow. Next we test for an empty/null variable:

```
if [[ ! -z "$FSLIMIT" && "$FSLIMIT" != '' ]]
```

Note that in the null sanity check there are double quotes around both of the *\$FSLIMIT* variables, "*\$FSLIMIT*". These are required! If you omit the double quotes and the variable is actually null, the test will fail and a shell error message is generated and displayed on the terminal. It never hurts to add double quotes around a variable, and sometimes it is required.

For the while loop we go back to our favorite loop structure. Feeding the while loop from the bottom, after done, is the fastest way to loop through a file line by line without using file descriptors. With the sanity checks complete, we just compare some numbers and give back a return code to the calling shell script. Please pay attention to the boldface code in Listing 17-10.

```
#!/usr/bin/ksh
#
# SCRIPT: fs_mon_AIX_MB_FREE_except.ksh
# AUTHOR: Randy Michael
# DATE: 08-22-2007
# REV: 2.1.P
# PURPOSE: This script is used to monitor for full filesystems,
# which are defined as "exceeding" the FSMAX value.
# A message is displayed for all "full" filesystems.
```

Listing 17-10 fs_mon_AIX_MB_FREE_except.ksh shell script

```
# PLATFORM: AIX
#
# REV LIST:
#
          Randy Michael - 08-27-2007
#
          Changed the code to use MB of free space instead of
          the %Used method.
#
#
          Randy Michael - 08-27-2007
#
          Added code to allow you to override the set script default
#
          for MIN_MB_FREE of FS Space
#
# set -n # Uncomment to check syntax without any execution
# set -x # Uncomment to debug this script
##### DEFINE FILES AND VARIABLES HERE ####
MIN_MB_FREE="50MB"
                     # Min. MB of Free FS Space
WORKFILE="/tmp/df.work" # Holds filesystem data
                        # Initialize to empty
>$WORKFILE
OUTFILE="/tmp/df.outfile" # Output display file
               # Initialize to empty
>$OUTFILE
EXCEPTIONS="/usr/local/bin/exceptions" # Override data file
DATA_EXCEPTIONS="/tmp/dfdata.out" # Exceptions file w/o # rows
THISHOST=`hostname` # Hostname of this machine
####### DEFINE FUNCTIONS HERE ########
function check_exceptions
# set -x # Uncomment to debug this function
while read FSNAME FSLIMIT
do
    # Do an NFS sanity check
    echo $FSNAME | grep ":" >/dev/null \
         && FSNAME=$(echo $FSNAME | cut -d ":" -f2)
    if [[ ! -z "$FSLIMIT" && "$FSLIMIT" != '' ]] # Check for empty/null
    then
        (( FSLIMIT = $(echo $FSLIMIT | sed s/MB//g) * 1024 ))
        if [[ $FSNAME = $FSMOUNT ]]
        then
            # Get rid of the "MB" if it exists
            FSLIMIT=$(echo $FSLIMIT | sed s/MB//g)
            if (( FSMB_FREE < FSLIMIT )) # Numerical Test
            then
                return 1 # Found out of limit
```

Listing 17-10 (continued)

```
else
               return 2 # Found OK
            fi
        fi
    fi
done < $DATA_EXCEPTIONS # Feed the loop from the bottom!!!
return 3 # Not found in $EXCEPTIONS file
}
# Load the $EXCEPTIONS file if it exists
if [[ -s $EXCEPTIONS ]]
then
    # Ignore all lines beginning with a pound sign, #
    # and omit all blank lines
    cat $EXCEPTIONS | grep -v "^#" | sed /^$/d > $DATA_EXCEPTIONS
fi
# Get the data of interest by stripping out /dev/cd#,
# /proc rows and keeping columns 1, 4 and 7
df -k | tail +2 | egrep -v '/dev/cd[0-9]|/proc' \
   awk '{print $1, $3, $7}' > $WORKFILE
# Format Variables for the proper MB value
(( MIN_MB_FREE = $(echo $MIN_MB_FREE | sed s/MB//g) * 1024 ))
# Loop through each line of the file and compare column 2
while read FSDEVICE FSMB_FREE FSMOUNT
do
    if [[ -s $EXCEPTIONS ]]
    then
     check_exceptions
     RC="$?" # Check the Return Code!
      if (( RC == 1 )) # Found out of exceptions limit
      then
         (( FS_FREE_OUT = $FSMB_FREE / 1000 ))
          echo "$FSDEVICE mounted on $FSMOUNT only has\
 ${FS_FREE_OUT}MB Free" \
               >> $OUTFILE
      elif (( RC == 2 )) # Found in exceptions to be OK
      then # Just a sanity check - We really do nothing here ...
           # The colon, :, is a NO-OP operator in KSH
```

Listing 17-10 (continued)

```
: # No-Op - Do Nothing!
      elif (( RC == 3 )) # Not found in the exceptions file
      then
          FSMB_FREE=$(echo $FSMB_FREE | sed s/MB//g) # Remove the "MB"
          if (( FSMB_FREE < MIN_MB_FREE ))
          then
              (( FS_FREE_OUT = FSMB_FREE / 1000 ))
              echo "$FSDEVICE mounted on $FSMOUNT only has\
 ${FS_FREE_OUT}MB Free" >> $OUTFILE
          fi
      fi
    else # No Exceptions file use the script default
      FSMB_FREE=$(echo $FSMB_FREE | sed s/MB//g) # Remove the "MB"
      if (( FSMB_FREE < MIN_MB_FREE ))
      then
          ((FS_FREE_OUT = FSMB_FREE / 1000))
          echo "$FSDEVICE mounted on $FSMOUNT only has\
 ${FS_FREE_OUT}MB Free" >> $OUTFILE
      fi
    fi
done < $WORKFILE
if [[ -s $OUTFILE ]]
then
      echo "\nFull Filesystem(s) on $THISHOST\n"
      cat $OUTFILE
      print
fi
```

Listing 17-10 (continued)

The script in Listing 17-10 is good, and we have covered all of the bases, right? If you want to stop here, you will be left with an incomplete picture of what we can accomplish. There are several more things to consider, and, of course, there are many more ways to do any of these tasks, and no single one is *correct*. Let's consider mixing the filesystem-percentage-used and the MB-of-free-filesystem-space techniques. With a mechanism to auto-detect the way we select the usage, the filesystem monitoring script could be a much more robust tool — and a *must-have* tool where you have a mix of regular and large filesystems to monitor.

Percentage Used – MB Free and Large Filesystems

Now we're talking! Even if most of your filesystems are large file enabled or are just huge in size, the small ones will still kill you in the end. For a combination of small and large filesystems, we need a mix of both the percent-used and MB-of-free-space

techniques. For this combination to work, we need a way to auto-detect the *correct usage*, which we still need to define. There are different combinations of these auto-detect techniques that can make the monitoring work differently. For the large filesystems we want to use the MB of free space, and for regular filesystems we use the percentage method.

We need to define a *trigger* that allows for this free space versus percentage monitoring transformation. The trigger value will vary by environment, but this example uses 1 GB as the transition point from percentage used to MB of free space. Of course, the value should be more like 4-6 GB, but we need an example. We also need to consider how the SEXCEPTIONS file is going to look. Options for the exceptions file are a combined file or two separate files, one for percentage used and one for MB free. The obvious choice is one combined file. What are combined entries to look like? How are we going to handle the wrong entry *type*? The entries need to conform to the specific test type the script is looking for. The best way to handle this is to require that either a % or MB be added as a suffix to each new entry in the exceptions file. With the MB or % suffix we could override not only the triggering level, but also the testing method! If an entry has only a number without the suffix, this exceptions file entry will be ignored and the shell script's default values will be used. This suffix method is the most flexible, but it, too, is prone to mistakes in the exceptions file. For the mistakes, we need to test the entries in the exceptions to see that they conform to the standard that we have decided on.

The easiest way to create this new, more robust script is to take large portions of the previous scripts and convert them into functions. We can simply insert the word *function* followed by a function name and enclose the code within curly braces — for example, function test_function {function_code}. Or if you prefer the GNU function method, we can use this example: test_function () {function_code}. The only difference between the two function methods is that one uses the word *function* to define the function, whereas the other just adds a set of parentheses after the function's name. But adding the parentheses is okay even using the function operator. When we use functions, it is easy to set up a logical framework from which to call the functions. It is always easiest to set up the framework first and then fill in the middle. The *logic code* for this script will look like Listing 17-11.

Listing 17-11 Logic code for a large and small filesystem free-space script

```
case $RC in
    1) Found exceeded by % method
    2) Found out-of-limit by MB Free method
    3) Found OK in exceptions file by a testing method
    4) Not found in exceptions file
    esac
    else # No exceptions file
        Use script defaults to compare
    fi
done

if we have anything out of limits
then
    display_output
fi
```

Listing 17-11 (continued)

This is very straightforward and easy to do with functions. From this logical description we already have the main body of the script written. Now we just need to modify the check_exceptions function to handle both types of data and create the load_FS_data, load_EXCEPTIONS_data, and display_output functions. For this script we are also going to do things a little differently because this *is* a learning process. As we all know, there are many ways to accomplish the same task in UNIX; shell scripting is a prime example. To make our scripts a little easier to read at a glance, we are going to change how we do numeric test comparisons. We currently use the standard bracketed test functions with the numeric operators, -lt, -le, -eq, -ne, -ge, and -gt:

if [\$VAR1 -gt \$VAR2]

We are now going to use the bracketed tests for character strings only and do all of our numerical comparisons with the double parentheses method:

if ((VAR1 > VAR2))

The operators for this method are <, <=, ==, !=, >=, and >. These test operators are for numerical data only. When we make this small change, it makes the script much easier to follow because we know immediately that we are dealing with either numeric data or a character string, without knowing much at all about the data being tested. Notice that we did *not* reference the variables with a \$ (dollar sign) for the numeric tests. The \$ omission is not the only difference, but it is the most obvious. The \$ is omitted because it is implied that anything that is not numeric is a variable. Other things to look for in this script are compound tests, math and math within tests, the use of curly braces with variables, \$VAR1 MB, a no-op using a : (colon), data validation, error checking, and error notification. These variables are a lot to look for, but you can learn much from studying the script shown in Listing 17-12.

Just remember that all functions must be defined before they can be used! Failure to define functions is the most common mistake when working with them. The second

most common mistake has to do with *scope*. Scope deals with *where* a variable and its value are known to other scripts and functions. *Top level down* is the best way to describe where scope lies. The basic rules say that *all* of a shell script's variables are known to the *internal*, lower-level, functions, but *none* of the function's variables are known to any *higher-calling* script or function, thus the top level down definition. We will cover a method called a co-process of dealing with scope in Chapter 10, "Process Monitoring and Enabling Pre-Processing, Startup, and Post-Processing Events."

So, in this script the check_exceptions function will use the *global* script's variables, which are known to all of the functions, and the function will, in turn, reply with a return code, as we defined in the logic flow of Listing 17-11. Scope is a very important concept, as is the placement of the function in the script. The comments in this script are extensive, so please study the code and pay particular attention to the boldface text.

NOTE Remember that you have to define a function before you can use it.

```
#!/usr/bin/ksh
# SCRIPT: fs_mon_AIX_PC_MBFREE_except.ksh
# AUTHOR: Randy Michael
# DATE: 08-22-2007
# REV: 4.3.P
# PURPOSE: This script is used to monitor for full filesystems,
     which are defined as "exceeding" the MAX_PERCENT value.
#
     A message is displayed for all "full" filesystems.
# PLATFORM: AIX
#
# REV LIST:
      Randy Michael - 08-27-2007
#
#
       Changed the code to use MB of free space instead of
#
      the %Used method.
#
#
      Randy Michael - 08-27-2007
#
      Added code to allow you to override the set script default
#
      for MIN_MB_FREE of FS Space
#
#
       Randy Michael - 08-28-2007
#
       Changed the code to handle both %Used and MB of Free Space.
#
       It does an "auto-detection" but has override capability
       of both the trigger level and the monitoring method using
       the exceptions file pointed to by the $EXCEPTIONS variable
# set -n # Uncomment to check syntax without any execution
# set -x # Uncomment to debug this script
#
```

Listing 17-12 fs_mon_AIX_PC_MBFREE_except.ksh shell script

```
##### DEFINE FILES AND VARIABLES HERE ####
MIN_MB_FREE="100MB"
                      # Min. MB of Free FS Space
MAX_PERCENT="85%"
                     # Max. FS percentage value
FSTRIGGER="1000MB"
                     # Trigger to switch from % Used to MB Free
WORKFILE="/tmp/df.work" # Holds filesystem data
>$WORKFILE
                        # Initialize to empty
OUTFILE="/tmp/df.outfile" # Output display file
>$OUTFILE
                        # Initialize to empty
EXCEPTIONS="/usr/local/bin/exceptions" # Override data file
DATA_EXCEPTIONS="/tmp/dfdata.out" # Exceptions file w/o # rows
EXCEPT FILE="N"
                     # Assume no $EXCEPTIONS FILE
THISHOST=`hostname`
                      # Hostname of this machine
###### FORMAT VARIABLES HERE ######
# Both of these variables need to be multiplied by 1024 blocks
(( MIN_MB_FREE = $(echo $MIN_MB_FREE | sed s/MB//g) * 1024 ))
(( FSTRIGGER = $(echo $FSTRIGGER | sed s/MB//g) * 1024 ))
####### DEFINE FUNCTIONS HERE ########
function check_exceptions
# set -x # Uncomment to debug this function
while read FSNAME FSLIMIT
do
    IN_FILE="N" # If found in file, which test type to use?
    # Do an NFS sanity check and get rid of any ":".
    # If this is found it is actually an error entry
    # but we will try to resolve it. It will
    # work only if it is an NFS cross mount to the same
    # mount point on both machines.
    echo $FSNAME | grep ':' >/dev/null \
         && FSNAME=$(echo $FSNAME | cut -d ':' -f2)
    # Check for empty and null variable
    if [[ ! -z "$FSLIMIT" && "$FSLIMIT" != '' ]]
    then
       if [[ $FSNAME = $FSMOUNT ]] # Found it!
        then
            # Check for "MB" Characters...Set IN_FILE=MB
          echo $FSLIMIT | grep MB >/dev/null && IN_FILE="MB" \
                 && (( FSLIMIT = (echo \ FSLIMIT \ )
                       sed s/MB//g) * 1024 ))
```

Listing 17-12 (continued)

```
# check for "%" Character...Set IN_FILE=PC, for %
           echo $FSLIMIT | grep "%" >/dev/null && IN_FILE="PC" \
                && FSLIMIT=$(echo $FSLIMIT | sed s/\%//g)
           case $IN FILE in
           MB) # Use Megabytes of free space to test
               # Up-case the characters, if they exist
               FSLIMIT=$(echo $FSLIMIT | tr '[a-z]' '[A-Z]')
               # Get rid of the "MB" if it exists
               FSLIMIT=$(echo $FSLIMIT | sed s/MB//g)
               # Test for blank and null values
               if [[ ! -z $FSLIMIT && $FSLIMIT != '' ]]
               then
                  # Test for a valid filesystem "MB" limit
                  if (( FSLIMIT >= 0 && FSLIMIT < FSSIZE ))
                  then # Check the limit
                     if (( FSMB_FREE < FSLIMIT ))
                     then
                        return 1 # Found out of limit
                                 # using MB Free method
                     else
                        return 3 # Found OK
                     fi
                  else
                    echo "\nERROR: Invalid filesystem MAX for\
$FSMOUNT - $FSLIMIT"
                    echo "
                                 Exceptions file value must be\
less than or"
                    echo "
                                 equal to the size of the filesystem\
measured"
                    echo "
                                 in 1024 bytes\n"
                  fi
               else
                  echo "\nERROR: Null value specified in exceptions\
file"
                  echo "
                               for the $FSMOUNT mount point.\n"
               fi
               ; ;
           PC) # Use the Percent used method to test
               # Strip out the % sign if it exists
               PC_USED=$(echo $PC_USED | sed s/\%//g)
               # Test for blank and null values
               if [[ ! -z $FSLIMIT && $FSLIMIT != '' ]]
               then
                  # Test for a valid percentage, i.e. 0-100
                  if (( FSLIMIT >= 0 && FSLIMIT <= 100 ))
                  then
                     if (( PC_USED > FSLIMIT ))
```

Listing 17-12 (continued)

```
then
                         return 2 # Found exceeded by % Used method
                      else
                         return 3 # Found OK
                      fi
                   else
                      echo "\nERROR: Invalid percentage for\
 $FSMOUNT - $FSLIMIT"
                      echo "
                                   Exceptions file values must be"
                                  between 0 and 100%\n"
                      echo "
                   fi
                else
                   echo "\nERROR: Null value specified in exceptions"
                                 file for the $FSMOUNT mount point.\n"
                   echo "
                fi
                ;;
            N) # Test type not specified in exception file, use default
                # Inform the user of the exceptions file error ...
                echo "\nERROR: Missing testing type in exceptions file"
                echo "
                             for the $FSMOUNT mount point. A \"%\" or"
                echo "
                             \"MB\" must be a suffix to the numerical"
                echo "
                             entry. Using script default values...\n"
                # Method Not Specified - Use Script Defaults
                if (( FSSIZE >= FSTRIGGER ))
                then # This is a "large" filesystem
                    if (( FSMB_FREE < MIN_MB_FREE ))
                    then
                         return 1 # Found out of limit using MB Free
                    else
                         return 3 # Found OK
                    fi
                else # This is a standard filesystem
                    PC_USED=$(echo $PC_USED | sed s/\%//g) #Remove the %
                    FSLIMIT=$(echo $FSLIMIT | sed s/\%//g) #Remove the %
                    if (( PC_USED > FSLIMIT ))
                    then
                        return 2 # Found exceeded by % Used method
                    else
                        return 3 # Found OK
                    fi
                fi
                ;;
            esac
        fi
    fi
done < $DATA_EXCEPTIONS # Feed the loop from the bottom!!!</pre>
```

Listing 17-12 (continued)

```
return 4 # Not found in SEXCEPTIONS file
}
****
function display_output
{
if [[ -s $OUTFILE ]]
then
    echo "\nFull Filesystem(s) on $THISHOST\n"
    cat $OUTFILE
    print
fi
}
*********
function load_EXCEPTIONS_data
{
# Ignore any line that begins with a pound sign, #
# and omit all blank lines
cat $EXCEPTIONS | grep -v "^#" | sed /^$/d > $DATA_EXCEPTIONS
}
*****
function load_FS_data
{
  df -k | tail +2 | egrep -v '/dev/cd[0-9] /proc' \
       awk '{print $1, $2, $3, $4, $7}' > $WORKFILE
3
*********
****
load FS data
# Do we have a nonzero size $EXCEPTIONS file?
if [[ -s $EXCEPTIONS ]]
then # Found a nonempty $EXCEPTIONS file
   load_EXCEPTIONS_data
   EXCEP_FILE="Y"
fi
```

```
while read FSDEVICE FSSIZE FSMB_FREE PC_USED FSMOUNT
do
     if [[ $EXCEP_FILE = "Y" ]]
     then
         check_exceptions
         CE_RC="$?" # Check Exceptions Return Code (CE_RC)
         case $CE_RC in
         1) # Found exceeded in exceptions file by MB Method
            ( FS_FREE_OUT = FSMB_FREE / 1000 ))
            echo "$FSDEVICE mounted on $FSMOUNT has ${FS_FREE_OUT}MB\
 Free" >> $OUTFILE
          ;;
         2) # Found exceeded in exceptions file by %Used method
            echo "$FSDEVICE mount on $FSMOUNT is ${PC_USED}%" \
                  >> $OUTFILE
          ;;
         3) # Found OK in exceptions file
            : # NO-OP Do Nothing
          ;;
         4) # Not found in exceptions file - Use Script Default Triggers
            if (( FSSIZE >= FSTRIGGER ))
            then # This is a "large" filesystem
              # Remove the "MB", if it exists
              FSMB_FREE=$(echo $FSMB_FREE | sed s/MB//g)
              typeset -i FSMB_FREE
              if (( FSMB_FREE < MIN_MB_FREE ))
              then
                (( FS_FREE_OUT = FSMB_FREE / 1000 ))
                echo "$FSDEVICE mounted on $FSMOUNT has\
 ${FS_FREE_OUT}MB Free" >> $OUTFILE
              fi
            else # This is a standard filesystem
                PC_USED=$(echo $PC_USED | sed s/\%//g)
                MAX_PERCENT=$(echo $MAX_PERCENT | sed s/\%//g)
                if (( PC_USED > MAX_PERCENT ))
                then
                    echo "$FSDEVICE mount on $FSMOUNT is ${PC_USED}%" \
                          >> $OUTFILE
                fi
            fi
          ;;
         esac
```

Listing 17-12 (continued)

```
else # NO $EXCEPTIONS FILE USE DEFAULT TRIGGER VALUES
          if (( FSSIZE >= FSTRIGGER ))
          then # This is a "large" filesystem - Use MB Free Method
           FSMB_FREE=$(echo $FSMB_FREE | sed s/MB//g) # Remove the "MB"
            if (( FSMB_FREE < MIN_MB_FREE ))
            then
              ((FS_FREE_OUT = FSMB_FREE / 1000))
              echo "$FSDEVICE mounted on $FSMOUNT has\
${FS_FREE_OUT}MB Free" >> $OUTFILE
            fi
          else # This is a standard filesystem - Use % Used Method
              PC_USED=$(echo $PC_USED | sed s/\%//g)
              MAX PERCENT=$(echo $MAX PERCENT | sed s/\%//g)
              if (( PC_USED > MAX_PERCENT ))
              then
                  echo "$FSDEVICE mount on $FSMOUNT is ${PC_USED}%" \
                       >> $OUTFILE
              fi
          fi
     fi
done < $WORKFILE # Feed the while loop from the bottom !!!
display_output
# End of Script
```

In the script shown in Listing 17-12, we made tests to confirm the data's integrity and for mistakes in the exceptions file (of course, we can go only so far with mistakes!). The reason is that we made the exceptions file more complicated to use. Two of my testers consistently had reverse logic on the MB free override option of the script by thinking *greater than* instead of *less than*. From this confusion, a new exceptions file was created that explained what the script is looking for and gave example entries. Of course, all of these lines begin with a pound sign, #, so they are ignored when data is loaded into the \$DATA_EXCEPTIONS file. Listing 17-13 shows the exceptions file that worked best with the testers.

```
# FILE: "exceptions"
#
# This file is used to override both the default
# trigger value in the filesystem monitoring script
```

Listing 17-13 Example exceptions file

```
# fs_mon_except.ksh, but also allows overriding the
# monitoring technique used, i.e. Max %Used and
# minimum MB of filesystem space. The syntax to
# override is a /mount-point and a trigger value.
# EXAMPLES:
#
# /usr 96% # Flag anything ABOVE 96%
# OR
# /usr 50MB # Flag anything BELOW 50 Megabytes
# All lines beginning with a # are ignored.
# NOTE: All Entries MUST have either "MB" or
      "%" as a suffix!!! Or else the script
      defaults are used. NO SPACES PLEASE!
#
/opt 95%
/ 50%
/usr 70MB
```

The requirement for either % or MB does help keep the entry mistakes down. In case mistakes are made, the error notifications seemed to get these cleared up very quickly — usually after an initial run.

Are we finished with filesystem monitoring? No way! What about the other four operating systems that we want to monitor? We need to be able to execute this script on AIX, Linux, HP-UX, OpenBSD, and SunOS without the need to change the script on each platform.

Running Filesystem Scripts on AIX, Linux, HP-UX, OpenBSD, and Solaris

Can we run the filesystem scripts on various UNIX flavors? You bet! Running our filesystem monitoring script is very easy because we used functions for most of the script. We are going to use the same script, but instead of hard-coding the loading of the filesystem data, we need to use variables to point to the correct OS syntax and columns of interest. Now we need a new function that will determine which flavor of UNIX we are running. Based on the OS, we set up the command syntax and command

output columns of interest that we want to extract, and load the filesystem data for this particular OS. For OS determination we just use the **uname** command. uname, and the get_OS_info function, will return the resident operating system, as shown in Table 17-1.

OPERATING SYSTEM	COMMAND RESULT	FUNCTION RESULT
Linux	Linux	LINUX
AIX	AIX	AIX
HP-UX	HP-UX	HP-UX
OpenBSD	OpenBSD	OPENBSD
Solaris	SunOS	SUNOS

Table 17-1 uname Command and Function Results

For the function's output, we want to use all *uppercase* characters, which makes testing much easier. In the following function please notice we use the typeset function to ensure that the result is in all uppercase characters:

To use the get_OS_info function, we can assign it to a variable using command substitution, use the function directly in a command statement, or redirect the output to a file. For this script modification, we are going to use the get_OS_info function directly in a case statement. Now we need five different load_FS_data functions, one for each of the five operating systems, and that is all the modification that is needed. Each of the load_FS_data functions will be unique in command syntax and the column fields to extract from the df command output, as well as the devices to exclude from testing. Because we wrote this script using functions, we will replace the original load_FS_data script, at the Beginning of Main, with a case statement that utilizes the get_OS_info function. The case statement will execute the appropriate load_FS_data function. Listing 17-14 shows how to test the OS flavor and execute the correct function.

```
case $(get_OS_info) in
   AIX) # Load filesystem data for AIX
         load_AIX_FS_data
     ;;
   HP-UX) # Load filesystem data for HP-UX
         load_HP_UX_FS_data
      ;;
   LINUX) # Load filesystem data for Linux
         load_LINUX_FS_data
      ;;
   OPENBSD) # Load filesystem data for OpenBSD
         load OPENBSD FS data
     ;;
   SUNOS) # Load filesystem data for Solaris
         load_Solaris_FS_data
     ;;
   * )
          # Unsupported in script
          echo "\nUnsupported Operating System...EXITING\n"
          exit 1
esac
```

Listing 17-14 Operating system test

Listing 17-14 shows simple enough replacement code. In this case statement we either execute one of the functions or exit if the OS is not in the list with a return code of 1, one. In these functions we will want to pay attention to the command syntax for each operating system, the columns to extract for the desired data, and the file systems that we want to ignore, if any. There is an egrep, or extended grep, in each statement that will allow for exclusions to the filesystems that are monitored. A typical example of this is a CD-ROM. Remember that a CD-ROM will always show that it is 100 percent utilized because it is mounted as read-only and you cannot write to it. Also, some operating systems list *mount points* that are really not meant to be monitored, such as /proc in AIX5L.

Command Syntax and Output Varies between Operating Systems

The command syntax and command output varies between UNIX operating systems. To get a similar output of the AIX df –k command on other operating systems, we sometimes have to change the command syntax. We also extract data from different columns in the output. The command syntax and resulting output for AIX, Linux, HP-UX, OpenBSD, and SUN/Solaris are listed in the text that follows as well as the columns of interest for each operating system output. Notice that HP-UX uses the bdf command instead of df –k. Please review Tables 17-2 through 17-11.

FILESYSTEM	1024-BLOCKS	FREE	%USED	IUSED	%IUSED	MOUNTED ON
/dev/hd4	32768	16376	51%	1663	11%	/
/dev/hd2	1212416	57592	96%	36386	13%	/usr
/dev/hd9var	53248	30824	43%	540	5%	/var
/dev/hd3	106496	99932	7%	135	1%	/tmp
/dev/hd1	4096	3916	5%	25	%	/home
/proc						/proc
/dev/hd10opt	638976	24456	97%	15457	10%	/opt
/dev/scripts_lv	102400	95264	7%	435	2%	/scripts
/dev/cd0	656756	0	100%	328378	100%	/cdrom

Table 17-2 AIX df -k Con	nmand Output
--------------------------	--------------

Table 17-3 AIX df -k Output Columns of Interest

DF OUTPUT COLUMNS	COLUMN CONTENTS
Column 1	The filesystem device name, Filesystem
Column 2	The size of the filesystem in 1024 blocks, 1024-blocks
Column 3	The kilobytes of free filesystem space, Free
Column 4	The percentage of used capacity, %Used
Column 7	The mount point of the filesystem, Mounted on

Table 17-4 Linux df -k Command Output

FILESYSTEM	1K-BLOCKS	USED	AVAILABLE	USE%	MOUNTED ON
/dev/hda16	101089	32949	62921	34%	/
/dev/hda5	1011928	104	960420	0%	/backup
/dev/hda1	54416	2647	48960	5%	/boot
/dev/hda8	202220	13	191767	0%	/download
/dev/hda9	202220	1619	190161	1%	/home
/dev/hda12	124427	19	117984	0%	/tmp
/dev/hda6	1011928	907580	52944	94% /usr	_
/dev/hda10	155545	36	147479	0%	/usr/local
/dev/hda11	124427	29670	88333	25%/var	

DF OUTPUT COLUMNS	COLUMN CONTENTS
Column 1	The filesystem device name, Filesystem
Column 2	The size of the filesystem in 1k-blocks, 1k-blocks
Column 4	The kilobytes of free filesystem space, Available
Column 5	The percentage of used capacity, Use%
Column 6	The mount point of the filesystem, $Mounted$ on

Table 17-5 Linux df -k Output Columns of Interest

Table 17-6 OpenBSD df -k Command Output

FILESYSTEM	1K-BLOCKS	USED	AVAIL	CAPACITY	MOUNTED ON
/dev/wd0a	4918134	39672	4633506	1%	/
/dev/wd0i	4399006	10844	4168212	0%	/backup
/dev/wd0e	489550	2	465072	0%	/home
/dev/wd0h	2458254	12	2335330	0%	/scripts
/dev/wd0g	9840894	6	9348844	0%	/tmp
/dev/wd0d	9840984	516170	8832680	6%	/usr
/dev/wd0f	49166	7666	39042	16%	/var

Table 17-7 OpenBSD df -kOutput Columns of Interest

DF OUTPUT COLUMNS	COLUMN CONTENTS
Column 1	The filesystem device name, Filesystem
Column 2	The size of the filesystem in 1k-blocks, ${\tt 1k-blocks}$
Column 4	The kilobytes of free filesystem space, Avail
Column 5	The percentage of used capacity, Capacity
Column 6	The mount point of the filesystem, Mounted on

Table 17-8 SUN/Solaris df -k Command Output

FILESYSTEM	KBYTES	USED	AVAIL	CAPACITY	MOUNTED ON
/dev/dsk/c0d0s0	192423	18206	154975	11%	
/dev/dsk/c0d0s6	1015542	488678	465932	52%	/usr
/proc	0	0	0	0%	/proc

(continued)

FILESYSTEM	KBYTES	USED	AVAIL	CAPACITY	MOUNTED ON
Fd	0	0	0	0%	/dev/fd
Mnttab	0	0	0	0%	/etc/mnttab
/dev/dsk/c0d0s3	96455	5931	80879	7%	/var
Swap	554132	0	55413	0%	/var/run
/dev/dsk/c0d0s5	47975	1221	41957	3%	/opt
Swap	554428	296	554132	1%	/tmp
/dev/dsk/c0d0s7	1015542	1	954598	1%	/export/home
/dev/dsk/c0d0s1	375255	214843	122887	64%	/usr/openwin

Table 17-8 (continued)

Table 17-9 Sun/Solaris df -k Output Columns of Interest

DF OUTPUT COLUMNS	COLUMN CONTENTS
Column 1	The filesystem device name, Filesystem
Column 2	The size of the filesystem in 1k-blocks, <code>kbytes</code>
Column 4	The kilobytes of free filesystem space, avail
Column 5	The percentage of used capacity, capacity
Column 6	The mount point of the filesystem, Mounted on

Table 17-10 HP-UX bdf Command Output

FILESYSTEM	KBYTES	USED	AVAIL	%USED	MOUNTED ON
/dev/vg00/lvol3	151552	89500	58669	60%	/
/dev/vg00/lvol1	47829	24109	18937	56%	/stand
/dev/vg00/lvol9	1310720	860829	422636	67%	/var
/dev/vg00/lvol8	972800	554392	392358	59%	/usr
/dev/vg13/lvol1	4190208	1155095	2850597	29%	/u2
/dev/vg00/lvol7	102400	4284	92256	4%	/tmp
/dev/vg00/lvol13	2039808	1664073	352294	83%	/test2
/dev/vg00/lvol6	720896	531295	177953	75%	/opt
/dev/vg00/lvol5	409600	225464	176663	56%	/home

DF OUTPUT COLUMNS	COLUMN CONTENTS
Column 1	The filesystem device name, Filesystem
Column 2	The size of the filesystem in 1k-blocks, <code>kbytes</code>
Column 4	The kilobytes of free filesystem space, avail
Column 5	The percentage of used capacity, %used
Column 6	The mount point of the filesystem, Mounted on

Table 17-11 HP-UX bdf Output Columns of Interest

Now that we know how the commands and output vary between operating systems, we can take this into account when creating the shell functions to load the correct filesystem data for each system. Note in each of the following functions that one or more filesystems or devices are set to be ignored, which is specified by the egrep part of the statement:

```
*****
function load_AIX_FS_data
{
  df -k | tail +2 | egrep -v '/dev/cd[0-9] //proc' \
       | awk '{print $1, $2, $3, $4, $7}' > $WORKFILE
}
**********
function load_HP_UX_FS_data
{
  bdf | tail +2 | egrep -v '/mnt/cdrom' \
     | awk '{print $1, $2, $4, $5, $6}' > $WORKFILE
}
****
function load_LINUX_FS_data
{
  df -k | tail +2 | egrep -v '/mnt/cdrom'
       awk '{print $1, $2, $4, $5, $6}' > $WORKFILE
}
*****
function load_OpenBSD_FS_data
{
  df -k | tail +2 | egrep -v '/mnt/cdrom'
       awk '{print $1, $2, $4, $5, $6}' > $WORKFILE
}
```

Each UNIX system is different, and these functions may need to be modified for your particular environment. The script modification to execute on all of the five operating systems includes entering the functions into the top part of the script, where functions are defined, and to replace the current load_FS_data function with a case statement that utilizes the get_OS_info function. This is an excellent example of how using functions can make life doing modifications much easier.

Programming a Shell-Neutral Script

What is the primary issue when executing a shell script in sh, ksh, and Bash environments? The echo command! Yep, I like to use the backslash operators for cursor control (n, c, t, b, and so on), and if the executing shell is not taken into consideration, the resulting output shows the cursor control operators as plain text. In Bourne and Bash shells, to use the backslash operators, the echo –e syntax must be used. The case statement to select the correct echo command is shown in Listing 17-15.

```
###### SHELL SPECIFIC VARIABLES ######
# Set the correct usage for the echo command for found $SHELL
case $SHELL in
*/bin/Bash) ECHO="echo -e"
    ;;
*/bin/ksh) ECHO="echo"
    ;;
*/bin/sh) ECHO="echo -e"
    ;;
*/bin/sh) ECHO="echo"
    ;;
esac
```

Listing 17-15 Using a case statement to use the echo command correctly

In the shell script, whenever an echo is needed, the script syntax uses \$ECHO, and the proper command usage is automatically used.

These scripts were written in Korn shell (ksh) to illustrate using different commands. Some systems' shells do not support the print command except ksh. The *final* script (it is never a final script!) will look like the following code, shown in Listing 17-16. Please scan through the boldface text in detail.

```
#!/usr/bin/ksh
# SCRIPT: fs_mon_ALL_OS.ksh
# AUTHOR: Randy Michael
# DATE: 09-25-2007
# REV: 6.0.P
# PURPOSE: This script is used to monitor for full filesystems,
      which are defined as exceeding. the MAX_PERCENT value.
#
#
      A message is displayed for all .full. filesystems.
# PLATFORM: AIX, Linux, HP-UX, OpenBSD, and Solaris
#
# REV LIST:
#
       Randy Michael - 08-27-2007
#
        Changed the code to use MB of free space instead of
#
       the %Used method.
#
#
       Randy Michael - 08-27-2007
#
       Added code to allow you to override the set script default
#
       for MIN_MB_FREE of FS Space
#
#
        Randy Michael - 08-28-2007
#
       Changed the code to handle both %Used and MB of Free Space.
#
        It does an .auto-detection. but has override capability
#
        of both the trigger level and the monitoring method using
#
        the exceptions file pointed to by the $EXCEPTIONS variable
#
#
       Randy Michael - 08-28-2007
#
       Added code to allow this script to be executed on
#
        AIX, Linux, HP-UX, and Solaris
#
#
        Randy Michael . 09=25=2007
#
        Added code for OpenBSD support
# set -n # Uncomment to check syntax without any execution
# set -x # Uncomment to debug this script
##### DEFINE FILES AND VARIABLES HERE ####
MIN_MB_FREE="100MB"
                       # Min. MB of Free FS Space
MAX PERCENT="85%"
                       # Max. FS percentage value
FSTRIGGER="1000MB"
                      # Trigger to switch from % Used to MB Free
WORKFILE="/tmp/df.work" # Holds filesystem data
>$WORKFILE
                        # Initialize to empty
OUTFILE="/tmp/df.outfile" # Output display file
```

Listing 17-16 fs_mon_ALL_OS.ksh shell script

```
>$OUTFILE
                  # Initialize to empty
EXCEPTIONS="/usr/local/bin/exceptions" # Override data file
DATA_EXCEPTIONS="/tmp/dfdata.out" # Exceptions file w/o # rows
EXCEPT_FILE="N"
                    # Assume no $EXCEPTIONS FILE
THISHOST=`hostname`
                   # Hostname of this machine
###### FORMAT VARIABLES HERE ######
# Both of these variables need to be multiplied by 1024 blocks
(( MIN_MB_FREE = $(echo $MIN_MB_FREE | sed s/MB//g) * 1024 ))
((FSTRIGGER = $(echo $FSTRIGGER | sed s/MB//g) * 1024 ))
###### SHELL SPECIFIC VARIABLES ######
# Set the correct usage for the echo command for found $SHELL
case $SHELL in
*/bin/Bash) ECHO="echo -e"
           ;;
 */bin/ksh) ECHO="echo"
           ;;
 */bin/sh) ECHO="echo -e"
           ;;
        *) ECHO="echo"
           ;;
esac
*****
####### DEFINE FUNCTIONS HERE ########
function get_OS_info
# For a few commands it is necessary to know the OS and its level
# to execute the proper command syntax. This will always return
# the OS in UPPERCASE
typeset -u OS # Use the UPPERCASE values for the OS variable
OS=`uname` # Grab the Operating system, i.e. AIX, HP-UX
print $05 # Send back the UPPERCASE value
3
function check_exceptions
# set -x # Uncomment to debug this function
```

```
while read FSNAME FSLIMIT
do
    IN FILE="N"
    # Do an NFS sanity check and get rid of any .:..
    # If this is found it is actually an error entry
    # but we will try to resolve it. It will only
    # work if it is an NFS cross mount to the same
    # mount point on both machines.
    $ECHO $FSNAME | grep ':' >/dev/null \
         && FSNAME=$($ECHO $FSNAME | cut -d ':' -f2)
    # Check for empty and null variable
    if [[ ! -z $FSLIMIT && $FSLIMIT != '' ]]
    then
        if [[ $FSNAME = $FSMOUNT ]] # Found it!
        then
            # Check for "MB" Characters...Set IN_FILE=MB
            $ECHO $FSLIMIT | grep MB >/dev/null && IN_FILE="MB" \
                 && (( FSLIMIT = $($ECHO $FSLIMIT \
                       | sed s/MB//g) * 1024 ))
            # check for '%' Character...Set IN_FILE=PC, for %
            $ECHO $FSLIMIT | grep '%' >/dev/null && IN_FILE="PC" \
                 && FSLIMIT=$($ECHO $FSLIMIT | sed s/\%//g)
            case $IN_FILE in
            MB) # Use MB of Free Space Method
                # Up-case the characters, if they exist
                FSLIMIT=$($ECHO $FSLIMIT | tr '[a-z]' '[A-Z]')
                # Get rid of the 'MB' if it exists
                FSLIMIT=$($ECHO $FSLIMIT | sed s/MB//g)
                # Test for blank and null values
                if [[ ! -z $FSLIMIT && $FSLIMIT != '' ]]
                then
                   # Test for a valid filesystem 'MB' limit
                   if (( FSLIMIT >= 0 && FSLIMIT < FSSIZE ))
                   then
                      if (( FSMB_FREE < FSLIMIT ))
                      then
                         return 1 # Found out of limit
                                  # using MB Free method
                      else
                         return 3 # Found OK
                      fi
                   else
                       $ECHO "\nERROR: Invalid filesystem MAX for\
 $FSMOUNT - $FSLIMIT"
                       $ECHO "
                                    Exceptions file value must be less
```

```
than or"
                     $ECHO "
                                   equal to the size of the filesystem
measured"
                      ŚECHO "
                                    in 1024 bytes\n"
                  fi
               else
                  $ECHO "\nERROR: Null value specified in exceptions\
file"
                  $ECHO "
                                for the $FSMOUNT mount point.\n"
               fi
               ;;
           PC) # Use Filesystem %Used Method
               # Strip out the % sign if it exists
               PC_USED=$($ECHO $PC_USED | sed s/\%//g)
               # Test for blank and null values
               if [[ ! -z "$FSLIMIT" && "$FSLIMIT" != '' ]]
               then
                  # Test for a valid percentage, i.e. 0-100
                  if (( FSLIMIT >= 0 && FSLIMIT <= 100 ))
                  then
                     if (( $PC_USED > $FSLIMIT ))
                     then
                        return 2 # Found exceeded by % Used method
                     else
                        return 3 # Found OK
                     fi
                  else
                     $ECHO "\nERROR: Invalid percentage for $FSMOUNT -\
$FSLIMIT"
                     $ECHO "
                                  Exceptions file values must be"
                     $ECHO "
                                  between 0 and 100%\n"
                  fi
               else
                  $ECHO "\nERROR: Null value specified in exceptions\
file"
                  $ECHO "
                               for the $FSMOUNT mount point.\n"
               fi
               ;;
           N) # Method Not Specified - Use Script Defaults
               if (( FSSIZE >= FSTRIGGER ))
               then # This is a "large" filesystem
                   if (( FSMB_FREE < MIN_MB_FREE ))
                   then
                        return 1 # Found out of limit
                                 # using MB Free method
                   else
                        return 3 # Found OK
                   fi
```

```
else # This is a standard filesystem
                  PC_USED=$($ECHO $PC_USED | sed s/\%//g) # Remove %
                  FSLIMIT=$($ECHO $FSLIMIT | sed s/\%//g) # Remove %
                  if (( PC_USED > FSLIMIT ))
                  then
                     return 2 # Found exceeded by % Used method
                  else
                     return 3 # Found OK
                  fi
              fi
              ;;
          esac
       fi
   fi
done < $DATA_EXCEPTIONS # Feed the loop from the bottom!!!
return 4 # Not found in SEXCEPTIONS file
}
*****
function display_output
{
if [[ -s $OUTFILE ]]
then
     $ECHO "\nFull Filesystem(s) on $THISHOST\n"
     cat $OUTFILE
     print
fi
}
*****
function load_EXCEPTIONS_data
{
# Ignore any line that begins with a pound sign, #
# and omit all blank lines
cat $EXCEPTIONS | grep -v "^#" | sed /^$/d > $DATA_EXCEPTIONS
}
****
function load_AIX_FS_data
{
  df -k | tail +2 | egrep -v "/dev/cd[0-9] /proc" \
        awk '{print $1, $2, $3, $4, $7}' > $WORKFILE
```

Listing 17-16 (continued)

```
}
*****
function load_HP_UX_FS_data
{
  bdf | tail +2 | egrep -v "/cdrom" \
      awk '{print $1, $2, $4, $5, $6}' > $WORKFILE
}
*****
function load_LINUX_FS_data
{
  df -k | tail -n 2 | egrep -v "/cdrom"\
      awk '{print $1, $2, $4, $5, $6}' > $WORKFILE
}
****
function load OpenBSD FS data
{
  df -k | tail +2 | egrep -v "/mnt/cdrom"\
      awk '{print $1, $2, $4, $5, $6}' > $WORKFILE
}
*****
function load_Solaris_FS_data
{
  df -k | tail +2 | egrep -v "/dev/fd|/etc/mnttab|/proc"\
      awk '{print $1, $2, $4, $5, $6}' > $WORKFILE
}
# Query the operating system to find the Unix flavor, then
# load the correct filesystem data for the resident OS
case $(get_OS_info) in
  AIX) # Load filesystem data for AIX
       load_AIX_FS_data
    ;;
```

```
HP-UX) # Load filesystem data for HP-UX
          load_HP_UX_FS_data
      ;;
   LINUX) # Load filesystem data for Linux
          load LINUX FS data
      ;;
   OPENBSD) # Load filesystem data for OpenBSD
          load_OpenBSD_FS_data
      ;;
   SUNOS) # Load filesystem data for Solaris
          load_Solaris_FS_data
     ;;
   *)
          # Unsupported in script
          $ECHO "\nUnsupported Operating System for this\
 Script...EXITING\n"
          exit 1
esac
# Do we have a nonzero size $EXCEPTIONS file?
if [[ -s $EXCEPTIONS ]]
then # Found a nonempty $EXCEPTIONS file
    load_EXCEPTIONS_data
    EXCEP_FILE="Y"
fi
while read FSDEVICE FSSIZE FSMB_FREE PC_USED FSMOUNT
do
     if [[ $EXCEP_FILE = "Y" ]]
     then
         check_exceptions
         CE_RC="$?" # Check Exceptions Return Code (CE_RC)
         case $CE_RC in
         1) # Found exceeded in exceptions file by MB Method
            ((FS FREE OUT = FSMB FREE / 1000))
            $ECHO "$FSDEVICE mounted on $FSMOUNT has ${FS_FREE_OUT}MB\
 Free" >> $OUTFILE
          ::
         2) # Found exceeded in exceptions file by %Used method
            $ECHO "$FSDEVICE mount on $FSMOUNT is ${PC_USED}%" \
                  >> $OUTFILE
          ;;
         3) # Found OK in exceptions file
            : # NO-OP Do Nothing. A ':' is a no-op!
          ;;
```

```
Listing 17-16 (continued)
```

```
4) # Not found in exceptions file - Use Default Triggers
            if (( FSSIZE >= FSTRIGGER ))
            then # This is a "large" filesystem
            FSMB_FREE=$($ECHO $FSMB_FREE | sed s/MB//g) # Remove the "MB"
              if (( FSMB_FREE < MIN_MB_FREE ))
              then
                (( FS_FREE_OUT = FSMB_FREE / 1000 ))
               $ECHO "$FSDEVICE mounted on $FSMOUNT has {FS FREE OUT}MB\
Free" >> SOUTFILE
              fi
            else # This is a standard filesystem
                PC_USED=$($ECHO $PC_USED | sed s/\%//g)
                MAX_PERCENT=$($ECHO $MAX_PERCENT | sed s/\%//g)
                if (( PC_USED > MAX_PERCENT ))
                then
                    $ECHO "$FSDEVICE mount on $FSMOUNT is ${PC_USED}%" \
                          >> $OUTFILE
                fi
            fi
          ;;
         esac
     else # NO $EXCEPTIONS FILE USE DEFAULT TRIGGER VALUES
          if (( FSSIZE >= FSTRIGGER ))
          then # This is a "large" filesystem - Use MB Free Method
           FSMB_FREE=$($ECHO $FSMB_FREE | sed s/MB//g) # Remove the 'MB'
            if (( FSMB_FREE < MIN_MB_FREE ))
            then
              ((FS_FREE_OUT = FSMB_FREE / 1000))
              ECHO "$FSDEVICE mounted on $FSMOUNT has \
                    ${FS_FREE_OUT}MB Free" >> $OUTFILE
            fi
          else # This is a standard filesystem - Use % Used Method
              PC_USED=$($ECHO $PC_USED | sed s/\%//g)
              MAX_PERCENT=$($ECHO $MAX_PERCENT | sed s/\%//g)
              if (( PC_USED > MAX_PERCENT ))
              then
                  $ECHO "$FSDEVICE mount on $FSMOUNT is ${PC_USED}%" \
                       >> $OUTFILE
              fi
          fi
     fi
done < $WORKFILE # Feed the while loop from the bottom !!!!!
display_output
# End of Script
```

A good study of the script in Listing 17-16 will reveal some nice ways to handle the different situations we encounter while writing shell scripts. As always, it is intuitively obvious!

The /usr/local/bin/exceptions file in Listing 17-17 is used on yogi.

```
# FILE: "exceptions"
#
# This file is used to override the default
# trigger value in the filesystem monitoring script
# fs_mon_ALL_OS_except.ksh, but also allows overriding the
# monitoring technique used, i.e. Max %Used and
# MINIMUM MB FREE of filesystem space. The syntax to
# override is a /mount-point and a "trigger value" with
# either "%" or "MB" as a suffix.
# EXAMPLES:
#
 /usr 96%
# OR
# /usr 50MB
#
# All lines beginning with a # are ignored.
# NOTE: All Entries MUST have either "MB" or
#
       "%" as a suffix !!! Or else the script
#
       defaults are used. NO SPACES PLEASE!
#
/opt 95%
/ 50%
/usr 70MB
/home 50MB
```

Listing 17-17 Sample exceptions file

Listing 17-17 should work, but it gives an error. If the monitoring script is executed using these exception file entries, it will result in the following output:

```
ERROR: Invalid filesystem MINIMUM_MB_FREE specified
    for /home - 50MB -- Current size is 4MB.
    Exceptions file value must be less than or equal
    to the size of the filesystem measured Megabytes
Full Filesystem(s) on yogi
/dev/hd4 mount on / is 51%
/dev/hd2 mounted on /usr has 57MB Free
/dev/hd10opt mount on /opt is 97%
```

The problem is with the /home filesystem entry in the \$EXCEPTIONS file. The value specified is 50 Megabytes, and the /home filesystem is only 4 MB in size. In a case like this the check_exceptions function will display an error message and then use the shell script default values to measure the filesystem and return an appropriate return code to the calling script. So, if a modification is made to the exceptions file, the script needs to be run to check for any errors.

The important thing to note is that error checking and data validation should take place before the data is used for measurement. This sequence will also prevent any messages from standard error (stderr) that the system may produce.

Other Options to Consider

With one-terabyte disk drives coming out this year, the data explosion is heading to every home. The Lab Assignments give you a chance to modify the shell scripts to measure filesystem usage in the GB range and above.

We can always improve on a script, and the full filesystems script is no exception. Here are some thoughts.

Event Notification

Because monitoring for full filesystems should involve event notification, it is wise to modify the display_output function to send some kind of message, whether by page or email, or otherwise this information needs to be made known so that we can call ourselves proactive. Sending an email to your pager and desktop would be a good start. An entry like the statement that follows might work, but its success depends on the mail server and firewall configurations:

```
echo "Full Filesystem(s) on $THISHOST\n" > $MAILFILE
cat $OUTFILE >> $MAILFILE
mailx -s "Full Filesystem(s) on $THISHOST" $MAIL_LIST < $MAILFILE</pre>
```

For pager notification, the text message must be *very short*, but descriptive enough to get the point across. You can also set up a mail alias in sendmail, maybe something like fs_support. A mail alias is easier and more intuitive to maintain for the Systems Administrators.

Automated Execution

If we are to monitor the system, we want the system to tell us when *it* has a problem. We want event notification, but we also want the event notification to be automated. For filesystem monitoring, a cron table entry is the best way to do this. An interval of about 10-15 minutes 24×7 is most common. We have the exceptions capability built in so that if pages become a problem, the exceptions file can be modified to stop the

filesystem from being *in error*, and thus stop the paging. The cron entry that follows will execute the script every 10 minutes, on the 5s, 24 hours a day, 7 days a week:

5,15,25,35,45,55 * * * * /usr/local/bin/fs_mon_ALL_OS.ksh 2>&1

To make this cron entry you can either edit a cron table with crontab -e or use the following command sequence to append an entry to the end of the cron table:

For this to work, the fs_mon_ALL_OS.ksh script must be modified to send notification by some method. Paging, email, SNMP traps, and modem dialing are the preferred methods. You could send this output to the systems console, but who would ever see it?

Modify the egrep Statement

It may be wise to remove the egrep part of the df statement, used for filesystem exclusion, and use another method. As pointed out previously, grepping can be a mistake. Grepping was done here because most of the time we can get a unique character string for a filesystem device to make grep and egrep work without error, but not always. If this is a problem, creating a list either in a variable assignment in the script or in a file is the best bet. Then the new \$IGNORE_LIST list can be searched and an exact match can be made.

Summary

Throughout this chapter we have changed our thinking about monitoring for full filesystems. The script that we use can be very simple for the average small shop, or more complex as we move to larger and larger storage solutions. All filesystems are not created equal in size, and when you get a mix of large and small filesystems on mixed operating systems, we have shown how to handle the mix with ease. Now we get to play with terabyte disk drives!

In the next chapter we will move into monitoring the paging and/or swap space. If we run out of paging or swap space, the system will start thrashing, and if the problem is chronic, the system may crash. We will look at the different monitoring methods for each operating system.

Lab Assignments

- 1. Modify the fs_mon_ALL_OS.ksh shell script so that the *exceptions* file will support not only MB (megabyte) limits, but will now support GB (gigabyte) exception limits. Name the new file fs_mon_ALL_OS_GB.ksh.
- 2. Modify the fs_mon_ALL_OS_GB.ksh shell script to use df -m (for megabyte) instead of df -k (kilobytes) to measure disk utilization. Name the new file fs_mon_ALL_OS_GB_DF-M.ksh.

CHAPTER 18

Monitoring Paging and Swap Space

Every Systems Administrator loves paging and swap space because they are the magic bullets to fix a system that does not have enough real physical memory — right? *Wrong!* This misconception is thought to be true by many people, at various levels, in a lot of organizations. The fact is that if your system does not have enough real memory to run your applications, adding more paging/swap space is not going to help. Depending on the system's hardware and the application(s) running on your system, swap space should start at least 1.5 times physical memory. Many high-performance applications such as SAP, Oracle, and DB2 require 4 to 10 GB of real memory, plus at least 2 times real memory in swap/paging space. *Did you notice I said GB of real memory?* At a minimum we need 8 GB of paging space, and it could be as high as 20 GB; so the actual amount of paging and swap space is variable, but 1.5 times is a good place to start. Use the application's recommended requirement, if one is suggested, as a starting point. Just be sure to take into account any other applications that are also running on the system.

Some of you may be asking, What is the difference between *paging* space and *swap* space? It depends on the UNIX flavor whether your system does swapping or paging, but both swap space and paging space are disk storage that makes up virtual memory along with real, or physical, memory. A *page fault* happens when a memory segment, or *page*, is needed in memory but is not currently resident in memory. When a page fault occurs, the system attempts to load the needed data into memory; this is called paging or swapping, depending on the UNIX system you are running. When the system is doing a lot of paging in and out of memory we need to be able to monitor this activity. If your system runs out of paging space or is in a state of continuous swapping, such that as soon as a segment is paged out of memory it is immediately needed again, the system is *thrashing*. If this thrashing condition continues for very long, you have a risk of the system crashing. In this chapter we are going to use the terms "paging" and "swapping" interchangeably.

Each of our five UNIX flavors (AIX, HP-UX, Linux, OpenBSD, and Solaris) uses different commands to list the swap-space usage. The output for each command and OS varies also. The goal of this chapter is to create six shell scripts: one script of each of the five operating systems and an all-in-one shell script that will run on any of our five UNIX flavors. Each of the shell scripts must produce the exact same output, which is shown in Listing 18-1.

Paging Space Report for yogi Wed Jun 5 21:48:16 EDT 2007 Total MB of Paging Space: 336MB Total MB of Paging Space Used: 33MB Percent of Paging Space Used: 10% Percent of Paging Space Free: 90%

Listing 18-1 Required paging- and swap-space report

Before we get started creating the shell scripts, we need the command syntax for each operating system. Each of the commands produces a different result, so this should be an interesting chapter in which we can try some varied techniques.

Syntax

As usual, we need the correct command syntax before we can write a shell script. As we go through each of the operating systems, the first thing I want you to notice is the command syntax used and the output received back. Because we want each UNIX flavor to produce the same output, as shown in Listing 18-1, we are going to have to do some math. This is not going to be hard math, but each of the paging- and swap-space command outputs is lacking some of the desired information, so we must calculate the missing pieces. Now we are going to see the syntax for each operating system.

AIX lsps Command

AIX does paging instead of swapping. This technique uses 4096-byte-block pages. When a page fault occurs, AIX has a complex algorithm that frees memory of the least used noncritical memory page to disk paging space. When the memory has space available, the page of data is paged in to memory. To monitor paging-space usage in AIX, you use the **lsps** command, which stands for *list paging space*. The lsps command has two command options: -a, to list each paging space separately, and -s, to show a summary of all paging spaces. Both lsps options are shown here:

```
# lsps -a
Page Space Physical Volume Volume Group Size %Used Active Auto Type
paging00 hdisk2 rootvg 1024MB 11 yes yes lv
hd6 hdisk0 rootvg 1024MB 9 yes yes lv
# lsps -s
Total Paging Space Percent Used
    2048MB 10%
```

From the first command output, lsps -a, on this system notice that there are two paging spaces defined, paging00 and hd6, both are the same size at 1 GB each, and each paging space is on a separate disk. This is an important point. In AIX, paging space is used in a round-robin fashion, starting with the paging space that has the largest area of free space. If one paging space is significantly larger, the round-robin technique is defeated, and the system will almost always use the larger paging space. This has a negative effect on performance because one disk will take all of the paging activity.

In the second output, lsps -s, we get a summary of all of the paging space usage. Notice that the only data that we get is the total size of the paging space and the percentage used. From these two pieces of data we must calculate the remaining parts of our required output, which is total paging space in MB, free space in MB, used space in MB, percent used, and percent free. We will cover these points in the scripting section for AIX later in this chapter.

HP-UX swapinfo Command

The HP-UX operating system uses swapping, which is evident by the command **swapinfo**. HP-UX does the best job of giving us the best detailed command output, so we need to calculate only one piece of data for our required output, percent of total swap space free. Everything else is provided with the swapinfo -tm command. The -m switch specifies to produce output in MB, and the -t switch specifies to produce a total line for a summary of all virtual memory. This command output is shown here:

[root@d	ino]/>	swapinfo	-tm					
	Mb	Mb	Mb	PCT	START/	Mb		
TYPE	AVAIL	USED	FREE	USED	LIMIT	RESERVE	PRI	NAME
dev	96	21	73	22%	928768	-	1	/dev/dsk/c0t6d0
reserve	-	46	-46					
memory	15	5	10	33%				
total	111	72	37	65%	-	0	-	

Notice in this output that HP-UX splits up virtual memory into three categories: dev, reserve, and memory. For our needs we could use the summary information that is shown in the total line at the bottom. As you can see on the total line,

the total virtual memory is 111 MB, and the system is consuming 72 MB of this total, which leaves 37 MB of free virtual memory. The fifth column shows that the system is consuming 65 percent of the available virtual memory. This total row is misleading, though, when we are interested only in the swap-space usage. The actual swap-space usage is located on the dev row of data at the top of the command output. As you can see, we need to calculate only the percent free, which is a simple calculation.

Linux free Command

Linux uses swapping and uses the **free** command to view memory and swap-space usage. The free command has several command switches, but the only one we are concerned with is the -m command switch to list output in MB. The swap information given by the free -m command is listed only in MB, and there are no percentages presented in the output. Therefore, from the total MB, used MB, and free MB, we must calculate the percentages for percentage used and percentage free. The following shows the free -m command output:

# free -m						
	total	used	free	shared	buffers	cached
Mem:	52	51	1	0	1	20
-/+ buffers/cache:		30	22			
Swap:	211	9	202			

The last line in this output has the swap information listed in MB, specified by the -m switch. This command output shows that the system has 211 MB of total swap-space, of which 9 MB has been used and 202 MB of swap space is free.

OpenBSD swapctl Command

OpenBSD uses swapping and uses the **swapctl** command to view swap-space usage. The swapctl command has several command switches, but the only ones we are concerned with are the -1 and -k command switches. The swap information given by the swapctl -1k command is listed in KB by using the -k switch. Percentages presented in the output show the percentage of current usage. Because we have our swap-space usage indicated in KB, we need to multiple by 1,000 for our standard MB output. The following shows the swapctl -1k command output:

# swapctl -	lk				
Device	1K-blocks	Used	Avail	Capacity	Priority
swap_device	500000	0	500000	0%	0

As we can see in this OpenBDS output, the system has 500 MB of swap space defined, and 0 percent and 0 MB are currently utilized.

Solaris swap Command

The Solaris operating system does swapping, as indicated by the command **swap**. Of the swap command switches we are concerned with only the -s switch, which produces a summary of swap-space usage. All output from this command is produced in KB so we have to do a little division by 1,000 to get our standard MB output. Like Linux, the Solaris swap output does not show the swap status using percentages, so we must calculate these values. The swap -s output is shown here:

```
# swap -s
total: 26788k bytes allocated + 7256k reserved = 34044k used, 557044k
available
```

This is an unusual output to decipher because the data is all on the same line, but because Solaris attempts to create a mathematical statement we will have to use our own mathematical statements to fill in the blanks to get our required script output. The swap -s command output shows that the system has used a total of 34 MB and it has 557 MB of free swap space. We must calculate the total MB, the percentage used, and the percentage of free swap space. These calculations are not too hard to handle as we will see in the shell scripting section for Solaris later in this chapter.

Creating the Shell Scripts

Now that we have the basic syntax of the commands to get paging- and swap-space statistics, we can start our scripting of the solutions. In each case you should notice which pieces of data are missing from our required output, as shown in Listing 18-1. All of these shell scripts are different. Some pipe command outputs to a while loop to assign the values to variables, and some use other techniques to extract the desired data from the output. Please study each shell script in detail, and you will learn how to handle the different situations you are challenged with when working in a heterogeneous environment.

AIX Paging Monitor

As we previously discussed, the AIX <code>lsps -s</code> command output shows only the total amount of paging space measured in MB and the percentage of paging space that is currently in use. To get our standard set of data to display we need to do a little math. This is not too difficult when you take one step at a time. In this shell script let's use a file to store the command output data. To refresh your memory the <code>lsps -s</code> command output is shown again here (this output is using a different AIX system):

The first thing we need to do is to remove the column headings. I like to use the tail command in a pipe for this purpose. The command syntax is shown in the next statement:

lsps -s | tail +2 336MB 2%

The tail + 2 says to start listing the end of the file starting at line 2. This resulting output contains only the data, without the column headings. The next step is to store these values in variables so that we can work with them for some calculations. We are going to use a file for initial storage and then use a while read loop, which we feed from the bottom using input redirection with the filename. Of course, we could have piped the command output to the while read loop, but I want to vary the techniques in each shell script in this chapter. Let's look at the first part of the data gathering and the use of the while read loop, as shown in Listing 18-2.

```
PAGING_STAT=/tmp/paging_stat.out # Paging Stat hold file
# Load the $PAGING_STAT file with data
lsps -s | tail +2 > $PAGING_STAT
# Use a while loop to assign the values to variables
while read TOTAL PERCENT
do
        DO CALCULATIONS HERE
done < $PAGING_STAT</pre>
```

Listing 18-2 Logical view of AIX lsps -s data gathering

Notice in Listing 18-2 that we first define a file to hold the data, which is pointed to by the \$PAGING_STAT variable. In the next step we redirect output of our paging space status command to the defined file. Next comes a while loop, where we read the file data and assign the first data field to the variable TOTAL and the second data field to the variable PERCENT.

Notice how the \$PAGING_STAT file is used to feed the while loop from the bottom. As you saw in Chapter 2, "24 Ways to Process a File Line-by-Line," this technique is one of the two fastest methods of reading data from a file. The middle of the while loop is where we do our calculations to fill in the blanks of our required output.

Speaking of calculations, we need to do three calculations for this script, but before we can perform the calculations on the data we currently have, we need to get rid of the suffixes attached to the variable data. The first step is to extract the MB from the \$TOTAL variable and then extract the percent sign, %, from the \$PERCENT variable. We do both of these operations using a **cut** command in a pipe, as shown here:

```
PAGING_MB=$(echo $TOTAL | cut -d 'MB' -f1)
PAGING_PC=$(echo $PERCENT | cut -d% -f1)
```

In both of these statements we use command substitution, specified by the $(command_statement)$ notation, to execute a command statement and assign the result to the variable specified. In the first statement we echo the TOTAL variable and pipe the output to the cut command. For the cut command we specify the *delimiter* to be MB, and we enclose it with single tic marks, 'MB'. Then we specify that we want the first field, specified by -f1. In the second statement we do the exact same thing, except that this time we specify that the percent sign, %, is the delimiter. The result of these two statements is that we have the PAGING_MB and PAGING_PC variables pointing to integer values without any other characters. Now we can do our calculations!

Let's do the most intuitive calculation first. We have the value of the percent of paging space used stored in the *\$PAGING_PC* variable as an integer value. To get the percent of free paging space, we need to subtract the percent used value from 100, as shown in the next command statement:

((PAGING_PC_FREE = 100 - PAGING_PC))

Notice that we used the double parentheses mathematical method, specified by the ((Math Statement)). I like this method because it is so intuitive to use. Also notice that you do *not* use the dollar sign, \$, with variables when using this method. Because the double parentheses method expects a mathematical statement, any character string that is not numeric is assumed to be a variable, so the dollar sign can be omitted.

The next calculation is not so intuitive to some. We want to calculate the MB of paging space that is currently in use. Now let's think about this. We have the percentage of paging space used, the percentage of paging space free, and the total amount of paging space measured in MB. To calculate the MB of used paging space, we can use the value of the total MB of paging space and the percentage of paging space used divided by 100, which converts the value of paging space used into a decimal value internally. See how this is done in the next statement:

((MB_USED = PAGING_MB * PAGING_PC / 100))

One thing to note in the preceding math statement: this will produce only an integer output. If you want to see the output in floating-point notation, you need to use the **bc** utility, which you will see in some of the following sections.

The last calculation is another intuitive calculation, to find the MB of free paging space. Because we already have the values for the total paging space in MB, and the MB of paging space in use, we need only to subtract the used value from the total. This is shown in the next statement:

((MB_FREE = PAGING_MB - MB_USED))

We have completed all of the calculations, so now we are ready to produce the required output for the AIX shell script. Take a look at the entire shell script shown in Listing 18-3, and pay particular attention to the boldface type.

```
#!/usr/bin/ksh
#
# SCRIPT: AIX_paging_mon.ksh
# AUTHOR: Randy Michael
# DATE: 5/31/2007
# REV: 1.1.P
# PLATFORM: AIX Only
#
# PURPOSE: This shell script is used to produce a report of
      the system's paging space statistics including:
#
#
#
      Total paging space in MB, MB of free paging space,
      MB of used paging space, % of paging space used, and
#
#
       % of paging space free
# REV LIST:
#
# set -x # Uncomment to debug this shell script
# set -n # Uncomment to check command syntax without any execution
PC_LIMIT=65
               # Percentage Upper limit of paging space
                # before notification
THISHOST=$(hostname)
               # Host name of this machine
PAGING_STAT=/tmp/paging_stat.out # Paging Stat hold file
echo "\nPaging Space Report for $THISHOST\n"
date
# Load the data in a file without the column headings
lsps -s | tail +2 > $PAGING_STAT
```

Listing 18-3 AIX_paging_mon.ksh shell script listing

```
# Start a while loop and feed the loop from the bottom using
# the $PAGING_STAT file as redirected input, after "done"
while read TOTAL PERCENT
do
     # Clean up the data by removing the suffixes
     PAGING_MB=$(echo $TOTAL | cut -d 'MB' -f1)
     PAGING_PC=$(echo $PERCENT | cut -d% -f1)
     # Calculate the missing data: %Free, MB used and MB free
     (( PAGING_PC_FREE = 100 - PAGING_PC ))
     (( MB USED = PAGING MB * PAGING PC / 100 ))
     (( MB_FREE = PAGING_MB - MB_USED ))
     # Produce the rest of the paging space report:
     echo "\nTotal MB of Paging Space:\t$TOTAL"
     echo "Total MB of Paging Space Used:\t${MB_USED}MB"
     echo "Total MB of Paging Space Free:\t${MB_FREE}MB"
     echo "\nPercent of Paging Space Used:\t${PERCENT}"
     echo "\nPercent of Paging Space Free:\t${PAGING_PC_FREE}%"
     # Check for paging space exceeded the predefined limit
     if ((PC_LIMIT <= PAGING_PC))</pre>
     then
          # Paging space is over the limit, send notification
         tput smso # Turn on reverse video!
         echo "\n\nWARNING: Paging Space has Exceeded the ${PC_LIMIT}% \
Upper Limit!\n"
         tput rmso # Turn off reverse video
     fi
done < $PAGING_STAT
rm -f $PAGING_STAT
# Add an extra new line to the output
echo "\n"
```

Listing 18-3 (continued)

There is one part of our shell script in Listing 18-3 that we have not covered yet. At the top of the script where we define variables, I added the PC_LIMIT variable. I normally set this threshold to 65 percent so that I will know when I have exceeded a safe system paging-space limit. When your system starts running at a

high paging-space level, you need to find the cause of this added activity. Sometimes developers do not write applications properly when it comes to deallocating memory. If a program runs for a long time and it is not written to clean up and release allocated memory, the program is said to have a *memory leak*. The result of running this memory leak program for a long time without a system reboot is that your system will run out of memory. When your system runs out of memory, it starts paging in and out to disk, and then your paging space starts edging up. The only way to correct this problem and regain your memory is to reboot the system, most of the time.

Notice at the end of the script that there is a test to see if the percentage of paging space used is greater than or equal to the limit that is set by the PC_LIMIT variable. If the value is exceeded, then reverse video is turned on so the WARNING message stands out on the screen. After the message is displayed, reverse video is turned back off. To turn on reverse video use the **tput smso** command. When reverse video is on, anything that you print to the screen appears in reverse video; however, do not forget to turn it off because this mode will continue *after* the shell script ends execution if you do not turn it off. To turn off the reverse video mode use the **tput rmso** command. Listings 18-4 and 18-5 show the shell script in action. Listing 18-4 shows a report of the system within the set 65 percent limit, and Listing 18-5 shows the report when the system has exceeded a 5 percent paging limit (I had to lower the threshold to 5 percent to test the shell script).

```
Paging Space Report for yogi
Fri Jun 7 15:47:08 EDT 2007
Total MB of Paging Space: 336MB
Total MB of Paging Space Used: 6MB
Total MB of Paging Space Free: 330MB
Percent of Paging Space Used: 2%
Percent of Paging Space Free: 98%
```

Listing 18-4 AIX_paging_mon.ksh in action

As you can see in Listing 18-4, yogi is not doing too much right now. Let's produce a little load on the system and set the trigger threshold in the AIX_paging_mon.ksh shell script to 5 percent so that we can see the threshold exceeded, as shown in Listing 18-5.

```
Paging Space Report for yogi
Fri Jun 7 15:54:30 EDT 2007
```

Listing 18-5 AIX_paging_mon.ksh exceeding a five percent paging limit

Total MB of Paging Space: 336MB Total MB of Paging Space Used: 23MB Total MB of Paging Space Free: 313MB Percent of Paging Space Used: 7% Percent of Paging Space Free: 93%

Listing 18-5 (continued)

This is still not much of a load, but it does make the point of the ability to set a trigger threshold for notification purposes. Of course, the reverse video of the warning message did not come to the page; believe me, it does show up in reverse video on the screen. Let's move on to the HP-UX system.

HP-UX Swap-Space Monitor

The HP-UX operating system does swapping, as shown by the swapinfo command. To check the statistics of swap space you use the swapinfo -tm command. The -t command switch adds a summary total line to the output, and the -m option specifies that the output space measurements are in MB, as opposed to the default of KB. As I said previously, HP-UX does the best job of producing the various virtual memory statistics, so we need to calculate only one piece of our required output, the percent of free swap space. Before we go any further, let's look at the command output we are dealing with, as shown in Listing 18-6.

swapinfo	-tm						
	Mb	Mb	Mb	PCT	START/	Mb	
TYPE	AVAIL	USED	FREE	USED	LIMIT	RESERVE	PRI NAME
dev	96	23	71	24% 92	28768	-	1 /dev/dsk/c0t6d0
reserve	-	45	-45				
memory	15	6	9	40%			
total	111	74	35	67%	-	0	-

Listing 18-6 HP-UX swapinfo -tm command output

As you can see, HP-UX shows paging space for devices, reserved memory, and real memory usage. I like to use the total row of output to get a good summary of what all of the virtual memory is doing. It really does not matter if you use the dev row or the total row to do your monitoring, but for this exercise I am going to use the dev row to monitor only the swap space and not worry about what real memory is doing.

The easiest way to extract the data we want on the dev row in the output is to use grep to pattern match on the string dev, because dev appears on only one row of data. Piping the swapinfo command output to a grep statement produces the following output:

```
# swapinfo -tm | grep dev
dev 96 23 71 24% 928768 - 1 /dev/dsk/c0t6d0
```

The output that we want to extract, Total MB, Used MB, Free MB, and Percent Used, is located in fields \$2, \$3, \$4, and \$5, respectively. From looking at this we have at least two options to assign the field values to variables. We can use five awk statements, or we can pipe the preceding command output to a while read loop. Of course, the while read loop runs for only one loop iteration. The easiest technique is to pipe to the while loop. The following command will get us started:

Notice in the while read portion of this statement how we assign unneeded fields to variables named junk and junk2. The first field, specified by the junk variable, targets dev; we are not interested in saving this field, so it gets a junk assignment. The last variable, junk2, is a catch-all for anything remaining on the line of output; specifically, "928768 - 1 /dev/dsk/c0t6d0" gets assigned to the variable junk2 as one field. This is an extremely important part of the while read statement because you must account for everything when reading in a line of data. Had I left out the junk2 variable, the PERCENT_USED variable would point to the data "24% 928768 - 1 /dev/dsk/c0t6d0", when the only thing we want is 24%. The junk2 variable catches all of the remaining data on the line and assigns it to the junk2 variable. This brings up another point. If you want to capture the entire line of data and assign it to a single variable, you can do this too by using the following syntax:

Using this syntax, all of the data is captured with a single variable, DATA_LINE, and the data is separated into fields just as it appears in the command output.

Back to our previous swapinfo statement, we have the data of interest stored in the following variables:

- SW_TOTAL Total swap space available on the system measured in MB
- SW_USED MB of swap space that is currently in use
- SW_FREE MB of swap space that is currently free
- PERCENT_USED Percentage of total swap space that is in use

The only part of our required output missing is the percentage of total swap space that is currently free. This is an easy calculation because we already have the \$PERCENT_USED. For the calculation we need to remove the percent sign, %, in the \$PERCENT_USED variable. The following statement does the removal of the percent sign and makes the calculation in one step:

```
((PERCENT_FREE = 100 - $(echo $PERCENT_USED | cut -d% -f1) ))
```

In the preceding mathematical statement, we assign 100 percent minus 24 percent to the variable PERCENT_FREE using command substitution to remove the percent sign from the \$PERCENT_USED variable using the cut command. In the cut part of the statement we define % to be the delimiter, or field separator, specified by -d%, then we extract the first field, 24 in this case, using the -f1 notation. Once the command substitution is complete, we are left with the following math statement:

 $((PERCENT_FREE = 100 - 24))$

Now let's examine the entire shell script that is shown in Listing 18-7.

```
#!/usr/bin/ksh
#
# SCRIPT: HP-UX_swap_mon.ksh
# AUTHOR: Randy Michael
# DATE: 5/31/2007
# REV: 1.1.P
# PLATFORM: HP-UX Only
#
# PURPOSE: This shell script is used to produce a report of
#
       the system's swap space statistics including:
#
#
   Total paging space in MB, MB of free paging space,
    MB of used paging space, % of paging space used, and
    % of paging space free
#
# REV LIST:
# set -x # Uncomment to debug this shell script
# set -n # Uncomment to check command syntax without any execution
PC_LIMIT=65
                     # Percentage Upper limit of paging space
                     # before notification
```

Listing 18-7 HP-UX_swap_mon.ksh shell script listing

```
THISHOST=$(hostname) # Host name of this machine
echo "\nSwap Space Report for $THISHOST\n"
date
# Start a while read loop by using the piped-in input from
# the swapinfo -tm command output.
swapinfo -tm | grep dev | while read junk SW_TOTAL SW_USED \
                         SW_FREE PERCENT_USED junk2
do
   # Calculate the percentage of free swap space
   ((PERCENT_FREE = 100 - $(echo $PERCENT_USED | cut -d% -f1) ))
   echo "\nTotal Amount of Swap Space:\t${SW_TOTAL}MB"
   echo "Total MB of Swap Space Used: \t$ {SW_USED}MB"
   echo "Total MB of Swap Space Free:\t${SW_FREE}MB"
   echo "\nPercent of Swap Space Used:\t${PERCENT_USED}"
   echo "\nPercent of Swap Space Free:\t${PERCENT FREE}%"
   # Check if paging space exceeded the predefined limit
   if (( PC_LIMIT <= $(echo $PERCENT_USED | cut -d% -f1) ))
   then
       # Swap space is over the predefined limit, send notification
       tput smso # Turn on reverse video!
       echo "\n\nWARNING: Swap Space has Exceeded the\
${PC_LIMIT}% Upper Limit!\n"
       tput rmso # Turn reverse video off!
   fi
done
echo "\n"
```

Listing 18-7 (continued)

There are a few things that I want to point out in Listing 18-7. The first point is that any time we use the *\$PERCENT_USED* value we always use command substitution to remove the percent sign, *%*, as shown in the following command substitution statement:

```
$(echo $PERCENT_USED | cut -d% -f1)
```

The next part I want to go over is our required report output. At the top of the shell script we initialize the report by stating a report header including the hostname of the machine and the date stamp of the time the report was executed. Then we do any calculation that is needed to gather any missing data for our required output. Once all of our required data is gathered, we have a series of echo statements that add to the report. In these echo statements we spell out the data in an easily readable list. I want you to look at each echo statement and then look at the report output in Listing 18-8.

```
Swap Space Report for dino
Sun Oct 21 17:27:20 EDT 2007
Total Amount of Swap Space: 96MB
Total MB of Swap Space Used: 24MB
Total MB of Swap Space Free: 70MB
Percent of Swap Space Used: 25%
Percent of Swap Space Free: 75%
```

Listing 18-8 HP-UX swap-space report

There are three things I want you to notice in the echo statements. First is the use of the \n when we want to add another new line to the output, which is a blank line in this case. In Bash shell, you must add the -e switch to the echo command to enable the use of the backslash operators. In Korn shell, this is not a requirement. Second is the use of the \t to add a TAB to align the data output. And finally, note the use of the curly braces, {VAR}, around the variable names. The curly braces are needed because we are adding characters to the output, and these characters are adjacent to the variable's *name* and there is not a space, which include MB and % suffixes. To separate the extra characters from the variable name we need to use curly braces to ensure the separation.

At the end of the script in Listing 18-8 we compare the percent used variable to the trigger threshold that is defined in the DEFINE VARIABLES section at the top of the shell script. If the threshold is exceeded, we turn on reverse video, print a warning message, and then turn reverse video back off. The over threshold warning message is shown in Listing 18-9.

```
Swap Space Report for dino

Sun Oct 21 17:40:35 EDT 2007

Total Amount of Swap Space: 96MB

Total MB of Swap Space Used: 24MB

Total MB of Swap Space Free: 70MB

Percent of Swap Space Used: 25%

Percent of Swap Space Free: 75%

WARNING: Swap Space has Exceeded the 20% Upper Limit!
```

Listing 18-9 HP-UX swap-space report with over limit warning

I edited the shell script and changed the PC_LIMIT variable assignment to 20 percent for this example. The reverse video does not show up on paper, but on the screen it stands out so that the user will always notice the warning message. I usually set this threshold to 65 percent. When you exceed this level of swap-space usage, you really need to find the cause of the increased swapping.

Linux Swap-Space Monitor

The Linux operating system does swapping, and the command to gather swap-space statistics is the free command. The free command output by default lists swap-space usage in KB, but the -m switch is available for listing the statistics in MB. Additionally, the free -m output does not include any statistics measured in percentages, so we must calculate the percentage of free swap space and the percentage of used swap space.

These percentage calculations are relatively easy, but we really want to measure the percentage using floating-point notation this time. We need to use the bc utility for the mathematical calculations. Chapter 13, "Floating-Point Math and the bc Utility," goes into great detail on floating-point math and the use of the bc utility. First, let's look at the following free -m output so that we know what we are dealing with:

# free -m						
	total	used	free	shared	buffers	cached
Mem:	52	51	0	0	0	18
-/+ buffer	rs/cache:	32	19			
Swap:	211	14	197			

The row of output that we are interested in is the last line of output, beginning with Swap:. This output shows that we have a total of 211 MB of swap space, where 14 MB is currently being used. This leaves 197 MB of free swap space. We have three out of five pieces of our required output, so we need to calculate only the percentage of free

swap space and the percentage of used space. For these calculations we need to look at the use of the bc utility.

The bc utility is a precision calculator language that is a UNIX level built-in program. For our purposes we have two techniques for using the bc utility. We can place our mathematical statement in an echo statement and pipe the output to bc. The second option is to use a *here* document with command substitution. For this exercise we are going to use the second option to look at the use of a here document.

To calculate the percentage of used swap space, we divide the total amount of swap space into MB of used swap space and multiply this total by 100, as shown in the following statement:

```
($SW_USED / $SW_TOTAL) * 100
```

This looks simple enough, but how do we get a floating-point output in a shell script? This is where the bc utility comes in. There is an option in bc called scale. The scale indicates how many decimal places to the right of the decimal point we want to use in the calculation. In our case we need to set scale=4. Now you are asking, Why four places? Because we are multiplying the result of the division by 100, we will have only two active decimal places with data, and the last two will have zeros in the end. Let's look at the following example to clear up any confusion:

```
PERCENT_USED=$(bc <<EOF
scale=4
($SW_USED / $SW_TOTAL) * 100
EOF
)</pre>
```

From the previous values, the result of this calculation is 7.1000 percent of used space because \$SW_USED is 15 MB and \$SW_TOTAL is 211 MB. Now let's look more closely at the use of the bc utility. We are using command substitution with an enclosed here document. A here document has the following form:

```
command <<LABEL
...
Input to the command
...
LABEL
```

This is a neat way of providing input to a command that usually requires user input, and this is why it is referred to as a *here document*, because the input is *here*, as opposed to being entered by the user at the command line.

In our case we use the here document inside command substitution, which is specified by the \$(commands) notation. The result is assigned to the PERCENT_USED variable. The calculation of the percent free is done in the same manner except that this time we divide the MB of free space into the MB of total swap space, as shown here:

```
PERCENT_FREE=$(bc <<EOF
scale=4</pre>
```

```
($SW_FREE / $SW_TOTAL) * 100
EOF
)
```

In our case, using the previously acquired data we get a result of 92.4100 percent. With these two calculations we have all of the data required for our standard output. We will cover the bc command, and we will use it again in the next section that deals with the Solaris swap-space monitor. Take a look at the entire shell script in Listing 18-10, and pay particular attention to the boldface type.

```
#!/usr/bin/ksh
# SCRIPT: linux_swap_mon.ksh
# AUTHOR: Randy Michael
# DATE: 5/31/2007
# REV: 1.1.P
# PLATFORM: Linux Only
#
#
# PURPOSE: This shell script is used to produce a report of
       the system's swap space statistics including:
#
#
     Total paging space in MB, MB of free paging space,
#
     MB of used paging space, % of paging space used, and
#
#
     % of paging space free
# REV LIST:
#
# set -x # Uncomment to debug this shell script
# set -n # Uncomment to check command syntax without any execution
THISHOST=$(hostname)  # Host name of this machine
PC_LIMIT=65
                # Upper limit of Swap space percentage
                 # before notification
******
echo "\nSwap Space Report for $THISHOST\n"
date
```

Listing 18-10 Linux_swap_mon.ksh shell script listing

```
free -m | grep -i swap | while read junk SW_TOTAL SW_USED SW_FREE
do
# Use the bc utility in a here document to calculate
# the percentage of free and used swap space.
PERCENT_USED=$ (bc <<EOF
scale=4
($SW_USED / $SW_TOTAL) * 100
EOF
)
PERCENT_FREE=$ (bc <<EOF
scale=4
($SW_FREE / $SW_TOTAL) * 100
EOF
)
    # Produce the rest of the paging space report:
    echo "\nTotal Amount of Swap Space:\t${SW_TOTAL}MB"
    echo "Total KB of Swap Space Used:\t${SW_USED}MB"
    echo "Total KB of Swap Space Free:\t${SW_FREE}MB"
    echo "\nPercent of Swap Space Used:\t${PERCENT_USED}%"
    echo "\nPercent of Swap Space Free:\t${PERCENT_FREE}%"
    # Grab the integer portion of the percent used to
    # test for the over limit threshold
    INT_PERCENT_USED=$(echo $PERCENT_USED | cut -d. -f1)
    if (( PC_LIMIT <= INT_PERCENT_USED ))
    then
        # Swap space limit has exceeded the threshold, send notification
         tput smso # Turn on reverse video!
         echo "\n\nWARNING: Paging Space has Exceeded the ${PC_LIMIT}%
Upper Limit!\n"
         tput rmso # Turn off reverse video!
    fi
done
echo "\n"
```

Notice the while read portion of the free -m command. We use the variable junk as a place to store the first field, which contains Swap:. This is the same technique that we used in the HP-UX section of this chapter. If we had additional data fields after the MB of free swap space, we could use a junk2 variable to hold this extra unneeded data, too.

Also notice that our bc calculations are done inside our while read loop. Even though I have indented everything else inside the loop for readability, you *cannot* use indention with these here documents! If you do indent anything, the calculation will fail, and this is extremely difficult to troubleshoot because it *looks* as if it should work.

Let's take a look at the shell script in Listing 18-10 in action in Listing 18-11.

```
# ./linux_swap_mon.ksh
Swap Space Report for bambam
Sun Jun 9 13:01:06 EDT 2007
Total Amount of Swap Space: 211MB
Total KB of Swap Space Used: 16MB
Total KB of Swap Space Free: 195MB
Percent of Swap Space Used: 7.5800%
Percent of Swap Space Free: 92.4100%
```

Listing 18-11 Linux_swap_mon.ksh in action

Notice that the last two numbers in the percentage of used and free swap space are zeros. I am leaving the task of removing these two numbers as an exercise for you to complete. Now let's move on to the OpenBSD swap-space monitor.

OpenBSD Swap-Space Monitor

As we discussed previously, OpenBSD uses swapping and utilizes the swapctl command to view swap-space usage. The swapctl command has several command switches, but the only ones we are concerned with are -l and -k. The swap information given by the swapctl -lk command is listed in KB by using the -k switch. Percentages presented in the output show the percentage of current usage, so we need a calculation here. Additionally, because we have our swap-space usage indicated in KB, we need to multiply by 1,000 for our standard MB output. The following shows the swapctl -lk command output:

swapctl -lk
Device 1K-blocks Used Avail Capacity Priority
swap_device 500000 0 500000 0% 0

As we can see in this OpenBDS output, the system has 500 MB of swap space defined, and currently 0 percent and 0 MB are utilized.

We start our shell script by extracting the fields of interest using an awk statement for the \$2, \$3, \$4, and \$5, as shown here:

Now that we have captured the data, let's do some calculations. We have the total, used, and free KB of paging-space statistics. We first need to multiply each of these values by 1,000 to get our standard MB values. The last step is to remove the % suffix and calculate the percent of free swap space. Pay attention to the boldface text in Listing 18-12.

```
#!/bin/Bash
# SCRIPT: OpenBSD_swap_mon.ksh
# AUTHOR: Randy Michael
# DATE: 10/31/2007
# REV: 1.2.p
# PLATFORM: OpenBSD Only
#
# PURPOSE: This shell script is used to produce a report of
       the system's paging space statistics including:
#
#
#
     Total paging space in MB, MB of Free paging space,
#
     MB of Used paging space, % of paging space Used, and
     % of paging space Free
# REV LIST:
#
# set -x # Uncomment to debug this shell script
# set -n # Uncomment to check command syntax without any execution
PC_LIMIT=65
                 # Percentage Upper limit of paging space
                 # before notification
THISHOST=$(hostname) # Host name of this machine
PAGING_STAT=/tmp/paging_stat.out # Paging Stat hold file
***********
echo "\nPaging Space Report for $THISHOST\n"
```

Listing 18-12 OpenBSD_swap_mon.ksh script

```
date
*****
# Load the data in a file without the column headings
swapct1 -1k | tai1 +2 | awk '{print $2, $3, $4, $5}' \
          | while read KB_TOT KB_USED KB_AVAIL PC_USED
do
    (( TOTAL = KB_TOT / 1000 ))
    (( MB USED = KB USED / 1000 ))
    (( MB_FREE = KB_AVAIL / 1000 ))
    PC_FREE_NO_PC=$(echo $PC_USED | awk -F '%' '{print $1}')
    ((PC_FREE = 100 - PC_FREE_NO_PC))
    # Produce the rest of the paging space report:
    echo "\nTotal MB of Paging Space:\t${TOTAL}MB"
    echo "Total MB of Paging Space Used:\t${MB_USED}MB"
    echo "Total MB of Paging Space Free:\t${MB_FREE}MB"
    echo "\nPercent of Paging Space Used:\t${PC_USED}"
    echo "\nPercent of Paging Space Free:\t${PC_FREE}%\n"
done
# Check for paging space exceeded the predefined limit
if (( PC_LIMIT <= PC_FREE_NO_PC ))
then
    echo "\nNARNING: Paging Space has Exceeded the ${PC_LIMIT}% \
Upper Limit!\n"
fi
```

Notice in Listing 18-12 that we used awk to extract the desired fields of data, as opposed to using the junk and junk2 variable assignments as we previously did. This is a cleaner method for extracting data.

We also used awk to extract the % suffix from the percent used, as opposed to using the cut command. We define % as the delimiter, or field separator, using the -F '%' switch notation with awk. Listing 18-13 shows the OpenBSD_swap_mon.ksh shell script in action.

```
# ksh ./OpenBSD_swap_mon.ksh
Paging Space Report for fred
```

Listing 18-13 OpenBSD_swap_mon.ksh script in action

Mon Oct 29 18:01:28 EST 2007 Total MB of Paging Space: 500MB Total MB of Paging Space Used: 0MB Total MB of Paging Space Free: 0% Percent of Paging Space Free: 100%

Listing 18-13 (continued)

Notice with OpenBSD that to execute a shell script on the command line, you must specify the executing shell on the command line. Let's move on to the Solaris swap-space monitor.

Solaris Swap-Space Monitor

The Solaris operating system does swapping, and the command to gather swap space statistics is swap -s. The output of the swap -s command is all on a single line, which is different from any of the previously studied operating systems. Additionally, all of the swap-space statistics are measured in KB as opposed to MB, which is our required measurement. Before we go any further, let's look at the Solaris swap -s output:

```
# swap -s
total: 56236k bytes allocated + 9972k reserved = 66208k used, 523884k
available
```

As you can see, the output is a little difficult to understand. We are interested in two fields for our purposes: the ninth field, 66208k, and the eleventh field, 523884k. The ninth field represents the total amount of used swap space, and the eleventh field represents the free swap space, where both are measured in KB. We are not interested in the amount of reserved and allocated swap space individually, but in the total, which is located in the ninth field.

When I say the ninth and eleventh fields I am specifying that each field in the output is separated by at least one blank space, also called white space. From this definition it is intuitively obvious that total:, +, =, and used are all individual fields in the command output. This is important to know because we are going to use two awk statements to extract the \$9 and \$11 fields and assign them to different variables.

As in the Linux section, we do not have any percentages given in the output, so we must calculate the percentage of free swap space and the percentage of used swap space. If you looked at the Linux section, then you already know how to use the bc utility. If you jumped to the Solaris section, we will cover this again here.

The bc utility is a precision calculator language that is a UNIX level built-in program. For our purposes, we have two techniques for using the bc utility. We can place our

mathematical statement in an echo statement and pipe the output to bc. The second option is to use a *here* document with command substitution. For this exercise we are going to use the second option and look at the use of a here document.

To calculate the percentage of used swap space, we divide the total amount of swap space into MB of used swap space and multiply this total by 100, as shown in the following statement:

```
($SW_USED / $SW_TOTAL) * 100
```

This looks simple enough, but how do we get a floating-point output in a shell script? This is where the bc utility comes in. There is an option in bc called **scale**. The scale indicates how many decimal places to the right of the decimal point we want to use in the calculation. In our case we need to set scale=4. Now you are asking, Why four places? Because we are multiplying the result of the division by 100, we will have only two active decimal places with data after this multiplication, and the last two will have zeros. Let's look at this next example to clear up any confusion:

```
PERCENT_USED=$(bc <<EOF
scale=4
($SW_USED / $SW_TOTAL) * 100
EOF
)</pre>
```

From the previous values the result of this calculation is 11.2200 percent of used space because \$SW_USED is 66 MB and \$SW_TOTAL is 590 MB. Now let's look more closely at the use of the bc utility. We are using command substitution with an enclosed here document. A here document has the following form:

```
command <<LABEL
...
Input to the command
...
LABEL
```

This is a neat way of providing input to a command that usually requires user input, and this is why it is referred to as a *here* document — the input is *here*, as opposed to being entered by the user at the command line.

In our case, we use the here document inside command substitution, which is specified by the \$(commands) notation. The result is assigned to the PERCENT_USED variable. The calculation of the percent free is done in the same manner except that this time we divide the MB of free space into the MB of total swap space, as shown in the code that follows:

```
PERCENT_FREE=$(bc <<EOF
scale=4
($SW_FREE / $SW_TOTAL) * 100</pre>
```

EOF

In our case, the percentage of used swap space is 11.220 percent, and the percentage of free swap space is 88.7800 percent. Of course, to get the total swap space we added the \$9 and \$11 fields together. The entire shell script is shown in Listing 18-14.

```
#!/usr/bin/ksh
# SCRIPT: SUN_swap_mon.ksh
# AUTHOR: Randy Michael
# DATE: 5/31/2007
# REV: 1.1.P
#
# PLATFORM: Solaris Only
#
# PURPOSE: This shell script is used to produce a report of
#
       the system's swap space statistics including:
#
     Total paging space in MB, MB of free paging space,
#
    MB of used paging space, % of paging space used, and
#
     % of paging space free
#
#
# REV LIST:
#
# set -x # Uncomment to debug this shell script
# set -n # Uncomment to check command syntax without any execution
# Upper limit of Swap space percentage
PC LIMIT=65
               # before notification
THISHOST=$(hostname) # Host name of this machine
echo "\nSwap Space Report for $THISHOST\n"
date
# Use two awk statements to extract the $9 and $11 fields
# from the swap -s command output
```

Listing 18-14 SUN_swap_mon.ksh shell script

```
SW_USED=$(swap -s | awk '{print $9}' | cut -dk -f1)
SW_FREE=$(swap -s | awk '{print $11}' | cut -dk -f1)
# Add SW_USED to SW_FREE to get the total swap space
((SW_TOTAL = SW_USED + SW_FREE))
# Calculate the percent used and percent free using the
# bc utility in a here document with command substitution
PERCENT_USED=$(bc <<EOF
scale=4
($SW USED / $SW TOTAL) * 100
EOF
)
PERCENT_FREE=$ (bc <<EOF
scale=4
($SW_FREE / $SW_TOTAL) * 100
EOF
)
# Convert the KB measurements to MB measurements
((SW_TOTAL_MB = SW_TOTAL / 1000))
((SW USED MB = SW USED / 1000))
((SW_FREE_MB = SW_FREE / 1000))
# Produce the remaining part of the report
echo "\nTotal Amount of Swap Space:\t${SW_TOTAL_MB}MB"
echo "Total KB of Swap Space Used:\t${SW_USED_MB}MB"
echo "Total KB of Swap Space Free:\t${SW_FREE_MB}MB"
echo "\nPercent of Swap Space Used:\t${PERCENT_USED}%"
echo "\nPercent of Swap Space Free:\t${PERCENT_FREE}%"
# Grab the integer portion of the percent used
INT_PERCENT_USED=$(echo $PERCENT_USED | cut -d. -f1)
# Check to see if the percentage used maximum threshold
# has been exceeded
if (( PC_LIMIT <= INT_PERCENT_USED ))
then
    # Percent used has exceeded the threshold, send notification
    tput smso # Turn on reverse video!
```

Listing 18-14 (continued)

```
echo "\n\nWARNING: Swap Space has Exceeded the ${PC_LIMIT}% \
Upper Limit!\n"
    tput rmso # Turn off reverse video!
fi
echo "\n"
```

Notice how we used two awk statements using two separate reads of the swap -s command output. These two measurements occur in such a short amount of time that it should not matter; however, you may want to change the method to a single read and store the output in a variable or file. I'm leaving this modification task for you to do as an exercise. In the next step we add the KB of free swap space to the KB of used swap space to find the total swap space on the system.

With these three KB measurements we calculate the percentage of used and free swap space using the bc utility inside a command substitution statement, while using a here document to provide input to the bc command. There is still one more step before we are ready to print the report — convert the KB measurements to MB. We only need to divide our KB measurements by 1,000, and we are ready to go. Next the remaining portions of the report are printed, and then the test is made to see if the percent used has exceeded the threshold limit, specified by the PC_LIMIT variable. If the percentage used limit is exceeded, then reverse video is turned on, the warning message is displayed, and reverse video is turned back off. The SUN_swap_mon.ksh shell script is shown in action in Listing 18-15.

```
# ./SUN_swap_mon.ksh
Swap Space Report for wilma
Mon Jun 10 03:50:29 EDT 2007
Total Amount of Swap Space: 590MB
Total KB of Swap Space Used: 66MB
Total KB of Swap Space Free: 524MB
Percent of Swap Space Free: 11.2200%
Percent of Swap Space Free: 88.7800%
```

Listing 18-15 SUN_swap_mon.ksh script in action

Notice that the percentages are given as floating-pointing numbers, but there are two extra zeros. These two extra zeros are the result of specifying in the bc here document that the scale=4 and then multiplying the result by 100. As an exercise, add a command to remove the two extra zeros. Are we finished? Not yet; we still need

a single shell script that will run on all five operating systems. Let's move on to the all-in-one section.

All-in-One Paging- and Swap-Space Monitor

Let's put everything together by making the five previous scripts into functions and use the **uname** command in a case statement to determine the UNIX flavor, and thus which function to run.

Let's look at this combined shell script, and we will go over the details at the end. The combined shell script is called all-in-one_swapmon.ksh and is shown in Listing 18-16.

```
#!/usr/bin/ksh
#
# SCRIPT: all-in-one_swapmon.ksh
# AUTHOR: Randy Michael
# DATE: 5/31/2007
# REV: 3.0.P
# PLATFORM: AIX, Solaris, HP-UX and Linux Only
#
# PURPOSE: This shell script is used to produce a report of
        the system's paging or swap space statistics including:
#
#
      Total paging space in MB, MB of Free paging space,
      MB of Used paging space, % of paging space Used, and
      % of paging space Free
#
# REV LIST:
# 11/5/2007 -- Randy Michael
# REVISION: Added a function for OpenBSD swap monitoring and
# changed the case statement controlling correct usage
# depending on the OS.
##############
# set -x # Uncomment to debug this shell script
# set -n # Uncomment to check command syntax without any execution
PC_LIMIT=65
                    # Upper limit of Swap space percentage
                    # before notification
```

Listing 18-16 all-in-one_swapmon.ksh shell script

```
THISHOST=$(hostname) # Host name of this machine
*******
echo "\nSwap Space Report for $THISHOST\n"
date
function SUN_swap_mon
# Use two awk statements to extract the $9 and $11 fields
# from the swap -s command output
SW_USED=$(swap -s | awk '{print $9}' | cut -dk -f1)
SW_FREE=$(swap -s | awk '{print $11}' | cut -dk -f1)
# Add SW_USED to SW_FREE to get the total swap space
((SW_TOTAL = SW_USED + SW_FREE))
# Calculate the percent used and percent free using the
# bc utility in a here documentation with command substitution
PERCENT USED=$ (bc <<EOF
scale=4
($SW_USED / $SW_TOTAL) * 100
EOF
)
PERCENT_FREE=$ (bc <<EOF
scale=4
($SW_FREE / $SW_TOTAL) * 100
EOF
)
# Convert the KB measurements to MB measurements
((SW_TOTAL_MB = SW_TOTAL / 1000))
((SW_USED_MB = SW_USED / 1000))
((SW_FREE_MB = SW_FREE / 1000))
# Produce the remaining part of the report
```

```
echo "\nTotal Amount of Swap Space:\t${SW_TOTAL_MB}MB"
echo "Total KB of Swap Space Used:\t${SW_USED_MB}MB"
echo "Total KB of Swap Space Free:\t${SW_FREE_MB}MB"
echo "\nPercent of Swap Space Used:\t${PERCENT_USED}%"
echo "\nPercent of Swap Space Free:\t${PERCENT_FREE}%"
# Grab the integer portion of the percent used
INT_PERCENT_USED=$(echo $PERCENT_USED | cut -d. -f1)
# Check to see if the percentage used maximum threshold
# has been exceeded
if (( PC_LIMIT <= INT_PERCENT_USED ))
then
   # Percent used has exceeded the threshold, send notification
   tput smso # Turn on reverse video!
   echo "\n\nWARNING: Swap Space has Exceeded the ${PC_LIMIT}% \
Upper Limit!\n"
   tput rmso # Turn off reverse video!
fi
echo "\n"
}
******
function Linux_swap_mon
free -m | grep -i swap | while read junk SW_TOTAL SW_USED SW_FREE
do
# Use the bc utility in a here document to calculate
# the percentage of free and used swap space.
PERCENT_USED=$ (bc <<EOF
scale=4
($SW_USED / $SW_TOTAL) * 100
EOF
)
PERCENT_FREE=$ (bc <<EOF
scale=4
($SW_FREE / $SW_TOTAL) * 100
EOF
```

```
)
     # Produce the rest of the paging space report:
    echo "\nTotal Amount of Swap Space:\t${SW_TOTAL}MB"
    echo "Total KB of Swap Space Used:\t${SW_USED}MB"
    echo "Total KB of Swap Space Free:\t${SW_FREE}MB"
    echo "\nPercent of Swap Space Used:\t${PERCENT_USED}%"
    echo "\nPercent of Swap Space Free:\t${PERCENT_FREE}%"
    # Grap the integer portion of the percent used to
     # test for the over limit threshold
    INT_PERCENT_USED=$(echo $PERCENT_USED | cut -d. -f1)
    if (( PC_LIMIT <= INT_PERCENT_USED ))
    then
         tput smso
         echo "\n\nWARNING: Paging Space has Exceeded the \
${PC_LIMIT}% Upper Limit!\n"
         tput rmso
    fi
done
echo "\n"
}
*****************
function HP_UX_swap_mon
{
# Start a while read loop by using the piped in input from
# the swapinfo -tm command output.
swapinfo -tm | grep dev | while read junk SW_TOTAL SW_USED \
                             SW FREE PERCENT USED junk2
do
   # Calculate the percentage of free swap space
   ((PERCENT_FREE = 100 - $(echo $PERCENT_USED | cut -d% -f1) ))
   echo "\nTotal Amount of Swap Space:\t${SW_TOTAL}MB"
   echo "Total MB of Swap Space Used:\t${SW_USED}MB"
   echo "Total MB of Swap Space Free: \t$ {SW_FREE }MB"
   echo "\nPercent of Swap Space Used:\t${PERCENT_USED}"
```

```
echo "\nPercent of Swap Space Free:\t${PERCENT_FREE}%"
   # Check for paging space exceeded the predefined limit
   if (( PC_LIMIT <= $(echo $PERCENT_USED | cut -d% -f1) ))
   then
      # Swap space is over the predefined limit, send notification
     tput smso # Turn on reverse video!
       echo "\n\nWARNING: Swap Space has Exceeded the\
${PC_LIMIT}% Upper Limit!\n"
       tput rmso # Turn reverse video off!
   fi
done
echo "\n"
}
function AIX_paging_mon
PAGING_STAT=/tmp/paging_stat.out # Paging Stat hold file
******
# Load the data in a file without the column headings
lsps -s | tail +2 > $PAGING_STAT
# Start a while loop and feed the loop from the bottom using
# the $PAGING_STAT file as redirected input
while read TOTAL PERCENT
do
    # Clean up the data by removing the suffixes
    PAGING_MB=$(echo $TOTAL | cut -d 'MB' -f1)
    PAGING_PC=$(echo $PERCENT | cut -d% -f1)
    # Calculate the missing data: %Free, MB used and MB free
    (( PAGING_PC_FREE = 100 - PAGING_PC ))
    (( MB_USED = PAGING_MB * PAGING_PC / 100 ))
    (( MB_FREE = PAGING_MB - MB_USED ))
```

```
# Produce the rest of the paging space report:
    echo "\nTotal MB of Paging Space:\t$TOTAL"
    echo "Total MB of Paging Space Used:\t${MB_USED}MB"
    echo "Total MB of Paging Space Free:\t${MB_FREE}MB"
    echo "\nPercent of Paging Space Used:\t${PERCENT}"
    echo "\nPercent of Paging Space Free:\t${PAGING_PC_FREE}%"
    # Check for paging space exceeded the predefined limit
    if ((PC_LIMIT <= PAGING_PC))
    then
         # Paging space is over the limit, send notification
         tput smso # Turn on reverse video!
        echo "\n\nWARNING: Paging Space has Exceeded the ${PC_LIMIT}% \
Upper Limit!\n"
         tput rmso # Turn off reverse video
    fi
done < $PAGING_STAT
rm -f $PAGING STAT
# Add an extra new line to the output
echo "\n"
}
*****
function OpenBSD_swap_mon
*************
# Load the data in a file without the column headings
swapct1 -1k | tail +2 | awk '{print $2, $3, $4, $5}' \
          while read KB_TOT KB_USED KB_AVAIL PC_USED
do
    (( TOTAL = KB_TOT / 1000 ))
    (( MB_USED = KB_USED / 1000 ))
    (( MB_FREE = KB_AVAIL / 1000 ))
    PC_FREE_NO_PC=$(echo $PC_USED | awk -F '%' '{print $1}')
    ( PC_FREE = 100 - PC_FREE_NO_PC ) )
    # Produce the rest of the paging space report:
```

```
echo "\nTotal MB of Paging Space:\t${TOTAL}MB"
    echo "Total MB of Paging Space Used:\t${MB_USED}MB"
    echo "Total MB of Paging Space Free:\t${MB_FREE}MB"
    echo "\nPercent of Paging Space Used:\t${PC_USED}"
    echo "\nPercent of Paging Space Free:\t${PC_FREE}%\n"
done
# Check for paging space exceeded the predefined limit
if (( PC_LIMIT <= PC_FREE_NO_PC ))
then
    echo "\n\nWARNING: Paging Space has Exceeded the ${PC_LIMIT}% \
Upper Limit!\n"
fi
}
# Find the Operating System and execute the correct function
case $(uname) in
    AIX) AIX_paging_mon
    ;;
    HP-UX) HP_UX_swap_mon
    ;;
    Linux) Linux_swap_mon
    ;;
    SunOS) SUN_swap_mon
    ;;
    OpenBSD) OpenBSD_swap_mon
    ;;
    *) echo "\nERROR: Unsupported Operating System...EXITING\n"
      exit 1
esac
# End of all-in-one_swapmon.ksh
```

As you can see, there is not much to converting a shell script into a function. The only thing required is that you extract out of each shell script the core code that makes up the shell script. The common code should remain in the main body of the new shell script. In our example, the common parts are the PC_LIMIT, which defines the over limit percentage threshold, and the hostname of the machine. Everything else is unique to each of the five shell scripts and functions here.

To turn a shell script into a function, all you need to do is a cut and paste in your favorite editor and copy the main body of the shell script into a new shell script. This new function can be enclosed into a function in two ways, as follows:

```
function was_a_shell_script
{
  shell_script_code_here
  }
or
  was_a_shell_script ()
```

```
{
shell_script_code_here
}
```

I tend to use the function definition instead of the POSIX C language type definition. This is a personal choice, but both types of function definitions produce the same result. If you are a C programmer you will most likely prefer the POSIX C type notation. I like to use the function definition so that a person coming behind me trying to edit the shell script will know intuitively that this is a function because it is spelled out in the definition.

Once we have each of the four functions defined inside a single shell script, we need to know only on which operating system we are running and to execute the appropriate function. To determine the UNIX flavor, we use the uname command. The following case statement runs the correct swap/paging function as defined by the UNIX flavor:

```
case $(uname) in
AIX) AIX_paging_mon
;;
HP-UX) HP_UX_swap_mon
;;
Linux) Linux_swap_mon
;;
SunOS) SUN_swap_mon
;;
OpenBSD) OpenBSD_swap_mon
;;
*) echo "\nERROR: Unsupported Operating System...EXITING...\n"
    exit 1
;;
esac
```

Notice that if the UNIX flavor is not AIX, HP-UX, Linux, OpenBSD, or SunOS, the shell script gives an error message and exits with a return code of 1.

Other Options to Consider

As usual, we can always improve on a shell script, and this chapter is no exception. Each of these shell scripts could stand a little improvement one way or another because there is not just one way to do anything! I have noted a few suggestions here.

Event Notification

The only event notification I have included in these shell scripts and functions is a warning message presented to the user in reverse video. I usually add email notification to my alphanumeric pager also. Just use your preferred method of remote notification, and you will have the upper hand on keeping your systems running smoothly.

Log File

A log file is a great idea for this type of monitoring. I suggest that any time the threshold is crossed that a log file receives an appended message. This is simple to do by using the tee -a command in a pipe. See the man page on tee for more information.

Scheduled Monitoring

If you are going to do paging/swap monitoring, it is an extremely good idea to do this monitoring on a scheduled basis. Different shops have different requirements. I like to monitor every 15 minutes from 6:00 a.m. until 10:00 p.m. This way I have covered everyone from the East Coast to the West Coast. If you have locations in other time zones around the world, you may want to extend your coverage to include these times as well. The monitoring is up to you, but it is best to take a proactive approach and find the problem before someone tells *you* about it. You may also discover some trends of heavy system loads to help in troubleshooting.

Summary

In this chapter we started out with a predetermined output that we had to adhere to for any UNIX flavor, and we held to it. Each operating system presented us with a new challenge because in each instance we lacked part of the required data and we had to do a little math to get into the correct format. Each of these shell scripts is a unique piece of work, but in the end we combined everything into a single multi-OS functioning shell script that determines what the UNIX flavor is and executes the proper function to get the desired result. If the UNIX flavor is not AIX, HP-UX, Linux, OpenBSD, or Solaris, the shell script gives an error message and exits cleanly.

In this chapter we covered various techniques to extract and calculate data to produce an identical output no matter the UNIX flavor. I hope you gained some valuable experience with dealing with the challenge of handling different types of data to produce a standard report. This type of experience is extremely important for heterogeneous environments. In the next chapter, we will look at some techniques to monitor the load on a system.

Lab Assignments

- 1. Schedule the all-in-one_swapmon.ksh shell script to execute, as the root user, every 10 minutes between the hours of 6:00 a.m. and 1:00 a.m., Monday through Friday, and 9:00 a.m. and 9:00 p.m. on Saturday and Sunday.
- 2. Add logging capability to the all-in-one_swapmon.ksh shell script.
- 3. Add email alerts when paging space is running low. This is a three-step alert system. Use a yellow alert for ranges between 65 percent and 75 percent. Use a red alert for ranges between 76 percent and 90 percent. Use a code blue alert for the most severe alerts, starting at 91 percent. The email alerts should specify the alert level and all swap/paging details. Code blue alerts must be repeated every 5 minutes until the problem is corrected, thus disabling the cron execution. (Tip: check for another instance of this same script executing. Remember that function calls show up in the process table also!)

CHAPTER 19

Monitoring System Load

Have you ever seen a system start slowing down as the wait state and uptime stats rise, and finally the system crashes? I have, and it is not a pretty sight when all of the heads start popping up over the cubes. In this chapter, we are going to look at some techniques to monitor the CPU load on a UNIX system. When the system is unhappy running under a heavy load, there are many possible causes. The system may have a runaway process that is producing a ton of zombie processes every second, or you have a tape drive failure and your database redo logs fill up a filesystem and cause the database and SAP to stop. In any case, we want to be proactive in catching a symptom in the early stages of loading down the system.

There are really only three basic things to look at when monitoring the CPU load on the system. First, look at the load statistics produced as part of the **uptime** command. This output indicates the average number of jobs in the run queue over the last 5, 10, and 15 minutes in AIX, and 1, 5, and 15 minutes for HP-UX, Linux, OpenBSD, and Solaris. The second measurement to look at is the percentages of CPU usage for system/kernel, user/application, I/O wait state, and idle time. These four measurements can be obtained from the **iostat**, **vmstat**, and **sar** outputs. We will look at each of these commands individually. The final step in monitoring the CPU load is to find the CPU hogs. Of course, to get a good feel for how the system is running we need to look at more system statistics, such as memory and swap-space utilizations, as well as disk and network I/O. Each of these can cause high I/O waits. Most systems have a top-like monitoring tool that shows the CPU process users in descending order of CPU usage. The output in Listing 19-1 is a sample output for the top command on a Linux machine.

```
top - 07:40:25 up 1 day, 21:32, 2 users, load average: 0.22, 0.08, 0.02
Tasks: 110 total, 2 running, 108 sleeping, 0 stopped, 0 zombie
Cpu(s): 49.1%us, 1.0%sy, 0.0%ni, 49.9%id, 0.0%wa, 0.0%hi, 0.0%si,
0.0%st
```

Listing 19-1 Example top output on a Linux machine

Mem:	1027744k	tot	al,	7322	264k u	sed,		29548	80k f	free, 173	800k buffers
Swap:	2031608k	tot	al,		0k u	used,	2	203160)8k f	free, 273	460k cached
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	4 TIME+	COMMAND
9536	root	20	0	72480	1188	996	R	100	0.1	L 0:17.04	
rando	om_file.bas	5									
9535	root	20	0	14580	1064	808	R	0	0.1	L 0:00.04	top
1	root	20	0	10328	692	576	S	0	0.1	L 0:01.37	init
2	root	15	-5	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	RT	-5	0	0	0	S	0	0.0	0:00.00	migration/0
4	root	15	-5	0	0	0	S	0	0.0	0:00.00	ksoftirqd/0
5	root	RT	-5	0	0	0	S	0	0.0	0:00.03	watchdog/0
6	root	RT	-5	0	0	0	S	0	0.0	0:00.00	migration/1
7	root	15	-5	0	0	0	S	0	0.0	0:00.00	ksoftirqd/1
8	root	RT	-5	0	0	0	S	0	0.0	0:00.04	watchdog/1
9	root	15	-5	0	0	0	S	0	0.0	0:00.23	events/0
61	root	15	-5	0	0	0	S	0	0.0	0:00.05	kblockd/1
64	root	15	-5	0	0	0	S	0	0.0	0:00.00	kacpid
65	root	15	-5	0	0	0	5	0	0.0	0:00.00	kacpi_notify

We can also use the **ps auxw** command that displays CPU % usage for each process in descending order from the top. This chapter is limited to looking at the CPU load and the run queue statistics. We will look at each of these in this chapter. First, let's look at installing the **sysstat** package on Linux and the command syntax for the commands we're using.

Installing the System-Statistics Programs in Linux

If you are running a Linux system and cannot find the iostat, sar, and other commands to gather system statistics, you need to install the sysstat package. The easiest way to do the installation is to use yum (apt works, too). Listing 19-2 shows the sysstat package network installation using yum.

```
[root@yogi ~]# yum install sysstat
Loading "installonlyn" plugin
Setting up Install Process
Setting up repositories
core [1/4]
updates [2/4]
adobe-linux-i386 [3/4]
extras [4/4]
Reading repository metadata in from local files
Parsing package install arguments
```

Listing 19-2 Example of installing the sysstat package using yum

```
Resolving Dependencies
--> Populating transaction set with selected packages. Please wait.
---> Downloading header for sysstat to pack into transaction set.
00:00
---> Package sysstat.i386 0:6.0.1-3.2.1 set to be updated
--> Running transaction check
Dependencies Resolved
_____
=========
         Arch Version Repository Size
Package
_____
_____
Installing:
              i386 6.0.1-3.2.1 core
sysstat
                                           154 k
Transaction Summary
_____
Install 1 Package(s)
Update 0 Package(s)
Remove 0 Package(s)
Total download size: 154 k
Is this ok [v/N]: v
Downloading Packages:
00:02
Running Transaction Test
Finished Transaction Test
Transaction Test Succeeded
Running Transaction
                          ############################### [1/1]
Installing: sysstat
Installed: sysstat.i386 0:6.0.1-3.2.1
Complete!
```

Listing 19-2 (continued)

The nice thing about yum and apt is that any dependencies the installation has are resolved prior to installing the target package. For more information on yum and apt, see the manual pages (**man yum** and **man apt**, respectively). You can also download and install the RPM packages.

NOTE You should pick either yum or apt and stick with that particular method. Do not use both, because some packages do not update cleanly if you do.

Syntax

As usual with UNIX, there is not just one way to monitor a system for load. We can use any of the following commands to get system load statistics: uptime, iostat, sar, and vmstat. To illustrate the ability of each of these commands, we are going to take a look at each one of the commands individually.

NOTE In the next sections we are selecting specific fields of data from command output. It is up to *you* to individually verify that the command outputs I present are indeed the output fields that your particular UNIX flavor *and OS version* produce for each of the five operating systems presented here. Just be aware that different versions of the same operating system sometimes produce different output for the same command.

Syntax for uptime

Using the uptime command is the most direct method of getting a measurement of the system load. Part of the output of the uptime command is a direct measure of the average length of the run queue over the last 5 minutes, 10 minutes, and the last measurement is averaged over 15 minutes on AIX. For HP-UX, Linux, OpenBSD, and Solaris the uptime command is a direct measure of the average length of the run queue over the last 1 minute, 5 minutes, and the last measurement is averaged over 15 minutes. The length of the run queue is a direct measurement of how busy the CPU is by the number of runable processes waiting for CPU time, as an average, over a period of time.

We do need to put a bit of logic into the use of the uptime command in a shell script because the output field's position varies depending on how long it has been since the last system reboot, and, possibly, which UNIX flavor we are running. We are going to test each of these options to show why it is important to consider the output data you are using in a shell script.

We have five possible variations to look at in this uptime command output, as you will see. The first is 1–59 minutes, the second is 1–23 hours, and the third measurement is when the system has been up for more than 24 hours. After the system has been up for at least one day, we have to consider hours and minutes again! Believe it or not, the load fields continue to float during each day. When the system reaches an *exact* hour, to the minute, of the reboot day, an hrs field is added; this is true for the *anniversary* first hour, too. In this case, the min field is added along with the day field. Follow along through the next few examples to see how the fields vary during these five stages.

We are going to look at the last three fields, the load average. This load average is the number of runable processes in the run queue averaged over the last 1 minute, 5 minutes, and 15 minutes, since this is a Linux system, but all UNIX flavors have the same type behavior.

Linux

This uptime output is shown when the Linux system has been up 20 minutes. The fields we want are in the \$9, \$10, and \$11 positions:

uptime
12:17pm up 20 min, 4 users, load average: 2.29, 2.17, 1.51

This uptime output is shown when the Linux system has been up for 1 hour and 7 minutes. The fields we want are in the \$8, \$9, and \$10 positions:

```
# uptime
1:04pm up 1:07, 4 users, load average: 1.74, 2.10, 2.09
```

This uptime output is shown when the Linux system has been up for 12 days, 19 hours, and 3 minutes. The fields we want are in the \$10, \$11, and \$12 positions:

```
# uptime
4:40pm up 12 days, 19:03, 4 users, load average: 1.52, 0.47, 0.16
```

This uptime output is shown when the Linux system has been up for 14 days and exactly 17 minutes. The fields we want are in the \$11, \$12, and \$13 positions:

```
# uptime
9:16pm up 14 days, 17 mins, 9 users, load average: 1.31, 1.82, 1.61
```

This uptime output is shown when the Linux system has been up for 14 days and exactly 5 hours. The fields we want are in the \$11, \$12, and \$13 positions:

```
# uptime
9:16pm up 14 days, 5 hr, 2 users, load average: 1.01, 1.69, 1.84
```

What's the Common Denominator?

The most obvious common denominator is that the load statistics are the last three fields of the output. The load average on a Linux machine shows the average number of runable processes over the preceding 1-, 5-, and 15-minute intervals. Because the field positions vary, we need to look at the total number of fields in the output of the uptime command and subtract 2 for the first field, subtract 1 for the second field, and the third field is the last field in the output. By knowing the exact fields where the data resides we can directly access the last three fields of output. In the "Scripting the Solutions" section we will look closely at the details.

Syntax for iostat

To get the CPU load statistics from the iostat command, we have to be a little flexible between UNIX flavors. For AIX and HP-UX machines, we need to use the -t command

switch. For Linux and Solaris, we use the -c switch. And for OpenBSD, we use the -C switch. Due to the UNIX flavor dependency, we need to first check the operating system using the uname command. Then, based on the operating system, we can assign the proper switch to the iostat command.

Let's look at the output of the iostat command for each of our UNIX flavors, AIX, HP-UX, Linux, OpenBSD, and Solaris.

AIX

# iost	at -t 1	0 2					
tty:	tin	tout	avg-cpu:	% user	% sys	% idle	% iowait
	0.2	33.6		2.4	8.2	84.0	5.4
	0.1	1188.4		16.8	83.2	0.0	0.0

In this AIX output, notice the last four fields, *%user*, *%sys*, *%idle*, and *%iowait*. These four fields are the ones that we want to extract. The field positions are *\$*3, *\$*4, *\$*5, and *\$*6, and we want just the last line of the output because the first line of data is an average since the last system reboot. Also, notice that the rows of actual data consist entirely of numeric characters. This will become important as we look at each operating system.

HP-UX

#	iostat -t	10 2						
				ty tout 2	us	cp ni 0		
	device	bps	sps	msps				
	c0t6d0	0	0.0	1.0				
		tt tin 0	y tout 0	us	cpi ni 41	sy	id 59	0
	device	bps	sps	msps				
	c0t6d0	1	0.1	1.0				

Notice that the HP-UX output differs greatly from the AIX iostat output. The only thing that distinguishes the CPU data from the rest of the data is the fact that the entire row of data is numeric. This is an important characteristic of the HP-UX data, and it will help us extract the data that we are looking for. Notice again that the first set of statistics is an average since the last system reboot.

Linux

Notice that the Linux iostat command switch for CPU statistics is -c, instead of the -t that we used for AIX and HP-UX. In this output we have the average of the CPU load since the last reboot, and then the current data is shown in the second command output. Also notice that the actual data presented is entirely numeric. It looks as if we have a trend.

OpenBSD

The OpenBSD iostat -C 10 2 output shows the load average statistics since the last system reboot on the first line of data and the most current data on the last line. Notice again that the actual data is a row of numeric characters. Knowing that the data is always on a row that is numeric characters allows us to greatly simplify writing this shell script.

Solaris

```
iostat -c 10 2
cpu
us sy wt id
3 14 0 83
17 81 0 2
```

The Solaris iostat -c output shows the load average statistics since the last system reboot on the first line of data and the most current data on the last line. Notice again that the actual data is a row of numeric characters. Knowing that the data is always on a row that is numeric characters allows us to greatly simplify writing this shell script.

What Is the Common Denominator?

The real common denominator for the iostat command data between each UNIX flavor is that we have a row of numeric data only. The only thing remaining is the fields for each OS, which vary by field and content. We want just the last line of data, which is the most current data. From this set of criteria, let's write a little code to see how we can specify the correct switch and set the proper fields to extract. We can do both of these tasks with a single case statement, as shown in Listing 19-3.

```
OS=$(uname)
case $0S in
AIX HP-UX)
              SWITCH='-t'
              F1=3
              F2 = 4
              F3=5
              F4=6
              echo "\nThe Operating System is $OS\n"
              ;;
Linux)
              SWITCH='-c'
              F1=1
              F2 = 3
              F3 = 4
              F4 = 5
              echo "\nThe Operating System is $OS\n"
              ;;
              SWITCH='-c'
SunOS)
              F1 = 1
              F_{2=2}
              F3=3
              F4 = 4
              echo "\nThe Operating System is $OS\n"
              ;;
OpenBSD)
              SWITCH="-C"
              F1=1
              F2 = 2
              F3=3
              F4=5
              ;;
*)
              echo "\nERROR: $0S is not a supported operating system\n"
              echo "\n\t...EXITING...\n"
              exit 1
              ;;
esac
```

Listing 19-3 case statement for the iostat fields of data

Notice in Listing 19-3 that we use a single case statement to set up the environment for the shell script to run the correct iostat command for each of the four

UNIX flavors. If the UNIX flavor is not in the list, the user receives an error message before the script exits with a return code of 1, one. Later we will cover the entire shell script.

Syntax for sar

The sar command stands for *system activity report*. Using the sar command we can take direct sample intervals for a specific time period. For example, we can take 4 samples that are 10 seconds each, and the sar command automatically averages the results for us.

Let's look at the output of the sar command for each of our UNIX flavors, AIX, HP-UX, Linux, and Solaris. OpenBSD does not support the sar command.

AIX

```
# sar 10 4
AIX yogi 3 5 00CF8DEF4C00 12/20/07
System configuration: lcpu=32
12:51:26 %usr %sys %wio %idle physc
12:51:36 6 2
                    0
                            92 16.00
12:51:46 8
12:51:56 8
               2
                      0
                            90 16.01
               2
                      1
                            89 15.97
          9
               5
                      2
12:52:06
                            84 16.00
           7
                 3
                       1
                            89
                               16.00
Average
```

Now let's look at the average of the samples directly:

sar 10 4 | grep Average

Average 7 3 1 89 16.00

HP-UX

sar 10 4

HP-UX dino B.10.20 A 9000/715 11/26/07

22:48:10	%usr	%sys	%wio	%idle	
22:48:20	40	60	0	0	
22:48:30	40	60	0	0	
22:48:40	12	19	0	68	
22:48:50	0	0	0	100	
Average	23	35	0	42	

Now let's look at the average of the samples directly:

sar 10 4 | grep Average Average 25 37 0 38

Linux

[root@booboo scripts]# sar 10 4
Linux 2.6.20-1.2316.fc5 (booboo) 12/20/2007

12:51:43 PM	CPU	%user	%nice	%system	%iowait	%idle
12:51:53 PM	all	95.10	0.00	4.90	0.00	0.00
12:52:03 PM	all	76.32	0.00	5.29	0.00	18.38
12:52:13 PM	all	22.68	0.00	2.70	0.00	74.63
12:52:23 PM	all	93.00	0.00	4.50	0.00	2.50
Average:	all	71.76	0.00	4.35	0.00	23.89

Now let's look at the average of the samples directly:

```
# sar 10 4 | grep Average
Average: all 71.76 0.00 4.35 0.00 23.89
```

Solaris

sar 10 4

SunOS wilma	5.8 Ger	eric i86	Брс	11/26/07
23:01:55	%usr	%sys	%wio	%idle
23:02:05 23:02:15	1 12	1 53	0 0	98 35
23:02:25	15	67	0	18
23:02:35	21	59	0	21
Average	12	45	0	43

Now let's look at the average of the samples directly:

sar 10 4 | grep Average Average 12 45 0 43

What Is the Common Denominator?

With the sar command the only common denominator is that we can always grep on the word "Average." Like the iostat command, the fields vary between some UNIX

flavors. We can use a similar case statement to extract the correct fields for each UNIX flavor, as shown in Listing 19-4.

```
OS=$(uname)
case $0S in
AIX | HP-UX | SunOS)
       F1=2
       F2 = 3
       F3=4
       F4 = 5
       echo "\nThe Operating System is $OS\n"
       ;;
Linux)
       F1=3
       F2=5
       F_{3=6}
       F4=7
       echo "\nThe Operating System is $OS\n"
       ;;
*)
       echo "\nERROR: $0S is not a supported operating system\n"
       echo "\n\t...EXITING...\n"
       exit 1
       ;;
esac
```

Listing 19-4 case statement for the sar fields of data

Notice in Listing 19-4 that a single case statement sets up the environment for the shell script to select the correct fields from the sar command for each of the five UNIX flavors. If the UNIX flavor is not in the list, the user receives an error message before the script exits with a return code of 1, one. Later we will cover the entire shell script.

Syntax for vmstat

The vmstat command stands for *virtual memory statistics*. Using the vmstat command, we can get a lot of data about the system, including memory, paging space, page faults, and CPU statistics. We are concentrating on the CPU statistics in this chapter, so let's stay on track. The vmstat commands also allow us to take direct samples over intervals for a specific time period. The vmstat command does not do any averaging for us; however, we are going to stick with two intervals. The first interval is the average of the system load since the last system reboot, like the iostat command. The last line contains the most current sample.

Let's look at the output of the vmstat command for each of our UNIX flavors, AIX, HP-UX, Linux, OpenBSD, and Solaris.

AIX

[ro	ot	:yogi]@,	/scrip	ts#	vmst	at 3	0 2									
kthr memory				page					t		сī	pu				
r	b	avm	fre	re	pi	ро	fr	sr	су	in	sy	CS	us	sy	id	wa
0	0	23936	580	0	0	0	0	2	0	103	2715	713	8	25	67	0
1	0	23938	578	0	0	0	0	0	0	115	9942	270	24	76	0	0

The last line of output is what we are looking for. This is the average of the CPU load over the length of the interval. We want just the last four columns in the output. The fields that we want to extract for AIX are in positions \$14, \$15, \$16, and \$17.

HP-UX

#	vmst	at	30 2														
		procs memory				page					faults cpu						
r	b	W	avm	free	re	at	pi	ро	fr	de	sr	in	sy	CS	us	sy	id
0	39	0	8382	290	122	26	2	0	0	0	3	128	2014	146	14	21	65
1	40	0	7532	148	345	71	0	0	0	0	0	108	5550	379	29	43	27

The HP-UX vmstat output is a long string of data. Notice for the CPU data that HP-UX supplies only three values: user part, system part, and the CPU idle time. The fields that we want to extract are in positions \$16, \$17, and \$18.

Linux

```
[root@yogi scripts]# vmstat 30 2
procs ------memory------swap-- ----io---- --system-- ----
r b swpd free buff cache si so bi bo in cs us sy id
wa st
0 0 197248 69768 4480 177328 0 0 2 14 17 7 5 1 93 0 0
3 0 197248 68892 4540 177344 0 0 0 41 1123 2376 74 5 21
0 0
```

Like AIX, the Linux vmstat output for CPU activity has four fields: user part, system part, the CPU idle time, and I/O wait. The fields that we want to extract are in positions \$13, \$14, \$15, and \$16.

OpenBSD

# vmst	at 30 2										
procs	memory	page				disk	S	traps		сŗ	bu
r b w	avm fre	flt re	pi po	fr	sr wd	0 cd0	int	sys	cs us	sy i	d
2 0 0	7556 446888	5 0	0 0	0	0	0 0	235	60	14 9	48	37
2 0 0	7560 446888	3 0	0 0	0	0 0	0 0	229	20157	16 73	27	0

Like HP-UX, the OpenBSD vmstat output for CPU activity has three fields: user part, system part, and the CPU idle time. The fields that we want to extract are in positions \$17, \$18, and \$19.

Solaris

# vmsta	at 30 2																		
procs	me	mory]	pag	е				d	isk			faul	ts		C	pu
r b w	swap	free	re	mf]	pi	ро	fr	de	sr	cd	f0	sO		in	sy	CS	us	sy	id
0 0 0	558316	33036	57	433	2	0	0	0	0	0	0	0	0	111	500	77	2	8	90
0 0 0	556192	29992	387	2928	8 0	0	0	0	0	1	0	0	0	155	2711	273	14	60	26

As with HP-UX and Linux, the Solaris vmstat output for CPU activity consists of the last three fields: user part, system part, and the CPU idle time. The fields that we want to extract are in positions \$20, \$21, and \$22.

What Is the Common Denominator?

There are at least two common denominators for the vmstat command output between the UNIX flavors. The first is that the CPU data is in the last fields. On AIX the data is in the last four fields with the added I/O wait state. HP-UX, Linux, OpenBSD, and Solaris do not list the wait state. The second common factor is that the data is always on a row that is entirely numeric. Again, we need a case statement to parse the correct fields for the command output. Take a look at Listing 19-5 for the exact fields we want to capture.

```
OS=$(uname)
case $0S in
AIX)
       F1=14
       F2=15
       F3=16
       F4=17
       echo "\nThe Operating System is $OS\n"
       ;;
HP-UX)
       F1=16
       F2=17
       F3=18
       F4=1 # This "F4=1" is bogus and not used for HP-UX
       echo "\nThe Operating System is $OS\n"
       : :
Linux)
```

Listing 19-5 case statement for the vmstat fields of data

```
F1=13
       F2=14
       F3=15
       F4=16
       echo "\nThe Operating System is $OS\n"
       ;;
OpenBSD)
       F1=17
       F2=18
       F3=19
       F4=1 # This "F4=1" is bogus and not used for Linux
       echo "\nThe Operating System is $OS\n"
       ;;
SunOS)
       F1=20
       F2=21
       F3=22
       F4=1 # This "F4=1" is bogus and not used for SunOS
       echo "\nThe Operating System is $OS\n"
       ::
*)
       echo "\nERROR: $0S is not a supported operating system\n"
       echo "\n\t...EXITING...\n"
       exit 1
       ;;
esac
```

Listing 19-5 (continued)

Notice in Listing 19-5 that the F4 variable gets a valid assignment only on the AIX match. For HP-UX, Linux, OpenBSD, and Solaris, the F4 variable is assigned the value of the \$1 field, specified by the F4=1 variable assignment. This bogus assignment is made so that we do not need a special vmstat command statement for each operating system. You will see how this works in detail in the "Scripting the Solutions" section.

Scripting the Solutions

Each of the techniques presented is slightly different in execution and output. Some options need to be timed over an interval for a user-defined amount of time, measured in seconds. We can get an immediate load measurement using the uptime command, but the sar, iostat, and vmstat commands require the user to specify a period of time to measure over and the number of intervals to sample the load. If you enter the sar, iostat, or vmstat commands without any arguments, the statistics presented are an average since the last system reboot. Because we want current statistics, the scripts must supply a period of time to sample. We are always going to initialize the INTERVAL variable to equal 2. The first line of output is measured since the last system reboot, and the second line is the current data that we are looking for.

Let's look at each of these commands in separate shell scripts in the following sections.

Using uptime to Measure the System Load

Using uptime is one of the best indicators of the system load. The last columns of the output represent the average of the *run queue* over the last 5, 10, and 15 minutes for an AIX machine, and over the last 1, 5, and 10 minutes for HP-UX, Linux, OpenBSD, and Solaris. A run queue is where jobs wanting CPU time line up for their turn for some processing time in the CPU. The priority of the process, or on some systems a *thread*, has a direct influence on how long a job has to wait in line before getting more CPU time. The lower the priority, the more CPU time. The higher the priority, the less CPU time.

The uptime command always has an average of the length of the run queue. The threshold trigger value that you set will depend on the normal load of your system. My little desktop AIX box starts getting very slow when the run queue hits 2, but the p570 at work typically runs with a run queue value over 10 because it is a multiprocessor machine running a multi-terabyte database, as shown here:

uptime
04:08PM up 96 days, 8:46, 2 users, load average: 14.13, 11.08, 6.37

With these differences in acceptable run queue levels, you will need to tailor the threshold level for notification on a machine-by-machine basis.

Scripting with the uptime Command

Scripting the uptime solution is a short shell script, and the response is immediate. As you remember in the "Syntax" section, we had to follow the floating load statistics as the time since the last reboot moved from minutes, to hours, and even days after the machine was rebooted. But knowing that the data always resides in the last three fields helps us create a simple method of directly accessing the data. The good thing is that the floating fields are consistent across the UNIX flavors studied in this book. Check out the uptime_loadmon.Bash script in Listing 19-6 and we will cover the details at the end.

```
#!/bin/Bash
#
# SCRIPT: uptime_loadmon.Bash
# AUTHOR: Randy Michael
# DATE: 12/16/2007
# REV: 1.0.P
# PLATFORM: AIX, HP-UX, Linux, OpenBSD, and Solaris
#
# PURPOSE: This shell script uses the "uptime" command to
```

Listing 19-6 uptime_loadmon.Bash shell script

```
extract the most current load average data, which
#
#
        in this case is the average number of jobs in the
#
        run queue.
# set -x # Uncomment to debug this shell script
# set -n # Uncomment to check script syntax without any execution
*****
*****
MAXLOAD=2.00
          # Trigger value to warn
# Extract the integer and decimal parts of $MAXLOAD value
MAXLOAD_INT=$(echo $MAXLOAD | awk -F '.' '{print $1}')
MAXLOAD_DEC=$(echo $MAXLOAD | awk -F '.' '{print $2}')
# Check the UNIX flavor for the correct uptime values
# AIX specifies load as the last 5, 10, and 15 minutes.
# The other UNIX flavors specify the load in the last
# 1, 5, and 15 minutes.
case $(uname) in
AIX)
      L1=5
       L2=10
      L3=15
       ;;
 *)
       L1=1
       L2=5
       L3=15
       ;;
esac
*****
# DEFINE FUNCTIONS HERE
*****
function get_max
{
# This function returns the number of arguments
# presented to the function
(($# == 0)) && return -1
echo $#
}
```

Listing 19-6 (continued)

```
**********
# BEGINNING OF MAIN
**********
echo -e "\nGathering System Load Average using the \"uptime\" command\n"
# This next command statement extracts the latest
# load statistics no matter what the UNIX flavor is.
NUM_ARGS=$(get_max $(uptime)) # Get the total number of fields in
uptime output
((NUM_ARGS == -1)) && echo "ERROR: get_max Function Error...EXITING..."\
                  && exit 2
# Find the exact fields that represent the run queue load stats
ARGM2=$(((NUM_ARGS - 2))) # Subtract 2 from the total
ARGM1=$(((NUM_ARGS - 1))) # Subtract 1 from the total
ARGM=$NUM_ARGS
                         # Last value in string
# Extract the run queue data in the last 3 fields of uptime output
uptime | sed s/,//g | awk '{print $'$ARGM2', $'$ARGM1', $'$ARGM'}' \
      | while read LAST5 LAST10 LAST15
do
    # Separate the values into integer and decimal parts
    echo $LAST5 | awk -F '.' '{print $1, $2}' \
    | while read INT DEC
     do
         # Test if the load is above the trigger threshold
        if (( INT > MAXLOAD_INT ))
      then
          echo -e "\nWARNING: System load has \
reached ${LAST5}\n"
      elif (( INT == MAXLOAD_INT ))
      then
          # Since the integer values are at threshold we need
          # to check the decimal values
            if (( DEC >= MAXLOAD_DEC ))
            then
              echo -e "\nWARNING: System load has \
reached ${LAST5}\n"
          fi
      fi
```

Listing 19-6 (continued)

```
echo "System load average for the last $L1 minutes is $LAST5"
echo "System load average for the last $L2 minutes is $LAST10"
echo "System load average for the last $L3 minutes is $LAST15"
echo -e "\nThe load threshold is set to ${MAXLOAD}\n"
done
done
```

Listing 19-6 (continued)

The uptime_loadmon.Bash shell script in Listing 19-6 shows how we can extract output fields that vary in number. We know that the run queue statistics always reside in the last three fields of the uptime command output. So, to know how many fields are present in the output, we call the function get_max, which takes as its \$1 argument the uptime command's output. This function returns -1 if the function is called but does not have any arguments, or a positive integer value of the number of fields present in the uptime command output. On my Fedora Linux machine that has been up for more than 15 days, the uptime command produces output with 12 fields of data. So, we are interested the data in fields \$10, \$11, and \$12 for our run queue measurements as shown here:

```
[root@booboo ~]# uptime
15:17:50 up 15 days, 6:36, 10 users, load average: 0.94, 0.63, 0.49
```

The fields vary in number depending on how long the system has been up since the last reboot. The "uptime" changes from minutes, hours, and days the longer the system is up and running. With this type of varying output, we cannot just hard code specific fields; we have to do a little math first. The code to find our fields of interest is shown here in Listing 19-7.

```
# Find the exact fields that represent the run queue load stats
ARGM2=$(((NUM_ARGS - 2))) # Subtract 2 from the total
ARGM1=$(((NUM_ARGS - 1))) # Subtract 1 from the total
ARGM=$NUM_ARGS # Last value in string
```

Listing 19-7 Code to extract the last three fields of command output

Knowing the total number of fields allows us to capture the first load value directly by subtracting 2 from the value returned by the get_max function. To get the second field, we subtract 1, and the last field is the final value we want to capture. We directly extract the correct fields using the following regular expression in the highlighted awk statement in bold text:

If our uptime output produces 12 fields of output, we are interested in fields \$10, \$11, and \$12. We use the three field values we calculated in Listing 19-7 here in a regular expression to pick out exactly the fields we are interested in. So, the regular expression \$'\$ARGM1' specifies the \$10 field. Following along with this, the regular expression \$'\$ARGM2' points to the \$11 field, and so on.

The three data fields are assigned to the LAST5, LAST10, and LAST15 variables. You can see the uptime_loadmon.ksh shell script in action in Listings 19-8 and 19-9.

```
# ./uptime_loadmon.ksh
Gathering System Load Average using the "uptime" command
System load value is currently at 1.86
The load threshold is set to 2.00
```

Listing 19-8 Script in action under "normal" load

Listing 19-8 shows the uptime_loadmon.ksh shell script in action on a machine that is under a normal load. Listing 19-9 shows the same machine under an excessive load — at least, it is excessive for this little machine.

```
# ./uptime_loadmon.ksh
Gathering System Load Average using the "uptime" command
WARNING: System load has reached 2.97
System load value is currently at 2.97
The load threshold is set to 2.00
```

Listing 19-9 Script in action under "excessive" load

This is about all there is to using the uptime command. Let's move on to the sar command.

Using sar to Measure the System Load

Some UNIX flavors have the sar data collection set up by default. This sar data is presented when the sar command is executed without any switches. The data that is displayed is automatically collected at scheduled intervals throughout the day and compiled into a report at day's end. By default, the system keeps a month's worth of data available for online viewing. This is great for seeing the basic trends of the

machine as it is loaded through the day. If we want to collect data at a specific time of day for a specific period of time, we need to add the number of seconds for each interval and the total number of intervals to the sar command. The final line in the output is an average of all of the previous sample intervals.

This is where our shell script comes into play. By using a shell script with the times and intervals defined, we can take samples of the system load over small or large increments of time without interfering with the system's collection of sar data. This can be a valuable tool for things like taking hundreds of small incremental samples as a development application is being tested. Of course, this technique can also help in troubleshooting just about any application. Let's look at how we script the solution.

Scripting with the sar Command

For each of our UNIX flavors, the sar command produces four CPU load statistics. The outputs vary somewhat, but the basic idea remains the same. In each case, we define an INTERVAL variable specifying the total number of samples to take and a SECS variable to define the total number of seconds for each sample interval. Notice that we used the variable SECS as opposed to SECONDS. We do not want to use the variable SECONDS, because it is a shell built-in variable used for timing in a shell and it increments automatically. As I stated in the introduction, this book uses variable names in uppercase so the reader will quickly know that the code is referencing a variable; however, in the real world you may want to use the lowercase version of the variable name. It really would not matter here because we are defining the variable value and then using it within the same second, hopefully.

The next step in this shell script is to define which positional fields we need to extract to get the sar data for each of the UNIX operating systems. For this step we use a case statement using the uname command output to define the fields of data. It turns out that AIX, HP-UX, and SunOS operating systems all have the sar data located in the \$2, \$3, \$4, and \$5 positions. Linux differs in this respect with the sar data residing in the \$3, \$5, \$6, and \$7 positions. In each case, these field numbers are assigned to the F1, F2, F3, and F4 variables inside the case statement. Let's look at the sar_loadmon.Bash shell script in Listing 19-10 and cover the remaining details at the end. OpenBSD does not support sar.

```
#!/bin/Bash
#
# SCRIPT: sar_loadmon.ksh
# AUTHOR: Randy Michael
# DATE: 12/20/2007
# REV: 1.0.P
# PLATFORM: AIX, HP-UX, Linux, and Solaris
#
# # PURPOSE: This shell script takes multiple samples of the CPU
```

Listing 19-10 sar_loadmon.Bash shell script

```
usage using the command. The average of
#
#
         sample periods is shown to the user based on the
#
         UNIX operating system that this shell script is
#
         executing on. Different UNIX flavors have differing
#
        outputs and the fields vary too.
#
# REV LIST:
#
# set -n # Uncomment to check the script syntax without any execution
# set -x # Uncomment to debug this shell script
*****
*****
SECS=30 # Defines the number of seconds for each sample
INTERVAL=10 # Defines the total number of sampling intervals
OS=$(uname) # Defines the UNIX flavor
##### SET UP THE ENVIRONMENT FOR EACH OS HERE ######
******
# These point to the correct field in the
# command output for each UNIX flavor.
case $0S in
AIX | HP-UX | SunOS)
    F1=2
    F2=3
    F3=4
    F4=5
     echo -e "\nThe Operating System is $OS\n"
     ;;
Linux)
    F1=3
    F_{2}=5
    F3=6
    F4=7
    echo -e "\nThe Operating System is $OS\n"
    ;;
*)
     echo -e "\nERROR: $0S is not a supported operating system\n"
     echo -e "\n\t...EXITING...\n"
     exit 1
     ;;
esac
```

Listing 19-10 (continued)

```
*********
*****
echo -e "Gathering CPU Statistics using sar...\n"
echo "There are $INTERVAL sampling periods with"
echo "each interval lasting $SECS seconds"
echo -e "\n...Please wait while gathering statistics...\n"
# This "sar" command takes $INTERVAL samples, each lasting
# $SECS seconds. The average of this output is captured.
sar $SECS $INTERVAL | grep Average \
        awk '{print $'$F1', $'$F2', $'$F3', $'$F4'}' \
        while read FIRST SECOND THIRD FOURTH
do
    # Based on the UNIX Flavor, tell the user the
    # result of the statistics gathered.
    case $OS in
    AIX | HP-UX | SunOS )
         echo -e "\nUser part is ${FIRST}%"
         echo "System part is ${SECOND}%"
         echo "I/O Wait is ${THIRD}%"
         echo -e "Idle time is ${FOURTH}%\n"
         ;;
    Linux)
         echo -e "\nUser part is ${FIRST}%"
         echo "Nice part is ${SECOND}%"
         echo "System part is ${THIRD}%"
         echo -e "Idle time is ${FOURTH}%\n"
         ;;
    esac
done
```

Listing 19-10 (continued)

In the shell script in Listing 19-10 we start by defining the data time intervals. In these definitions we are taking 10 interval samples of 30 seconds each, for a total of 300 seconds, or 5 minutes. Then we grab the UNIX flavor using the uname command and assigning the operating system value to the OS variable. Following these definitions we define the data fields that contain the sar data for each operating system.

Now we get to the interesting part where we actually take the data sample. Look at the following sar command statement, and we will decipher how it works:

We really need to look at the statement one pipe at a time. In the very first part of the statement we take the sample(s) over the defined number of intervals. Consider the following statement and output:

```
SECS=30
INTERVAL=10
# sar $SECS $INTERVAL
AIX yogi 3 5 00CF7EFE1D00 12/19/07
System configuration: lcpu=32
14:38:38 %usr %sys
                      %wio %idle
                                  physc
14:38:41 6 2 0
                              92 16.01
          5
14:38:44
                 3
                       0
                              92 15.99
          6 2
8 3
6 3
14:38:47
                       0
                              92 15.99
14:38:50
                       0
                              89 16.00
14:38:53
                       0
                              91 16.01
2
14:39:02
14:39:05
10
14:20
10
10
          6
                2
                       0
                              91 15.99
                       0
                              92
                                16.01
                       0
                              84 15.99
                       0
                              85 15.97
          5
14:39:08
                4
                        0
                              92 16.01
            7
                  3
                        Ο
                              90
                                16.00
Average
```

The preceding output is produced by the first part of the sar command statement. Then, all of this output is piped to the next part of the statement, as shown here:

```
sar $SECS $INTERVAL | grep Average
Average 7 3 0 90 16.00
```

Now we have the row of data that we want to work with, which we grepped out using the word Average as a pattern match. The next step is to extract the positional fields that contain the data for user, system, I/O wait, and idle time for AIX. Remember in the previous script section that we defined the field numbers and assigned them to the F1, F2, F3, and F4 variables, which in our case results in F1=2, F2=3, F3=4, and F4=5. Using the following extension to our previous command we get the following statement:

Notice that we continued the command statement on the next line by placing a backslash (\) at the end of the first line of the statement. In the awk part of the statement you can see a confusing list of dollar signs and "F" variables. The purpose of this set of

characters is to *directly* access what the "F" variables are pointing to. Let's run through this in detail by example.

The F1 variable has the value 2 assigned to it. This value is the positional location of the first data field that we want to extract. So we want to access the value at the \$2 position. Make sense? When we extract the \$2 data we get the value 7, as defined in the previous step. Instead of going in this roundabout method, we want to *directly* access the *field* that the F1 variable points to. Just remember that a variable is only a pointer to a value, nothing more! We want to point directly to what another variable is pointing to. The solution is to use the following regular expression:

\$'\$F1'

or

\$\\$F1

In any case, the innermost pointer (\$) must be *escaped*, which removes the special meaning. For this shell script we use the \$'\$F1' notation. The result of this notation, in this example, is 7, which is the value that we want. This is not smoke and mirrors when you understand how it works.

The final part of the sar command statement is to pipe the four data fields to a while loop so that we can do something with the data, which is where we end the sar statement and enter the while loop.

The only thing that we do in the while loop is to display the results based on the UNIX flavor. The sar_loadmon.ksh shell script is in action in Listing 19-11.

```
# ./sar_loadmon.ksh
The Operating System is AIX
Gathering CPU Statistics using sar...
There are 10 sampling periods with
each interval lasting 30 seconds
...Please wait while gathering statistics...
User part is 7%
System part is 3%
I/O wait state is 0%
Idle time is 90%
```

Listing 19-11 sar_loadmon.ksh shell script in action

From the output presented in Listing 19-11 you can see that the shell script queries the system for its operating system, which is AIX here. Then the user is notified of the sampling periods and the length of each sample period. The output is displayed to the user by field. That is it for using the sar command. Now let's move on to the iostat command.

Using iostat to Measure the System Load

The iostat command is mostly used to collect disk storage statistics, but by using the -t, -c, or, -C command switch, depending on the operating system, we can see the CPU statistics as we saw them in the syntax section for the iostat command. We are going to create a shell script using the iostat command and use almost the same technique as we did in the last section.

Scripting with the iostat Command

In this shell script we are going to use a very similar technique to the sar shell script in the previous section. The difference is that we are going to take only two intervals with a long sampling period. As an example, the INTERVAL variable is set to 2, and the SECS variable is set to 300 seconds, which is 5 minutes. Also, because we have two possible switch values, -t and -c, we need to add a new variable called SWITCH. Let's look at the iostat_loadmon.Bash shell script in Listing 19-12, and we will cover the differences at the end in more detail.

```
#!/bin/Bash
#
# SCRIPT: iostat_loadmon.ksh
# AUTHOR: Randy Michael
# DATE: 12/20/2007
# REV: 1.0.P
# PLATFORM: AIX, HP-UX, Linux, OpenBSD, and Solaris
# PURPOSE: This shell script take two samples of the CPU
#
         usage using the command. The first set of
         data is an average since the last system reboot. The
#
         second set of data is an average over the sampling
#
         period, or $INTERVAL. The result of the data acquired
#
         during the sampling period is shown to the user based
         on the UNIX operating system that this shell script is
         executing on. Different UNIX flavors have differing
         outputs and the fields vary too.
#
# REV LIST:
#
```

Listing 19-12 iostat_loadmon.Bash shell script

```
# set -n # Uncomment to check the script syntax without any execution
# set -x # Uncomment to debug this shell script
*****
*****
SECS=300 # Defines the number of seconds for each sample
          # Defines the total number of sampling intervals
INTERVAL=2
STATCOUNT=0 # Initializes a loop counter to 0, zero
OS=$(uname) # Defines the UNIX flavor
*****
##### SET UP THE ENVIRONMENT FOR EACH OS HERE ######
*****
# These "F-numbers" point to the correct field in the
# command output for each UNIX flavor.
case $0S in
AIX | HP-UX) SWITCH= '-t'
         F1=3
         F2=4
         F3=5
         F4=6
         echo -e "\nThe Operating System is $OS\n"
         ;;
Linux)
         SWITCH='-c'
         F1=1
         F2=3
         F3 = 4
         F4=6
         echo -e "\nThe Operating System is $OS\n"
         ;;
         SWITCH='-c'
SunOS)
         F1=1
         F2=2
         F3=3
         F4 = 4
         echo -e "\nThe Operating System is $OS\n"
         ;;
OpenBSD)
         SWITCH='-C'
         F1=1
         F2 = 2
         F3=3
         F4=5
         echo -e "\nThe Operating System is $OS\n"
```

```
Listing 19-12 (continued)
```

```
;;
*)
           echo -e "\nERROR: $OS is not a supported operating system\n"
           echo -e "\n\t...EXITING...\n"
           exit 1
           ;;
esac
*****
*****
echo -e "Gathering CPU Statistics using vmstat...\n"
echo "There are $INTERVAL sampling periods with"
echo "each interval lasting $SECS seconds"
echo -e "\n...Please wait while gathering statistics...\n"
# Use "iostat" to monitor the CPU utilization and
# remove all lines that contain alphabetic characters
# and blank spaces. Then use the previously defined
# field numbers, for example, F1=4, to point directly
# to the 4th position, for this example. The syntax
# for this techniques is ==> $'$F1'.
iostat $SWITCH $SECS $INTERVAL | egrep -v '[a-zA-Z]|^$' \
        | awk '{print $'$F1', $'$F2', $'$F3', $'$F4'}' \
        while read FIRST SECOND THIRD FOURTH
do
 if ((STATCOUNT == 1)) # Loop counter to get the second set
 then
                     # of data produced by "iostat"
    case $OS in # Show the results based on the UNIX flavor
    AIX)
          echo -e "\nUser part is ${FIRST}%"
          echo "System part is ${SECOND}%"
          echo "Idle part is ${THIRD}%"
          echo -e "I/O wait state is ${FOURTH}%\n"
          ;;
    HP-UX OpenBSD)
          echo -e "\nUser part is ${FIRST}%"
          echo "Nice part is ${SECOND}%"
          echo "System part is ${THIRD}%"
          echo -e "Idle time is ${FOURTH}%\n"
          ;;
    SunOS |Linux)
          echo -e "\nUser part is ${FIRST}%"
          echo "System part is ${SECOND}%"
          echo "I/O Wait is ${THIRD}%"
          echo -e "Idle time is ${FOURTH}%\n"
```

Listing 19-12 (continued)

```
;;
esac
fi
((STATCOUNT = STATCOUNT + 1)) # Increment the loop counter
done
```

Listing 19-12 (continued)

The similarities are striking between the sar implementation and the iostat script shown in Listing 19-12. At the top of the shell script we define an extra variable, STATCOUNT. This variable is used as a loop counter, and it is initialized to 0, zero. We need this counter because we have only two intervals, and the first line of the output is the load average since the last system reboot. The second, and final, set of data is the CPU load statistics collected during our sampling period, so it is the most current data. Using a counter variable, STATCOUNT, we collect the data and assign it to variables on the second loop iteration, or when the STATCOUNT is equal to 1, one.

In the next section we use the UNIX flavor given by the uname command in a case statement to assign the correct switch to use in the iostat command. This is also where the F1, F2, F3, and F4 variables are defined with the positional placement of the data we want to extract from the command output.

Now comes the fun part. Let's look at the iostat command statement we use to extract the CPU statistics here:

The beginning of the iostat command statement uses the correct command switch, as defined by the operating system, and the sampling time and the number of intervals, which is two this time. From this first part of the iostat statement we get the following output on a Linux system:

```
SWITCH='-c'
SECS=300
INTERVAL=2
iostat $SWITCH $SECS $INTERVAL
Linux 2.6.23.8-34.fc7 (booboo) 12/19/2007
avg-cpu: %user %nice %system %iowait %steal %idle
0.01 0.03 0.01 0.03 0.00 99.92
avg-cpu: %user %nice %system %iowait %steal %idle
23.30 0.00 26.60 0.40 0.00 49.70
```

Remember that the first row of data is an average of the CPU load since the last system reboot, so we are interested in the last row of output. If you remember from the syntax section for the iostat command, the common denominator for this output is that the data rows are entirely numeric characters. Using this as the criteria to extract data, we add to our iostat command statement as shown here:

iostat \$SWITCH \$SECS \$INTERVAL | egrep -v '[a-zA-Z] ^\$'

The egrep addition to the preceding command statement does two things for us. First, it excludes all lines of the output that have alphabetic characters, leaving only the rows with numbers. The second thing we get is the removal of all blank lines from the output. Let's look at each of these.

To omit the alpha characters we use the egrep command with the -v option, which says to display everything in the output *except* the rows that the pattern matched. To specify all alpha characters we use the following regular expression:

[a-zA-Z]

Then to remove all blank lines we use the following expression:

^\$

The caret character means *begins with*, and to specify blank lines we use the dollar sign (\$). If you want to remove all of the lines in a file that are commented out with a hash mark (#), use ^#.

When we join these two expressions in a single extended grep (egrep), we get the following extended regular expression:

egrep -v '[a-zA-Z]|^\$'

At this point we are left with the following output:

0.01 0.03 0.01 0.03 0.00 99.92 23.30 0.00 26.60 0.40 0.00 49.70

This brings us to the next addition to the iostat command statement in the shell script. This is where we add the awk part of the statement using the F1, F2, F3, and F4 variables, as shown here:

This is the same code that we covered in the preceding section, where we point *directly* to what another pointer is pointing to. For Linux, F1=1, F2=2, F3=3, and F4=4. With this information we know that \$'\$F1' on the first line of output is

equal to 0.01, and on the second row this same expression is equal to 23.30. Now that we have the values we have a final pipe to a while loop. Remember that in the while loop we have added a loop counter, STATCOUNT. On the first loop iteration, the while loop does nothing. On the second loop iteration, the values 23.30, 0.26.60, 0.40, and 49.70 are assigned to the variables FIRST, SECOND, THIRD, and FOURTH, respectively.

Using another case statement with the \$0S value the output is presented to the user based on the operating system fields, as shown in Listing 19-13.

```
The Operating System is Linux
Gathering CPU Statistics using vmstat...
There are 2 sampling periods with
each interval lasting 300 seconds
...Please wait while gathering statistics...
User part is 39.35%
Nice part is 0.00%
System part is 31.59%
Idle time is 29.06%
```

Listing 19-13 iostat_loadmon.ksh shell script in action

Notice that the output is in the same format as the sar script output. This is all there is to the iostat shell script. Let's now move on to the vmstat solution.

Using vmstat to Measure the System Load

The vmstat shell script uses the exact same technique as the iostat shell script in the previous section. Only AIX produces four fields of output; the remaining UNIX flavors have only three data points to measure for the CPU load statistics. The rest of the vmstat output is for virtual memory statistics, which is the main purpose of this command anyway. Let's look at the vmstat script.

Scripting with the vmstat Command

When you look at this shell script for vmstat you will think that you just saw this shell script in the previous section. Most of these two shell scripts are the same, with only minor exceptions. Let's look at the vmstat_loadmon.Bash shell script in Listing 19-14 and cover the differences in detail at the end.

```
#!/bin/Bash
# SCRIPT: vmstat_loadmon.ksh
# AUTHOR: Randy Michael
# DATE: 12/20/2007
# REV: 1.0.P
# PLATFORM: AIX, HP-UX, Linux, OpenBSD, and Solaris
#
# PURPOSE: This shell script takes two samples of the CPU
        usage using the command. The first set of
#
        data is an average since the last system reboot. The
#
#
        second set of data is an average over the sampling
        period, or $INTERVAL. The result of the data acquired
#
         during the sampling period is shown to the user based
#
        on the UNIX operating system that this shell script is
#
         executing on. Different UNIX flavors have differing
#
        outputs and the fields vary too.
#
# REV LIST:
# set -n # Uncomment to check the script syntax without any execution
# set -x # Uncomment to debug this shell script
****
SECS=300 # Defines the number of seconds for each sample
INTERVAL=2 # Defines the total number of sampling intervals
STATCOUNT=0 # Initializes a loop counter to 0, zero
OS=$(uname) # Defines the UNIX flavor
*****
##### SET UP THE ENVIRONMENT FOR EACH OS HERE ######
*****
# These "F-numbers" point to the correct field in the
# command output for each UNIX flavor.
OS=$(uname)
case $0S in
AIX)
```

Listing 19-14 vmstat_loadmon.Bash shell script

```
F1=14
     F2=15
     F3=16
     F4 = 17
     echo -e "\nThe Operating System is $OS\n"
     ;;
HP-UX)
     F1=16
     F2=17
     F3=18
     F4=1 # This "F4=1" is bogus and not used for HP-UX
     echo -e "\nThe Operating System is $OS\n"
     ;;
Linux)
     F1=13
     F2=14
     F3=15
     F4=16
     echo -e "\nThe Operating System is $OS\n"
     ;;
OpenBSD)
     F1=17
     F2=18
     F3=19
     F4=1 # This "F4=1" is bogus and not used for Linux
     echo -e "\nThe Operating System is $OS\n"
     ;;
SunOS)
     F1=20
     F2=21
     F3=22
     F4=1 # This "F4=1" is bogus and not used for SunOS
     echo -e "\nThe Operating System is $OS\n"
     ;;
*)
     echo -e "\nERROR: $OS is not a supported operating system\n"
     echo -e "\n\t...EXITING...\n"
     exit 1
     ;;
esac
*****
******
echo -e "Gathering CPU Statistics using vmstat...\n"
echo "There are $INTERVAL sampling periods with"
echo "each interval lasting $SECS seconds"
echo -e "\n...Please wait while gathering statistics...\n"
```

Listing 19-14 (continued)

```
# Use "vmstat" to monitor the CPU utilization and
# remove all lines that contain alphabetic characters
# and blank spaces. Then use the previously defined
# field numbers, for example F1=20, to point directly
# to the 20th position, for this example. The syntax
# for this technique is ==> $'$F1' and points directly
# to the $20 positional parameter.
vmstat $SECS $INTERVAL | egrep -v '[a-zA-Z]|^$' \
         | awk '{print $'$F1', $'$F2', $'$F3', $'$F4'}' \
         | while read FIRST SECOND THIRD FOURTH
do
 if ((STATCOUNT == 1)) # Loop counter to get the second set
                       # of data produced by
 then
     case $0S in # Show the results based on the UNIX flavor
     AIX Linux)
           echo -e "\nUser part is ${FIRST}%"
           echo "System part is ${SECOND}%"
           echo "Idle part is ${THIRD}%"
           echo -e "I/O wait state is ${FOURTH}%\n"
           ;;
     HP-UX OpenBSD SunOS)
           echo -e "\nUser part is ${FIRST}%"
           echo "System part is ${SECOND}%"
           echo -e "Idle time is ${THIRD}%\n"
           ;;
     esac
 fi
 ((STATCOUNT = STATCOUNT + 1)) # Increment the loop counter
done
```

Listing 19-14 (continued)

We use the same variables in Listing 19-14 as we did in Listing 19-12 with the iostat script. The differences come when we define the "F" variables to indicate the fields to extract from the output and the presentation of the data to the user. As I stated before, only AIX produces a fourth field output.

In the first case statement, where we assign the F1, F2, F3, and F4 variables to the field positions that we want to extract for each operating system, notice that only AIX and Linux assign the F4 variable to a valid field. HP-UX, OpenBSD, and SunOS all have the F4 variable assigned the field \$1, F4=1. I did it this way so that I would not have to rewrite the vmstat command statement for a second time to extract just three fields. This method helps to make the code shorter and less confusing — at least I hope it is less confusing! There is a comment next to each F4 variable assignment that states that this field assignment is bogus and not used in the shell script.

Other than these minor changes the shell script for the vmstat solution is the same as the solution for the iostat command. The vmstat_loadmon.ksh shell script is in action in Listing 19-15 on a Solaris machine.

```
# ./vmstat_loadmon.ksh
The Operating System is SunOS
Gathering CPU Statistics using vmstat...
There are 2 sampling periods with
each interval lasting 300 seconds
...Please wait while gathering statistics...
User part is 14%
System part is 54%
Idle time is 31%
```

Listing 19-15 vmstat_loadmon.ksh shell script in action

Notice that the Solaris output shown in Listing 19-15 does not show the I/O wait state. This information is available only on AIX for the vmstat shell script. The output format is the same as the last few shell scripts. It is up to you how you want to use this information. Let's look at some other options that you may be interested in next.

Other Options to Consider

As with any shell script there is always room for improvement, and this set of shell scripts is no exception. I have a few suggestions, but I'm sure that you can think of a few more.

Try to Detect Any Possible Problems for the User

One thing that would be valuable when looking at the CPU load statistics is to try to detect any problems. For example, if the system percentage plus the user percentage is consistently greater than 90 percent, the system may be CPU bound. This is easy to code into any of these shell scripts using the following statement:

((SYSTEM + USER > 90)) && echo "\nWarning: This system is CPU-bound\n"

Another possible problem happens when the I/O wait percentage is consistently over 80 percent; then the system may be I/O bound. This, too, is easy to code into the shell scripts. System problem thresholds vary widely depending on whom you are talking to, so I will leave the details up to you. I'm sure you can come up with some other problem detection techniques.

Show the User the Top CPU Hogs

Whenever the system is stressed under load, the cause of the problem may be a runaway process or a developer trying out the fork() system call during the middle of the day (same problem, different cause!). To show the user the top CPU hogs, you can use the ps auxw command. Notice that there is *not* a hyphen before auxw! Something like the following command syntax will work:

ps auxw | head -n 15

The output is sorted by CPU usage in descending order from the top. Also, most UNIX operating systems have a top-like command. In AIX, it is topas; in HP-UX and Linux, it is top; and in Solaris, it is prstat. Any of these commands will show you real-time process statistics.

Gathering a Large Amount of Data for Plotting

Another method is to get a lot of short intervals over a longer period of time. The sar command is perfect for this type of data gathering. Using this method of short intervals over a long period, maybe eight hours, gives you a detailed picture of how the load fluctuates through the day. This is the perfect kind of detailed data for graphing on a line chart. It is very easy to take the sar data and use a standard spreadsheet program to create graphs of the system load versus time.

Summary

This chapter presented some different concepts that are not in any other chapter, and it is always intended that way throughout this book. Play around with these shell scripts, and see how you can improve the usefulness of each script. It is always fun to find a new use for a shell script by playing with the code.

In the next chapter, we are going to study some techniques to monitor for stale disk partitions on an AIX system. I hope you gained some knowledge in this chapter, and every chapter. See you in the next chapter.

Lab Assignments

- Modify the uptime_loadmon.Bash shell script to add an email alert if the trigger threshold is exceeded.
- Modify the sar_loadmon.Bash, iostst_loadmon.Bash, and vmstat_loadmon .Bash shell scripts to add a threshold trigger when the average user time exceeds 95% during the sampling period.
- 3. Modify the sar_loadmon.Bash shell script so that it will collect data over a long period of time and save this data for plotting. The idea is to set up 60-second sample intervals sampling 30 times per hour. This data should be stored in a flat text file with the following data on each separate line: date/time stamp, user time, system time, I/O wait (for systems that support this), and idle time.

CHAPTER 20

Monitoring for Stale Disk Partitions (AIX-Specific)

Monitoring for stale disk partitions is an AIX thing. To understand this chapter you need to be familiar with the Logical Volume Manager (LVM) that is at the heart of the AIX disk subsystem. We will get to the LVM in the next section. At the high level a stale disk partition means that the mirrored disks are not in sync. Sometimes when you find stale disk partitions you can resync the mirrors, and all is well. If the mirrors will not sync up, you may be seeing the first signs of a failing disk.

We are going to look at three methods of monitoring for stale partitions:

- Monitoring at the logical volume (LV) level
- Monitoring at the physical volume (PV), or disk, level
- Monitoring at the volume group (VG), PV, and LV levels to get the full picture

All three methods will report the number of stale disk partitions, but it is nice to know the VG, PV, and the LV that are involved in the unsynced mirrors. We are going to step through the entire process of building these shell scripts, starting with the command syntax required to query the system. Before we start our scripting effort, I want to give you a high-level overview of the AIX LVM and the commands we are going to use.

AIX Logical Volume Manager (LVM)

Unlike most UNIX operating systems, IBM manages disk resources with what IBM calls the *Logical Volume Manager* (LVM). The LVM consists of the following components, starting with the smallest.

Starting at the physical volume (PV) level, or a Logical Unit Number (LUN) presented in a SAN environment, AIX's LVM recognizes this device (disk drive or LUN) as a new *physical volume*, or PV. Each PV, or disk, in the LVM is broken down into small partitions called *physical partitions* (PP). The default PP size depends on the disk drives

the system found when cfgmgr (Configuration Manager) was executed either by a reboot or by executing the cfgmgr –V command on the command line as root.

The LVM uses groups of these PPs to create a *logical map* to point to the actual PPs on the disk. These mapped partitions are called *logical partitions* (LP). The sizes of an LP and PP are exactly the same because an LP is just a pointer to a PP. Therefore, the operating system always sees contiguous disk space.

At the next level up, we have the logical volume (LV). An LV consists of one or more LPs. The LV can span multiple PVs. This is the level at which the Systems Administrator creates mirrors or various RAID configurations. When an LV is first created, the LV is considered *raw*, at least in AIX, meaning that it does not have a filesystem mount point. Raw LVs are commonly used for databases.

On top of an LV we can create a filesystem, which has a mount point — for example, /scripts. The LV does not require a filesystem if you want the LV to remain raw, but you can create one.

A volume group (VG) is a collection of one or more physical volumes (PVs), or disks. A PV is listed on the system as an hdisk#, where # is an integer value. A VG is the largest component of the LMV. A VG contains one or more LVs, so this is the mechanism that allows an LV to span multiple PVs.

That is the high-level overview of the LVM and its components. For this chapter we are going to focus our attention at the VG, PV, LV, and PP levels, and we are concerned only with disks in a mirrored configuration. If you want more information on the AIX LVM, there are plenty of IBM Red Book books that go into great detail about AIX system management at www.redbooks.ibm.com.

The Commands and Methods

As usual, we need the command syntax before we can write a shell script. We will work with three LVM commands in this chapter. Each of these commands queries the system for specific information on the components and status of the disk subsystem. Before we proceed, it is important to know what each of these commands is used for and what type of information can be gathered from the system.

Disk Subsystem Commands

The **lsvg** command queries the system for VG information. To see which VGs are *varied-on*, or active, we add the -o switch to the lsvg command. We also have the -1 flag that allows the lsvg command to query the system for the contents of a specific VG. We are interested in one of the fields in the lsvg <VG_name> command output, called STALE_PPs:, which has a value representing the number of stale PVs in the target VG. Ideally, we want this number to be zero.

Then we move to the LV command, **lslv**. The lslv command will query the system for the status information of a specific LV, which is entered as a command parameter. One of the fields in the output of the lslv <LV_name> command is STALE PP:. This output shows the number of stale PPs for the LV specified on the command line.

Ideally, we want this number to be zero. If we add the -1 flag to the lslv command, we can see which PVs are associated with the LV in the first column of the command output.

Next we can move down to the PV, or disk, level. The **lspv** command queries the system for information on a specific PV, which is passed as a command parameter to the lspv command. Like lslv, the lspv command also reports the number of STALE PARTITIONS: as a field in the output.

You will see the output of each of these commands as we write the scripts for this chapter. We have the commands defined, so we are now ready to start creating our first shell script to monitor for stale disk partitions.

Method 1: Monitoring for Stale PPs at the LV Level

The easiest, but not always the quickest, method of checking for stale disk partitions is to work at the LV level of the LVM structure. Querying the system for LV stale-partition information gives the high-level overview for each LV. If, however, the LV spans more than one PV, or disk, another step must be taken to find the actual mirrored disks that are not in sync. We will get to this finer granularity of monitoring in the next section of this chapter.

We start our monitoring by issuing an LVM query to find each of the active VGs on the system, or the VGs that are *varied online*. For this step we use the <code>lsvg -o</code> command. The <code>-o</code> flag tells the <code>lsvg</code> command to list only the VGs that are *currently* varied online. Many more VGs may exist on the system, but if they are not varied online we cannot check the status of any of the LVs that reside within the VG, because the entire VG is inactive. Let's assign the VG list to a variable called <code>ACTIVE_VG_LIST</code>:

```
ACTIVE_VG_LIST=$(lsvg -o)
```

My test machine has two VGs, and both are active:

rootvg appvg2

The previous command saves the active VGs in a variable. Using the ACTIVE_VG_ LIST variable contents, we next create a list of active LVs on the system. Each VG will have one or more LVs that may or may not be active, or *open*. Using the \$ACTIVE_VG_LIST data we can query the system to list each active LV within each active VG. The lsvg -1 \$VG command queries the system at the VG level to display the contents. Listing 20-1 shows the output of the lsvg -1 appvg2 rootvg command on my test machine.

appvg2:						
LV NAME	TYPE	LPs	PPs	PVs	LV STATE	MOUNT POINT
tel_lv	jfs	2	2	1	open/syncd	/usr/telalert
oracle_lv	jfs	128	128	1	open/syncd	/oracle

Listing 20-1 Output of the lsvg -l appvg2 rootvg command

oradata_1	lv	jfs	128	128	1	open/syncd	/oradata
ar_lv		jfs	16	16	1	open/syncd	/usr/ar
remp_tmp(01	jfs	128	128	1	open/syncd	/remd_tmp01
export_l	v	jfs	100	100	1	open/syncd	/export
loglv00		jfslog	1	1	1	open/syncd	N/A
remp2_ct	101	jfs	1	1	1	open/syncd	/remd_ctl01
remp2_ct	102	jfs	1	1	1	open/syncd	/remd_ct102
remp2_ct	103	jfs	1	1	1	open/syncd	/remd_ct103
rempR2_da	at01	jfs	192	192	1	open/syncd	/remd_dat01
R2remedy_	_lv	jfs	10	10	1	open/syncd	/usr/remedy
remp2_log	g1a	jfs	1	1	1	open/syncd	/remd_log1a
remp2_log	g1b	jfs	1	1	1	open/syncd	/remd_log1b
remp2_log	g2a	jfs	1	1	1	open/syncd	/remd_log2a
remp2_log	g2b	jfs	1	1	1	open/syncd	/remd_log2b
remp2_log	g3a	jfs	1	1	1	open/syncd	/remd_log3a
remp2_log	g3b	jfs	1	1	1	open/syncd	/remd_log3b
remp2_log	g4a	jfs	1	1	1	open/syncd	/remd_log4a
remp2_log	g4b	jfs	1	1	1	open/syncd	/remd_log4b
remp2_log	g5a	jfs	1	1	1	open/syncd	/remd_log5a
remp2_log	g5b	jfs	1	1	1	open/syncd	/remd_log5b
remp2_rbs	s01	jfs	47	47	1	open/syncd	/remd_rbs01
remp2_sys	s01	jfs	4	4	1	open/syncd	/remd_sys01
arlogs_lv	V	jfs	35	35	1	open/syncd	/usr/ar/logs
remp2_usi	r01	jfs	6	6	1	open/syncd	/remd_usr01
rootvg:							
LV NAME		TYPE	LPs	PPs	PVs	LV STATE	MOUNT POINT
hd5		boot	1	2	2	closed/syncd	N/A
hd6		paging	80	160	2	open/syncd	N/A
hd8		jfslog	1	2	2	open/syncd	N/A
hd4		jfs	4	8	2	open/syncd	/
hd2		jfs	40	80	2	open/syncd	/usr
hd9var		jfs	10	20	2	open/syncd	/var
hd3		jfs	10	20	2	open/syncd	/tmp
hd1		jfs	3	6	2	open/syncd	/home
local_lv		jfs	9	18	2	open/syncd	/usr/local

Listing 20-1 (continued)

The list of LVs is shown in column one. Notice the sixth column in the output in Listing 20-1, LV STATE. Most of the LVs are open/synced, but one LV, hd5, is closed/synced. The hd5 LV that is closed is the *boot logical volume* and is active only when the system is booting up. Because we want only active LVs, all we need to do is to grep on the string open and then awk out the first column. The next command saves the list of currently active LVs in a variable called ACTIVE_LV_LIST:

ACTIVE_LV_LIST=\$(lsvg -1 \$ACTIVE_VG_LIST | grep open | awk '{print \$1}')

In the preceding command, we use our $ACTIVE_VG_LIST$ as a command parameter for the lsvg -l command. Then we pipe (|) to grep the lsvg output for only the

rows that contain the string open. Next, another pipe is used to awk out the first column, specified by awk '{print \$1}'. The result is a list of currently active LV names. If you think about a two-dimensional array, the grep command works on the *rows* and the awk command works on the *columns*.

The only thing left to do is to query each LV for the number of stale PPs, specified by the STALE PP: field. To check every LV we need to set up a for loop to run the same command on each LV in the active list. The command we use to query the LV is lslv -L \$LV_NAME. The output for a single LV is shown in Listing 20-2.

LOGICAL MOLINE	01									
LOGICAL VOLUME:	remp_tmp01	VOLUME GROUP:	appvg2							
LV IDENTIFIER:	00011151b819f83a.5	PERMISSION:	read/write							
VG STATE:	active/complete	LV STATE:	opened/syncd							
TYPE:	jfs	WRITE VERIFY:	off							
MAX LPs:	512	PP SIZE:	32 megabyte(s)							
COPIES:	1	SCHED POLICY:	parallel							
LPs:	128	PPs:	128							
STALE PPs:	0	BB POLICY:	relocatable							
INTER-POLICY:	minimum	RELOCATABLE:	yes							
INTRA-POLICY:	middle	UPPER BOUND:	32							
MOUNT POINT:	/remd_tmp01	LABEL:	/remd_tmp01							
MIRROR WRITE CONSI	MIRROR WRITE CONSISTENCY: on									
EACH LP COPY ON A	SEPARATE PV ?: yes									

Listing 20-2 LV statistics for the remp_tmp01 logical volume

Notice in the command output in Listing 20-2 the eighth row, where the field STALE PPs: is listed. The second column of this row contains the number of stale partitions in the LV. Ideally, we want this value to be zero, 0. If the value is greater than zero we have a problem. Specifically, the mirrored disks associated with this LV are not in sync, which translates to a worthless mirror. Looking at this output, how are we supposed to get the number of stale disk partitions? It turns out that this is a very simple combination of grep and awk. Take a look at the following command statement:

NUM_STALE_PP=\$(lslv -L \$LV | grep "STALE PP" | awk '{print \$3}'

This statement saves the number of stale PPs into the NUM_STALE_PP variable. We accomplish this feat by command substitution, specified by the VARIABLE= \$(commands) notation. The way to make this task easy is to do the parsing one step at a time. First, the row containing the STALE PP string is extracted and is provided as input to the next command in the pipe. The next command in the pipe is an awk statement that extracts only the third field, specified by '{print \$3}'. At this point you may be asking why we used the third field instead of the second. By default, awk uses white space as a field separator, and because STALE PPs: 0 contains two areas of white space, we need the third field instead of the second.

Now that we have all of the commands, all we need to do is set up a loop to run the previous command against each LV stored in the *\$ACTIVE_LV_LIST* variable. A little for loop will work just fine for this script. The loop is shown in Listing 20-3.

```
THIS_HOST=$(hostname)
for LV in $ACTIVE_LV_LIST
do
     NUM_STALE_PP=$(lslv -L $LV | grep "STALE PP" | awk '{print $3}'
     if ((NUM_STALE_PP > 0))
     then
        echo "\n${THIS_HOST}: $LV has $NUM_STALE_PP stale PPs"
     fi
done
```

Listing 20-3 Loop to show the number of stale PPs from each LV

I want to point out several things in Listing 20-3. First, notice that we save the hostname of the machine in a variable called THIS_HOST. When creating any type of report we need to know which machine we are reporting on. When you have more than 100 machines, things can get a little confusing if you do not have a hostname to go with the report.

A for loop needs a list of items to loop through. To get the list of active LVs, we use the \$ACTIVE_LV_LIST to provide our for loop with a list. The next step is to run the lslv -L command for each LV listed and extract the field that shows the number of stale PPs. For this command we again use command substitution to assign the value to a variable called NUM_STALE_PP. Using this saved value we do a numeric test in the if statement. Notice that we did not add a dollar sign (\$) in front of the NUM_STALE_PP variable. Because we used the double parentheses numeric test method, the command assumes that every nonnumeric string is a variable, so the dollar sign (\$) is not needed.

If we find that the number of stale PPs is greater than zero, we use an echo statement to show the hostname of the machine followed by the LV name that has stale partitions and, last, the number of stale partitions that were found. These steps are followed for every active LV in every active VG on the entire system. The full shell script is shown in Listing 20-4.

```
#!/bin/ksh
#
# SCRIPT: stale_LV_mon.ksh
#
# AUTHOR: Randy Michael
# DATE: 01/22/2007
# REV: 1.1.P
#
# PLATFORM: AIX only
#
# PURPOSE: This shell script is used to query the system
# for stale PPs in every active LV within every active
```

Listing 20-4 stale_LV_mon.ksh shell script

```
VG.
# REVISION LIST:
# set -x # Uncomment to debug this script
# set -n # Uncomment to check command syntax without any execution
THIS_HOST=`hostname` # Hostname of this machine
STALE_PP_COUNT=0
                     # Initialize to zero
# Find all active VGs
echo "\nGathering a list of active Volume Groups"
ACTIVE_VG_LIST=$(lsvg -o)
# Find all active LVs in every active VG.
echo "\nCreating a list of all active Logical Volume"
ACTIVE_LV_LIST=$(lsvg -1 $ACTIVE_VG_LIST | grep open | awk '{print $1}')
# Loop through each active LV and query for stale disk partitions
echo "\nLooping through each Logical Volume searching for stale PPs"
echo "...Please be patient; this may take several minutes to complete..."
for LV in $ACTIVE_LV_LIST
do
     # Extract the number of STALE PPs for each active LV
     NUM_STALE_PP=`lslv -L $LV | grep "STALE PP" | awk '{print $3}'`
     # Check for a value greater than zero
      if ((NUM_STALE_PP > 0))
      then
           # Increment the stale PP counter
           (( STALE_PP_COUNT = $STALE_PP_COUNT + 1))
           # Report on all LVs containing stale disk partitions
           echo "\n${THIS_HOST}: $LV has $NUM_STALE_PP PPs"
      fi
done
# Give some feedback if no stale disk partitions were found
if ((STALE_PP_COUNT == 0))
then
      echo "\nNo stale PPs were found in any active LV...EXITING...\n"
fi
```

Listing 20-4 (continued)

Notice in the script in Listing 20-4 that we added notification at each step in the process. As always, we need to let the user know what is going on. Before each command I added an echo statement to show the user how we progress through the

shell script. I also added a STALE_PP_COUNT variable to give feedback if no stale PPs were found. Now let's move on to searching for stale PPs at the PV level instead of the LV level.

Method 2: Monitoring for Stale PPs at the PV Level

Checking for stale disk partitions at the LV level will let you know that one or more LVs have stale PPs. To get a better picture of where the unsynced mirrors reside, we need to look at the hdisk level. In this section we are going to change the query point for searching for stale PPs from the LV to the PV, or disk level. The savings in execution time between these two methods is threefold in favor of working directly with the disks by my measurements. On my test machine the LV query took 40.77 seconds in real time, 0.36 seconds of system time, and 0.02 seconds of user time. Using the PV query method I reduced the execution time to 12.77 seconds in real time and 0.17 seconds of system time, and I had the same 0.02 seconds for user time. To understand the LV and PV configuration I have 18 mirrored disks, which are 9 mirror pairs of 9.1 GB disk drives, and a total of 32 LVs. Because an LV query takes longer to execute than a PV query, it is understandable that the PV query won. Depending on the system configuration, this timing advantage may not always hold.

In the PV monitoring method we still are concerned only with the hdisks that are in currently varied-on volume groups (VGs), as we did in the LV method using the lsvg -o command. Using this active VG list we can query each active VG and extract all of the hdisks that belong to each VG. Once we have a complete list of all of the hdisks we can start a loop and query each of the PVs independently. The output of a PV query is similar to the LV query statistics in Listing 20-2. Take a look at the PV query of hdisk5 using the lspv -1 hdisk5 command in Listing 20-5.

PHYSICAL VOLUME:	hdisk5	VOLUME GROUP:	appvg2
PV IDENTIFIER:	00011150e33c3f14 VG IDEN	TIFIER 0001115	0e33ce9bb
PV STATE:	active		
STALE PARTITIONS:	0	ALLOCATABLE:	yes
PP SIZE:	16 megabyte(s)	LOGICAL VOLUMES:	2
TOTAL PPs:	542 (8672 megabytes)	VG DESCRIPTORS:	1
FREE PPs:	397 (6352 megabytes)		
USED PPs:	145 (2320 megabytes)		
FREE DISTRIBUTION:	890091108109		
USED DISTRIBUTION:	20108170000		

Listing 20-5 PV statistics for the hdisk5 physical volume

In the output in Listing 20-5 the STALE PARTITIONS: field in row four and its value are the third field in the row. If the stale partition value ever exceeds zero, we use the same type of reporting technique that we used in the LV query in Method 1. If no stale partitions are found, we can give the "all is well" message and exit the script.

Because we have the basic idea of the process, let's take a look at the shell script in Listing 20-6. We will explain the technique in further detail at the end of the code listing.

```
#!/usr/bin/ksh
# SCRIPT: stale_PP_mon.ksh
# AUTHOR: Randy Michael
# DATE: 01/29/2007
# REV: 1.2.P
#
# PLATFORM: AIX only
#
# PURPOSE: This shell script is used to query the system for stale PPs.
#
          The method queries the system for all of the currently
         varied-on volume groups and then builds a list
#
#
          of the PVs to query. If a PV query detects any stale
          partitions notification is sent to the screen. Each step in
#
#
          the process has user notification
# REVISION LIST:
#
# set -x # Uncomment to debug this shell script
# set -n # Uncomment to check command syntax without any execution
THIS_HOST=$(hostname)  # Hostname of this machine
HDISK_LIST=
                        # Initialize to NULL
                        # Initialize to zero
STALE PP COUNT=0
# Inform the user at each step
echo "\nGathering a list of hdisks to query\n"
# Loop through each currently varied-on VG
for VG in $(lsvg -o)
do
    # Build a list of hdisks that belong to currently varied-on VGs
    echo "Querying $VG for a list of disks"
   HDISK_LIST="$HDISK_LIST $(lsvg -p $VG |grep disk \
                                           awk '{print $1}')"
done
echo "\nStarting the hdisk query on individual disks\n"
# Loop through each of the hdisks found in the previous loop
for HDISK in $(echo $HDISK_LIST)
do
    # Query a new hdisk on each loop iteration
```

Listing 20-6 stale_PP_mon.ksh shell script

```
echo "Querying $HDISK for stale partitions"
    NUM_STALE_PP=$(lspv -L $HDISK | grep "STALE PARTITIONS:" \
                   | awk '{print $3}')
    # Check to see if the stale partition count is greater than zero
    if ((NUM_STALE_PP > 0))
    then
         # This hdisk has at least one stale partition - Report it!
         echo "\n${THIS_HOST}: $HDISK has $NUM_STALE_PP Stale
Partitions"
         # Build a list of hdisks that have stale disk partitions
         STALE_HDISK_LIST=$(echo $STALE_HDISK_LIST; echo $HDISK)
   fi
done
# If no stale partitions were found send an "all is good" message
((NUM_STALE_PP > 0)) \setminus
  || echo "\n${THIS_HOST}: No Stale PPs have been found...EXITING...\n"
```

Listing 20-6 (continued)

We totally changed our viewpoint in our search for stale disk partitions. Instead of working at each LV we are scanning each disk, or PV, independently. The search time on my test machine was three times faster, but my machine configuration does not mean that your system query will be as fast. I want to start at the top of our stale_PP_mon.ksh shell script in Listing 20-6 and work to the bottom.

We start off the script by initializing three variables, THIS_HOST (the hostname of the reporting machine), HDISK_LIST (the list of PVs to query, which we initialize to NULL), and STALE_PP_COUNT (the total number of stale disk partitions on all disks, which is initialized to zero). We will show how each of these variables is used as we progress through the script.

The next step is to use the list of currently varied-on VGs (using the lsvg -o command) to create a list of currently available hdisks — at least they should be available. We do this in a for loop by appending to the HDISK_LIST variable during each loop iteration. Once we have a list of available system disks, we start a for loop to query each hdisk individually. During the query statement

we capture the number of stale disk partitions by using grep and awk together in the same statement. Just remember that the grep command acts on the *rows* and the awk statement acts on the *columns*. On each loop iteration we check the value of the \$NUM_STALE_PP variable. If the count is greater than zero we do two things: report

the disk to the screen, and append to the STALE_HDISK_LIST variable. Notice how we append to a variable that currently has data in it. By initializing the variable to NULL (specified by VARIABLE=), by creating an assignment to nothing, we can always append to the variable using the following syntax:

VARIABLE="\$VARIABLE \$NEW_VALUE"

Because the \$VARIABLE has an initial value of nothing, NULL, the first value assigned is a new value, and all subsequent values are appended to the VARIABLE variable on each loop iteration.

At the end of the script we test the \$NUM_STALE_PP variable, which has a running count of all stale disk partitions. If the value is zero, we let the end user know that everything is OK. Notice how we do the test. We do a numerical test on the \$NUM_STALE_PP variable to see if it is greater than zero. If the value is one or more, the statement is true. On a true statement the logical OR (||) passes control to the second part of the statement, which states "No stale PPs have been found." The logical OR saves an if statement and is faster to execute than an if statement.

Now, that was a fun little script. We can improve on both scripts that have been presented thus far. There is a procedure to attempt to resync the disks containing stale partitions. In the next section we are going to combine the LV and PP query methods and add in a VG query as the top-level query to search for stale disk partitions. We will also attempt to resync all of the stale LVs that we find, if the ATTEMPT_RESYNC variable is set to TRUE.

Method 3: VG, LV, and PV Monitoring with a resync

We have looked at stale disk mirrors from two angles, but we can look for stale disk partitions at a higher level, the VG level. By using the lsvg command we can find which VG has disks that have stale PPs. Using the lsvg <VG_name> command we can shorten our queries to a limited number of disks, although with Murphy's Law working, it just might be the largest VG on the planet!

The strategy that we want to follow is first to query each active VG for stale PVs, which we find using the lsvg <VG_name> command. Then, for each VG that has the STALE PV: field greater than zero, we query the current VG in the loop to get a list of associated PVs, or disks. Using a list of all of the PVs that we found, we conduct a query of each disk to find both the list of LVs the PV is associated with and the value of the STALE PARTITIONS: field. For each PV found, to have at least one stale partition, we query the PV for a list of LVs that reside on the current PV. Please don't get confused now! The steps involved are a natural progression through the food chain to the source. The final result of all of these queries is that we know which VG, PV, and LV have unsynced mirrors, which is the complete picture that we want.

The process that we follow in this section is faster to execute and easier to follow, so let's start. The commands we are going to use are shown in Listing 20-7.

lsvg -o	Produces a list of active VGs
lsvg \$VG_NAME	Queries the target VG for status information
lsvg -p \$VG_NAME	Produces a list of hdisks that belong to the VG
lspv \$PV_NAME	Queries the hdisk specified by \$PV_NAME
lspv -l \$PV_NAME	Produces a list of LVs on the target hdisk
lslv \$LV_NAME	Queries the target LV for status information
syncvg \$HDISK_LIST	Synchronizes the mirrors at the hdisk level
syncvg -l \$LV_LIST	Synchronizes the mirrors at the LV level
varyonvg	Synchronizes only the stale partitions

Listing 20-7 Command summary for the Method 3 shell script

Using the nine commands in Listing 20-7 we can produce a fast-executing shell script that produces the full picture of exactly where all of the unsynced mirrors reside, and we can even attempt to fix the problem!

For this shell script I want to present you with the entire script; then we will step through and explain the philosophy behind the techniques used. In studying Listing 20-8, pay close attention to the bold text.

```
#!/usr/bin/ksh
# SCRIPT: stale_VG_PV_LV_PP_mon.ksh
# AUTHOR: Randy Michael
# DATE: 01/29/2007
# REV: 1.2.P
# PLATFORM: AIX only
# PURPOSE: This shell script is used to query the system for stale PPs.
#
         The method queries the system for all of the currently
#
         varied-on volume groups and then builds a list of the PVs to
#
         query. If a PV query detects any stale partitions notification
#
         is sent to the screen. Each step in the process has user
         notification.
# REVISION LIST:
# set -x # Uncomment to debug this shell script
# set -n # Uncomment to check command syntax without any execution
# EXIT CODES: 0 ==> Normal execution or no stale PP were found
             1 ==> Trap EXIT
#
#
             2 ==> Auto resyncing failed
#
```

```
Listing 20-8 stale_VG_PV_LV_PP_mon.ksh shell script
```

```
**********
ATTEMPT_RESYNC=FALSE  # Flag to enable auto resync, "TRUE" will resync
LOGFILE="/tmp/stale_PP_log" # Stale PP logfile
THIS_HOST=$(hostname)  # Hostname of this machine
                    # Initialize to zero
STALE_PP_COUNT=0
STALE_PV_COUNT=0
                   # Initialize to zero
HDISK_LIST=
                    # Initialize to NULL
                   # Initialize to NULL
INACTIVE_PP_LIST=
STALE_PV_LIST=
                    # Initialize to NULL
STALE LV LIST=
                   # Initialize to NULL
STALE_VG_LIST=
                    # Initialize to NULL
RESYNC_LV_LIST=
                   # Initialize to NULL
PV_LIST=
                    # Initialize to NULL
****
#### INITIALIZE THE LOG FILE ####
>$LOGFILE # Initialize the log file to empty
date >> $LOGFILE # Date the log file was created
echo "\n$THIS_HOST \n" >> $LOGFILE # Hostname for this report
# Trap Exit function
function trap_exit
echo "\n\t...EXITING on a TRAPPED signal...\n"
}
****
# Set a trap...
trap 'trap_exit; exit 1' 1 2 3 5 15
*****
*****
# Inform the user at each step
# Loop through each currently varied-on VG and query VG for stale PVs.
# For any VG that has at least one stale PV we then query the VG
# for the list of associated PVs and build the $PV_LIST
```

```
echo "\nSearching each Volume Group for stale Physical Volumes...\c" \
        | tee -a $LOGFILE
# Search each VG for stale PVs, then build a list of VGs and PVs
# that have stale disk partitions
for VG in $(lsvg -o)
do
    NUM_STALE_PV=$(lsvg $VG | grep 'STALE PVs:' | awk '{print $3}')
    if ((NUM_STALE_PV > 0))
    then
          STALE_VG_LIST="$STALE_VG_LIST $VG"
         PV_LIST="$PV_LIST $(lsvg -p $VG | tail +3 | awk '{print $1}')"
          ((STALE_PV_COUNT = STALE_PV_COUNT + 1))
     fi
done
# Test to see if any stale PVs were found; if not then
# exit with return code 0
if ((STALE_PV_COUNT == 0))
then
     echo "\nNo Stale Disk Mirrors Found...EXITING...\n" \
| tee -a $LOGFILE
    exit 0
else
     echo "\nStale Disk Mirrors Found!...Searching each hdisk for \
stale PPs...\c" | tee -a $LOGFILE
fi
# Now we have a list of PVs from every VG that reported stale PVs
# The next step is to query each PV to make sure each PV is in
# an "active" state and then query each PV for stale PPs.
# If a PV is found to be inactive then we will not query
# the PV for stale partitions, but move on to the next PV in
# the $PV LIST.
for HDISK in $PV LIST
do
    PV_STATE=$(lspv $HDISK | grep 'PV STATE:' | awk '{print $3}')
    if [[ $PV_STATE != 'active' ]]
     then
         INACTIVE_PV_LIST="$INACTIVE_PV_LIST $HDISK"
     fi
     if ! $(echo $INACTIVE_PV_LIST | grep $HDISK) >/dev/null 2>&1
     then
```

Listing 20-8 (continued)

```
NUM_STALE_PP=$(lspv $HDISK | grep 'STALE PARTITIONS:' \
                         | awk '{print $3}')
          if ((NUM_STALE_PP > 0))
          then
               STALE_PV_LIST="$STALE_PV_LIST $HDISK"
               ((STALE_PP_COUNT = $STALE_PP_COUNT + 1))
          fi
     fi
done
# Now we have the list of PVs that contain the stale PPs.
# Next we want to get a list of all of the LVs affected.
echo "\nSearching each disk with stale PPs for associated LVs\c" \
        tee -a $LOGFILE
for PV in $STALE_PV_LIST
do
     STALE_LV_LIST="$STALE_LV_LIST $(lspv -1 $PV | tail +3 \
                    awk '{print $1}')"
done
# Using the STALE_LV_LIST variable list we want to query
# each LV to find which ones need to be resynced
echo "\nSearch each LV for stale partitions to build a resync LV \
list\c" | tee -a $LOGFILE
for LV in $STALE_LV_LIST
do
     LV_NUM_STALE_PP=$(lslv $LV | grep "STALE PPs:" | awk '{print $3}')
     (($LV_NUM_STALE_PP != 0)) && RESYNC_LV_LIST="$RESYNC_LV_LIST $LV"
done
# If any inactive PVs were found we need to inform the user
# of each inactive PV
# Check for a NULL variable
if [[ -n "$INACTIVE_PV_LIST" && "$INACTIVE_PV_LIST" != '' ]]
then
     for PV in $INACTIVE_PV_LIST
     do
          echo "\nWARNING: Inactive Physical Volume Found:" \
| tee -a $LOGFILE
          echo "\n$PV is currently inactive:\n" | tee -a $LOGFILE
          echo "\nThis script is not suitable to to correct this \
```

```
Listing 20-8 (continued)
```

```
problem..." | tee -a $LOGFILE
         echo " ...CALL IBM SUPPORT ABOUT ${PV}..." \
| tee -a $LOGFILE
    done
fi
echo "\nStale Partitions have been found on at least one disk!" \
       | tee -a $LOGFILE
echo "\nThe following Volume Group(s) have stale PVs:\n" \
       | tee -a $LOGFILE
echo $STALE_VG_LIST | tee -a $LOGFILE
echo "\nThe stale disk(s) involved include the following:n" \
        | tee -a $LOGFILE
echo $STALE_PV_LIST | tee -a $LOGFILE
echo "\nThe following Logical Volumes need to be resynced:\n" \
        | tee -a $LOGFILE
echo $RESYNC_LV_LIST | tee -a $LOGFILE
if [[ $ATTEMPT_RESYNC = "TRUE" ]]
then
    echo "\nAttempting to resync the LVs on $RESYNC_PV_LIST ...\n" \
             | tee -a $LOGFILE
     syncvg -1 $RESYNC_LV_LIST | tee -a $LOGFILE 2>&1
     if (( $? == 0))
     then
          echo "\nResyncing all of the LVs SUCCESSFUL...EXITING..." \
                  tee -a $LOGFILE
     else
         echo "\nResyncing FAILED...EXITING...\n" | tee -a $LOGFILE
          exit 2
     fi
else
    echo "\nAuto resync is not enabled...set to TRUE to automatically \
resync\n" | tee -a $LOGFILE
     echo "\n\t...EXITING...\n" | tee -a $LOGFILE
fi
echo "\nThe log file is: $LOGFILE\n"
```

The shell script in Listing 20-8 is interesting because of the techniques used. As we start at the top of the shell script, notice the first variable definition, ATTEMPT_RESYNC. I initialize this variable to FALSE because resyncing at the LV level can cause a significant system load. A better method is to run the **varyonvg** command without any arguments. This method will only resync the stale partitions. Because of the possibility of loading down the system and slowing production response time, I initialize this

variable to FALSE. If I am working on a test/development or sandbox machine, I usually set the ATTEMPT_RESYNC variable to TRUE, in uppercase. The TRUE setting will attempt to resync, at the LV level, every stale LV.

The remaining variables initialize the LOGFILE and THIS_HOST variables to the log filename and hostname, respectively. A couple of counters are initialized to zero, and seven other variables are initialized to NULL. In the next section we initialize the \$LOGFILE with header information.

The only function in this script is the trap_exit function. The trap_exit function displays to standard output (stdout) ... EXITING on a TRAPPED signal... when a trap is captured. The trap is set for exit codes 1, 2, 3, 5, and 15 and then the script exits with return code 1. This functionality is just a notification measure for the user. Now we are at BEGINNING OF MAIN in our script.

At each step through this shell script we want to give the user feedback so that he or she will know what is going on. When writing shell scripts you need to do two things: comment *everything* and give your users feedback so that they know what is going on. In our first query we inform the users that we are searching each VG for stale PVs. For this step we use the <code>lsvg -o</code> command to get a list of currently varied-on VGs. Using this active VG list, we use a for loop to loop through each active VG and query for the STALE PVs: field using the <code>lsvg \$VG</code> command to extract the number of stale PVs in each VG using both <code>grep</code> and <code>awk</code>. When any stale PVs are detected, the VG is added to the <code>STALE_VG_LIST</code> variable, and all of the PVs in the VG are then added to the <code>PV_LIST</code> variable, specified by the <code>lsvg -p</code> \$VG command. Next the <code>STALE_PV_COUNT</code> variable is incremented by one for each PV using the math notation (<code>(STALE_PV_COUNT = STALE_PV_COUNT + 1)</code>). At this point we have a list of all of the VGs that have stale PVs identified.

If the STALE_PV_COUNT variable is zero, there are no stale disk partitions to report in the system for the currently varied-on VGs. If the count is zero, we inform the user that no stale disk mirrors were found, and we exit the script with a return code of 0, zero. If no stale disk partitions exist, this shell script executes in seconds. If the count is greater than zero, we inform the user that stale disk mirrors were found, and we continue to the next step, which is to query each PV in the *\$PV_LIST*, searching for stale disk partitions.

To query each PV that is part of a VG that has stale PVs identified, we use a for loop to loop through each hdisk assigned to the \$PV_LIST variable. Before we can query the disk, we need to ensure that the PV is in an *active* state. If the disk is inactive, we cannot query that disk. In this section of the shell script we use the 1spv \$HDISK command within the for loop twice. The first time we are ensuring that the disk is active, and in the second step we query the disk for the value of the STALE PARTITIONS: field. If the disk is found to be inactive, we just add the disk to the INACTIVE_PV_LIST variable. If the disk is in an active state and the query detects any stale partitions, we add the hdisk to the STALE_PV_LIST variable. Notice in this section the if statement syntax that is used to check for inactive PVs before the disk query is initiated:

if ! \$(echo \$INACTIVE_PV_LIST | grep \$HDISK) >/dev/null 2>&1

This test ensures that the disk is not listed in the *\$INACTIVE_PV_LIST* variable. The nice thing about using this syntax is that we use the *if* statement to check the return code of the enclosed command. We also negate the response so that we are testing for the disk not being listed in the variable by using the ! operator. To stop any screen output, the command is redirected to the bit bucket, and standard error is redirected to standard output, specified by the 2>&1 notation. Through the process of this for loop we populate the STALE_PV_LIST variable, which is a list of each of the active disks on the system that have stale disk partitions. We also keep a running count of the stale PPs found.

In the next section we use the populated \$STALE_PV_LIST variable to get a list of all of the LVs that are part of each disk in the stale disk list. In this step we use another for loop to loop through each stale PV and populate the STALE_LV_LIST variable using the lspv -1 \$PV command. Then we use this newly populated \$STALE_LV_LIST to query each LV to find which ones have stale PPs. For this section we query each LV using the lslv \$LV command and extract the value of the STALE PP: field using a combination of grep and awk commands in a pipe. Each LV found to have at least one stale PP is added to the RESYNC_LV_LIST variable, which is used later to resync each of the LVs, if enabled, and in the log report.

Now we use the list of inactive PVs, using the <code>\$INACTIVE_PV_LIST</code> variable, to produce notification to the user of each inactive PV found on the system. We start with an if statement and test for the <code>\$INACTIVE_PV_LIST</code> variable being NULL, or empty. If the variable is not NULL, we loop through each PV in the list and issue a warning message to the user for each inactive PV. This information is also logged in the <code>\$LOGFILE</code> using a pipe to the <code>tee -a</code> command to append to the <code>\$LOGFILE</code> and display the information to the screen at the same time.

In the next step, we give the user a list of each VG, PV, and LV that is affected by the stale disk partitions. After this notification is both logged and displayed we attempt to resync the mirrors at the LV level. Sometimes there are just one or two LVs on a PV that have stale disk partitions, so the LV is where we want to attempt to resync. We will attempt a resync only if the \$ATTEMPT_RESYNC variable is initialized to TRUE. Any other value will cause this step to be skipped, but the user is notified that the resync option is disabled. If a resync is enabled, the syncvg -1 \$RESYNC_LV_LIST command is executed. The return code is checked for a zero value, indicating a successful resync operation. If the return code is not zero, you need to call IBM support and replace the disk before it goes dead on you. The steps involved in replacing a disk are beyond the scope of this book. We can also use the varyonvg command to resync only the stale partitions.

Other Options to Consider

As usual, any shell script can be improved, and this set of shell scripts is no exception.

SSA Disks

The **ssaxlate** command is used with a type of disk developed by IBM known as Serial Storage Architecture (SSA). The SSA disks not only use the hdisk# but also have an

associated pdisk#. Normally the hdisk# and the pdisk# differ on the system. The ssaxlate command gives a cross-reference between the two disk representations. It is always a good idea to have this extra information if we are dealing with SSA disks, especially if you are replacing an SSA disk. To use the ssaxlate command, you need to know the specific hdisk# to translate to the corresponding pdisk#, or vice versa. As an example, we want to know what pdisk# translates to hdisk36. The command syntax to do the translation is shown here:

```
# ssaxlate -1 hdisk36
pdisk32
```

In this example, hdisk36 translates to pdisk32. From this you can infer that hdisk0 through hdisk3 are not SSA disks. Usually the first few disks on an AIX system are SCSI, SATA, or FATA disk drives. You can also translate a pdisk# into the corresponding hdisk# by running the ssaxlate command against the pdisk#.

Log Files

In the first two shell scripts in this chapter we did not use a log file as we did in Method 3. It is always nice to have a log file to look at after the fact when you are running any type of system query. Creating a log file is a simple process of defining a variable to point to a filename that you want to use for a log file and appending output to the log file variable. If your system tends to fill up the /tmp filesystem, I recommend creating a log directory, maybe in /usr/local/logs, or creating a separate filesystem just for log files. You can still have the mount point /usr/local/logs, or anything you want. If /tmp fills up, you will not be able to write anything to the log file. You may also want to keep a month's worth of log files to review in case of system problems. To do this you can add a date stamp as a filename extension and remove all files older than 30 days with the find command.

Automated Execution

You can make a cron entry in the root cron table to execute this shell script to automate running the script daily. A sample cron table entry is shown here:

05 23 * * * /usr/local/bin/stale_PP_mon.ksh >/dev/null 2>&1

The preceding cron table entry will execute the stale_PP_mon.ksh shell script every day at 11:05 p.m., 365 days a year. The output is redirected to the bit bucket, but the log file will be created for review the next day.

Event Notification

If you use the previous cron table entry to execute the shell script every day, you may want to get some kind of notification by way of an email or a page. The easiest way is to email the log file to yourself every day. You can also modify the shell script to

produce a very short message as a page. As an example, you could send one of the following text messages to an alphanumeric pager:

\$THIS_HOST: stale PP check OK
\$THIS_HOST: stale PP check FAILED

These are short messages to get the point across, and you will know which machine the page came from.

Summary

In this chapter we looked at a logical progression of creating a shell script by starting at the basics. I hope you have gained at least some knowledge of the AIX Logical Volume Manager (LVM) through this experience. As you can see in this chapter, the first attempt to solve a challenge may not always be the best, or fastest, method; but this is how we learn. If we take these small steps and work up a full-blown shell script with all of the bells and whistles, we have learned a great deal. I know a lot of you do not work on AIX systems, but this is still a valuable exercise.

In the next chapter we are continuing with another AIX topic, turning on/off SSA disk identification lights. See you in the next chapter!

Lab Assignment

1. Rewrite the shell script in Listing 20-8 to utilize functions for each test type: VG, PV, and LV.

CHAPTER 21

Turning On/Off SSA Identification Lights

If your system utilizes the Serial Storage Architecture (SSA) disk subsystem from IBM, you understand how difficult it is to find a specific failed disk in the hundreds of disks that populate the racks. Other needs for SSA disk identification include finding all of the drives attached to a particular system. Then you may also want to see only the drives that are in currently varied-on volume groups or a specific group of disks. In revising this book, I thought about deleting this chapter; however, many shops still use SSA, and this chapter has some very good shell scripting techniques, so I decided to keep this topic. Even if you do not utilize SSA disks, please stick around and follow through the scripting techniques to pick up a few scripting tips.

In identifying hardware components in a system, you usually have a set of tools for this function. This chapter is going to concentrate on AIX systems. The script presented in this chapter is valid only for AIX, but with a few modifications it can run on other UNIX flavors that utilize the SSA subsystem. I am sticking to AIX because this script has an option to query *volume groups*, which not all UNIX flavors support. If your systems are running the Veritas filesystem, then only a few commands need to be modified for my identification script to work because Veritas supports the concept of a volume group.

In identifying an SSA disk you have two ways of referencing the disk. In AIX all disks are represented as an hdisk#. As an example, hdisk0 almost always contains the operating system, and it is part of the rootvg volume group. It is not often an SSA disk; it is usually an internal SCSI disk. If an hdisk is an SSA disk, it has a second disk name that is used within the SSA subsystem, which is called the pdisk#. Not often are the hdisk# and the pdisk# the same number because the first couple of disks are usually SCSI drives. We need to be able to translate an hdisk to its associated pdisk, and vice versa.

Syntax

As always, we need to start out with the commands to accomplish the task. With the SSA subsystem we are concerned about two commands that relate to hdisks and pdisks. The first command, **ssaxlate**, translates an hdisk# into a pdisk#, or vice versa. The second command we use is the *ssaidentify* command, which requires a pdisk representation of the SSA disk drive. This command is used to turn the SSA disk-identification lights on and off. We want the script to identify the SSA disks to recognize either disk format, hdisk or pdisk. With the ssaxlate command this is not a problem.

To use these commands you need to know only the SSA disk to act on and add the appropriate command switch. Let's look at both commands here.

Translating an hdisk to a pdisk

```
# ssaxlate -1 hdisk43
pdisk41
```

In this example hdisk43 translates to pdisk41. This tells me that the hdisk-to-pdisk offset is 2, which I have to assume means that hdisk0 and hdisk1 are both SCSI disks, and hdisk3 through at least hdisk43 are all SSA disks. This is not always the case. It depends on how the AIX configuration manager discovered the disks in the first place, but my statement is a fair assumption. We could just as easily translate pdisk41 to hdisk43 by specifying pdisk41 in the ssaxlate command.

The next step is to actually turn on the identification light for hdisk43, which we discovered to be pdisk41. The ssaidentify command wants the disks represented as pdisks, so we need to use pdisk41 for this command.

Identifying an SSA Disk

```
# ssaidentify -1 pdisk41 -y
```

The ssaidentify command will just return a return code of success or failure, but no text is returned. If the return code is 0, zero, the command was successful. If the return code is nonzero, the command failed for some reason and a message is sent to standard error, which is file descriptor 2. All we are interested in is whether or not the return code is zero.

The Scripting Process

In the SSA identification script we are going to use a lot of functions. These functions perform the work, so we just need the logic to decide which function to execute. An important thing you need to understand about functions is that the function *must* be declared, or written, in the code previous to when you want to execute the function.

This makes sense if you think about it: you have to write the code before you can use it! The functions involved in this shell script are listed in Table 21-1 for your convenience.

FUNCTION NAME	PURPOSE
usage	Shows the user how to use the shell script
man_page	Shows detailed information on how to use the shell script
cleanup	Executes when a trapped exit signal is detected
twirl	Used to give the user feedback that processing continues
all_defined_ pdisks	Controls SSA identification lights for all system SSA disks
all_varied_on_pdisks	Controls SSA disks only in currently varied-on volume groups
list_of_disks	Controls SSA identification of a list of one or more disks

Table 21-1 SSA Identification Functions

Usage and User Feedback Functions

As you can see, we have our work cut out for us, so let's get started. The first function is the usage function. When a user input error is detected you want to give the user some feedback on how to properly use the shell script. Always create a usage function. I want to show you this function because I did something you may not know that you can do. I used a single echo command and have 15 separate lines of output. Take a look at the function in Listing 21-1 to see the method.

```
function usage
{
echo "\nUSAGE ERROR...
\nMAN PAGE ==> $SCRIPTNAME -?
\nTo Turn ALL Lights Either ON or OFF:
\nUSAGE: SSAidentify.ksh [-v] [on] [off]
EXAMPLE: SSAidentify.ksh -v on
\nWill turn ON ALL of the system's currently VARIED ON
SSA identify lights. NOTE: The default is all DEFINED SSA disks
\nTo Turn SPECIFIC LIGHTS Either ON or OFF Using EITHER
the pdisk#(s) AND/OR the hdisk#(s):
\nUSAGE: SSAidentify.ksh [on] [off] pdisk{#1} [hdisk{#2}]...
EXAMPLE: SSAidentify.ksh on hdisk36 pdisk44 pdisk47
\nWill turn ON the lights to all of the associated pdisk#(s)
that hdisk36 translates to and PDISKS pdisk44 and pdisk47.
\nNOTE: Can use all pdisks, all hdisks or BOTH hdisk
```

Listing 21-1 Usage function with a single echo command

```
and pdisk together if you want..."
exit 1
}
```

As you can see in Listing 21-1, I enclose the entire text that I want to echo to the screen within double quotes, "usage text". To place text on the next line, just press the ENTER key. If you want an extra blank line or a TAB, use one or more of the many cursor functions available with the echo command, as shown in Table 21-2.

ECHO FUNCTION	PURPOSE
\n	Inserts a new line with a carriage return
\t	Tabs over on TAB length characters for each \t entered
\b	Backs the cursor up one space for each $\begin{tabular}{lllllllllllllllllllllllllllllllllll$
\c	Leaves the cursor at the current position, without a carriage return or line feed

Table 21-2 Cursor Control Commands for the echo Command

There are many more in the man pages on your system. When incorrect usage of the shell script is detected, which you have to build into the script, the proper usage message in Listing 21-2 is displayed on the screen.

```
USAGE ERROR...
MAN PAGE ==> SSAidentify.ksh -?
To Turn ALL Lights Either ON or OFF:
USAGE: SSAidentify.ksh [-v] [on] [off]
EXAMPLE: SSAidentify.ksh -v on
Will turn ON ALL of the system's currently VARIED ON
SSA identify lights. NOTE: The default is all DEFINED SSA disks
To Turn SPECIFIC LIGHTS Either ON or OFF Using EITHER
the pdisk#(s) AND/OR the hdisk#(s):
USAGE: SSAidentify.ksh [on] [off] pdisk(#1) [hdisk(#2)]...
EXAMPLE: SSAidentify.ksh on hdisk36 pdisk44 pdisk47
Will turn ON the lights to all of the associated pdisk#(s)
```

Listing 21-2 Example of cursor control using the echo command

```
that hdisk36 translates to and PDISKS pdisk44 and pdisk47.
NOTE: Can use all pdisks, all hdisks or BOTH hdisk
and pdisk together if you want...
```

By using cursor control with the echo command, we can eliminate using a separate echo command on every separate line of text we want to display. I do the same thing in the man_page function. You can see this function in its entirety in the full shell script shown in Listing 21-9.

Before I show you the cleanup function, I want to show you the twirl function. The twirl function is used to give feedback to the user, which you saw back in Chapter 4, "Progress Indicators Using a Series of Dots, a Rotating Line, or Elapsed Time." As a brief review, the twirl function displays the appearance of a line rotating. And this is accomplished through? You guessed it, cursor control using the echo command. I like the twirl function because it is not too hard to understand and it is very short. This function works by starting an infinite while loop, which is done using the : (colon) no-op operator. A no-op does nothing and always has a zero return code, so it is perfect to create an infinite loop. The next step is to have a counter that counts only from 0 to 4. When the counter reaches 4 it is reset back to 0, zero. At each count a case statement is used to decide which of the four lines, $-, \setminus, |$, and /, is to be displayed. At the same time, the cursor is backed up so it is ready to overwrite the previous line character with a new one. There is a sleep for one second on each loop iteration. You must leave the sleep statement in the code or you will see a big load on the system by all of the continuous updates to the screen. I use this function for giving user feedback when a time-consuming job is executing. When the job is finished I kill the twirl function and move on. The easiest way to kill a background function is to capture the PID just after kicking off the background job, which is assigned to the \$! shell variable. This is similar to the way \$? is used to see the return code of the last command. The twirl function is shown in Listing 21-3.

```
function twirl
{
TCOUNT="0" # For each TCOUNT the line twirls one increment
while : # Loop forever...until you break out of the loop
do
TCOUNT=$(expr ${TCOUNT} + 1) # Increment the TCOUNT
case ${TCOUNT} in
    "1") echo '-'"\b\c"
    sleep 1
    ;;
    "2") echo '\\'"\b\c"
    sleep 1
```

```
Listing 21-3 Twirl function listing
```

```
;;
 "3") echo "|\b\c"
 sleep 1
 ;;
 "4") echo "/\b\c"
 sleep 1
 ;;
 *) TCOUNT="0";; # Reset the TCOUNT to "0", zero.
 esac
done
# End of twirl function
}
```

When I have a time-consuming job starting, I start the twirl function with the following commands:

twirl & TWIRL_PID=\$!

This leads into the next function, cleanup. In normal operation the twirl function is killed in the main body of the script, or in the function that it is called in, by using the kill command and the previously saved PID, which is pointed to by the TWIRL_PID variable. Life, though, is not always normal. In the top of the main body of the shell script we set a **trap**. The trap is used to execute one or more commands, programs, or shell scripts when a specified exit code is captured. Of course, *you cannot* trap *a* kill -9! In this shell script we execute the cleanup function on exit codes 1, 2, 3, 5, and 15. You can add more exit codes if you want. This cleanup function displays a message on the screen that a trap has occurred and runs the kill -9 \$TWIRL_PID command before exiting the shell script. If you omit the trap and the twirl function is running in the background, it will continue to run in the background! You cannot miss it — you always have a twirling line on your screen. Of course, you can kill the PID if you can find it in the process table with the ps command. The cleanup function is shown in Listing 21-4.

```
function cleanup
{
  echo "\n...Exiting on a trapped signal...EXITING STAGE LEFT...\n"
  kill -9 $TWIRL_PID
  # End of cleanup function
}
```

Listing 21-4 Cleanup function listing

When an exit code is captured the user is informed that the shell script is exiting, and then the kill command is executed on the PID saved in the \$TWIRL_PID variable.

Control Functions

Now we get into the real work of turning on and off the SSA identification lights starting with the all_defined_pdisks function. This function is the simplest of the SSA identification functions in this chapter. The goal is to get a list of every SSA disk on the system and use the pdisk# to control the identification lights by turning all lights on or off in sequence.

To understand this function you need to understand an AIX command called **lsdev** and the switches we use to extract only the pdisk information. The <code>lsdev</code> command is used to display devices in the system and the characteristics of devices. The <code>-C</code> switch tells the <code>lsdev</code> command to look at only the currently defined devices. Then the <code>-c</code> command switch is added to specify the particular *class* of device; in our case the device class is <code>pdisk</code>. So far our <code>lsdev</code> command looks like the following statement:

```
# lsdev -Cc pdisk
```

But we want to drill down a little deeper in the system. We can also specify a *subclass* to the previously defined class by adding the -s switch with our subclass ssar. We also want to have a formatted output with column headers, so we add the -H switch. These headers just help ensure that we have good separation between fields. Now we have the following command:

```
# lsdev -Cc pdisk -s ssar -H
```

When you use this command on a system with SSA disks you see an output similar to the one in Listing 21-5.

```
status location
name
                                 description
pdisk0 Available 34-08-5B91-01-P SSA160 Physical Disk Drive
pdisk1 Available 34-08-5B91-02-P SSA160 Physical Disk Drive
pdisk2 Available 34-08-5B91-03-P SSA160 Physical Disk Drive
pdisk3 Available 34-08-5B91-04-P SSA160 Physical Disk Drive
pdisk4 Available 24-08-5B91-05-P SSA160 Physical Disk Drive
pdisk5 Available 24-08-5B91-07-P SSA160 Physical Disk Drive
pdisk6 Available 24-08-5B91-06-P SSA160 Physical Disk Drive
pdisk7 Available 24-08-5B91-08-P SSA160 Physical Disk Drive
pdisk8 Available 24-08-5B91-09-P SSA160 Physical Disk Drive
pdisk9 Available 24-08-5B91-10-P SSA160 Physical Disk Drive
pdisk10 Available 24-08-5B91-11-P SSA160 Physical Disk Drive
pdisk11 Available 24-08-5B91-12-P SSA160 Physical Disk Drive
pdisk12 Available 34-08-5B91-13-P SSA160 Physical Disk Drive
```

```
Listing 21-5 lsdev listing of pdisks
```

```
pdisk13 Available 34-08-5B91-14-P SSA160 Physical Disk Drive
pdisk14 Available 34-08-5B91-16-P SSA160 Physical Disk Drive
pdisk15 Available 34-08-5B91-15-P SSA160 Physical Disk Drive
```

In Listing 21-5 we have more information than we need. The only part of this lsdev command output that we are interested in is in the first column, and only the lines that have "pdisk" in the first column. To filter this output we need to expand our lsdev command by adding awk and grep to filter the output. Our expanded command is shown here:

```
# lsdev -Cc pdisk -s ssar -H | awk '{print $1}' | grep pdisk
```

In this command statement we extract the first column using the awk statement in a pipe, while specifying the first column with the '{print \$1}' notation. Then we use grep to extract only the lines that contain the pattern pdisk. The result is a list of all currently defined pdisks on the system.

To control the identification lights for the pdisks in this list we use a for loop and use our lsdev command to create the list of pdisks with command substitution. These steps are shown in Listing 21-6.

```
function all_defined_pdisks
{
   # TURN ON/OFF ALL LIGHTS:
   # Loop through each of the system's pdisks by using the "lsdev"
   # command with the "-Cc pdisk" switch while using "awk" to extract
   # out the actual pdisk number. We will either
   # turn the identifier lights on or off, specified by the $SWITCH
   # variable:
   #
      Turn lights on: -y
     Turn lights off: -n
   #
   # as the $SWITCH value to the "ssaidentify" command, as used below..
echo "\nTurning $STATE ALL of the system's pdisks...Please Wait...\n"
for PDISK in $(1sdev -Cc pdisk -s ssar -H | awk '{print $1}' \
              grep pdisk)
do
   echo "Turning $STATE ==> $PDISK"
    ssaidentify -1 $PDISK -${SWITCH} \
   || echo "Turning $STATE $PDISK Failed"
```

Listing 21-6 all_defined_pdisks function listing

```
done
echo "\n...TASK COMPLETE...\n"
}
```

In Listing 21-6 notice the command substitution used in the for loop, which is in bold text. The command substitution produces the list arguments that are assigned to the \$PDISK variable on each loop iteration. As each pdisk is assigned, the ssaidentify command is executed using the \$PDISK definition as the target and uses the -\$SWITCH as the action to take, which can be either -y for light on or -n for light off. These values are defined in the main body of the shell script. As each light is being turned on or off the user is notified. If the action fails the user is notified of the failure also. This failure notification is done using a logical OR, specified by the double pipes, ||.

The next function is all_varied_on_pdisks. This function is different in that we must approach the task of getting a list of SSA disks to act on using completely different strategy. The result we want is the ability to control the SSA disks that are in volume groups that are currently varied-on. To get this list we must first get a list of the varied-on volume groups using the lsvg -o command. This command gives a list of varied-on volume groups directly without any added text, so we are okay with this command's output. Using this list of volume groups we can now use the lspv command to get a full listing of defined hdisks. From this list we use grep to extract the hdisks that are in currently varied-on volume groups. Notice that all of this activity so far is at the hdisk level. We need to have pdisks to control the identification lights. To build a list of hdisks to convert, we use a for loop, tagging on the volume groups with the VG variable. For each \$VG we run the following command to build a list:

lspv | grep \$VG >> \$HDISKFILE

Notice that we use a file to store this list. A file is needed because if a variable were used we might exceed the character limit for a variable, which is 2,048 on most systems. As you know, most large shops have systems with hundreds, if not thousands, of SSA disks. To be safe we use a file for storage here.

Using this list of hdisks we are going to use another for loop to translate each of the hdisks into the associated pdisk. Because we may still have a huge list containing pdisks we again use a file to hold the list. The translation takes place using the ssaxlate command, but what if some of these hdisks are not SSA disks? Well, the translation will fail! To get around this little problem we first test each translation and send all of the output to the bit bucket and check the return code of the ssaxlate command. If the return code is 0, zero, then the hdisk is an SSA disk. If the return code is nonzero, the hdisk is not an SSA disk. The result is that only pdisks are added to the new pdisk list file, which is pointed to by the PDISKFILE variable. Because this translation may take quite a while we start the twirl function, which is our progress indicator, in the background before the translation begins. As soon as the translation process ends, the twirl function is killed using the saved PID.

The only thing left to do is to perform the desired action on each of the pdisk identification lights. We do this by starting yet another for loop. This time we use command substitution to produce a list of pdisks by listing the pdisk list file with the cat command. On each loop iteration the ssaidentify command is executed for each pdisk in the list file. The all_varied_on_pdisks function is shown in Listing 21-7.

```
function all_varied_on_pdisks
trap 'kill -9 $TWIRL_PID; return 1' 1 2 3 15
cat /dev/null > $HDISKFILE
cat /dev/null > $PDISKFILE
echo "\nGathering a list of Varied on system SSA disks...
Please wait...\c"
VG_LIST=$(lsvg -o) # Get the list of Varied ON Volume Groups
for VG in $VG_LIST
do
        lspv | grep $VG >> $HDISKFILE # List of Varied ON PVs
done
twirl & # Gives the user some feedback during long processing times...
TWIRL_PID=$!
echo "\nTranslating hdisk(s) into the associated pdisk(s)
         ... Please Wait...\c"
for DISK in $(cat $HDISKFILE) # Translate hdisk# into pdisk#(s)
do
     # Checking for an SSA disk
     /usr/sbin/ssaxlate -1 $DISK # 2>/dev/null 1>/dev/null
     if (($? == 0))
     then
         /usr/sbin/ssaxlate -1 $DISK >> $PDISKFILE # Add to pdisk List
     fi
done
kill -9 $TWIRL_PID # Kill the user feedback function...
echo "\b "
               # Clean up the screen by overwriting the last character
echo "\nTurning $STATE all VARIED ON system pdisks...Please Wait...\n"
# Act on each pdisk individually...
for PDISK in $(cat $PDISKFILE)
```

Listing 21-7 all_varied_on_pdisks function listing

Notice that there is a trap at the beginning of this function in Listing 21-7. Because we are using the twirl function for user feedback we need a way to kill off the rotating line, so we added a trap inside the function. In the next step we initialized both of the list files to empty files. Then the fun starts. This is where we filter through all of the hdisks to find the ones that are in currently varied-on volume groups. With this hdisk list we loop through each of the disks looking for SSA disks. As we find each hdisk it is translated into a pdisk and added to the pdisk list. With all of the pdisks of interest found, we loop through each one and turn on/off the SSA identification lights.

The last function is list_of_disks, which acts on one or more hdisks or pdisks that are specified on the command line when the shell script is executed. In the main body of the shell script we do all of the parsing of the command-line arguments because if you tried to parse the command line inside a function the parsing would act on the *function's* argument, not the shell script's arguments. Therefore this is a short function.

In the main body of the shell script a variable, PDISKLIST, is populated with a list of pdisks. Because the user can specify hdisks or pdisks or both on the command line, the only verification that has been done is on the hdisks only, when they were translated to pdisks. We need to do a sanity check to make sure that each of the pdisks we act on has a character special file in the /dev filesystem. This is done using the **-c** switch in an if...then test. If the pdisk listed has a character special file associated with it, an attempt is made to turn the SSA identification light on/off; otherwise, the user is notified that the specified pdisk is not defined on the system. The list_of_disks function is shown in Listing 21-8.

```
function list_of_disks
```

{

```
# TURN ON/OFF INDIVDUAL LIGHTS:
# Loop through each of the disks that was passed to this script
# via the positional parameters greater than $1, i.e., $2, $3, $4...
# We first determine if each of the parameters is a pdisk or an
# hdisk. For each hdisk passed to the script we first need to
# translate the hdisk definition into a pdisk definition. This
# script has been set up to accept a combinition of hdisks and
# pdisks.
#
# We will either turn the identifier lights on or off, specified by
# the $SWITCH variable for each pdisk#:
```

Listing 21-8 list_of_disks function listing

```
#
   #
       Turn lights on: -y
   #
        Turn lights off: -n
   #
   # as the $SWITCH value to the "ssaidentify" command, as used below...
   echo "\n"
   # The disks passed to this script can be all hdisks, all pdisks,
   # or a combination of pdisks and hdisks; it just does not matter.
   # We translate each hdisk into the associated pdisk(s).
echo "\nTurning $STATE individual SSA disk lights...\n"
for PDISK in $PDISKLIST
do
     # Is it a real pdisk??
     if [ -c /dev/${PDISK} ] 2>/dev/null
     then # Yep - act on it...
          /usr/sbin/ssaidentify -1 $PDISK -${SWITCH} >/dev/null
          if (($? == 0))
          then
                /usr/bin/ssaxlate -1 $PDISK -${SWITCH}
                if ((\$? == 0))
                then
                     echo "Light on $PDISK is $STATE"
                else
                     echo "Turning $STATE $PDISK Failed"
                fi
          fi
     else
else
          echo "\nERROR: $PDISK is not a defined device on $THISHOST\n"
     fi
done
echo "\n...TASK COMPLETE...\n"
}
```

Notice in the boldface text in Listing 21-8 where we do the test to see if the pdisk listed is a real pdisk by using the -c switch in the if statement. We have covered the rest of the function, so let's move on to the main body of the shell script.

The Full Shell Script

This is a good point to show the entire shell script and go through the details at the end of the listing. The SSAidentify.ksh shell script is shown in Listing 21-9.

```
#!/bin/ksh
#
# SCRIPT: SSAidentify.ksh
# AUTHOR: Randy Michael
#
# DATE: 11/7/2007
# REV: 2.5.A
# PURPOSE: This script is used to turn on, or off, the
# identify lights on the system's SSA disks
#
# REV LIST:
#
    11/27/2007: Added code to allow the user to turn on/off
#
    individual pdisk lights
#
#
    12/10/2007: Added code to accept a combination of pdisks
    and hdisks. For each hdisk passed the script translates
#
#
    the hdisk# into the associated pdisk#(s).
#
#
    12/10/2007: Added code to ALLOW using the currently VARIED ON
    Volume Group's disks (-v switch), as opposed to ALL DEFINED
#
#
    SSA disks, which is the default behavior. Very helpful in an
#
    HACMP environment.
#
#
    12/11/2007: Added the "twirl" function to give the user feedback
#
    during long processing periods, i.e., translating a few hundred
#
    hdisks into associated pdisks. The twirl function is just a
     rotating cursor, and it twirls during the translation processing.
#
# set -n
         # Uncomment to check syntax without any execution
# set -x # Uncomment to debug this script
SCRIPTNAME=$(basename $0)
*****
function usage
```

Listing 21-9 SSAidentify.ksh shell script listing

```
echo "\nUSAGE ERROR...
\nMAN PAGE ==> $SCRIPTNAME -?
\nTo Turn ALL Lights Either ON or OFF:
\nUSAGE: SSAidentify.ksh [-v] [on] [off]
EXAMPLE: SSAidentify.ksh -v on
\nWill turn ON ALL of the system's currently VARIED ON
SSA identify lights. NOTE: The default is all DEFINED SSA disks
\nTo Turn SPECIFIC LIGHTS Either ON or OFF Using EITHER
the pdisk#(s) AND/OR the hdisk#(s):
\nUSAGE: SSAidentify.ksh [on] [off] pdisk{#1} [hdisk{#2}]...
EXAMPLE: SSAidentify.ksh on hdisk36 pdisk44 pdisk47
\nWill turn ON the lights to all of the associated pdisk#(s)
that hdisk36 translates to and PDISKS pdisk44 and pdisk47.
\nNOTE: Can use all pdisks, all hdisks or BOTH hdisk
and pdisk together if you want..."
exit 1
}
*****
function man_page
MAN_FILE="/tmp/man_file.out"
>$MAN FILE
# Text for the man page ...
echo "\n\t\tMAN PAGE FOR SSAidentify.ksh SHELL SCRIPT\n
This script is used to turn on, or off, the system's SSA disk drive
identification lights. You can use this script in the following ways:\n
To turn on/off ALL DEFINED SSA drive identification lights, ALL
VARIED-ON SSA
drive identification lights (-v switch), AN INDIVIDUAL SSA drive
identification
light or A LIST OF SSA drive identification lights.\n
SSA disk drives can be specified by EITHER the pdisk OR the hdisk, or
a COMBINATION OF BOTH. The script translates all hdisks into the
associated pdisk(s) using the system's /usr/sbin/ssaxlate command
and turns
the SSA identification light on/off using the system's /usr/sbin/
ssaidentify
command.\n
This script has four switches that control its action:\n
-? - Displays this man page.\n
on - Turns the SSA identify light(s) ON.\n
```

```
off - Turns the SSA identify light(s) OFF.\n
-v - Specifies to only act on SSA disks that are in currently varied-on
volume groups. The default action is to act on ALL DEFINED SSA disks.\n
NOTE: This switch is ignored for turning on/off individual SSA drive
lights, only valid when turning on/off ALL lights. This option is very
helpful in an HACMP environment because ALL DEFINED, the default action,
will turn on/off all of the SSA drive lights even if the SSA disk is in
a volume group that is not currently varied-on. This can be confusing
in an HA cluster.\n Using this script is very straightforward. The
following examples show the correct use of this script:\n" >> $MAN_FILE
echo "\nUSAGE: SSAidentify.ksh [-v] [on] [off] [pdisk#/hdisk#]
[pdisk#/hdisk# list]
\n\nTo Turn ALL Lights Either ON or OFF:
\nUSAGE: SSAidentify.ksh [-v] [on] [off]
\nEXAMPLE: $SCRIPTNAME on
\nWill turn ON ALL of the system's DEFINED SSA identify lights.
This is the default.
EXAMPLE: SSAidentify.ksh -v on
\nWill turn ON ALL of the system's currently VARIED-ON
SSA identify lights. OVERRIDES THE DEFAULT ACTION OF ALL DEFINED
SSA DISKS
\nTo Turn SPECIFIC LIGHTS Either ON or OFF Using EITHER
the pdisk#(s) AND/OR the hdisk#(s):
\nUSAGE: $SCRIPTNAME [on] [off] pdisk{#1} [hdisk{#2}]...
\nEXAMPLE: $SCRIPTNAME on hdisk36 pdisk44 pdisk47
\nWill turn ON the lights to all of the associated pdisk#(s)
that hdisk36 translates to and PDISKS pdisk44 and pdisk47.
\nNOTE: Can use all pdisks, all hdisks or BOTH hdisk
and pdisk together if you want...\n\n" >> $MAN_FILE
more $MAN_FILE
# End of man_page function
3
****
function cleanup
{
echo "\n...Exiting on a trapped signal...EXITING STAGE LEFT...\n"
kill $TWIRL_PID
# End of cleanup function
}
*****
```

```
function twirl
{
TCOUNT="0"
               # For each TCOUNT the line twirls one increment
while :
               # Loop forever...until you break out of the loop
do
      TCOUNT=$(expr ${TCOUNT} + 1) # Increment the TCOUNT
      case ${TCOUNT} in
          "1") echo '-'"\b\c"
                  sleep 1
                  ;;
           "2")
                  echo '\\'"\b\c"
                  sleep 1
                  ;;
           "3")
                  echo "|\b\c"
                  sleep 1
                  ;;
           "4")
                  echo "/\b\c"
                  sleep 1
                  ;;
           *)
                  TCOUNT="0" ;; # Reset the TCOUNT to "0", zero.
      esac
done
# End of twirl function
}
*****
function all_defined_pdisks
{
   # TURN ON/OFF ALL LIGHTS:
   # Loop through each of the system's pdisks by using the "lsdev"
   # command with the "-Cc pdisk" switch while using "awk" to extract
   # out the actual pdisk number. We will either
   # turn the identifier lights on or off, specified by the
   # $SWITCH variable:
   #
       Turn lights on: -y
   #
       Turn lights off: -n
   #
   # as the $SWITCH value to the "ssaidentify" command, as used below...
echo "\nTurning $STATE ALL of the system's pdisks...Please Wait...\n"
for PDISK in $(lsdev -Cc pdisk -s ssar -H | awk '{print $1}' | grep pdisk)
do
     echo "Turning $STATE ==> $PDISK"
```

```
ssaidentify -1 $PDISK -${SWITCH} || echo "Turning $STATE $PDISK
    Failed"
done
echo "\n...TASK COMPLETE...\n"
}
*****
function all_varied_on_pdisks
trap 'kill -9 $TWIRL_PID; return 1' 1 2 3 15
cat /dev/null > $HDISKFILE
echo "\nGathering a list of Varied on system SSA disks...Please
wait...\c"
VG_LIST=$(lsvg -o) # Get the list of Varied ON Volume Groups
for VG in $(echo $VG_LIST)
do
       lspv | grep $VG >> $HDISKFILE # List of Varied ON PVs
done
twirl & # Gives the user some feedback during long processing times...
TWIRL_PID=$!
echo "\nTranslating hdisk(s) into the associated pdisk(s)...Please
Wait... \c"
for DISK in $(cat $HDISKFILE) # Translate hdisk# into pdisk#(s)
do
     # Checking for an SSA disk
    /usr/sbin/ssaxlate -1 $DISK # 2>/dev/null 1>/dev/null
    if (($? == 0))
     then
          /usr/sbin/ssaxlate -1 $DISK >> $PDISKFILE # Add to pdisk List
     fi
done
kill -9 $TWIRL_PID # Kill the user feedback function...
echo "\b "
echo "\nTurning $STATE all VARIED-ON system pdisks...Please Wait...\n"
```

```
for PDISK in $(cat $PDISKFILE)
do # Act on each pdisk individually...
    echo "Turning $STATE ==> $PDISK"
    /usr/sbin/ssaidentify -1 $PDISK -${SWITCH} || echo "Turning $STATE
$PDISK Failed"
done
echo "\n\t...TASK COMPLETE...\n"
}
****
function list_of_disks
{
# TURN ON/OFF INDIVDUAL LIGHTS:
# Loop through each of the disks that was passed to this script
# via the positional parameters greater than $1, i.e., $2, $3, $4...
# We first determine if each of the parameters is a pdisk or an hdisk.
# For each hdisk passed to the script we first need to translate
# the hdisk definition into a pdisk definition. This script has
# been set up to accept a combination of hdisks and pdisks.
# We will either turn the identifier lights on or off, specified by
# the $SWITCH variable for each pdisk#:
#
    Turn lights on: -y
#
    Turn lights off: -n
# as the $SWITCH value to the "ssaidentify" command.
echo "\n"
# The disks passed to this script can be all hdisks, all pdisks
# or a combination of pdisks and hdisks; it just does not matter.
# We translate each hdisk into the associated pdisk(s).
echo "\nTurning $STATE individual SSA disk lights...\n"
for PDISK in $PDISKLIST
do
     # Is it a real pdisk??
    if [ -c /dev/${PDISK} ] 2>/dev/null
    then # Yep - act on it...
          /usr/sbin/ssaidentify -1 $PDISK -${SWITCH}
         if [ $? -eq 0 ]
```

```
then
            /usr/bin/ssaxlate -1 $PDISK -${SWITCH}
            if (($? == 0))
            then
                echo "Light on $PDISK is $STATE"
            else
                echo "Turning $STATE $PDISK Failed"
            fi
       fi
    else
       echo "\nERROR: $PDISK is not a defined device on $THISHOST\n"
    fi
done
echo "\n...TASK COMPLETE...\n"
}
*****
*****
# Set a trap...
# Remember...Cannot trap a "kill -9" !!!
trap 'cleanup; exit 1' 1 2 3 15
*****
# Check for the correct number of arguments (1)
if (($# == 0))
then
   usage
fi
*****
# See if the system has any pdisks defined before proceeding
PCOUNT=$(lsdev -Cc pdisk -s ssar | grep -c pdisk)
if ((PCOUNT == 0))
then
   echo "\nERROR: This system has no SSA disks defined\n"
   echo "\t\t...EXITING...\n"
   exit 1
fi
```

```
# Make sure that the ssaidentify program is
# executable on this system...
if [ ! -x /usr/sbin/ssaidentify ]
then
   echo "\nERROR: /usr/sbin/ssaidentify is NOT an executable"
   echo "program on $THISHOST"
   echo "\n...EXITING...\n"
   exit 1
fi
*****
# Make sure that the ssaxlate program is
# executable on this system...
if [ ! -x /usr/sbin/ssaxlate ]
then
   echo "\nERROR: /usr/sbin/ssaxlate is NOT an executable"
   echo "program on $THISHOST"
   echo "\n...EXITING...\n"
   exit 1
fi
*****
****
#
# Okay, we should have valid data at this point
# Let's do a light show.
*****
*****
# Always use the UPPERCASED value for the $STATE, $MODE,
# and $PASSED variables...
typeset -u MODE
MODE= "DEFINED_DISKS"
typeset -u STATE
STATE=UNKNOWN
typeset -u PASSED
# Use lowercase for the argument list
typeset -1 ARGUMENT
```

```
# Grab the system hostname
THISHOST=$(hostname)
# Define the hdisk and pdisk FILES
HDISKFILE="/tmp/disklist.out"
>$HDISKFILE
PDISKFILE="/tmp/pdisklist.identify"
>$PDISKFILE
# Define the hdisk and pdisk list VARIABLES
HDISKLIST=
PDISKLIST=
# Use getopts to parse the command-line arguments
while getopts ":vV" ARGUMENT 2>/dev/null
do
     case $ARGUMENT in
                  MODE="VARIED ON"
          v V)
                 ;;
          \?)
                 man_page
                  ;;
     esac
done
****
# Decide if we are to turn the lights on or off...
(echo $@ | grep -i -w on >/dev/null) && STATE=ON
(echo $@ | grep -i -w off >/dev/null) && STATE=OFF
case SSTATE in
ON)
 # Turn all of the lights ON...
    SWITCH="y"
    ;;
OFF)
    # Turn all of the lights OFF...
    SWITCH="n"
    ;;
*)
```

```
# Unknown Option...
   echo "\nERROR: Please indicate the action to turn lights ON or OFF\n"
    usage
    exit 1
    ;;
esac
*****
*****
*****
*****
if (($# == 1)) && [[ $MODE = "DEFINED_DISKS" ]]
then
     # This function will turn all lights on/off
    all_defined_pdisks
elif [[ $MODE = "VARIED_ON" ]] && (($# = 2))
then
     # This function will turn on/off SSA disk lights
     # in currently varied-on volume groups only
    all_varied_on_pdisks
# Now check for hdisk and pdisk arguments
elif [ $MODE = DEFINED_DISKS ] && (echo $@ | grep disk >/dev/null) \
                          && (($# >= 2))
then
     # If we are here we must have a list of hdisks
     # and/or pdisks
     # Look for hdisks and pdisks in the command-line arguments
    for DISK in $(echo $@ | grep disk)
     do
        case $DISK in
        hdisk*) HDISKLIST="$HDISKLIST $DISK"
               ;;
        pdisk*) PDISKLIST="$PDISKLIST $DISK"
               ;;
         *)
               : # No-Op - Do nothing
               ;;
         esac
     done
```

Listing 21-9 (continued)

```
if [[ ! -z "$HDISKLIST" ]] # Check for hdisks to convert to pdisks
     then
     # We have some hdisks that need to be converted to pdisks
     # so start converting the hdisks to pdisks
           # Give the user some feedback
           echo "\nConverting hdisks to pdisk definitions"
           echo "\n
                     ...Please be patient...\n"
           # Start converting the hdisks to pdisks
           for HDISK in $HDISKLIST
           do
               PDISK=$(ssaxlate -1 $HDISK)
               if (($? == 0))
               then
                     echo "$HDISK translates to ${PDISK}"
               else
                    echo "ERROR: hdisk to pdisk translation FAILED for
$HDISK"
               fi
               # Build a list of pdisks
                                        # Add pdisk to the pdisk list
               PDISKLIST="$PDISKLIST $PDISK"
            done
     fi
     if [[ -z "$PDISKLIST" ]]
     then
          echo "\nERROR: You must specify at least one hdisk or pdisk\n"
          man_page
          exit 1
     else
           # Turn on/off the SSA identification lights
           list_of_disks
     fi
fi
*****
          END OF SCRIPT
#
*****
```

Let's start at the BEGINNING OF MAIN in Listing 21-9. The very first thing that we do is set a trap. This trap is set for exit codes 1, 2, 3, 5, and 15. On any of these captured signals the cleanup function is executed, and then the shell script exits with a return code of 1. It is nice to be able to clean up before the shell script just exits.

In the next series of tests we first make sure that there is at least one argument present on the command line. If no arguments are given, the script presents the usage function, which displays proper usage and exits. If we pass the argument test then I thought it would be a good idea to see if the system has any SSA disks defined on the system. For this step we use the PCOUNT=\$(lsdev -Cc pdisk -s ssar | grep -c pdisk). The grep -c returns the count of SSA disks found on the system and assigns the value to the PCOUNT variable. If the value is zero there are no SSA disks, so inform the user and exit. If we do have some SSA disks, the next thing we do is make sure that the ssaidentify and ssaxlate commands exist and are executable on this system. At this point we know we are in an SSA environment, so we define and initialize all of the script's variables.

Then we get to use the getopts function to parse the command-line arguments. We expect and recognize just two arguments, -v and -V, to specify varied-on volume groups only. Any other argument, specified by a preceding hyphen, -, displays the man_page function. Anything else on the command line is ignored by the getopts function, which is a shell built-in function.

On the command line we *must* have either on or off present, or we do not have enough information to do anything. We check the command-line arguments by echoing out the full list and grepping for on and off. At the next case statement the \$STATE variable is tested. If on or off was not found, the usage function is displayed and the script exits. If we get past this point we know that we have the minimal data to do some work.

When we start playing with the lights we have to do some tests to decide what action we need to take and on what set of SSA disks. The first one is simple. If we have only one command-line argument and it is either on or off, then we know to turn on or off all defined SSA disk-identification lights on the system without regard to volume group status. So, here we run the all_defined_pdisks function. If we have two arguments on the command line and one of them is -v or -V, we know to act only on SSA disks in currently varied-on volume groups by turning every one of the SSA identification lights on or off.

The last option is to have hdisks or pdisks listed on the command line. For this option we know to act on only the disks that the user specified and to turn on or off only these disks. Because we allow both hdisks and pdisks we need to convert everything to pdisk definitions before we call the <code>list_of_disks</code> function. To do this we echo the entire list of command-line arguments and grep for the word <code>disk</code>. Using this list in a case statement, we can detect the presence of an hdisk or a pdisk. For each one found it is added to either the HDISKLIST or the PDISKLIST variables. After the test we check to see if the HDISKLIST variable has anything assigned, which means that the variable is not null. If there are entries, we convert each hdisk to its

associated pdisk and build up the pdisk list in the PDISKLIST variable. When the list is complete, and it is not an empty list, we call the list_of_disks function. That is it for this shell script.

Other Things to Consider

I cannot always fit all of the options into a chapter, and this chapter is no exception. Here are a few things to consider to modify this shell script.

Error Log

When I created this shell script it was for a personal need because I have so many SSA disk trays. For my purposes I did not need an error log, but you may find one necessary. In the places that I sent everything to the bit bucket, especially standard error or file descriptor 2, redirect this output to append to an error log. This may help you find something in the system that you missed.

Cross-Reference

Because it is rare for the hdisk and pdisk associations to match by numbers, you may find that a shell script to cross-reference the numbers is beneficial. You should be able to knock this out in about one hour. Look through the code where I first test the hdisk to see if it is an SSA disk and then do the translation. Using these few lines of code you can build a nice little cross-reference sheet for your staff.

Root Access and sudo

Both of the SSA commands need root privilege to execute. If your systems have strict root access rules you may just want to define this shell script in your /etc/sudoers file. Please *never* edit this file directly! There is a special wrapper program around the **vi** editor in the /usr/local/sbin directory called **visudo**. This command starts a vi session and opens up the /etc/sudoers file automatically. When you are finished editing and save the file, this program checks the /etc/sudoers file for errors.

Summary

In this chapter we learned a few new things about controlling the SSA subsystem on an AIX machine. These principles apply to any other UNIX system utilizing SSA. As always, there are many different ways to write a shell script, and some are lean and mean with no comments. I like to make the shell scripts easier to understand and maintain. But I do have a few things that you may want to consider. I hope you learned something in this chapter. In the next chapter we will look at techniques of pinging hosts and notifying staff if any hosts are not reachable. See you in the next chapter!

Lab Assignment

1. Add a command-line switch, -c, to the **SSAidentify.ksh** script in Listing 21-9 that will execute a new function to produce a cross-reference that matches pdisks to hdisks. This information should be both displayed and saved in a file.

CHAPTER 22

Automated Hosts Pinging with Notification of Failure

In every shop there is a critical need to keep the servers serving. For system availability, the quicker you know that a system is unreachable, the quicker you can act to resolve the problem and reduce company losses. At the lowest level of system access, we can **ping** each machine in the "critical machine" list. If the ping works, it will tell you if the network adapter is working, but it does not guarantee that the machine is booted and applications are working. For this level of checks you need to actually access the application or operating system.

In this chapter we are going to create a shell script that will ping hosts using a list of machines, which is stored in a separate file that is easily edited. Other options to this scenario include pinging all the machines in the /etc/hosts file, using **ftp** to transfer a file, querying a database, or interacting with an application, to name a few. Our interest in this chapter is to work at the lowest level and use the ping command to ensure that the machines are reachable from the network. When a machine is found unreachable, we send notification to alert staff that the machine is down. Due to the fact that in some shops the network can become saturated with network traffic, we are going to add an extra level of testing on a failed ping test, which we will get into later in this chapter. But before we go any further let's look at the command syntax for each of our operating systems (AIX, HP-UX, Linux, OpenBSD, and Solaris) to see if we can find a command syntax that will produce the same output for all of the operating systems that we are working with.

Syntax

As always, we need the correct command syntax before we can write a shell script. Our goal is to find the command syntax that produces the same output for each operating system. For this shell script we want to ping each host multiple times to ensure that the

node is reachable; the default is three pings. The standard output we want to produce on each OS is shown here:

```
# ping -c3 dino
PING dino: (10.10.10.4): 56 data bytes
64 bytes from 10.10.10.4: icmp_seq=0 ttl=255 time=2 ms
64 bytes from 10.10.10.4: icmp_seq=1 ttl=255 time=1 ms
64 bytes from 10.10.10.4: icmp_seq=2 ttl=255 time=1 ms
----dino PING Statistics----
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 1/1/2 ms
```

This is the command, and the output is from an AIX machine. Notice the PING Statistics at the bottom of the command output, where I have highlighted 3 packets received. This is the output line that we are interested in for every operating system. Now, how do we produce the same output for each OS? Instead of leaving you in the dark I am just going to list each one in Table 22-1, showing you how to ping the host dino.

In Table 22-1 notice that AIX, Linux, and OpenBSD have the same command syntax. For HP-UX and Solaris notice the two numbers, 56 and 3. The 56 specifies the packet size to send on each ping, and the 3 is the number of times to try to reach the host. For a packet size, 56 is a standard packet, and we are not going to change from this standard. It is important to know the differences in command structure for each operating system because we are creating one shell script and we will ping each node using a function, which selects the correct command to execute based on the UNIX flavor. To find the OS, we use the uname command. By using the output of the uname command in a case statement, we are assured that the correct command is executed on any of the five operating systems. This is really all we have for the syntax, but we need to do some checks and create some variables, so we are going to build the shell script around these commands, listed in Table 22-1.

OPERATING SYSTEM	PING COMMAND
AIX	# ping -c3 dino
HP-UX	# ping dino 56 3
Linux	# ping -c3 dino
OpenBSD	# ping -c3 dino
Solaris	# ping -s dino 56 3

Table 22-1 Ping Command for Each Operating System

Creating the Shell Script

In scripting this solution we want to add a couple of options for convenience. The first option is to have a means of stopping the pinging without interrupting the scheduled script execution, which is usually executed through a cron table entry. The second option is to have a means of stopping the *notification* for nodes that are unreachable. For each of these we can use a flag variable that must have a value of TRUE to enable the option. There are many times that we want to disable these two options, but the main reason is during a maintenance window when many of the machines are unreachable at the same time. If you have only one or two machines that are down, then commenting out the node name(s) in the ping.list file, which contains a list of nodes to ping, is preferable. You can also comment out the cron table entry to disable the monitoring altogether.

Now we need to define the pinging technique that we want to use. I like to use a two-level approach in checking for a system's reachability. In a two-level testing scenario, when a node is unreachable we go to sleep for a few seconds and try the test again. We do this to eliminate "false positives" due to a heavy network load. This is a major concern at some shops where I have worked, and finger-pointing back and forth between the network team and the Systems Administrators always happens, so I try to stay out of this argument. This second-level test adds just a few seconds to the testing window for each unreachable node. This is a relatively simple shell script to create, so keep reading!

Define the Variables

The first thing that we want to do in almost any shell script is to define the variables and files that are used in the script. We have already discussed two variables, which enable pinging and notification. For pinging we use the PINGHOSTS variable, and MAILOUT as the variable to permit or disable notification. Additionally, for ease of testing we are going to **typeset** both of these variables to force all text assignments to these variables to uppercase, as shown here:

```
typeset -u PINGHOSTS
typeset -u MAILOUT
PINGHOSTS=TRUE
MAILOUT=TRUE
```

We can also typeset the variables and assign the values in the same step, as shown here:

typeset -u PINGHOSTS=true typeset -u MAILOUT=true Notice that I assign a lowercase true to both variables, but when you print or test the variables you will see that the assignments have been changed to uppercase characters:

echo \$MAILOUT TRUE

There are a few more variables that we also need to define, including PING_COUNT and PACKET_SIZE, that specify the number of times to ping the target host and the packet size for each packet, which we discussed earlier.

```
integer PING_COUNT=3
integer PACKET_SIZE=56
```

Notice the *integer* notation used to define these variables as integers. This notation produces the exact same results that the typeset -i command produces.

Next we need the UNIX flavor that this shell script is running. This shell script recognizes AIX, HP-UX, Linux, OpenBSD, and Solaris. For this step we use the uname command, as shown here:

UNAME=\$(uname)

In this UNAME assignment we used command substitution to assign the result of the uname command to the variable UNAME.

The next two steps in this definition section involve defining the PINGFILE and MAILFILE file assignments. The PINGFILE contains a list of nodes that we want to ping. The shell script is expecting one node, or hostname, per line as shown here:

```
yogi
booboo
bamBam
mrranger
fred
```

If you no longer want to ping a node in the list file, you can comment the host out using a hash mark (#). For this shell script I specified that the ping list is located in /usr/local/bin/ping.list.

Similarly, the MAILFILE has a list of email addresses that are to be notified when a node is not reachable. We can also set up a sendmail alias. Our email list is located in /usr/local/bin/mail.list. The variable assignments are shown here:

```
PINGFILE="/usr/local/bin/ping.list" # List of nodes to ping
MAILFILE="/usr/local/bin/mail.list" # List of persons to notify
```

For these two files we are going to check for a nonzero length file, which implies the file exists and its size is greater than zero bytes. If the *\$PINGFILE* does not exist,

we need to send an ERROR message to the user and exit the shell script because we do not have a list of nodes to ping. If the \$MAILFILE does not exist, we are just going to display a message to notify the user that there will not be any email notification sent for unreachable nodes.

We also need a file to hold the data that is emailed out when a node is unreachable. The file is located in /tmp/pingfile.out and is assigned to the PING_OUTFILE variable:

PING_OUTFILE="/tmp/pingfile.out" # File for emailed notification
>\$PING_OUTFILE # Initialize to an empty file

Notice how we created an empty file by redirecting nothing to the file, which is pointed to by the *\$PING_OUTFILE* variable. You could also use cat /dev/null to accomplish the same task, as shown here:

```
cat /dev/null > $PING_OUTFILE
```

Next we need three variables that are to hold numeric values — at least we hope they are numeric. The first variable is called INTERVAL, and it contains a value specifying the number of seconds to sleep before trying to ping an unreachable node for the second time. I like to use three seconds:

integer INTERVAL="3" # Number of seconds to sleep between retries

As we discussed before, in our standard ping output we are interested in the PING Statistics line of output. Specifically, we want to extract the numeric value for the "3 packets received", which should be greater than zero if the node is reachable. To hold the value for the number of pings received back, we need two variables, one for the first try and one for the second attempt, in case the node is unreachable the first time. These two variables are PINGSTAT and PINGSTAT2 and are initialized to NULL, as shown here:

PINGSTAT= # Number of pings received back from pinging a node PINGSTAT2= # Number of pings received back on the second try

The last variable we need to assign is the hostname of the machine that is running this script. We need the hostname because we may have two hosts pinging each node in case one pinging node fails. For this variable we again use command substitution, as shown here:

THISHOST=`hostname` # The hostname of this machine

Notice that this time we used the back tics (`command`) instead of the dollar-doubleparentheses method (\$(command)) for command substitution. Both commandsubstitution options produce the same result, which is yogi on this machine.

Creating a Trap

To start out our shell script we are going to set a **trap**, which allows us to take some kind of action when an exit signal is captured, such as a user pressing Ctrl+C. We can capture most exit signals *except* for kill -9. The only *action* that we want to take in this shell script is to inform the user that the shell script has detected an exit signal and the script is exiting. This trap is added in this shell script so that you get used to putting traps in all of your shell scripts. We are going to capture exit signals 1, 2, 3, 5, and 15 only. You can add many more, but it is overkill in this case. For a complete list of exit signals, use the kill -1 (lowercase L) command. The command syntax for the trap is shown here:

```
trap 'echo "\nExiting on a trapped signal...\n";exit 1' 1 2 3 5 15
```

Using this trap command statement, the following message is displayed before the shell script exits with exit signal 1.

```
Exiting on a trapped signal...
```

The Whole Shell Script

We have all of the initializations complete and know what the ping syntax is for each operating system, so let's look at the whole shell script and cover some other issues at the end of Listing 22-1. Pay close attention to the boldface text.

Listing 22-1 pingnodes.ksh shell script

```
trap 'echo "\n\nExiting on trapped signal...\n" \
       ;exit 1' 1 2 3 15
****
# Define and initialize variables here...
PING_COUNT="3" # The number of times to ping each node
PACKET_SIZE="56"
                  # Packet size of each ping
typeset -u PINGNODES # Always use the UPPERCASE value for $PINGNODES
PINGNODES="TRUE" # To enable or disable pinging FROM this node - "TRUE"
typeset -u MAILOUT # Always use the UPPERCASE value for $MAILOUT
MAILOUT="TRUE"
                  # TRUE enables outbound mail notification of events
UNAME=$(uname)
                  # Get the Unix flavor of this machine
PINGFILE="/usr/local/bin/ping.list" # List of nodes to ping
if [ -s $PINGFILE ]
then
     # Ping all nodes in the list that are not commented out and blank
     PINGLIST=$(grep -v '^#' $PINGFILE)
else
     echo "\nERROR: Missing file - $PINGFILE"
     echo "\nList of nodes to ping is unknown...EXITING...\n"
     exit 2
fi
MAILFILE="/usr/local/bin/mail.list" # List of persons to notify
if [ -s $MAILFILE ]
then
       # Ping all nodes in the list that are not commented out and blank
       MAILLIST=$(cat $MAILFILE | egrep -v '^#')
else
       echo "\nERROR: Missing file - $MAILFILE"
       echo "\nList of persons to notify is unknown...\n"
       echo "No one will be notified of unreachable nodes...\n"
fi
PING_OUTFILE="/tmp/pingfile.out" # File for emailed notification
>$PING_OUTFILE # Initialize to an empty file
integer INTERVAL="3" # Number of seconds to sleep between retries
# Initialize the next two variables to NULL
```

```
PINGSTAT= # Number of pings received back from pinging a node
PINGSTAT2= # Number of pings received back on the second try
THISHOST=`hostname`
                   # The hostname of this machine
****
**********
function ping_host
# This function pings a single node based on the Unix flavor
# set -x # Uncomment to debug this function
# set -n # Uncomment to check the syntax without any execution
# Look for exactly one argument, the host to ping
if (( $# != 1 ))
then
    echo "\nERROR: Incorrect number of arguments - $#"
    echo " Expecting exactly one argument\n"
    echo "\t...EXITING...\n"
    exit 1
fi
HOST=$1 # Grab the host to ping from ARG1.
# This next case statement executes the correct ping
# command based on the Unix flavor
case $UNAME in
AIX OpenBSD Linux)
         ping -c${PING_COUNT} $HOST 2>/dev/null
         ;;
HP-UX)
         ping $HOST $PACKET_SIZE $PING_COUNT 2>/dev/null
         ;;
SunOS)
         ping -s $HOST $PACKET_SIZE $PING_COUNT 2>/dev/null
         ;;
*)
         echo "\nERROR: Unsupported Operating System - $(uname)"
         echo "\n\t...EXITING...\n"
```

```
Listing 22-1 (continued)
```

```
exit 1
esac
}
*****
function ping_nodes
*****
# Ping the other systems check
# This can be disabled if you do not want every node to be pinging all
# of the other nodes. It is not necessary for all nodes to ping all
# other nodes, although you do want more than one node doing the pinging
# just in case the pinging node is down. To activate pinging the
# "$PINGNODES" variable must be set to "TRUE". Any other value will
# disable pinging from this node.
# set -x # Uncomment to debug this function
# set -n # Uncomment to check command syntax without any execution
if [[ $PINGNODES = "TRUE" ]]
then
    echo # Add a single line to the output
    # Loop through each node in the $PINGLIST
    for HOSTPINGING in $PINGLIST # Spaces between nodes in the
                               # list are assumed
    do
         # Inform the user what is going on
         echo "Pinging --> ${HOSTPINGING}...\c"
         # If the pings received back are equal to "0" then you have a
         # problem.
         # Ping $PING_COUNT times, extract the value for the pings
         # received back.
         PINGSTAT=$(ping_host $HOSTPINGING | grep transmitted \
                   awk '{print $4}')
         # If the value of $PINGSTAT is NULL, then the node is
         # unknown to this host
```

```
if [[ -z "$PINGSTAT" && "$PINGSTAT" = '' ]]
         then
              echo "Unknown host"
              continue
         fi
         if (( PINGSTAT == 0 ))
         then # Let's do it again to make sure it really is unreachable
              echo "Unreachable...Trying one more time...\c"
              sleep $INTERVAL
              PINGSTAT2=$(ping_host $HOSTPINGING | grep transmitted \
                        | awk '{print $4}')
              if (( PINGSTAT2 == 0 ))
              then # It REALLY IS unreachable...Notify!!
                   echo "Unreachable"
                   echo "Unable to ping $HOSTPINGING from $THISHOST" \
                         tee -a $PING_OUTFILE
              else
                   echo "OK"
              fi
         else
              echo "OK"
         fi
    done
fi
}
****
function send_notification
{
if [ -s $PING_OUTFILE -a "$MAILOUT" = "TRUE" ];
then
       case $UNAME in
       AIX | HP-UX | Linux | OpenBSD) SENDMAIL= "/usr/sbin/sendmail"
       ;;
       SunOS) SENDMAIL="/usr/lib/sendmail"
       ;;
       esac
       echo "\nSending email notification"
       $SENDMAIL -f randy@$THISHOST $MAILLIST < $PING_OUTFILE
fi
}
```

Listing 22-1 (continued)

Now we get to the fun stuff! Let's start out with the three functions, because they do all of the work anyway. The first function is ping_host. The idea here is to set up a case statement, and based on the response from the uname command, which was assigned to the UNAME variable in the definitions section, we execute the specific ping command for the particular UNIX flavor. If an unlisted UNIX flavor is found, an ERROR message is given to the user, and this shell script exits with a return code 1. We must do this because we have no idea what the correct syntax for a ping command should be for an unknown operating system.

The ping_host function is called from the ping_nodes function on every loop iteration. Inside the ping_nodes function we first ensure that the \$PINGNODES variable is set to TRUE; otherwise, the pinging of nodes is disabled.

We use the *\$PINGFILE* file to load a variable, *PINGLIST*, with a list of nodes that we want to *ping*. This extra step is done to give the user the ability to comment out specific node(s) in the *\$PINGFILE*. Without this ability you would leave the user in a state of annoyance for all of the notifications because of a single node being down for a period of time. The command to strip out the commented lines and leave the remaining nodes in the list is shown here:

```
PINGLIST=$(grep -v '^#' $PINGFILE)
```

Notice how this command substitution works. We use grep with the -v switch. The -v switch tells grep to list everything *except* for the following pattern, which is '^#' in this case. Now let's look at the ^# part. When we put a caret character (^) in front of a pattern in this grep statement, we are ignoring any line that *begins with* a hash mark (#) in the \$PINGFILE. The caret (^) means *begins with*.

A for loop is started using the \$PINGLIST variable as a list, which contains each host in the /usr/local/bin/ping.list file that is *not* commented out. For each host in the listing we echo to the screen the target hostname and call the ping_host function inside of a command-substitution statement on each loop iteration, which is shown here:

```
echo "Pinging --> ${HOSTPINGING}...\c"
PINGSTAT=$(ping_host $HOSTPINGING | grep transmitted | awk '{print $4}')
```

For each node in the \$PINGLIST the echo statement and the command-substitution statement are executed. There are three possible results for the command-substitution statement, and we test for two; the last one is assumed. (1) The PINGSTAT value is 0, zero. If the packets received are 0, zero, then we sleep for \$INTERVAL seconds and try to reach the node again, this time assigning the packets received to the PINGSTAT2 variable. (2) The PINGSTAT value is NULL. This results when you try to ping a node that is unknown to the system. In this case we echo to the screen Unknown host and continue to the next node in the list. (3) The PINGSTAT value is nonzero and non-NULL, which means that the ping was successful. Please study each of these tests in the ping_nodes function.

Notice the tests used in the *if* statements. Each of these is a mathematical test, so we use the double parentheses method of testing, as shown here:

if ((PINGSTAT == 0))

There are two things to notice in this if statement. The first is that there is no dollar sign (\$) in front of the PINGSTAT variable. The dollar sign is not needed in a mathematical test when using the double parentheses method because the shell assumes that any nonnumeric string is a variable for this type of mathematical test. The second point I want to make in the preceding if statement is the use of the double equal signs (==). Using this type of mathematical test, a single equal sign is an *assignment*, not an equality test. This sounds a little strange, but you can actually assign a value to a variable in a test. To test for equality in the ((math test)) method, always use double equal signs (==) with this test method.

The last function in this shell script is the send_notification function. This function is used to send an email notification to each address listed in the /usr/local/bin/ mail.list file, which is pointed to by the MAILFILE variable. Before attempting any notification, the function tests to see if the \$PING_OUTFILE file has anything in it or if its size is greater than zero bytes. The second test is to ensure that the MAILOUT variable is set to TRUE. If the \$PING_OUTFILE has some data and the MAILOUT variable is set to TRUE, the function will attempt to notify each email address in the \$MAILFILE.

In the send_notification function, notice that I am using the sendmail command, as opposed to the mail or mailx commands. I use the sendmail command because I worked at a shop where I had a lot of trouble getting mail through the firewall because I was sending the mail as root. You need some valid email address, and I found a solution by using the sendmail command because I can specify a valid user who *sent* the email. The command I used is shown here:

```
sendmail -f randy@$THISHOST $MAILLIST < $PING_OUTFILE</pre>
```

In this statement the -f <user@THISHOST> specifies who is sending the email. We can also use the following mailx command syntax to accomplish the same task:

mailx -r randy@\$THISHOST \$MAILLIST < \$PING_OUTFILE</pre>

In this statement, the -r <user@THISHOST> specifies who is sending the email. The \$MAILLIST is the list of persons who should receive the email, and the < \$PING_OUTFILE input redirection is the body text of the email, which is stored in

a file. I still have one little problem, though. The sendmail command is not always located in the same directory, and sometimes it is not in the \$PATH. On AIX, HP-UX, Linux, and OpenBSD, the sendmail command is located in /usr/sbin. On Solaris the sendmail command is located in the /usr/lib directory. To get around this little problem we need a little case statement that utilizes the \$UNAME variable that we used in the ping_host function. With a little modification we have the function shown in Listing 22-2.

```
function send_notification
{
    if [ -s $PING_OUTFILE -a "$MAILOUT" = "TRUE" ];
    then
        case $UNAME in
        AIX|HP-UX|Linux) SENDMAIL="/usr/sbin/sendmail"
        ;;
        SunOS) SENDMAIL="/usr/lib/sendmail"
        ;;
        esac
        echo "\nSending email notification"
        $SENDMAIL -f randy@$THISHOST $MAILLIST < $PING_OUTFILE
fi
}</pre>
```

Listing 22-2 send_notification function listing

Notice that we used a single line for AIX, HP-UX, Linux, and OpenBSD in the case statement. At the end of the function we use the \$SENDMAIL variable to point to the correct full path of the sendmail command for the specific operating system.

Let's not forget to look at the pingnodes.ksh shell script in action! In the following output, shown in Listing 22-3, the node dino is unknown to the system, and the mrranger node is powered down, so there is no response from the ping to the system.

```
# ./pinghostfile.ksh.new
Pinging --> yogi...0K
Pinging --> bambam...0K
Pinging --> booboo...0K
Pinging --> dino...Unknown host
Pinging --> wilma...0K
Pinging --> mrranger...Unreachable...Trying one more time...Unreachable
Unable to ping mrranger from yogi
Sending email notification
```

Listing 22-3 pingnodes.ksh shell script in action

From the output in Listing 22-3, notice the result of pinging the node dino. I commented out the hostname dino in the /etc/hosts file. By doing so I made the node unknown to the system because DNS is not configured on this system. The mrranger node is powered down, so it is known but not reachable. Notice the difference in the outputs for these two similar, but very different, situations. Please study the code related to both of these tests in the ping_nodes function.

Other Options to Consider

As always, we can improve on any shell script, and this one is no exception. I have listed some options that you may want to consider.

\$PINGLIST Variable-Length-Limit Problem

In this scripting solution we gave the user the capability to comment out specific nodes in the <code>\$PINGFILE</code>. We assigned the list of nodes, which is a list without the comments, to a variable. This is fine for a relatively short list of nodes, but a problem arises when the maximum variable length, which is usually 2,048 bytes, is exceeded. If you have a long list of nodes that you want to ping and you notice that the script never gets to the end of the ping list, you have a problem. Or if you see a funny-looking node name, which is probably a hostname that has been cut off by the variable limit and associated with a system error message, you have a problem. To resolve this issue, define a new file to point to the <code>PINGLIST</code> variable, and then we will use the *file* to store the ping list data instead of a *variable*. To use <code>PINGLIST</code> as a file, add/change the following lines.

Add this line:

PINGLIST=/tmp/pinglist.out

Change this line:

PINGLIST=\$(cat \$PINGFILE | grep -v '^#')

to this line:

cat \$PINGFILE | grep -v '^#' > \$PINGLIST

Change this line:

for HOSTPINGING in \$(echo \$PINGLIST)

to this line:

for HOSTPINGING in \$(cat \$PINGLIST)

Using the file to store the ping list data removes the variable assignment limit. This modified shell script is located on this book's companion web site (www.wiley.com /go/michael2e). The script name is pingnodes_using_a_file.ksh.

Ping the /etc/hosts File Instead of a List File

This may be overkill for any large shop, but it is easy to modify the shell script to accomplish this task. You want to make the following change to the shell script after completing the tasks in the previous section, "\$PINGLIST Variable-Length-Limit Problem," to the shell script shown in Listing 22-1.

Change these lines:

```
if [ -s $PINGFILE ]
then
PINGLIST=$(cat $PINGFILE | grep -v '^#')
```

to these lines:

In this changed code we cat the /etc/hosts file and pipe the output to a sed statement, sed /^#/d. This sed statement removes every line in the /etc/hosts file that begins with a pound sign (#). The output of this sed statement is then piped to another sed statement, sed /^\$/d, which removes all of the blank lines in the /etc/hosts file (the blank lines are specified by the ^\$). This sed output is sent to a grep command that removes the loopback address (127.0.0.1) from the list. Finally, the remaining output is piped to an awk statement that extracts the hostname out of the second field. The resulting output is redirected to the \$PINGLIST file.

Before you sed purists gig me on using pipes, a more proper sed statement has the form:

Notice that we removed the cat /etc/hosts | and just added the filename as the last argument to the first sed command.

The modified shell script to ping the /etc/hosts file is included on the web site that accompanies the book (www.wiley.com/go/michael2e). Just download the script and function package. The filename we want to modify is pinghostsfile.ksh.

Logging

I have not added any logging capability to this shell script. Adding a log file, in addition to user notification, can help you find trends of when nodes are unreachable. Adding a log file is not too difficult to do. The first step is to define a unique log filename in the definitions section and assign the filename to a variable, maybe LOGFILE. In the script test for the existence of the file, using a test similar to the following statement will work.

```
Add these lines:
```

```
LOGPATH=/usr/local/log
LOGFILE=${LOGPATH}/pingnodes.log
if [ ! -s $LOGFILE ]
then
     if [ ! -d $LOGPATH ]
     then
          echo "\nCreating directory ==> $LOGPATH\c"
          mkdir /usr/local/log
          if (( $? != 0 ))
          then
            echo "\nUnable to create the $LOGPATH directory...EXITING \n"
             exit 1
          fi
          chown $USER /usr/local/log
          chmod 755 $LOGPATH
          echo
     fi
     echo "\nCreating Logfile ==> $LOGFILE\c"
     cp /dev/null > $LOGFILE
     chown $USER $LOGFILE
     echo
fi
```

After adding these lines of code, use the tee -a \$LOGFILE command in a pipe to both display the text on the screen and log the data in the \$LOGFILE.

Notification of "Unknown Host"

You may want to add notification, and maybe logging too, for nodes that are not known to the system. This usually occurs when the machine cannot resolve the node name into an IP address. This can be caused by the node not being listed in the /etc/hosts file, failure of the DNS lookup, or a typo in a hostname definition. You can check DNS using the nslookup command. Check all three conditions when you get the Unknown host message. Currently, this shell script only echoes this information to the screen. You may want to add this message to the notification.

Notification Method

In this shell script we use email notification. I like email notification, but if you have a network failure this is not going to help you. To get around the "network down" problem with email, you may want to set up a modem, for *dial-out* only, to dial your alphanumeric pager number and leave you a message. At least you will always get the message. I have had times, though, when I received the message two hours later due to a message overflow to the modem.

You may just want to change the notification to another method, such as SNMP traps. If you execute this shell script from an enterprise management tool, the response

required back to the program is usually an SNMP trap. Refer to the documentation of the program you are using for details.

Automated Execution Using a Cron Table Entry

I know you do not want to execute this shell script from the command line every 15 minutes yourself! I use a root cron table entry to execute this shell script every 15 minutes, 24 hours a day, Monday through Saturday, and 8:00 a.m. to midnight on Sunday; of course, this requires two cron table entries. Because weekly backups and reboots happen early Sunday morning, I do not want to be awakened every Sunday morning when a machine reboots, so I have a special cron entry for Sunday. Both root cron table entries shown execute this script every 15 minutes:

```
5,20,35,50 * * * 1-6 /usr/local/bin/pingnodes.ksh >/dev/null 2>&1
5,20,35,50 8-23 * * 0 /usr/local/bin/pingnodes.ksh </dev/null 2>&1
```

The first entry executes the pingnodes.ksh shell script at 5, 20, 35, and 50 minutes of every hour from Monday through Saturday. The second entry executes the pingnodes.ksh shell script at 5, 20, 35, and 50 minutes from 8:00 a.m. until 11:59 p.m., with the last ping test running at 11:50 p.m. Sunday.

Summary

In this chapter we took a different approach than we have in some other shell scripts in this book. Instead of creating a different function for each operating system, we created a single shell script and then used a separate function to execute the correct command syntax for the specific operating system. The uname command is a very useful tool for shell scripting solutions for various UNIX flavors in a single shell script.

I hope you enjoyed this chapter. I think we covered some unique ways to solve the scripting problems that arise when programming for multiple UNIX flavors in the same script. In the next chapter we will dive into the task of taking a system *snapshot*. The idea is to get a point-in-time system configuration for later comparison if a system problem has you puzzled. See you in the next chapter!

Lab Assignments

 Write a shell script that will ping all hosts listed in the /etc/hosts file that are not commented out with a pound sign (#), or just download the pinghostsfile .ksh shell script from www.wiley.com/go/michael2e.Modify this shell script to allow *exceptions* capability, as we used in Chapter 17, "Filesystem Monitoring," so that we can omit hosts from the ping monitoring process. Name this new shell script ping_hosts_exceptions.ksh.

- 2. Modify the ping_hosts_exceptions.ksh script to add logging for unreachable hosts with a date/time stamp. Name this new script ping_hosts_exceptions_ logging.ksh.
- 3. Create a sendmail alias named net_support to replace the \$MAILLIST variable functionality. Don't forget to run the newaliases command!

BONUS: Modify the pingnodes.ksh shell script in Listing 22-1 to troubleshoot a failed ping. The modifications should include the following:

- a. If the ping fails, try doing a DNS lookup; if successful, ping the IP address instead of the hostname.
- b. Keep a list of unreachable hosts and try to analyze if you have a total network failure, or just one segment or subnet is down. HINT: Use the ifconfig -a and netstat -rn commands to extract each adapter's network adapter IP and gateway to isolate the outage. If *all* nodes are unreachable, then log it!

CHAPTER 23

Creating a System-Configuration Snapshot

Have you ever rebooted a system and it came up in an unusual state? Any time that you reboot a system, you run a risk that the system will not come back up properly. When problems arise it is nice to have before and after pictures of the state of the machine. In this chapter we are going to look at some options for shell scripts that execute a series of commands to take a *snapshot* of the state of the machine. Some of the things to consider for this system snapshot include filesystems that are mounted, NFS mounts, processes that are running, network statistics and configuration, routing tables, and a list of defined system resources, just to name a few. This is different from gathering a snapshot of performance statistics, which is gathered over a period of time. All we are looking for is system-configuration data and the system's state at a point in time, specifically before the system is rebooted or when it is running in a normal state with all of the applications running properly.

With this information captured before a system reboot, you have a better chance of fixing a reboot problem quickly and reducing down time. I like to store snapshot information in a directory called /usr/local/reboot with the command names used for filenames. For this shell script all of the system information is stored in a single file with a section header added for each command output. Overall, this is not a difficult shell script to write, but gathering the list of commands that you want to run can sometimes be a challenge. For example, if you want to gather an application's configuration you need to find the commands that will produce the desired output. I always prefer having too much information, rather than not enough information, to troubleshoot a problem.

In this chapter I have put together a list of commands and created a bunch of functions to execute in the shell script. The commands selected are the most critical for troubleshooting an AIX machine; however, you will need to tailor this set of commands to suit your particular needs, operating system, and environment. Every shop is different, but they are all the same in some sense, especially when it comes to troubleshooting a problem. Let's look at some commands and the syntax that is required.

Syntax

As always, we need the commands and the proper syntax for these commands before we can write a shell script. The commands presented in this section are just a sample of the information that you can gather from the system. This set of commands is for an AIX system, but most apply to other UNIX flavors with modified syntax. The list of AIX commands is shown in Listing 23-1.

```
# Hostname of the machine
hostname
OR
uname -n
# UNIX flavor
uname -s
# AIX OS version
oslevel
# AIX maintenance level patch set
instfix -i | grep AIX_ML
OR
oslevel -r
oslevel -s # For AIX 5.3 and later
# Print system configuration
ptrconf
# Time zone for this system
cat /etc/environment | grep TZ | awk -F'=' '{print $2}'
# Real memory in the system
echo "$(bootinfo -r)KB"
OR
lsattr -El -a realmem | awk '{print $2}'
# Machine type/architecture
uname -M
OR - Depending on the architecture
uname -p
# List of defined system devices
lsdev -C
# Long directory listing of /dev
ls -l /dev
# List of all defined disks
lsdev -Cc disk
# List of all defined pdisks for SSA disks
lsdev -Cc pdisk
# List of defined tape drives
lsdev -Cc tape
# List of defined CD-ROMs
lsdev -Cc cdrom
# List of all defined adapters
```

Listing 23-1 System-snapshot commands for AIX

```
lsdev -Cc adapter
# List of network routing table
netstat -rn
# Network adapter statistics
netstat -i
# Filesystem Statistics
df -k
AND
mount
# List of defined Volume Groups
lsvg | sort -r
# List of varied-on Volume Groups
lsvg -o | sort -r
# List of Logical Volumes in each Volume Group
for VG in $(lsvg -o | sort -r)
do
    lsvg -l $VG
done
# Paging space definitions and usage
lsps -a
AND
lsps -s
# List of all hdisks in the system
lspv
# Disk drives listed by Volume Group assignment
for VG in $(lsvg -o | sort -r)
do
    lsvg -p $VG
done
# List the HACMP configuration, if installed
if [ -x /usr/sbin/cluster/utilities/cllsif ]
then
    /usr/sbin/cluster/utilities/cllsif
    echo "\n"
fi
if [ -x /usr/sbin/cluster/utilities/clshowres ]
then
    /usr/sbin/cluster/utilities/clshowres
fi
# List of all defined printers
lpstat -W | tail +3
AND
cat /etc/qconfig
# List of active processes
ps -ef
# Show SNA configuration, if installed
sna -d s
```

```
Listing 23-1 (continued)
```

```
if (($? != 0))
then
   lssrc -s sna -l
fi
# List of udp and x25 processes, if any
ps -ef | egrep 'udp|x25' | grep -v grep
# Short listing of the system configuration
lscfq
# Long listing of the system configuration
lscfg -vp
# List of all system installed filesets
lslpp -L
# List of broken or inconsistent filesets
lppchk -v 2>&1
# List of the last 100 users to log in to the system
last | tail -100
```

As you can see in Listing 23-1, we can add anything that we want to the snapshot shell script to get as much detail as needed to troubleshoot a problem. Every environment is different, so this list of commands should be modified, or added to, to suit the needs of your shop. Additional tests include a list of databases that are running, application configurations, specific application processes that are critical, and a ping list of machines that are critical to the operation of any applications. You can add anything that you want or need here. Always try to gather more information than you think you may need to troubleshoot a problem.

Using this snapshot technique allows us to go back and look at what the system looked like under normal conditions and load. By looking at the snapshot script output file, the problem usually stands out when comparing it to the currently running system that has a problem.

Creating the Shell Script

For this shell script we are going to take the commands shown in Listing 23-1 and create a function for each one. Using functions greatly simplifies both creating and modifying the entire shell script. When we want to add a new test, or configuration output, we just create a new function and add the function-name in the main body of the shell script exactly where we want it to run. In this shell script all of the function definitions use the POSIX, or C-like, function statement, as shown here:

```
get_last_logins ()
{
Commands to execute
}
```

A lot of script programmers like this function-definition technique. I prefer defining a function using the function statement method, as shown here:

```
function get_last_logins
{
Commands to execute
}
```

This last method of defining a function is more intuitive to understand for the people who will follow in your footsteps and modify this shell script. I hope you noticed the use of the word *will* in the last sentence. No matter what the shell script does, there is always someone who will come along, after you have moved on to bigger and better things, who will modify the shell script. It is usually not because there is a problem with the script coding, but more likely a need for added functionality. For the people who follow me, I like to make sure that the shell script is easy to follow and understand. Use your own judgment and preference when defining functions in a shell script; just be consistent.

Because we have all of the commands listed in Listing 23-1, let's look at the entire shell script in Listing 23-2 and see how we created all of these functions.

```
#!/usr/bin/ksh
#
# SCRIPT: AIXsysconfig.ksh
#
# AUTHOR: Randy Michael
# REV: 2.1.A
# DATE: 06/14/2007
# PLATFORM: AIX only
#
# PURPOSE: Take a snapshot of the system for later comparison
   in the event of system problems. All data is stored in
#
#
    /usr/local/reboot in the file defined to the $SYSINFO FILE
    variable below.
#
#
#
# REV LIST:
    7/11/2007: Changed this script to use a single output file
#
#
                that receives data from a series of commands
#
               within a bunch of functions.
#
#
  10/11/2007: Added the following commands to capture
                the AIX technology level (TL) and patch set,
#
#
                and print the system configuration
#
#
               oslevel -s # Show TL and patch levels
#
```

Listing 23-2 AIXsysconfig.ksh shell script listing

```
#
          prtconf # Print the system configuration
#
############
# set -x # Uncomment to debug this script
# set -n # Uncomment to verify command syntax without execution
*****
*****
THISHOST=$(/usr/bin/hostname)
DATETIME=$(date +%m%d%y %H%M%S)
WORKDIR="/usr/local/reboot"
SYSINFO_FILE="${WORKDIR}/sys_snapshot.${THISHOST}.$DATETIME"
*****
*****
get_host ()
{
# Hostname of this machine
hostname
# uname -n works too
******
get_OS ()
{
# Operating System - AIX or exit
uname -s
}
get_OS_level ()
{
# Query for the operating system release and version level
oslevel -r
OSL=$(oslevel -r | cut -c1-2)
if (( OSL >= 53 ))
```

```
then
echo "Technology Level: $(oslevel -s)"
fi
}
****
get_ML_for_AIX ()
{
# Query the system for the maintenance level patch set
instfix -i | grep AIX_ML
}
****
print_sys_config ()
{
prtconf
}
****
get_TZ ()
{
# Get the time zone that the system is operating in.
cat /etc/environment | grep TZ | awk -F'=' '{print $2}'
}
****
get_real_mem ()
{
# Query the system for the total real memory
echo "$(bootinfo -r)KB"
# lsattr -El sys0 -a realmem | awk '{print $2}' Works too
}
****
get_arch ()
{
# Query the system for the hardware architecture. Newer
# machines use the -M switch and the older Micro-Channel
# architecture (MCA) machines use the -p option for
```

```
# the "uname" command.
ARCH=$(uname -M)
if [[ -z "$ARCH" && "$ARCH" = '' ]]
then
   ARCH=$(uname -p)
fi
echo "$ARCH"
}
*****
get_devices ()
{
# Query the system for all configured devices
lsdev -C
}
****
get_long_devdir_listing ()
{
# Long listing of the /dev directory. This shows the
# device major and minor numbers and raw device ownership
ls -l /dev
}
****
get_tape_drives ()
{
# Query the system for all configured tape drives
lsdev -Cc tape
}
*****
get_cdrom ()
{
# Query the system for all configured CD-ROM devices
lsdev -Cc cdrom
}
```

Listing 23-2 (continued)

```
*****
get_adapters ()
{
# List all configured adapters in the system
lsdev -Cc adapter
}
*****
get_routes ()
{
# Save the network routes defined on the system
netstat -rn
}
*****
get_netstats ()
{
# Save the network adapter statistics
netstat -i
}
*****
get_fs_stats ()
{
# Save the file system statistics
df -k
echo "\n"
mount
}
*****
get_VGs ()
{
# List all defined Volume Groups
lsvg | sort -r
}
```

Listing 23-2 (continued)

```
*****
get_varied_on_VGs ()
{
# List all varied-on Volume Groups
lsvg -o | sort -r
}
****
get_LV_info ()
{
# List the Logical Volumes in each varied-on Volume Group
for VG in $(get_varied_on_VGs)
do
   lsvg -l $VG
done
}
******
get_paging_space ()
{
# List the paging space definitions and usage
lsps -a
echo "\n"
lsps -s
}
****
get_disk_info ()
{
# List of all "hdisk"s (hard drives) on the system
lspv
}
****
get_VG_disk_info ()
{
# List disks by Volume Group assignment
for VG in $(get_varied_on_VGs)
```

```
do
   lsvg -p $VG
done
}
****
get_HACMP_info ()
{
# If the System is running HACMP then save the
# HACMP configuration
if [ -x /usr/sbin/cluster/utilities/cllsif ]
then
   /usr/sbin/cluster/utilities/cllsif
   echo "\n\n"
fi
if [ -x /usr/sbin/cluster/utilities/clshowres ]
then
   /usr/sbin/cluster/utilities/clshowres
fi
}
*****
get_printer_info ()
{
# Wide listing of all defined printers
lpstat -W | tail +3
}
******
get_process_info ()
{
# List of all active processes
ps -ef
}
****
get_sna_info ()
{
# If the system is using SNA save the SNA configuration
```

```
sna -d s
                # Syntax for 2.x SNA
if (( $? != 0 ))
then
   lssrc -s sna -l # must be SNA 1.x
fi
}
****
get_udp_x25_procs ()
{
# Listing of all "udp" and "x25" processes, if
# any are running
ps -ef | egrep 'udp|x25' | grep -v grep
}
get_sys_cfg ()
{
# Short listing of the system configuration
lscfg
}
*****
get_long_sys_config ()
{
# Long detailed listing of the system configuration
lscfg -vp
}
****
get_installed_filesets ()
{
# Listing of all installed LPP filesets (system installed)
lslpp -L
}
****
check_for_broken_filesets ()
{
```

Listing 23-2 (continued)

```
# Check the system for broken filesets
lppchk -vm3 2>&1
}
****
last_logins ()
# List the last 100 system logins
last | tail -100
}
****
******
# Check for AIX as the operating system
if [[ $(get_OS) != 'AIX' ]]
then
   echo "\nERROR: Incorrect operating system. This
     shell script is written for AIX.\n"
   echo "\n\t...EXITING...\n"
   exit 1
fi
*****
# Define the working directory and create this
# directory if it does not exist.
if [ ! -d $WORKDIR ]
then
   mkdir -p $WORKDIR >/dev/null 2>&1
   if (($? != 0))
    then
       echo "\nERROR: Permissions do not allow you to create the
     $WORKDIR directory. This script must exit.
     Please create the $WORKDIR directory and
     execute this script again.\n"
       echo "\n\t...EXITING...\n"
       exit 2
    fi
fi
```

```
Listing 23-2 (continued)
```

```
*****
 # Everything enclosed between this opening bracket and the
{
  # later closing bracket is both displayed on the screen and
  # also saved in the log file defined as $SYSINFO FILE.
echo "\n\n[ $(basename $0) - $(date) ]\n"
echo "Saving system information for $THISHOST..."
echo "\nSystem:\t\t\t$(get_host)"
echo "Time Zone:\t\t$(get_TZ)"
echo "Real Memory:\t\t$(get_real_mem)"
echo "Machine Type:\t\t$(get_arch)"
echo "Operating System:\t$(get_OS)"
echo "AIX Version Level:\t$(get_OS_level)"
echo "\nCurrent Maintenance/Technology Level:\n \
$(get_ML_for_AIX)"
echo "Print System Configuration\n"
print sys config
echo "Installed and Configured Devices\n"
get_devices
echo "Long Device Directory Listing - /dev\n"
get long devdir listing
echo "System Tape Drives\n"
get_tape_drives
echo "System CD-ROM Drives\n"
get_cdrom
echo "Defined Adapters in the System\n"
get adapters
echo "Network Routes\n"
get routes
echo "Network Interface Statictics\n"
get netstats
echo "Filesystem Statistics\n"
get_fs_stats
```

Listing 23-2 (continued)

```
echo "Defined Volume Groups\n"
get_VGs
echo "Varied-on Volume Groups\n"
get varied on VGs
echo "Logical Volume Information by Volume Group\n"
get LV info
echo "Paging Space Information\n"
get_paging_space
echo "Hard Disks Defined\n"
get_disk_info
echo "Volume Group Hard Drives\n"
get_VG_disk_info
echo "HACMP Configuration\n"
get_HACMP_info
echo "Printer Information\n"
get printer info
echo "Active Process List\n"
get_process_info
echo "SNA Information\n"
get sna info
echo "x25 and udp Processes\n"
get_udp_x25_procs
echo "System Configuration Overview\n"
get_sys_cfg
echo "Detailed System Configuration\n"
get_long_sys_config
echo "System Installed Filesets\n"
get_installed_filesets
echo "Looking for Broken Filesets\n"
check for broken filesets
echo "List of the last 100 users to login to $THISHOST\n"
last_logins
```

echo "\n\nThis report is saved in: \$SYSINFO_FILE \n"
Send all output to both the screen and the \$SYSINFO_FILE
using a pipe to the "tee -a" command"
} | tee -a \$SYSINFO_FILE

Listing 23-2 (continued)

As you can see in Listing 23-2, we have a lot of functions in this shell script. When I created these functions, I tried to place each one in the order that I wanted to execute in the shell script. This is not necessary as long as you do *not* try to use a function before it is defined. Because a shell script is interpreted, as opposed to a compiled language like C and C++, the flow goes from the top to the bottom. It makes sense that you have to define a function in the code above where the function is used. If we slip up and the function is defined *below* where it is used, we may or may not get an error message. Getting an error message depends on what the function is supposed to do and how the function is executed in the shell script.

From the top of the shell script in Listing 23-2 we first define the variables that we need. The hostname of the machine is always nice to know, and it is required for the report-file definition and in the report itself. Next we create a date/time stamp. This \$DATETIME variable is used in the report-file definition as well. We want the date *and* time because this script may be executed more than once in a single day. Next we define the working directory. I like to use /usr/local/reboot, but you can use any directory that you want. Finally, we define the report file, which is assigned to the \$SYSINFO_FILE variable.

The next section is where all of the functions are defined. Notice that some of these functions contain only a single command, and some have a bit more code. In a shell script like this one it is a good idea to place every command in a separate function. Using this method allows you to change the commands to a different operating system simply by editing some functions and leaving the basic shell script operation intact. There are too many functions in this shell script to go over them one at a time, but an output of this shell script is shown in Listing 23-3. For details on the specific AIX commands please refer to the AIX documentation and man pages on an AIX system.

At START OF MAIN we begin the real work. The first step is to ensure that the operating system is AIX. If this shell script is executed on another UNIX flavor, a lot of the commands will fail. If a non-AIX UNIX flavor is detected, the user receives an error message and the script exits with a return code of 1, one. Step two is to test for the existence of the \$WORKDIR directory, which is defined as /usr/local/reboot in this shell script. If the directory does not exist, an attempt is made to create the directory. Not all users will have permission to create a directory here. If the directory manually and run the shell script again.

If the operating system is AIX and the \$WORKDIR exists, we create the report file and begin creating the report. Notice that the entire list of functions and commands for the report is enclosed in braces, code. Then, after the final brace, at the end of the shell

script, all of the output is piped to the tee -a \$SYSINFO_FILE command. Using this pipe to the tee -a \$SYSINFO_FILE command allows the user to see the report as it is being created and the output is written to the \$SYSINFO_FILE file. Enclosing *all* of the code for the report within the braces saves a lot of effort to get the output to the screen and to the report file. The basic syntax is shown here:

```
####### BEGINNING OF MAIN ########
{
  report command
  .
  .
  .
  report command
} | tee -a $SYSINFO_FILE
```

Within the braces we start by setting up the report header information, which includes the hostname, time zone, real memory, machine type, operating system, operating system version, and the technology level and patch levels of the operating system version. When the header is complete, the script executes the functions listed in the DEFINE FUNCTIONS HERE section. As I stated before, I tried to define the functions in the order of execution. Before each function is executed, a line of hash marks is written out to separate each report section, and then some section header information is written for the specific task. At the end, and just before the ending brace, the report filename is shown to the user to indicate where the report file is located.

Let's take a look at an abbreviated report output in Listing 23-3.

```
[ AIXsysconfig.ksh - Tue Nov 13 12:27:16 EST 2007 ]
Saving system information for yogi...
System: yogi
Time Zone: EST5EDT
EST5EDT,M3.2.0,M11.1.0
Real Memory: 1048576KB
Machine Type: IBM,7029-6C3
Operating System: AIX
AIX Version Level: 5300-06
Technology Level: 5300-06-02-0727
Current Maintenance/Technology Level:
    All filesets for 5.3.0.0_AIX_ML were found.
    All filesets for 5300-01_AIX_ML were found.
    All filesets for 5300-02_AIX_ML were found.
```

Listing 23-3 AIXsysconfig.ksh shell script in action

```
All filesets for 5300-03_AIX_ML were found.
   All filesets for 5300-04_AIX_ML were found.
   All filesets for 5300-05_AIX_ML were found.
   All filesets for 5300-06_AIX_ML were found.
*****
Print System Configuration
System Model: IBM,7029-6C3
Machine Serial Number: 10P1234
Processor Type: PowerPC_POWER4
Number Of Processors: 1
Processor Clock Speed: 1200 MHz
CPU Type: 64-bit
Kernel Type: 64-bit
LPAR Info: 1 NULL
Memory Size: 1024 MB
Good Memory Size: 1024 MB
Platform Firmware level: 3F070425
Firmware Version: IBM, RG070425_d79e20_r
Console Login: enable
Auto Restart: true
Full Core: false
Network Information
   Host Name: yogi
   IP Address: 192.168.1.100
   Sub Netmask: 255.255.255.0
   Gateway: 192.168.1.1
   Name Server:
   Domain Name: tampa.rr.com
Paging Space Information
   Total Paging Space: 512MB
    Percent Used: 1%
Volume Groups Information
_____
rootvg:
       PV STATE TOTAL PPs FREE PPs
PV_NAME
FREE DISTRIBUTION
                         542 482
hdisk0
        active
108..87..70..108..109
hdisk1 active
                           542
                                       272
60..00..103..109
```

Listing 23-3 (continued)

```
INSTALLED RESOURCE LIST
The following resources are installed on the machine.
+/- = Added or deleted from Resource List.
  = Diagnostic support not available.
 Model Architecture: chrp
 Model Implementation: Multiple Processor, PCI bus
+ sys0
                                     System Object
+ sysplanar0
                                     System Planar
* pcil
                  U0.1-P1
                                     PCI Bus
* pci6
                  U0.1-P1
                                    PCI Bus
                  U0.1-P1
                                    PCI Bus
* pci7
* pci8
                  U0.1-P1
                                    PCI Bus
+ dpmpa0
                                    IBM 2-Port Multiprotocol
                  U0.1-P1-I3
Adapter (331121b9)
                 U0.1-P1-I3/Q0 IBM HDLC Network Device Driver
U0.1-P1-I3/Q1 IBM HDLC Network Device Driver
+ hdlc0
+ hdlc1
* pci9
                  U0.1-P1
                                    PCI Bus
+ ent1
                  U0.1-P1/E2
                                    10/100/1000 Base-TX PCI-X
Adapter (14106902)
* pci10
                 U0.1-P1
                                    PCI Bus
                  U0.1-P1
* pci0
                                    PCI Bus
                                   ISA Bus
                 U0.1-P1-X1
* isa0
                 U0.1-P1-X1/D1Standard I/O Diskette AdapterU0.1-P1-X1/K1Keyboard/Mouse AdapterU0.1-P1-X1/K1Keyboard AdapterU0.1-P1-X1/K1Mouse Adapter
+ fda0
* siokma0
+ sioka0
+ sioma0
                  U0.1-P1-X1/R1 CHRP IEEE1284 (ECP) Parallel
+ ppa0
Port Adapter
+ sa0
                  U0.1-P1-X1/S1 Standard I/O Serial Port
                 U0.1-P1-X1/S1-L0 Asynchronous Terminal
+ tty0
                  U0.1-P1-X1/S2 Standard I/O Serial Port
+ sa1
                 U0.1-P1-X1/S3 Standard I/O Serial Port
U0.1-P1-X1/Q6 ATA/IDE Controller Device
+ sa2
* ide0
                  U0.1-P1-X1/Q6-A0 IDE DVD-ROM Drive
+ cd0
* pci2
                 U0.1-P1 PCI Bus
                                    PCI Bus
* pci3
                  U0.1-P1
+ ent0
                  U0.1-P1/E1 10/100 Mbps Ethernet PCI
Adapter II (1410ff01)
* pci4
                  U0.1-P1
                                   PCI Bus
+ sisscsia0
                  U0.1-P1
                                    PCI-X Dual Channel Ultra320
SCSI Adapter
+ scsi0
                  U0.1-P1/Z1 PCI-X Dual Channel Ultra320
SCSI Adapter bus
+ hdisk0
           U0.1-P1/Z1-A5
                                  16 Bit LVD SCSI Disk Drive (36400 MB)
```

Listing 23-3 (continued)

```
+ hdisk1
                 U0.1-P1/Z1-A8
                                 16 Bit LVD SCSI Disk Drive
(36400 MB)
+ ses0
                 U0.1-P1/Z1-AF
                                SCSI Enclosure Services Device
+ scsil
                 U0.1-P1/Z2
                                 PCI-X Dual Channel Ultra320
SCSI Adapter bus
                                 PCI Bus
* pci5
                 U0.1-P1
+ L2cache0
                                 L2 Cache
+ mem0
                                 Memory
+ procl
                 U0.1-P1
                                 Processor
*****
Installed and Configured Devices
L2cache0 Available
                              L2 Cache
aio0
        Available
                               Asynchronous I/O (Legacy)
cd0
        Available 1G-19-00
                              IDE DVD-ROM Drive
dlc8023 Available
                               IEEE Ethernet (802.3) Data Link
Control
dlcether Available
                               Standard Ethernet Data Link Control
dlcfddi Available
                              FDDI Data Link Control
dlcgllc Available
                              X.25 QLLC Data Link Control
dlcsdlc Available
                               SDLC Data Link Control
dpmpa0 Available 1f-08
                              IBM 2-Port Multiprotocol Adapter
(331121b9)
en0
        Available 1L-08
                               Standard Ethernet Network Interface
         Defined 1j-08
                               Standard Ethernet Network Interface
en1
ent0
        Available 1L-08
                              10/100 Mbps Ethernet PCI
Adapter II (1410ff01)
ent1
        Available 1j-08
                               10/100/1000 Base-TX PCI-X
Adapter (14106902)
et0
        Defined 1L-08
                              IEEE 802.3 Ethernet Network Interface
         Defined 1j-08
et1
                               IEEE 802.3 Ethernet Network Interface
fda0
         Available 01-D1
                               Standard I/O Diskette Adapter
. (skipping forward)
hdisk0
        Available 1S-08-00-5,0 16 Bit LVD SCSI Disk Drive
hdisk1
         Available 1S-08-00-8,0 16 Bit LVD SCSI Disk Drive
hdlc0
        Available 1f-08-00
                              IBM HDLC Network Device Driver
hdlc1
        Available 1f-08-01
                              IBM HDLC Network Device Driver
ide0
        Available 1G-19
                              ATA/IDE Controller Device
         Available
inet0
                               Internet Network Extension
        Available 1G-18
isa0
                              ISA Bus
iscsi0
         Available
                               iSCSI Protocol Device
100
         Available
                              Loopback Network Interface
lvdd
         Available
                               LVM Device Driver
```

```
mem0
         Available
                              Memorv
        Available
                              SDLC COMIO Device Driver Emulator
mpc0
         Available
                              SDLC COMIO Device Driver Emulator
mpc1
paging00 Defined
                             N/A
       Available
pci0
                             PCI Bus
pci1
        Available
                             PCI Bus
pci2
         Available 1G-10
                             PCI Bus
        Available 1G-12
                             PCI Bus
pci3
        Available 1G-14
                             PCI Bus
pci4
pci5
        Available 1G-16
                             PCI Bus
pci6
        Available 1Y-10
                             PCI Bus
        Available 1Y-12
pci7
                             PCI Bus
        Available 1Y-13
                             PCI Bus
pci8
posix_aio0 Defined
                             Posix Asynchronous I/O
                              CHRP IEEE1284 (ECP) Parallel
ppa0
       Available 01-R1
Port Adapter
proc1
        Available 00-01
                             Processor
pty0
        Available
                              Asynchronous Pseudo-Terminal
rcm0
        Defined
                             Rendering Context Manager Subsystem
rootvg
        Defined
                              N/A
                             Standard I/O Serial Port
sa0
        Available 01-S1
sa1
        Available 01-S2
                             Standard I/O Serial Port
        Available 01-S3
                             Standard I/O Serial Port
sa2
         Available 1S-08-00
                             PCI-X Dual Channel Ultra320
scsi0
SCSI Adapter bus
         Available 1S-08-01 PCI-X Dual Channel Ultra320
scsi1
SCSI Adapter bus
      Available 1S-08-00-15,0 SCSI Enclosure Services Device
ses0
        Available 01-K1-00 Keyboard Adapter
sioka0
siokma0 Available 01-K1
                             Keyboard/Mouse Adapter
sioma0
        Available 01-K1-01
                            Mouse Adapter
sisscsia0 Available 1S-08
                             PCI-X Dual Channel Ultra320
SCSI Adapter
sys0 Available
                             System Object
sysplanar0 Available
                              System Planar
tty0
       Available 01-S1-00-00 Asynchronous Terminal
*****
Long Device Directory Listing - /dev
total 40
drwxrwx--- 2 root
                                4096 Nov 13 12:27 .SRC-unix
                    system
crw-rw---- 1 root
                   system
                               10, 0 Apr 13 2005 IPL rootvg
srwxrwxrwx 1 root
                    system
                                    0 Nov 13 11:48 SRC
crw----- 1 root system
                                10, 0 Nov 13 11:36 __vg10
cr--r---T 1 root system
                                8, 0 Apr 13 2005 audit
                                22, 0 Apr 13 2005 cd0
br--r--r-- 1 root
                  system
```

crw-rw-rw-	1	root	system	13,	0	Apr	13	2005	clone	
CIMMM-	1	root	system	4,	0	Apr	13	2005	console	
crw-rw-rwT	1	root	system	50,	0	Apr	15	2005	dlc8023	
crw-rw-rwT	1	root	system	49,	0	Apr	15	2005	dlcether	
crw-rw-rwT	1	root	system	48,	0	Apr	15	2005	dlcfddi	
crw-rw-rwT	1	root	system	52,	0	Nov	12	14:40	dlcmpc	
crw-rw-rwT	1	root	system	53,	0	Nov	13	10:39	dlcqllc	
crw-rw-rwT	1	root	system	47,	0	Apr	15	2005	dlcsdlc	
crw-rw-rwT	1	root	system	46,	0	Apr	15	2005	dlctoken	
crw-rw-rw-	1	root	system	13,	31	Apr	13	2005	echo	
CIMMM-	1	root	system	б,	0	Nov	13	11:49	error	
crw	1	root	system	б,	1	Apr	13	2005	errorctl	
brw-rw	1	root	system	10,	11	May	5	2005	fslv00	
brw-rw	1	root	system	10,	12	Nov	12	15 : 37	fslv01	
brw-rw	1	root	system	10,	8	Apr	13	2005	hd1	
brw-rw	1	root	system	10,	9	Apr	13	2005	hd10opt	
brw-rw	1	root	system	10,	5	Apr	13	2005	hd2	
brw-rw	1	root	system	10,	7	Apr	13	2005	hd3	
brw-rw		root	system	10,				11:36		
brw-rw		root	system	10,				11:48		
brw-rw		root	system	10,		-		2005	hd6	
brw-rw		root	system	10,		-		2005	hd8	
brw-rw		root	system	10,		-		2005	hd9var	
brw		root	system	23,					hdisk0	
brw	1	root	system	23,	1	Nov	13	11:22	hdisk1	
•	C									
. (skipping	LOI	rward)								
•										
crw-rw-rw-	1	root	system	29,	З	∆nr	13	2005	ntum3	
crw-rw-rw-		root	system	29,		-		2005	ptyp3 ptyp4	
crw-rw-rw-		root	system	29,		_		2005	ptyp5	
crw-rw-rw-		root	system	29,		_		2005	ptyp6	
crw-rw-rw-		root	system	29,		-		2005	ptyp7	
crw-rw-rw-		root	system	29,		-		2005	ptyp8	
crw-rw-rw-		root	system	29,		_		2005	ptyp9	
crw-rw-rw-		root	system	29,		_		2005	ptypa	
crw-rw		root	system	10,		-		2005	rootvg	
crw-rw		root	system	10,		-		2005	rpaging00	
crw-rw-rw-		root	system			-		2005	sad	
crw-rw-rw-	1	root	system	19,		_		2005	scsi0	
crw-rw-rw-		root	system	19,				2005	scsi1	
crw-rw-rw-		root	system	25,		_		2005	ses0	
crw-rw-rw-		root	system	19,		_			sisscsia0	
crw-rw-rw-		root	system					2005	slog	
crw-rw-rwT		root	system	44,		-		11:48	-	
crw-rw-rwT		root	system	41,					sna_trace	
			-						—	_

```
. (skipping forward)
                   system 2, 3 Apr 13 2005 zero
crw-rw-rw- 1 root
******
System Tape Drives
****
System CD-ROM Drives
cd0 Available 1G-19-00 IDE DVD-ROM Drive
****
Defined Adapters in the System
      Available 1f-08 IBM 2-Port Multiprotocol Adapter (331121b9)
dpmpa0
ent0
      Available 1L-08 10/100 Mbps Ethernet PCI
Adapter II (1410ff01)
ent1
       Available 1j-08 10/100/1000 Base-TX PCI-X
Adapter (14106902)
fda0 Available 01-D1 Standard I/O Diskette Adapter
ide0
      Available 1G-19 ATA/IDE Controller Device
ppa0
      Available 01-R1 CHRP IEEE1284 (ECP) Parallel Port
Adapter
     Available 01-S1 Standard I/O Serial Port
sa0
sa1
      Available 01-S2 Standard I/O Serial Port
      Available 01-S3 Standard I/O Serial Port
sa2
sioka0 Available 01-K1-00 Keyboard Adapter
siokma0 Available 01-K1 Keyboard/Mouse Adapter
sioma0 Available 01-K1-01 Mouse Adapter
sisscsia0 Available 1S-08 PCI-X Dual Channel Ultra320 SCSI Adapter
****
Network Routes
Routing tables
Destination Gateway
                          Flags Refs
                                       Use If Exp Groups
Route tree for Protocol Family 2 (Internet):
default
              192.168.1.1 UG 3
                                      7497 en0
                                                 _
```

Listing 23-3 (continued)

U 127/8 127.0.0.1 3 100 100 192.168.62.0 192.168.62.25 UHSb 0 0 en0 ****** Network Interface Statistics Name Mtu Network Address Ipkts Ierrs Opkts Oerrs Coll en0 1500 link#2 0.9.6b.2e.7c.da 0 7508 8898 0 0 en0 1500 172.27.62 ift00s 0 7508 8898 0 0 1o0 16896 link#1 4559 0 4562 0 0 4559 0 4562 100 16896 127 localhost 0 0 4559 0 4562 0 100 16896 ::1 0 ****** Filesystem Statistics Filesystem 1024-blocks Free %Used Iused %Iused Mounted on /dev/hd4 131072 107820 18% 2257 9% / 1245184 85020 94% 24985 52% /usr /dev/hd2 524288 494852 6% 391 1%/var /dev/hd9var 131072 119048 10% /dev/hd3 62 1% /tmp /dev/hd1 65536 63680 3% 92 1% /home -- -- /proc -/proc /dev/hd10opt 131072 16960 88% 3187 43% /opt 1048576 1008184 4% 523 1% /apps /dev/fslv00 /dev/fslv01 13762560 7541816 46% 1983 1% /install node mounted mounted over vfs date options _____ _____ /dev/hd4 / jfs2 Nov 13 11:47 rw,log=/dev/hd8 /dev/hd2 /usr jfs2 Nov 13 11:47 rw,log=/dev/hd8 /dev/hd9var /var jfs2 Nov 13 11:47 rw,log=/dev/hd8 /dev/hd3 /tmp jfs2 Nov 13 11:47 rw,log=/dev/hd8 /dev/hd1 /home jfs2 Nov 13 11:48 rw,log=/dev/hd8 /proc /proc procfs Nov 13 11:48 rw /dev/hd10opt /opt jfs2 Nov 13 11:48 rw,log=/dev/hd8 /dev/fslv00 /apps jfs2 Nov 13 11:48 rw,log=/dev/hd8 /dev/fslv01 /install jfs2 Nov 13 11:48 rw,log=/dev/hd8 **** Defined Volume Groups rootvg

Listing 23-3 (continued)

###########	###########	#####	#####	+#####	#######		
Varied-on V	olume Groups						
rootvg							
###########	###########	#####	#####	######	*########		
Logical Vol	ume Informat:	ion by	Volur	ne Grou	p		
rootvg:							
LV NAME	TYPE	LPs	PPs	PVs	LV STATE	MOUNT	r point
hd5	boot	1	2	2	closed/syncd	N/A	
hd6	paging	5	10	2	open/syncd	N/A	
hd8	jfs2log	1	2	2	open/syncd	N/A	
hd4	jfs2	2	4	2	open/syncd	/	
hd2	jfs2	19	38	2	open/syncd	/usr	
hd9var	jfs2	8	16	2	open/syncd	/var	
hd3	jfs2	2	4	2	open/syncd	/tmp	
hd1	jfs2	1	2	2	open/syncd	/home	e
hd10opt	jfs2	2	4	2	open/syncd	/opt	
paging00	paging	3	6	2	open/syncd	N/A	
fslv00	jfs2	16	32	2	open/syncd	/apps	3
fslv01	jfs2	210	210	1	open/syncd	/inst	all
	############# e Information		#####	****	****		
Page Space Auto Type	Physical	l Volur	ne T	7olume	Group Size	%Used	Active
paging00	hdisk0		1	rootvg	192MB	2	yes
yes lv hd6	hdisk1		,	cootvg	320MB	1	VOC
yes lv	nuiski		Ţ	JUULVY	520MB	Ţ	yes
Total Pagin 512MB		rcent (1%					
	###########	#####	#####	+#####	########		
Hard Disks	Defined						
hdisk0	000b123d3bf5	557c	1	cootvg	active		
hdisk1	000b123dfed50	d3ca	1	rootvg	active		

Listing 23-3 (continued)

```
Volume Group Hard Drives
rootvg:
PV_NAME PV STATE TOTAL PPs FREE PPs FREE DISTRIBUTION
hdisk0 active 542 482 108..87..70..108..109
hdisk1
       active
                542
                       272
                              60..00..00..103..109
*****
HACMP Configuration
*****
Printer Information
Active Process List
UID PID PPID C STIME TTY TIME CMD
root 1 0 0 11:47:57 - 0:00 /etc/init
           1 0 11:48:38 - 0:00 /usr/sbin/syncd 60
root 77920
root 98316
           1 0 11:48:38 - 0:00 /usr/lib/errdemon
           1 0 11:48:38 - 0:00 /usr/ccs/bin/shlap64
root 426198
root 430126 1 0 11:48:46 - 0:00 /usr/sbin/srcmstr
super 438298 528400 0 11:51:44 - 0:00 sshd: super@pts/0
root 446478 430126 0 11:48:54 - 0:00 /usr/sbin/portmap
root 471232 430126 0 11:48:54 - 0:00 /usr/sbin/inetd
root 475352 1 0 11:48:48 - 0:00 snadaemon64
root 479276 475352 0 11:48:49 - 0:00 snaslpdae -p11 -a
root 483566 475352 0 11:48:48 - 0:00 snaerrlog -p7
. (skipping forward)
root 647302 635004 0 12:00:49 pts/1 0:00 -ksh
******
SNA Information
  Local
               Partner
                         Mode Link
  LU name
               LU name
                          name
                                station
                                         State
_____ ____
SNA is not active. Restart SNA and retry your command.
0513-085 The sna Subsystem is not on file.
```

Listing 23-3 (continued)

<pre>x25 and udp Processes ********************************</pre>	+	****									
System Configuration Overview INSTALLED RESOURCE LIST The following resources are installed on the machine. +/- = Added or deleted from Resource List. * = Diagnostic support not available. Model Architecture: chrp Model Implementation: Multiple Processor, PCI bus * sys0 System Object * sysplanar0 System Planar * pci1 U0.1-P1 PCI Bus * pci6 U0.1-P1 PCI Bus * pci6 U0.1-P1 PCI Bus * pci8 U0.1-P1 PCI Bus * hdlc0 U0.1-P1-IJ/Q1 IBM HDLC Network Device Driver * hdlc1 U0.1-P1-PI PCI Bus * pci0 U0.1-P1 PCI Bus * isa0 U0.1-P1.XI/XI Keyboard/Mouse Adapter * siokma0 U0.1-P1-XI/XI Keyboard Adapter * siokma0 U0.1-P1-XI/XI Keyboard Adapter * siokma0 U0.1-P1-XI/SI Standard I/O Serial Port * tty0 U0.1-P1-XI/SI Standard I/O Serial Port * sal U0.1-P1-XI/S3 Standard I/O Serial Port * sal U0.1-P1-XI/S4 OATA/IDE Controller Device * cd0 U0.1-P1-XI/S4 OA DIE DVD-ROM Drive * pci2 U0.1-P1 PCI Bus * ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus * ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus * E2Cache0 * mem0 Memory	2	x25 and udp Processes									
System Configuration Overview INSTALLED RESOURCE LIST The following resources are installed on the machine. +/- = Added or deleted from Resource List. * = Diagnostic support not available. Model Architecture: chrp Model Implementation: Multiple Processor, PCI bus * sys0 System Object * sysplanar0 System Planar * pci1 U0.1-P1 PCI Bus * pci6 U0.1-P1 PCI Bus * pci6 U0.1-P1 PCI Bus * pci8 U0.1-P1 PCI Bus * hdlc0 U0.1-P1-IJ/Q1 IBM HDLC Network Device Driver * hdlc1 U0.1-P1-PI PCI Bus * pci0 U0.1-P1 PCI Bus * isa0 U0.1-P1.XI/XI Keyboard/Mouse Adapter * siokma0 U0.1-P1-XI/XI Keyboard Adapter * siokma0 U0.1-P1-XI/XI Keyboard Adapter * siokma0 U0.1-P1-XI/SI Standard I/O Serial Port * tty0 U0.1-P1-XI/SI Standard I/O Serial Port * sal U0.1-P1-XI/S3 Standard I/O Serial Port * sal U0.1-P1-XI/S4 OATA/IDE Controller Device * cd0 U0.1-P1-XI/S4 OA DIE DVD-ROM Drive * pci2 U0.1-P1 PCI Bus * ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus * ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus * E2Cache0 * mem0 Memory											
<pre>INSTALLED RESOURCE LIST The following resources are installed on the machine. +/- = Added or deleted from Resource List. * = Diagnostic support not available. Model Architecture: chrp Model Implementation: Multiple Processor, PCI bus * sys0 System Object * sysplanar0 System Planar * pci1 U0.1-P1 PCI Bus * pci6 U0.1-P1 PCI Bus * pci8 U0.1-P1 PCI Bus * pci8 U0.1-P1 PCI Bus * hdlc0 U0.1-P1-T3/Q0 IBM HDLC Network Device Driver * hdlc1 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * pci10 U0.1-P1 PCI Bus * isa0 U0.1-P1-X1/K1 Keyboard/Mouse Adapter * siokma0 U0.1-P1-X1/K1 Keyboard/Mouse Adapter * siokma0 U0.1-P1-X1/K1 Keyboard J/O Serial Port * tty0 U0.1-P1-X1/S1 Standard I/O Serial Port * sa1 U0.1-P1-X1/S3 Standard I/O Serial Port * sa2 U0.1-P1-X1/S4 Standard I/O Serial Port * sa2 U0.1-P1-X1/S5 Standard I/O Serial Port * sa2 U0.1-P1 PCI Bus * ses0 U0.1-P1</pre>	+	******	*****	****							
The following resources are installed on the machine. +/- = Added or deleted from Resource List. * = Diagnostic support not available. Model Architecture: chrp Model Implementation: Multiple Processor, PCI bus * sys0 System Planar * pci1 U0.1-P1 PCI Bus * pci6 U0.1-P1 PCI Bus * pci6 U0.1-P1 PCI Bus * pci7 U0.1-P1 PCI Bus * pci8 U0.1-P1 PCI Bus * pci8 U0.1-P1 PCI Bus * pci8 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * pci10 U0.1-P1 PCI Bus * pci10 U0.1-P1 PCI Bus * pci10 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * isa0 U0.1-P1-X1/I ISA Bus f fda0 U0.1-P1-X1/K1 Keyboard/Mouse Adapter * siokma0 U0.1-P1-X1/K1 Keyboard Adapter * siokma0 U0.1-P1-X1/K1 Keyboard Adapter * siokma0 U0.1-P1-X1/S1 Standard I/O Serial Port * tty0 U0.1-P1-X1/S1 Standard I/O Serial Port * tty0 U0.1-P1-X1/S2 Standard I/O Serial Port * tde0 U0.1-P1-X1/S3 Standard I/O Serial Port * sa1 U0.1-P1-X1/S3 Standard I/O Serial Port * sa2 U0.1-P1-X1/S3 Standard I/O Serial Port * sa2 U0.1-P1-X1/S4 Standard I/O Serial Port * sa2 U0.1-P1-X1/S5 Standard I/O Serial Port * sa2 U0.1-P1-X1/G6 ATA/IDE Controller Device * cd0 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus * pci5 U0.1-P1 PCI Bus * ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus	C.	System Configuration Overview									
<pre>+/- = Added or deleted from Resource List. * = Diagnostic support not available. Model Architecture: chrp Model Implementation: Multiple Processor, PCI bus * sys0 System Object sysplanar0 System Planar * pci1 U0.1-P1 PCI Bus * pci6 U0.1-P1 PCI Bus * pci7 U0.1-P1 PCI Bus * pci8 U0.1-P1 PCI Bus * pci8 U0.1-P1 PCI Bus * hdlc0 U0.1-P1-T3/Q0 IBM HDLC Network Device Driver * hdlc1 U0.1-P1 PCI Bus * pci10 U0.1-P1 PCI Bus * pci10 U0.1-P1 PCI Bus * pci10 U0.1-P1 PCI Bus * isa0 U0.1-P1-X1/NI ISA Bus * fda0 U0.1-P1-X1/NI Standard I/O Diskette Adapter * sioka0 U0.1-P1-X1/K1 Keyboard Adapter * sioka0 U0.1-P1-X1/K1 Keyboard Adapter * sioka0 U0.1-P1-X1/S1 Standard I/O Serial Port * tty0 U0.1-P1-X1/S3 Standard I/O Serial Port * sa2 U0.1-P1-X1/G6 ATA/IDE Controller Device * cd0 U0.1-P1-X1/Q6 ATA/IDE Controller Device * cd0 U0.1-P1-X1/Q6 ATA/IDE Controller Device * pci3 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus * pci5 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus * pci5 U0.1-P1 PCI PUS * pci5 U0.1-P1 PCI PCI Bus * pci5 U0.1-P1 PCI PCI Bus * pci5 U0.1-P1 PCI PCI PCI PCI PCI PCI PCI PCI PCI PCI</pre>]	INSTALLED RESOURCE	LIST								
 * = Diagnostic support not available. Model Architecture: chrp Model Implementation: Multiple Processor, PCI bus * sys0 System Object * sysplanar0 System Planar * pci1 U0.1-P1 PCI Bus * pci6 U0.1-P1 PCI Bus * pci8 U0.1-P1 PCI Bus * hdlc0 U0.1-P1-I3/Q0 IBM HDLC Network Device Driver * hdlc1 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * solo U0.1-P1 PCI Bus * sisa0 U0.1-P1 PCI Bus * sisa0 U0.1-P1-X1/D1 Standard I/O Diskette Adapter * siokma0 U0.1-P1-X1/K1 Keyboard Adapter * siokma0 U0.1-P1-X1/K1 Mouse Adapter * sioma0 U0.1-P1-X1/S1 Standard I/O Serial Port * tty0 U0.1-P1-X1/S1 Standard I/O Serial Port * tty0 U0.1-P1-X1/S2 Standard I/O Serial Port * sal U0.1-P1-X1/S2 Standard I/O Serial Port * sal U0.1-P1-X1/G6 ATA/IDE Controller Device * cd0 U0.1-P1-X1/G6 ATA/IDE Controller Device * pci2 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus * pci5 U0.1-P1 PCI Bus * pci5	Г	The following reso	urces are installe	d on the machine.							
Model Architecture: chrp Model Implementation: Multiple Processor, PCI bus+ sys0System Object+ sysplanar0System Planar* pci1U0.1-P1PCI Bus* pci6U0.1-P1PCI Bus* pci7U0.1-P1PCI Bus* pci8U0.1-P1PCI Bus+ hdlc0U0.1-P1PCI Bus+ hdlc1U0.1-P1PCI Bus* pci9U0.1-P1PCI Bus* pci0U0.1-P1PCI Bus* pci0U0.1-P1PCI Bus* pci10U0.1-P1PCI Bus* pci0U0.1-P1PCI Bus* soldU0.1-P1PCI Bus* sisa0U0.1-P1-X1/I ISA Bus* fda0U0.1-P1-X1/K1* sioka0U0.1-P1-X1/K1* sioka0U0.1-P1-X1/K1* sioma0U0.1-P1-X1/S1* stadU0.1-P1-X1/S1* sa1U0.1-P1-X1/S1* sa2U0.1-P1-X1/S3* sa4U0.1-P1-X1/Q6* sa2U0.1-P1-X1/Q6* sa2U0.1-P1-X1/Q6* pci3U0.1-P1* pci3U0.1-P1* pci3U0.1-P1* pci4U0.1-P1* pci5U0.1-P1* pci5U0.1-P1* pci5U0.1-P1* pci5U0.1-P1* pci4U0.1-P1* pci5U0.1-P1* pci5U0.1-P1* pci5U0.1-P1* pci5U0.1-P1* pci4U0.1-P1* pci5U0.1-P1* pci5U0.1-P1* pci5	4	-/- = Added or del	eted from Resource	List.							
Model Implementation: Multiple Processor, PCI bus+ sys0System Object+ sysplanar0System Planar* pci1U0.1-P1PCI Bus* pci6U0.1-P1PCI Bus* pci7U0.1-P1PCI Bus* pci8U0.1-P1PCI Bus* hdlc0U0.1-P1-T3/Q0IEM HDLC Network Device Driver+ hdlc1U0.1-P1-T3/Q1IEM HDLC Network Device Driver+ hdlc1U0.1-P1PCI Bus* pci9U0.1-P1PCI Bus* pci0U0.1-P1PCI Bus* pci0U0.1-P1PCI Bus* isa0U0.1-P1-X1/D1Standard I/O Diskette Adapter* sioka0U0.1-P1-X1/K1Keyboard Adapter+ sioka0U0.1-P1-X1/K1Keyboard Adapter+ sioma0U0.1-P1-X1/K1Mouse Adapter+ sa0U0.1-P1-X1/S1Standard I/O Serial Port+ tty0U0.1-P1-X1/S2Standard I/O Serial Port+ sa1U0.1-P1-X1/S3Standard I/O Serial Port+ sa2U0.1-P1-X1/S3Standard I/O Serial Port* sa2U0.1-P1-X1/S4Standard I/O Serial Port* sa2U0.1-P1-X1/G6ATA/IDE Controller Device* pci3U0.1-P1PCI Bus* pci3U0.1-P1PCI Bus* pci5U0.1-P1PCI Bus* pci5U0.1-P1PCI Bus* pci5U0.1-P1PCI Bus* pci5U0.1-P1PCI Bus* pci5U0.1-P1PCI Bus* pci5U0.1-P1PCI Bus <trr>* pci5U0.1-P1<td>ł</td><td>= Diagnostic s</td><td>upport not availab</td><td>le.</td></trr>	ł	= Diagnostic s	upport not availab	le.							
Model Implementation: Multiple Processor, PCI bus+ sys0System Object+ sysplanar0System Planar* pci1U0.1-P1PCI Bus* pci6U0.1-P1PCI Bus* pci7U0.1-P1PCI Bus* pci8U0.1-P1PCI Bus* hdlc0U0.1-P1-T3/Q0IEM HDLC Network Device Driver+ hdlc1U0.1-P1-T3/Q1IEM HDLC Network Device Driver+ hdlc1U0.1-P1PCI Bus* pci9U0.1-P1PCI Bus* pci0U0.1-P1PCI Bus* pci0U0.1-P1PCI Bus* isa0U0.1-P1-X1/D1Standard I/O Diskette Adapter* sioka0U0.1-P1-X1/K1Keyboard Adapter* sioka0U0.1-P1-X1/K1Keyboard Adapter* sioma0U0.1-P1-X1/K1Mouse Adapter* sa0U0.1-P1-X1/S1Standard I/O Serial Port* tty0U0.1-P1-X1/S1Standard I/O Serial Port* sa1U0.1-P1-X1/S3Standard I/O Serial Port* sa2U0.1-P1-X1/G6ATA/IDE Controller Device* cd0U0.1-P1-X1/G6ATA/IDE Controller Device* pci3U0.1-P1PCI Bus* pci3U0.1-P1PCI Bus* pci5U0.1-P1PCI Bus <td></td> <td>Model Architectu</td> <td>ro, chro</td> <td></td>		Model Architectu	ro, chro								
<pre>+ sys0 System Object + sysplanar0 System Planar * pci1 U0.1-P1 PCI Bus * pci6 U0.1-P1 PCI Bus * pci7 U0.1-P1 PCI Bus * pci8 U0.1-P1 PCI Bus + hdlc0 U0.1-P1-I3/Q0 IBM HDLC Network Device Driver + hdlc1 U0.1-P1-I3/Q1 IBM HDLC Network Device Driver * pci9 U0.1-P1 PCI Bus * pci10 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * isa0 U0.1-P1 PCI Bus * isa0 U0.1-P1 PCI Bus * isa0 U0.1-P1-X1 ISA Bus + fda0 U0.1-P1-X1/D1 Standard I/O Diskette Adapter * siokma0 U0.1-P1-X1/K1 Keyboard Adapter + siokma0 U0.1-P1-X1/K1 Keyboard Adapter + siokma0 U0.1-P1-X1/K1 Standard I/O Serial Port + sioka0 U0.1-P1-X1/K1 Standard I/O Serial Port + sioka0 U0.1-P1-X1/S1 Standard I/O Serial Port + sa1 U0.1-P1-X1/S2 Standard I/O Serial Port + sa2 U0.1-P1-X1/Q6 ATA/IDE Controller Device + cd0 U0.1-P1-X1/Q6 ATA/IDE Controller Device + cd0 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus + ses0 U0.1-P1 PCI Bus + ses0 U0.1-P1 PCI Bus + ses0 U0.1-P1 PCI Bus + ses0 U0.1-P1 PCI Bus * pci5 U0.1-P1 PCI Bus + ses0 U0.1-P1 P</pre>			_	cessor. PCT bus							
+ System Planar * pci1 U0.1-P1 PCI Bus * pci6 U0.1-P1 PCI Bus * pci7 U0.1-P1 PCI Bus * pci8 U0.1-P1 PCI Bus * pci8 U0.1-P1 PCI Bus * hdlc0 U0.1-P1-I3/Q0 IBM HDLC Network Device Driver * hdlc1 U0.1-P1-I3/Q1 IBM HDLC Network Device Driver * hdlc1 U0.1-P1 PCI Bus * pci9 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * pci0 U0.1-P1-X1 ISA Bus * fda0 U0.1-P1-X1/K1 Keyboard/Mouse Adapter * sioka0 U0.1-P1-X1/K1 Keyboard Adapter * sioma0 U0.1-P1-X1/K1 Mouse Adapter * sa0 U0.1-P1-X1/S1 Standard I/O Serial Port * tty0 U0.1-P1-X1/S3 Standard I/O Serial Port * sa2 U0.1-P1-X1/Q6 ATA/IDE Controller Device *		model impremented	cion, narcipie iio								
* pci1 U0.1-P1 PCI Bus * pci6 U0.1-P1 PCI Bus * pci7 U0.1-P1 PCI Bus * pci8 U0.1-P1 PCI Bus * hdlc0 U0.1-P1-I3/Q0 IBM HDLC Network Device Driver * hdlc1 U0.1-P1-I3/Q1 IBM HDLC Network Device Driver * hdlc1 U0.1-P1-I3/Q1 IBM HDLC Network Device Driver * pci9 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * isa0 U0.1-P1 PCI Bus * isa0 U0.1-P1 Standard I/O Diskette Adapter * siokma0 U0.1-P1-X1/K1 Keyboard/Mouse Adapter * sioka0 U0.1-P1-X1/K1 Keyboard Adapter * sioka0 U0.1-P1-X1/K1 Keyboard Adapter * sioma0 U0.1-P1-X1/K1 Keyboard Adapter * sa0 U0.1-P1-X1/K1 Keyboard Adapter * sa1 U0.1-P1-X1/S1 Standard I/O Serial Port * sa2 U0.1-P1-X1/S2 Standard I/O Serial Port * sa2 U0.1-P1-X1/Q6 ATA/IDE Controller Device * pci3<	4	- sys0		System Object							
* pci6 U0.1-P1 PCI Bus * pci7 U0.1-P1 PCI Bus * pci8 U0.1-P1 PCI Bus * hdlc0 U0.1-P1-I3/Q0 IBM HDLC Network Device Driver * hdlc1 U0.1-P1-I3/Q1 IBM HDLC Network Device Driver * hdlc1 U0.1-P1-I3/Q1 IBM HDLC Network Device Driver * pci9 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * isa0 U0.1-P1-X1 ISA Bus + fda0 U0.1-P1-X1/K1 Keyboard/Mouse Adapter * siokma0 U0.1-P1-X1/K1 Keyboard Adapter + sioka0 U0.1-P1-X1/K1 Mouse Adapter + sioka0 U0.1-P1-X1/S1 Standard I/O Serial Port + sa0 U0.1-P1-X1/S2 Standard I/O Serial Port + tty0 U0.1-P1-X1/S2 Standard I/O Serial Port + sa1 U0.1-P1-X1/S2 Standard I/O Serial Port + sa2 U0.1-P1-X1/Q6 ATA/IDE Controller Device + cd0 U0.1-P1 PCI Bus * pci3	-	- sysplanar0		System Planar							
* pci7 U0.1-P1 PCI Bus * pci8 U0.1-P1 PCI Bus * hdlc0 U0.1-P1-I3/Q0 IBM HDLC Network Device Driver * hdlc1 U0.1-P1-I3/Q1 IBM HDLC Network Device Driver * pci9 U0.1-P1 PCI Bus * pci10 U0.1-P1 PCI Bus * pci0 U0.1-P1.X1 ISA Bus * fda0 U0.1-P1-X1/K1 Keyboard/Mouse Adapter * sioka0 U0.1-P1-X1/K1 Keyboard Adapter * sioka0 U0.1-P1-X1/K1 Keyboard Adapter * sioka0 U0.1-P1-X1/S1 Standard I/O Serial Port * sa0 U0.1-P1-X1/S2 Standard I/O Serial Port * sa2 U0.1-P1-X1/Q6 ATA/IDE Controller Device<	ł	pcil	U0.1-P1	PCI Bus							
* pci8 U0.1-P1 PCI Bus + hdlc0 U0.1-P1-I3/Q0 IBM HDLC Network Device Driver + hdlc1 U0.1-P1-I3/Q1 IBM HDLC Network Device Driver * pci9 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * isa0 U0.1-P1 Standard I/O Diskette Adapter * siokma0 U0.1-P1-X1/K1 Keyboard/Mouse Adapter * siokma0 U0.1-P1-X1/K1 Keyboard Adapter * sioma0 U0.1-P1-X1/S1 Standard I/O Serial Port * tty0 U0.1-P1-X1/S2 Standard I/O Serial Port * tty0 U0.1-P1-X1/Q6 ATA/IDE Controller Device * cd0 U0.1-P1-X1/Q6 IDE DVD-ROM Drive * pci2 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus	4	pci6	U0.1-P1	PCI Bus							
+ hdlc0 U0.1-P1-I3/Q0 IBM HDLC Network Device Driver + hdlc1 U0.1-P1-I3/Q1 IBM HDLC Network Device Driver * pci9 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * isa0 U0.1-P1 PCI Bus * isa0 U0.1-P1-X1 ISA Bus + fda0 U0.1-P1-X1/D1 Standard I/O Diskette Adapter * siokma0 U0.1-P1-X1/K1 Keyboard/Mouse Adapter + sioka0 U0.1-P1-X1/K1 Keyboard Adapter + sioma0 U0.1-P1-X1/S1 Standard I/O Serial Port + sa0 U0.1-P1-X1/S1 Standard I/O Serial Port + tty0 U0.1-P1-X1/S2 Standard I/O Serial Port + sa1 U0.1-P1-X1/S3 Standard I/O Serial Port * sa2 U0.1-P1-X1/Q6 ATA/IDE Controller Device + cd0 U0.1-P1-X1/Q6 ATA/IDE Controller Device * pci3 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus * pci5 U0.1-P1 PCI Bus * pci	ł	pci7	U0.1-P1	PCI Bus							
+ hdlc1 U0.1-P1-I3/Q1 IBM HDLC Network Device Driver * pci9 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * isa0 U0.1-P1-X1 ISA Bus + fda0 U0.1-P1-X1/D1 Standard I/O Diskette Adapter * siokma0 U0.1-P1-X1/K1 Keyboard Adapter + sioka0 U0.1-P1-X1/K1 Keyboard Adapter + sioma0 U0.1-P1-X1/K1 Keyboard Adapter + sioma0 U0.1-P1-X1/K1 Keyboard Adapter + sioma0 U0.1-P1-X1/S1 Standard I/O Serial Port + sa0 U0.1-P1-X1/S1 Standard I/O Serial Port + tty0 U0.1-P1-X1/S2 Standard I/O Serial Port + sa1 U0.1-P1-X1/S3 Standard I/O Serial Port * sa2 U0.1-P1-X1/S3 Standard I/O Serial Port * ide0 U0.1-P1-X1/Q6 ATA/IDE Controller Device + cd0 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus * pci5 U0.1-P1 PCI Bus + ses0 U0.1-P1 PCI Bus + pci5 </td <td>ł</td> <td>pci8</td> <td>U0.1-P1</td> <td>PCI Bus</td>	ł	pci8	U0.1-P1	PCI Bus							
* pci9 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * isa0 U0.1-P1 PCI Bus * isa0 U0.1-P1-X1 ISA Bus + fda0 U0.1-P1-X1/D1 Standard I/O Diskette Adapter * siokma0 U0.1-P1-X1/K1 Keyboard/Mouse Adapter * sioka0 U0.1-P1-X1/K1 Keyboard Adapter + sioma0 U0.1-P1-X1/K1 Mouse Adapter * sioma0 U0.1-P1-X1/S1 Standard I/O Serial Port * tty0 U0.1-P1-X1/S1 Standard I/O Serial Port * sa1 U0.1-P1-X1/S2 Standard I/O Serial Port * sa2 U0.1-P1-X1/S3 Standard I/O Serial Port * ide0 U0.1-P1-X1/Q6 ATA/IDE Controller Device + cd0 U0.1-P1-X1/Q6-A0 IDE DVD-ROM Drive * pci2 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus * ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus + L2cache0 L2 Cache L2 Cache + mem0 <t< td=""><td>-1</td><td>- hdlc0</td><td>U0.1-P1-I3/Q0</td><td>IBM HDLC Network Device Driver</td></t<>	-1	- hdlc0	U0.1-P1-I3/Q0	IBM HDLC Network Device Driver							
* pci10 U0.1-P1 PCI Bus * pci0 U0.1-P1 PCI Bus * isa0 U0.1-P1-X1 ISA Bus + fda0 U0.1-P1-X1/D1 Standard I/O Diskette Adapter * siokma0 U0.1-P1-X1/K1 Keyboard/Mouse Adapter + sioka0 U0.1-P1-X1/K1 Keyboard Adapter + sioma0 U0.1-P1-X1/K1 Mouse Adapter + sioma0 U0.1-P1-X1/S1 Standard I/O Serial Port + sa0 U0.1-P1-X1/S1 Standard I/O Serial Port + tty0 U0.1-P1-X1/S2 Standard I/O Serial Port + sa1 U0.1-P1-X1/S3 Standard I/O Serial Port + sa2 U0.1-P1-X1/Q6 ATA/IDE Controller Device + cd0 U0.1-P1-X1/Q6-A0 IDE DVD-ROM Drive * pci2 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus + ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus + L2cache0 L2 Cache Memory	-	- hdlc1	U0.1-P1-I3/Q1	IBM HDLC Network Device Driver							
<pre>* pci0 U0.1-P1 PCI Bus * isa0 U0.1-P1-X1 ISA Bus + fda0 U0.1-P1-X1/D1 Standard I/O Diskette Adapter * siokma0 U0.1-P1-X1/K1 Keyboard/Mouse Adapter + sioka0 U0.1-P1-X1/K1 Keyboard Adapter + sioma0 U0.1-P1-X1/K1 Mouse Adapter + sa0 U0.1-P1-X1/K1 Mouse Adapter + sa0 U0.1-P1-X1/S1 Standard I/O Serial Port + tty0 U0.1-P1-X1/S1 Standard I/O Serial Port + tty0 U0.1-P1-X1/S2 Standard I/O Serial Port + sa2 U0.1-P1-X1/S3 Standard I/O Serial Port * ide0 U0.1-P1-X1/Q6 ATA/IDE Controller Device + cd0 U0.1-P1-X1/Q6-A0 IDE DVD-ROM Drive * pci2 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus * ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus + L2cache0 Memory</pre>	ł	pci9	U0.1-P1	PCI Bus							
<pre>* isa0 U0.1-P1-X1 ISA Bus + fda0 U0.1-P1-X1/D1 Standard I/O Diskette Adapter * siokma0 U0.1-P1-X1/K1 Keyboard/Mouse Adapter + sioka0 U0.1-P1-X1/K1 Keyboard Adapter + sioma0 U0.1-P1-X1/K1 Mouse Adapter + sa0 U0.1-P1-X1/S1 Standard I/O Serial Port + tty0 U0.1-P1-X1/S1 Standard I/O Serial Port + sa1 U0.1-P1-X1/S2 Standard I/O Serial Port + sa2 U0.1-P1-X1/S3 Standard I/O Serial Port * ide0 U0.1-P1-X1/Q6 ATA/IDE Controller Device + cd0 U0.1-P1-X1/Q6-A0 IDE DVD-ROM Drive * pci2 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus * ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus + L2cache0 + mem0 Memory</pre>	4	pcil0	U0.1-P1	PCI Bus							
+ fda0U0.1-P1-X1/D1Standard I/O Diskette Adapter* siokma0U0.1-P1-X1/K1Keyboard/Mouse Adapter+ sioka0U0.1-P1-X1/K1Keyboard Adapter+ sioma0U0.1-P1-X1/K1Mouse Adapter+ sa0U0.1-P1-X1/S1Standard I/O Serial Port+ tty0U0.1-P1-X1/S1Asynchronous Terminal+ sa1U0.1-P1-X1/S2Standard I/O Serial Port+ sa2U0.1-P1-X1/S3Standard I/O Serial Port* ide0U0.1-P1-X1/Q6ATA/IDE Controller Device+ cd0U0.1-P1-X1/Q6-A0IDE DVD-ROM Drive* pci2U0.1-P1PCI Bus* pci3U0.1-P1PCI Bus* ses0U0.1-P1/Z1-AFSCSI Enclosure Services Device* pci5U0.1-P1PCI Bus+ ses0U0.1-P1PCI Bus+ ses0U0.1-P1PCI Bus+ ses0U0.1-P1PCI Bus+ mem0L2 Cache+ mem0Memory	ł	pci0	U0.1-P1	PCI Bus							
<pre>* siokma0 U0.1-P1-X1/K1 Keyboard/Mouse Adapter + sioka0 U0.1-P1-X1/K1 Keyboard Adapter + sioma0 U0.1-P1-X1/K1 Mouse Adapter + sa0 U0.1-P1-X1/S1 Standard I/O Serial Port + tty0 U0.1-P1-X1/S1 Asynchronous Terminal + sa1 U0.1-P1-X1/S2 Standard I/O Serial Port + sa2 U0.1-P1-X1/S3 Standard I/O Serial Port * ide0 U0.1-P1-X1/Q6 ATA/IDE Controller Device + cd0 U0.1-P1-X1/Q6-A0 IDE DVD-ROM Drive * pci2 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus + ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus + L2cache0 L2 Cache + mem0 Memory</pre>	ł	isa0	U0.1-P1-X1	ISA Bus							
<pre>+ sioka0 U0.1-P1-X1/K1 Keyboard Adapter + sioma0 U0.1-P1-X1/K1 Mouse Adapter + sa0 U0.1-P1-X1/S1 Standard I/O Serial Port + tty0 U0.1-P1-X1/S1-L0 Asynchronous Terminal + sa1 U0.1-P1-X1/S2 Standard I/O Serial Port + sa2 U0.1-P1-X1/S3 Standard I/O Serial Port * ide0 U0.1-P1-X1/Q6 ATA/IDE Controller Device + cd0 U0.1-P1-X1/Q6-A0 IDE DVD-ROM Drive * pci2 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus + ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus + L2cache0 L2 Cache + mem0 Memory</pre>	4	- fda0	U0.1-P1-X1/D1	Standard I/O Diskette Adapter							
<pre>+ sioma0 U0.1-P1-X1/K1 Mouse Adapter + sa0 U0.1-P1-X1/S1 Standard I/O Serial Port + tty0 U0.1-P1-X1/S1 Asynchronous Terminal + sa1 U0.1-P1-X1/S2 Standard I/O Serial Port + sa2 U0.1-P1-X1/S3 Standard I/O Serial Port * ide0 U0.1-P1-X1/Q6 ATA/IDE Controller Device + cd0 U0.1-P1-X1/Q6-A0 IDE DVD-ROM Drive * pci2 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus + ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus + L2cache0 L2 Cache + mem0 Memory</pre>	ł	siokma0	U0.1-P1-X1/K1	Keyboard/Mouse Adapter							
<pre>+ sa0 U0.1-P1-X1/S1 Standard I/O Serial Port + tty0 U0.1-P1-X1/S1-L0 Asynchronous Terminal + sa1 U0.1-P1-X1/S2 Standard I/O Serial Port + sa2 U0.1-P1-X1/S3 Standard I/O Serial Port * ide0 U0.1-P1-X1/Q6 ATA/IDE Controller Device + cd0 U0.1-P1-X1/Q6-A0 IDE DVD-ROM Drive * pci2 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus + ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus + L2cache0 L2 Cache + mem0 Memory</pre>	+	- sioka0	U0.1-P1-X1/K1	Keyboard Adapter							
+ tty0U0.1-P1-X1/S1-L0Asynchronous Terminal+ sa1U0.1-P1-X1/S2Standard I/O Serial Port+ sa2U0.1-P1-X1/S3Standard I/O Serial Port* ide0U0.1-P1-X1/Q6ATA/IDE Controller Device+ cd0U0.1-P1-X1/Q6-A0IDE DVD-ROM Drive* pci2U0.1-P1PCI Bus* pci3U0.1-P1PCI Bus* pci4U0.1-P1PCI Bus+ ses0U0.1-P1/Z1-AFSCSI Enclosure Services Device* pci5U0.1-P1PCI Bus+ hem0Memory	-	- sioma0	U0.1-P1-X1/K1	Mouse Adapter							
<pre>+ sa1 U0.1-P1-X1/S2 Standard I/O Serial Port + sa2 U0.1-P1-X1/S3 Standard I/O Serial Port * ide0 U0.1-P1-X1/Q6 ATA/IDE Controller Device + cd0 U0.1-P1-X1/Q6-A0 IDE DVD-ROM Drive * pci2 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus + ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus + L2cache0 L2 Cache + mem0 Memory</pre>	-1	- sa0	U0.1-P1-X1/S1	Standard I/O Serial Port							
<pre>+ sa2 U0.1-P1-X1/S3 Standard I/O Serial Port * ide0 U0.1-P1-X1/Q6 ATA/IDE Controller Device + cd0 U0.1-P1-X1/Q6-A0 IDE DVD-ROM Drive * pci2 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus + ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus + L2cache0 L2 Cache + mem0 Memory</pre>	+	- tty0	U0.1-P1-X1/S1-L0	Asynchronous Terminal							
<pre>* ide0 U0.1-P1-X1/Q6 ATA/IDE Controller Device + cd0 U0.1-P1-X1/Q6-A0 IDE DVD-ROM Drive * pci2 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus + ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus + L2cache0 L2 Cache + mem0 Memory</pre>	+	- sal	U0.1-P1-X1/S2	Standard I/O Serial Port							
+ cd0U0.1-P1-X1/Q6-A0IDE DVD-ROM Drive* pci2U0.1-P1PCI Bus* pci3U0.1-P1PCI Bus* pci4U0.1-P1PCI Bus+ ses0U0.1-P1/Z1-AFSCSI Enclosure Services Device* pci5U0.1-P1PCI Bus+ L2cache0L2 Cache+ mem0Memory	+	- sa2	U0.1-P1-X1/S3	Standard I/O Serial Port							
* pci2 U0.1-P1 PCI Bus * pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus + ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus + L2cache0 L2 Cache + mem0 Memory	4	ide0	U0.1-P1-X1/Q6	ATA/IDE Controller Device							
<pre>* pci3 U0.1-P1 PCI Bus * pci4 U0.1-P1 PCI Bus + ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus + L2cache0 L2 Cache + mem0 Memory</pre>			U0.1-P1-X1/Q6-A0	IDE DVD-ROM Drive							
<pre>* pci4 U0.1-P1 PCI Bus + ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus + L2cache0 L2 Cache + mem0 Memory</pre>	ł	pci2	U0.1-P1	PCI Bus							
+ ses0 U0.1-P1/Z1-AF SCSI Enclosure Services Device * pci5 U0.1-P1 PCI Bus + L2cache0 L2 Cache + mem0 Memory	4	-	U0.1-P1	PCI Bus							
* pci5 U0.1-P1 PCI Bus + L2cache0 L2 Cache + mem0 Memory	4	pci4	U0.1-P1	PCI Bus							
+ L2cache0 L2 Cache + mem0 Memory	4	- ses0	U0.1-P1/Z1-AF	SCSI Enclosure Services Device							
+ mem0 Memory	ł	pci5	U0.1-P1	PCI Bus							
*	4	- L2cache0		L2 Cache							
+ procl U0.1-P1 Processor	4	- mem0		Memory							
	4	- procl	U0.1-P1	Processor							

```
******
Detailed System Configuration
INSTALLED RESOURCE LIST WITH VPD
The following resources are installed on your machine.
 Model Architecture: chrp
 Model Implementation: Multiple Processor, PCI bus
                            System Object
 sys0
 sysplanar0
                            System Planar
 pci1
              U0.1-P1
                            PCI Bus
      Device Specific.(YL).....U0.1-P1
 pci6
              U0.1-P1
                      PCI Bus
      pci7
              U0.1-P1
                            PCI Bus
      Device Specific.(YL).....U0.1-P1
 pci8
              U0.1-P1
                            PCI Bus
      Device Specific.(YL).....U0.1-P1
 hdlc0
              U0.1-P1-I3/Q0
                           IBM HDLC Network Device Driver
                           IBM HDLC Network Device Driver
 hdlc1
              U0.1-P1-I3/Q1
              U0.1-P1
                            PCI Bus
 pci9
      ent1
              U0.1-P1/E2
                            10/100/1000 Base-TX PCI-X
Adapter (14106902)
    10/100/1000 Base-TX PCI-X Adapter:
      Part Number.....00P3056
      Manufacture ID.....YL1021
      Network Address.....00096BBE7FEE
      ROM Level (alterable).....GOL002
      Device Specific.(YL).....U0.1-P1/E2
 pci10
              U0.1-P1
                            PCI Bus
```

Listing 23-3 (continued)

```
Device Specific.(YL).....U0.1-P1
 (skipping forward)
hdisk0
         U0.1-P1/Z1-A5 16 Bit LVD SCSI Disk Drive (36400 MB)
    Manufacturer.....IBM
    Machine Type and Model.....ST336607LC
    ROS Level and ID.....43353048
    Serial Number.....000BCAF0
    Part Number.....00P2676
    Device Specific.(Z0).....000003129F00013E
    Device Specific.(Z1).....1217C511
    Device Specific.(Z2).....0002
    Device Specific.(Z3).....04216
    Device Specific. (Z4).....0001
    Device Specific. (Z5).....22
    Device Specific. (Z6).....H12094
hdisk1
         U0.1-P1/Z1-A8 16 Bit LVD SCSI Disk Drive (36400 MB)
    Manufacturer.....IBM
    Machine Type and Model.....ST336607LC
    ROS Level and ID.....43353048
    Serial Number.....000BC353
    Device Specific.(Z0).....000003129F00013E
    Device Specific. (Z1) .....1217C511
    Device Specific.(Z2).....0002
    Device Specific. (Z3) .....04216
    Device Specific.(Z4).....0001
    Device Specific. (Z5) .....22
    Device Specific. (Z6).....H12094
. (skipping forward)
PLATFORM SPECIFIC
Name: IBM, 7029-6C3
```

Listing 23-3 (continued)

```
Model: IBM, 7029-6C3
Node: /
Device Type: chrp
 Platform Firmware:
   ROM Level (alterable).....3F070425
   Version.....RS6K
   System Info Specific.(YL)...U0.1-P1-X1/Y1
 Physical Location: U0.1-P1-X1/Y1
 System Firmware:
   ROM Level (alterable).....RG070425_d79e20_regatta
   Version.....RS6K
   System Info Specific.(YL)...U0.1-P1-X1/Y2
 Physical Location: U0.1-P1-X1/Y2
 System VPD:
   Machine/Cabinet Serial No...10P1234
   Machine Type/Model.....7029-6C3
   Manufacture ID.....IBM980
   Version.....RS6K
   Op Panel Installed.....Y
   Brand.....IO
   System Info Specific.(YL)...U0.1
 Physical Location: U0.1
 PS CEC OP PANEL :
   Serial Number.....YL1124567P9D
   Customer Card ID Number....28D3
   FRU Number..... 97P3352
   Action Code, Timestamp.....BD 200210290851
   Version.....RS6K
   System Info Specific.(YL)...U0.1-L1
 Physical Location: U0.1-L1
 1 WAY BACKPLANE :
   Serial Number.....YL1124567P27
   Customer Card ID Number.....26F4
   CCIN Extender.....1
   FRU Number..... 80P2749
   Version.....RS6K
   System Info Specific.(YL)...U0.1-P1
 Physical Location: U0.1-P1
```

```
. (skipping forward)
*****
System Installed Filesets
                 Level State Type Description (Uninstaller)
Fileset
_____
X11.Dt.ToolTalk
                 5.3.0.60 C F AIX CDE ToolTalk Support
                 5.3.0.0 C F AIX CDE Bitmaps
X11.Dt.bitmaps
X11.Dt.helpmin
                 5.3.0.0 C F AIX CDE Minimum Help Files
X11.Dt.helprun
                 5.3.0.0 C F AIX CDE Runtime Help
X11.Dt.lib
                 5.3.0.60 C F AIX CDE Runtime Libraries
X11.Dt.rte
                 5.3.0.60 C F AIX Common Desktop
                                Environment (CDE) 1.0
X11.adt.motif 5.3.0.50 C F AIXwindows Application
                                 Development Toolkit Motif
X11.apps.aixterm 5.3.0.60 C F AIXwindows aixterm
                                Application
X11.apps.clients
                 5.3.0.60 C F AIXwindows Client
                                Applications
X11.apps.custom 5.3.0.0 C F AIXwindows Customizing Tool
                  5.3.0.60 C F AIXwindows Runtime
X11.apps.rte
                                Configuration Applications
X11.apps.xterm 5.3.0.0 C F AIXwindows xterm
                                Application
X11.base.common 5.3.0.0 C F AIXwindows Runtime Common
                                Directories
X11.base.lib 5.3.0.61 A F AIXwindows Runtime
                                Libraries
X11.base.rte 5.3.0.61 A F AIXwindows Runtime
                                Environment
X11.loc.en_US.base.rte 5.3.0.0 C F AIXwindows Locale
                                Configuration - U.S.
                                English
                   5.3.0.60 C F AIXwindows Motif Libraries
 X11.motif.lib
 X11.motif.mwm
                   5.3.0.60 C F AIXwindows Motif Window
                                Manager
 bos.64bit
              5.3.0.62 A F Base Operating System
                               64 bit Runtime
 bos.acct
                   5.3.0.60 C F Accounting Services
 bos.adt.base
                   5.3.0.61 A F Base Application
                                 Development Toolkit
 bos.adt.graphics
                   5.3.0.60 C F Base Application
```

```
Development Graphics
                                  Include Files
                  5.3.0.62 A F Base Application
 bos.adt.include
                                  Development Include Files
 bos.adt.lib 5.3.0.60 C F Base Application
                                  Development Libraries
 bos.adt.libm
                   5.3.0.60 C F Base Application
                                  Development Math Library
                5.3.0.62 A F Base Profiling Support
 bos.adt.prof
bos.cdmount
                    5.3.0.50 C F CD/DVD Automount Facility
bos.diag.com
                   5.3.0.62 A F Common Hardware
                                  Diagnostics
bos.diag.rte 5.3.0.62 A F Hardware Diagnostics
 . (skipping forward)
 .
                    1.9-1 C R A utility for retrieving
wget
                                  files using the HTTP or
                                  FTP protocols. (/bin/rpm)
State codes:
A -- Applied.
B -- Broken.
C -- Committed.
E -- EFIX Locked.
0 -- Obsolete. (partially migrated to newer version)
? -- Inconsistent State...Run lppchk -v.
Type codes:
F -- Installp Fileset
P -- Product
C -- Component
T -- Feature
R -- RPM Package
E -- Interim Fix
*****
Looking for Broken Filesets
*****
```

Listing 23-3 (continued)

	List of t	the last 10	00 users to login	to i:	ft()0s			
			dino	36	<u></u>	14 01		14 00	(00.01)
	randy	pts/0				14:01			(00:21)
	randy	pts/0	dino			13:34			(00:05)
	randy	pts/1	dino			09:47			(00:03)
	randy	ftp	dino			09:46			(00:01)
	root	pts/0	192.168.37.224			10:39			(23:29)
	root	pts/0	booboo			12:38			(00:01)
	root	pts/0	booboo			12:34			(00:01)
	root	pts/0	booboo			12:32			(00:00)
	root	pts/0	booboo			15:41			(00:05)
	randy	pts/1	dino			15:37			(18:43)
	root	pts/0	booboo			15:36	-	15:38	(00:01)
	reboot	~				15:34			
	randy	pts/1	dino			15:10			(00:24)
	randy	pts/1	dino			14:19			(00:30)
	randy	pts/1	dino			12:56			(00:10)
	randy	pts/1	dino			10:43			(00:08)
	randy	pts/1	dino			14:43			(00:02)
	root	pts/0	booboo			14:33			(01:20)
	root	pts/0	booboo			14:07			(00:22)
	root	pts/1	booboo						(2+21:15)
	root	pts/1	booboo			13:14			(00:06)
	randy	pts/0	dino			12:59			(02:21)
	randy	ftp	192.168.62.26			15:13			(00:00)
	randy	pts/0	dino						(1+05:41)
	randy	pts/0	dino			08:06			(00:02)
	randy	pts/1	dino	Mar 1	13	15:22	-	08:06	(16:44)
	randy	pts/1	dino			13:47			(00:33)
	root	pts/0	192.168.63.17			10:34			(00:05)
	root	pts/0	booboo			17:23			(00:19)
	root	pts/0	booboo	Feb (01	13:20	-	13:22	(00:01)
	root	pts/0	booboo	Feb (01	11:23	-	11:24	(00:01)
	randy	pts/0	192.168.63.64	Jan 2	24	16:01	-	16:01	(00:00)
	reboot	~		Jun 1	11	12:08			
	shutdown	tty0		May (05	18:19			
	root	pts/0	booboo	May (05	18:16	-	18:19	(00:02)
	reboot	~		Apr 2	24	08:03			
	shutdown	tty0		Apr 2	21	17:55			
	reboot	~		Feb 2	26	05:38			
	. (skippi	ing forward	d)						
	•								
_									

Listing 23-3 (continued)

```
root tty0 Apr 13 11:10 - down (00:00)
reboot ~ Apr 13 10:58
wtmp begins Apr 13 10:58
This report is saved in:
/usr/local/reboot/sys_snapshot.yogi.111307_122716
```

From Listing 23-3 you can see that we collected a lot of information about the system configuration. This is just a sample of what you can collect, and I will leave the specifics of the information you gather up to you. For each function that you add or change, be sure to test the response. Sometimes you may be surprised that you do not see any output. Some of the command output shown in Listing 23-3 does not have any output, because my little system does not have the hardware that the query is looking for. If you expect output and there is not any, try redirecting standard error to standard output by using the following syntax:

command 2>&1

Many commands send information-type output to standard error, specified by file descriptor 2, instead of standard output, specified by file descriptor 1. First try the command without this redirection.

Other Options to Consider

There can always be improvements to any shell script. The shell script presented in this chapter is intended to be an example of the process of gathering system information. You always want to query the system for as much information as you can. Notice that I did not add any database or application configuration/statistics gathering here. The amount of information gathered is up to you. As I said before, every shop is different, but they are all the same when troubleshooting a problem. The AIXsysconfig.ksh shell script looks only at system-level statistics and configuration, so there is a large gap that you need to fill in. This gap is where your specific application comes into play. Look at your database and application documentation for the best method of gathering information about these products. By running the configuration-gathering script at least once a week, you will save yourself a lot of effort *when* a problem arises.

Summary

In this chapter we strictly looked at AIX. The process is the same for any UNIX flavor, but the information gathered will vary in each shop. No rocket science is needed here,

but you do need a good understanding of how your system is configured. You need to understand the applications and databases and what determines a failed application. You may be looking for a set of processes, or it could be a database query with an SQL statement. These are the things that need research on your part to make this type of shell script really beneficial.

In the next chapter we are going to move on to installing, configuring, and using sudo. The sudo program stands for *super user do*, and it allows us to set up specific commands that a user can execute as root. I hope you enjoyed this chapter, and I'll see you in the next chapter!

Lab Assignment

 Bruce Hamilton's Rosetta Stone for UNIX can be found at http://bhami.com/rosetta.html. The web site shows equivalent commands for various UNIX operating systems. Write an equivalent shell script to AIXsysconfig.ksh, but substitute your particular UNIX flavor — for example, Linuxsysconfig.ksh, OpenBSDsysconfig.ksh, and so on.

CHAPTER 24

Compiling, Installing, Configuring, and Using sudo

The main job of any good Systems Administrator is to protect the *root* password. No matter how firm and diligent we are about protecting the root password, we always have the application-support group and DBAs wanting root access for one reason or another. But, alas, there is a way to give specific users the ability to run selected commands, or *all* commands, as the root user without the need to know the root password. Facilitating this restricted root access is a *free software* program called sudo, which stands for *superuser do*. In this chapter we are going to show how to compile, install, configure, and use the sudo program. You can download the current distribution at http://sudo.ws (www.sudo.org was already taken).

Because sudo is not a shell script you may be asking, "Why is sudo included in *this* book?" I am including the sudo chapter because I have not found any reference to sudo in any scripting book, and it is a nice tool to use. We will cover a short shell script at the end of this chapter showing how to use sudo in a shell script.

The Need for sudo

In UNIX the root user is almighty and has absolutely no restrictions. All security is bypassed, and anyone with root access can perform any task, with some possibly resulting in major damage to the system, without any restrictions at all. UNIX systems do *not* ask, "Are you sure?" They just run the command specified by the root user and assume you know what you are doing. The sudo program allows the Systems Administrator to set up specific commands (or all commands) to be executed as the root user and specify only certain users (or groups of users) to execute the individual commands. Additionally, with sudo we have the ability to execute commands to *run as* another user. In addition, all commands and command arguments are logged either to a defined file, the system **syslog**, or a central syslog server, or as many as you configure for each system. The logging allows the Systems Administrator to have an

audit trail and to monitor user sudo activities as well as failed sudo attempts! The user executes a restricted command by preceding the command with the word sudo, as in this example:

sudo chmod 600 /etc/sudoers
Password:

When a user executes the preceding command, a password prompt is displayed. The password that the sudo program is asking for is *not* the root password but the password of the user who wants root access. When the password is entered, the /etc/sudoers file is searched to determine if root authority should be granted to run the specified command. If both the password is correct and the /etc/sudoers search grants access, the command will execute with root authority. After this initial sudo command, the user may submit more sudo commands without the need for a password until a sudo timeout, typically five minutes without issuing another sudo command. After the timeout period the user will again be prompted for his or her password when a sudo command is entered. We can also configure commands to be executed in sudo without a password. We will look at each method.

Configuring sudo on Solaris

If you are running Solaris, sudo should already be installed on the system. If you cannot find the files in any of the normal locations, you need to add a few soft links. To get sudo working on Solaris, you need to link the /opt/csw/bin/sudo file to /usr/bin/sudo. Then you need to link the /opt/csw/etc/sudoers file to /etc/sudoers. Finally, you need to link the /opt/csw/sbin/visudo file to /usr/sbin/visudo. You create these soft links using the following commands:

```
ln -s /opt/csw/bin/sudo /usr/bin/sudo
ln -s /opt/csw/etc/sudoers /etc/sudoers
ln -s /opt/csw/sbin/visudo /usr/sbin/visudo
```

After creating these links, as root, edit the /etc/sudoers file using visudo. For details, see the "Configuring sudo" section later in this chapter.

Downloading and Compiling sudo

The sudo program can be downloaded from various FTP mirror sites. The main sudo download web site is located at http://sudo.ws/sudo/download.html. The sudo program is *free software* and is distributed under a ISC-style license. As of this writing the current stable version of sudo is 1.6.9p9 and was released December 3, 2007. Todd Miller currently maintains the sudo program, and if you would like to tip Todd for his fine work, you can do so by clicking the Make a Donation button on the sudo main page (http://sudo.ws).

There are two ways to download the files. You can download the precompiled binaries for your UNIX flavor and version or download the source code distribution and compile the sudo program for your particular machine. I always download the source code and compile it on each individual system. The process takes just a few minutes, and you can be assured that it will run on your system. If you have a boatload of systems to install sudo on, you may want to consider using the precompiled binaries and pushing the binaries out to each system, or writing a shell script to push and install the product! Either way you choose, you will need only about 6 MB of free space to work with. Once sudo is installed you can remove the downloaded files if you need to regain the disk space. In this chapter we are going to download the source code and compile sudo for a particular system.

Compiling sudo

To compile sudo, you will need a C compiler. gcc is preferred, but cc works fine. You can download gcc from one of the mirror sites listed at http://gcc.gnu.org/ mirrors.html.

The sudo source code distribution is in a compressed tar format, where gzip is used for compression. A file compressed using gzip has a .gz extension — for example, sudo-1.6.9p9.tar.gz. When you download the file, put the software distribution in a directory that has at least 6 MB of free space. In our example we will use /usr/local, which is a separate filesystem from /usr on my machine. You *must* have root access to compile, install, and configure sudo!

After the sudo distribution file is placed in a work directory, the first step is to uncompress the source code tarball file:

```
[root@yogi local]# ls -l sudo-1.6.9p9.tar.gz
-rw-r--r-- 1 root root 581462 Dec 13 12:01 sudo-1.6.9p9.tar.gz
[root@yogi local]# gunzip sudo-1.6.9p9.tar.gz
[root@yogi local]# ls -l sudo-1.6.9p9.tar
-rw-r--r-- 1 root root 2611200 Dec 13 12:01 sudo-1.6.9p9.tar
```

The compressed sudo distribution file is 581,462 bytes. After the gunzip command uncompresses the file, it grows to 2,611,200 bytes. This tells us that gzip produces a compression ratio of about 4:1 for this particular archive.

When we *untar* the archive, a subdirectory will be created called sudo-1.6.9p9 that will contain all of the source code, license, readme, manuals, configure, and Makefile files. In the directory containing the sudo-1.6.6.tar file, in our case /usr/local, issue the following command:

tar -xvf sudo-1.6.9p9.tar

The extraction of the files is shown in Listing 24-1.

```
[root@yogi local]# tar -xvf sudo-1.6.9p9.tar
sudo-1.6.9p9/alloc.c
sudo-1.6.9p9/alloca.c
sudo-1.6.9p9/check.c
```

Listing 24-1 Untarring the sudo distribution

sudo-1.6.9p9/closefrom.c sudo-1.6.9p9/def_data.c sudo-1.6.9p9/defaults.c sudo-1.6.9p9/env.c sudo-1.6.9p9/err.c sudo-1.6.9p9/fileops.c sudo-1.6.9p9/find_path.c sudo-1.6.9p9/fnmatch.c sudo-1.6.9p9/getcwd.c sudo-1.6.9p9/getprogname.c sudo-1.6.9p9/getspwuid.c sudo-1.6.9p9/gettime.c sudo-1.6.9p9/glob.c sudo-1.6.9p9/goodpath.c sudo-1.6.9p9/interfaces.c sudo-1.6.9p9/ldap.c sudo-1.6.9p9/lex.yy.c sudo-1.6.9p9/lsearch.c sudo-1.6.9p9/logging.c sudo-1.6.9p9/memrchr.c sudo-1.6.9p9/mkstemp.c sudo-1.6.9p9/parse.c sudo-1.6.9p9/parse.lex sudo-1.6.9p9/parse.yacc sudo-1.6.9p9/set_perms.c sudo-1.6.9p9/sigaction.c sudo-1.6.9p9/snprintf.c sudo-1.6.9p9/strcasecmp.c sudo-1.6.9p9/strerror.c sudo-1.6.9p9/strlcat.c sudo-1.6.9p9/strlcpy.c sudo-1.6.9p9/sudo.c sudo-1.6.9p9/sudo_noexec.c sudo-1.6.9p9/sudo.tab.c sudo-1.6.9p9/sudo_edit.c sudo-1.6.9p9/testsudoers.c sudo-1.6.9p9/tgetpass.c sudo-1.6.9p9/utimes.c sudo-1.6.9p9/visudo.c sudo-1.6.9p9/zero_bytes.c sudo-1.6.9p9/auth/afs.c sudo-1.6.9p9/auth/aix_auth.c sudo-1.6.9p9/auth/bsdauth.c sudo-1.6.9p9/auth/dce.c sudo-1.6.9p9/auth/fwtk.c sudo-1.6.9p9/auth/kerb4.c sudo-1.6.9p9/auth/kerb5.c sudo-1.6.9p9/auth/pam.c

Listing 24-1 (continued)

sudo-1.6.9p9/auth/passwd.c sudo-1.6.9p9/auth/rfc1938.c sudo-1.6.9p9/auth/secureware.c sudo-1.6.9p9/auth/securid.c sudo-1.6.9p9/auth/securid5.c sudo-1.6.9p9/auth/sia.c sudo-1.6.9p9/auth/sudo_auth.c sudo-1.6.9p9/compat.h sudo-1.6.9p9/def_data.h sudo-1.6.9p9/defaults.h sudo-1.6.9p9/ins_2001.h sudo-1.6.9p9/ins_classic.h sudo-1.6.9p9/ins_csops.h sudo-1.6.9p9/ins_goons.h sudo-1.6.9p9/insults.h sudo-1.6.9p9/interfaces.h sudo-1.6.9p9/logging.h sudo-1.6.9p9/parse.h sudo-1.6.9p9/sudo.h sudo-1.6.9p9/sudo.tab.h sudo-1.6.9p9/version.h sudo-1.6.9p9/auth/sudo_auth.h sudo-1.6.9p9/emul/err.h sudo-1.6.9p9/emul/fnmatch.h sudo-1.6.9p9/emul/search.h sudo-1.6.9p9/emul/utime.h sudo-1.6.9p9/emul/glob.h sudo-1.6.9p9/emul/timespec.h sudo-1.6.9p9/BUGS sudo-1.6.9p9/CHANGES sudo-1.6.9p9/HISTORY sudo-1.6.9p9/INSTALL sudo-1.6.9p9/INSTALL.configure sudo-1.6.9p9/LICENSE sudo-1.6.9p9/Makefile.in sudo-1.6.9p9/PORTING sudo-1.6.9p9/README sudo-1.6.9p9/README.LDAP sudo-1.6.9p9/TROUBLESHOOTING sudo-1.6.9p9/UPGRADE sudo-1.6.9p9/aclocal.m4 sudo-1.6.9p9/acsite.m4 sudo-1.6.9p9/aixcrypt.exp sudo-1.6.9p9/config.guess sudo-1.6.9p9/config.h.in sudo-1.6.9p9/config.sub sudo-1.6.9p9/configure sudo-1.6.9p9/configure.in

Listing 24-1 (continued)

sudo-1.6.9p9/def_data.in sudo-1.6.9p9/fnmatch.3 sudo-1.6.9p9/indent.pro sudo-1.6.9p9/install-sh sudo-1.6.9p9/ltmain.sh sudo-1.6.9p9/mkdefaults sudo-1.6.9p9/mkinstalldirs sudo-1.6.9p9/pathnames.h.in sudo-1.6.9p9/sample.pam sudo-1.6.9p9/sample.syslog.conf sudo-1.6.9p9/sample.sudoers sudo-1.6.9p9/schema.OpenLDAP sudo-1.6.9p9/schema.iPlanet sudo-1.6.9p9/sudo.cat sudo-1.6.9p9/sudo.man.in sudo-1.6.9p9/sudo.pod sudo-1.6.9p9/sudoers sudo-1.6.9p9/sudoers.cat sudo-1.6.9p9/sudoers.man.in sudo-1.6.9p9/sudoers.pod sudo-1.6.9p9/sudoers2ldif sudo-1.6.9p9/visudo.cat sudo-1.6.9p9/visudo.man.in sudo-1.6.9p9/visudo.pod sudo-1.6.9p9/auth/API

Listing 24-1 (continued)

As you can see in Listing 24-1, a new subdirectory was created called sudo-1-6-9p9. After the program distribution file is uncompressed and untarred, we can proceed to the installation process. This is not a difficult process, so if you have never worked with the make command and Makefile before, don't worry. The first step is to configure the Makefile for your system. As you might expect, this is done with the configure command. The configure command comes with the sudo distribution. First, change directories to where the source code is located, in our example /usr/local/sudo-1.6.9p9, and run the configure command:

cd /usr/local/sudo-1.6.9p9
./configure

The configure command goes through system checks and builds a Makefile and the config.h file used to build sudo for your system. The configure command output for my Linux Fedora system is shown in Listing 24-2.

```
[root@yogi sudo-1.6.9p9]# ./configure
configure: Configuring Sudo version 1.6.9
checking whether to lecture users the first time they run sudo... yes
```

Listing 24-2 Example of running the configure command

```
checking whether sudo should log via syslog or to a file by default...
syslog
checking which syslog facility sudo should log with... local2
checking at which syslog priority to log commands... notice
checking at which syslog priority to log failures... alert
checking how long a line in the log file should be... 80
checking whether sudo should ignore '.' or '' in $PATH... no
checking whether to send mail when a user is not in sudoers... yes
checking whether to send mail when user listed but not for this host...
no
checking whether to send mail when a user tries a disallowed command...
no
checking who should get the mail that sudo sends... root
checking for bad password prompt... Password:
checking for bad password message ... Sorry, try again.
checking whether to expect fully qualified hosts in sudoers... no
checking for umask programs should be run with... 0022
checking for default user to run commands as ... root
checking for editor that visudo should use ... vi
checking whether to obey EDITOR and VISUAL environment variables... no
checking number of tries a user gets to enter their password... 3
checking time in minutes after which sudo will ask for a password
again... 5
checking time in minutes after the password prompt will time out... 5
checking whether to use per-tty ticket files ... no
checking whether to include insults... no
checking whether to override the user's path ... no
checking whether to get ip addresses from the network interfaces... yes
checking whether stow should be used ... no
checking whether to do user authentication by default... yes
checking whether to disable running the mailer as root... no
checking whether to disable shadow password support... no
checking whether root should be allowed to use sudo... yes
checking whether to log the hostname in the log file ... no
checking whether to invoke a shell if sudo is given no arguments... no
checking whether to set $HOME to target user in shell mode... no
checking whether to disable 'command not found' messages... no
checking for egrep... egrep
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling ... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler ... yes
checking whether gcc accepts -g... (cached) no
checking for gcc option to accept ISO C89... none needed
checking for library containing strerror... none required
```

```
Listing 24-2 (continued)
```

checking how to run the C preprocessor... gcc -E checking build system type... i686-pc-linux-gnu checking host system type... i686-pc-linux-gnu checking target system type... i686-pc-linux-gnu checking for a sed that does not truncate output... /bin/sed checking for grep that handles long lines and -e... /bin/grep checking for egrep... /bin/grep -E checking for 1d used by gcc... /usr/bin/ld checking if the linker (/usr/bin/ld) is GNU ld... yes checking for /usr/bin/ld option to reload object files... -r checking for BSD-compatible nm... /usr/bin/nm -B checking whether ln -s works... yes checking how to recognize dependent libraries... pass_all checking for ANSI C header files... yes checking for sys/types.h... yes checking for sys/stat.h... yes checking for stdlib.h... yes checking for string.h... yes checking for memory.h... yes checking for strings.h... yes checking for inttypes.h... yes checking for stdint.h... yes checking for unistd.h... yes checking dlfcn.h usability... yes checking dlfcn.h presence... yes checking for dlfcn.h... yes checking the maximum length of command line arguments... 98304 checking command to parse /usr/bin/nm -B output from gcc object... ok checking for objdir... .libs checking for ar... ar checking for ranlib... ranlib checking for strip... strip checking if gcc supports -fno-rtti -fno-exceptions... no checking for gcc option to produce PIC... -fPIC checking if gcc PIC flag -fPIC works... yes checking if gcc static flag -static works ... yes checking if gcc supports -c -o file.o... yes checking whether the gcc linker (/usr/bin/ld) supports shared libraries ... yes checking whether -lc should be explicitly linked in... no checking dynamic linker characteristics... GNU/Linux ld.so checking how to hardcode library paths into programs... immediate checking whether stripping libraries is possible ... yes checking if libtool supports shared libraries... yes checking whether to build shared libraries... yes checking whether to build static libraries... no configure: creating libtool checking path to sudo_noexec.so... \${exec_prefix}/libexec/sudo_noexec.so

Listing 24-2 (continued)

```
checking for uname... uname
checking for tr... tr
checking for nroff... nroff
checking whether gcc needs -traditional... no
checking for an ANSI C-conforming const... yes
checking for working volatile ... yes
checking for bison... bison -y
checking for mv... /bin/mv
checking for bourne shell... /bin/sh
checking for sendmail... /usr/sbin/sendmail
checking for vi... /bin/vi
checking for ANSI C header files... (cached) yes
checking for dirent.h that defines DIR... yes
checking for library containing opendir... none required
checking whether time.h and sys/time.h may both be inclued... yes
checking malloc.h usability... yes
checking malloc.h presence... yes
checking for malloc.h... yes
checking paths.h usability... yes
checking paths.h presence... yes
checking for paths.h... yes
checking utime.h usability... yes
checking utime.h presence... yes
checking for utime.h... yes
checking netgroup.h usability... no
checking netgroup.h presence... no
checking for netgroup.h... no
checking sys/sockio.h usability... no
checking sys/sockio.h presence... no
checking for sys/sockio.h... no
checking sys/bsdtypes.h usability... no
checking sys/bsdtypes.h presence... no
checking for sys/bsdtypes.h... no
checking sys/select.h usability... yes
checking sys/select.h presence... yes
checking for sys/select.h... yes
checking err.h usability... yes
checking err.h presence... yes
checking for err.h... yes
checking POSIX termios... yes
checking for mode_t... yes
checking for uid_t in sys/types.h... yes
checking for sig_atomic_t... yes
checking for sigaction_t... no
checking for struct timespec... yes
checking for struct in6_addr... yes
checking for size_t... yes
checking for ssize_t... yes
```

checking for dev_t... yes checking for ino_t... yes checking for full void implementation... yes checking max length of uid_t... 10 checking for long long ... yes checking for long and long long equivalence... no checking for sa_len field in struct sockaddr... no checking return type of signal handlers... void checking type of array argument to getgroups... gid_t checking for size_t... yes checking for getgroups... yes checking for working getgroups... yes checking for strchr... yes checking for strrchr... yes checking for memchr... yes checking for memcpy... yes checking for memset... yes checking for sysconf... yes checking for tzset... yes checking for strftime... yes checking for setrlimit ... yes checking for initgroups... yes checking for getgroups... (cached) yes checking for fstat... yes checking for gettimeofday... yes checking for setlocale ... yes checking for getaddrinfo... yes checking for setresuid ... yes checking for seteuid... yes checking for getifaddrs... yes checking for freeifaddrs... yes checking for getcwd... yes checking for glob... yes checking for GLOB_BRACE and GLOB_TILDE in glob.h... yes checking for lockf... yes checking for waitpid ... yes checking for innetgr... yes checking for getdomainname... yes checking for lsearch... yes checking for utimes... yes checking for futimes... yes checking for working fnmatch with FNM_CASEFOLD... yes checking for isblank ... yes checking for memrchr... yes checking for strerror... yes checking for strcasecmp... yes checking for sigaction ... yes checking for strlcpy... no

```
checking for strlcat... no
checking for closefrom... no
checking whether F_CLOSEM is declared... no
checking for mkstemp... yes
checking for snprintf... yes
checking for vsnprintf... yes
checking for asprintf... yes
checking for vasprintf... yes
checking for struct stat.st_mtim... yes
checking for two-parameter timespecsub... no
checking for socket... yes
checking for inet_addr... yes
checking for syslog... yes
checking for working alloca.h... yes
checking for alloca... yes
checking for getprogname... no
checking for __progname... yes
checking for main in -ldl... yes
checking security/pam_appl.h usability... yes
checking security/pam_appl.h presence... yes
checking for security/pam_appl.h... yes
checking whether to use PAM session support ... yes
checking for dgettext... yes
checking for log file location... /var/log/sudo.log
checking for timestamp file location... /var/run/sudo
configure: using the following authentication methods: pam
configure: creating ./config.status
config.status: creating Makefile
config.status: creating sudo.man
config.status: creating visudo.man
config.status: creating sudoers.man
config.status: creating config.h
config.status: creating pathnames.h
configure: You will need to customize sample.pam and install it as
/etc/pam.d/sudo
```

After the configure command completes without error, you have a customized Makefile for your system. You can, if you need to, edit the Makefile and change the default paths and the compiler to use. Now that we have a new customized Makefile we can now compile the sudo program on the system. Issue the following command, assuming you are still in the /usr/local/sudo-1.6.9p9 directory:

make

The **make** command is located in /usr/bin/make on most systems, and it uses the Makefile in the current directory to compile, in our case /usr/local/sudo-1.6.9p9/Makefile. The make command output is shown in Listing 24-3. Notice that my system uses the gcc compiler.

```
[root@yogi sudo-1.6.9p9]# make
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 check.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 env.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 getspwuid.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS GID=0 -DSUDOERS MODE=0440 gettime.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 goodpath.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 fileops.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 find_path.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 interfaces.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 logging.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 parse.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 set_perms.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 sudo.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 sudo_edit.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 tgetpass.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 zero_bytes.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 ./auth/sudo_auth.c
```

Listing 24-3 Command output – make command

```
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 ./auth/pam.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 sudo.tab.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 lex.yy.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 alloc.c
gcc -c -I. -I. -O2 -D GNU SOURCE -D PATH SUDDERS=\"/etc/sudders\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 defaults.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 strlcpy.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 strlcat.c
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 closefrom.c
gcc -o sudo check.o env.o getspwuid.o gettime.o goodpath.o fileops.o
find_path.o interfaces.o logging.o parse.o set_perms.o sudo.o
sudo_edit.o tgetpass.o zero_bytes.o sudo_auth.o pam.o
sudo.tab.o lex.yy.o alloc.o defaults.o strlcpy.o strlcat.o closefrom.o
-lpam -ldl
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\"
-D_PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0
-DSUDOERS_GID=0 -DSUDOERS_MODE=0440 visudo.c
gcc -o visudo visudo.o fileops.o gettime.o goodpath.o find_path.o sudo.
tab.o lex.yy.o alloc.o defaults.o strlcpy.o strlcat.o closefrom.o
/bin/sh ./libtool --mode=compile gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_
PATH_SUDOERS=\"/ etc/sudoers\" -D_PATH_SUDOERS_TMP=\"/etc/
sudoers.tmp\" -DSUDOERS_UID=0 -DSUDOERS_GID=0 -DSUDOERS_MODE=0440
./sudo_noexec.c
mkdir .libs
gcc -c -I. -I. -O2 -D_GNU_SOURCE -D_PATH_SUDOERS=\"/etc/sudoers\" -D_
PATH_SUDOERS_TMP=\"/etc/sudoers.tmp\" -DSUDOERS_UID=0 -DSUDOERS_GID=0
-DSUDOERS_MODE=0440 ./sudo_noexec.c -fPIC -o .libs/sudo_noexec.o
/bin/sh ./libtool --mode=link gcc -o sudo_noexec.la sudo_noexec.lo
-avoid -version -rpath /usr/local/libexec
gcc -shared .libs/sudo_noexec.o -Wl,-soname -Wl,sudo_noexec.so -o
.libs/sudo_noexec.so
creating sudo_noexec.la
(cd .libs && rm -f sudo_noexec.la && ln -s ../sudo_noexec.la
sudo_noexec.la)
```

After the make command completes, we have custom compiled code for our system, but we still have one more installation step to complete before we are ready to configure sudo. This last step is to install the compiled files created with the make command. The next command handles the installation of sudo:

make install

Remember that the make command is usually located in /usr/bin and should be in your \$PATH. The output of the make install command for my machine is shown in Listing 24-4.

```
[root@yogi sudo-1.6.9p9]# make install
/bin/sh ./mkinstalldirs /usr/local/bin \
    /usr/local/sbin /etc \
    /usr/local/share/man/man8 /usr/local/share/man/man5 \
    /usr/local/libexec
/bin/sh ./install-sh -c -O O -G O -M 4111 -s sudo /usr/local/bin/sudo
rm -f /usr/local/bin/sudoedit
ln /usr/local/bin/sudo /usr/local/bin/sudoedit
/bin/sh ./install-sh -c -O O -G O -M 0111 -s visudo /usr/local/sbin/
visudo
/bin/sh ./libtool --mode=install /bin/sh ./install-sh -c sudo_
noexec.la /usr/local/libexec
/bin/sh ./install-sh -c .libs/sudo_noexec.so /usr/local/libexec/
sudo_noexec.so
/bin/sh ./install-sh -c .libs/sudo_noexec.lai /usr/local/libexec/
sudo noexec.la
test -f /etc/sudoers || \
    /bin/sh ./install-sh -c -O 0 -G 0 -M 0440 \
      ./sudoers /etc/sudoers
/bin/sh ./install-sh -c -O 0 -G 0 -M 0444 ./sudo.man /usr/local/share/
man/man8/sudo.8
ln /usr/local/share/man/man8/sudo.8 /usr/local/share/man/man8/
sudoedit.8
/bin/sh ./install-sh -c -O 0 -G 0 -M 0444 ./visudo.man /usr/local/share/
man/man8/visudo.8
/bin/sh ./install-sh -c -O 0 -G 0 -M 0444 ./sudoers.man /usr/local/share/
man/man5/sudoers.5
```

Listing 24-4 Command output – make install

If you did not have any failures during the compilation and installation processes, then sudo is installed but not yet configured. In the next section we will look at two sample configuration files.

Configuring sudo

Configuring sudo is where a lot of people get a bit confused. The configuration is not too difficult if you take small steps and test each part as you build the configuration

file. If you look in the /etc directory after the installation is complete, you will see a file called sudgers. The sudgers file is used to configure the commands and users for the sudgers program. Be very careful to *never directly edit* the sudgers file! A special program is supplied that has a wrapper around the vi editor called **visudo**, or *vi sudo*.

The visudo program resides in /usr/local/sbin by default. The nice thing about visudo is that it checks the /etc/sudoers file for any errors before saving the file. If errors are detected, the visudo program will tell you exactly what the error is, and in most cases the line the error is on. If you directly edit the /etc/sudoers file and you make a mistake, the editor will just let you save the file, with the mistake, and it can be difficult to find the error. If you save the sudoers file with an error in it, sudo may not work at all for any users. The visudo program checks for the correct file format and ensures that the command/user references are consistent. If you make a mistake with a username, the visudo editor will not catch the mistake, but this type of error should be easy to find and correct after an initial run.

I am presenting enclosing two samples of a /etc/sudoers file for you to use as a template in Listings 24-5 and 24-6.

NOTE The sudgers file in Listing 24-5 is used with the permission of Todd Miller at http://sudg.ws and is included in the sudg distribution as a sample. Thank you, Todd!

```
# Sample /etc/sudoers file.
 This file MUST be edited with the 'visudo' command as root.
# See the sudoers man page for the details on how to write a sudoers file.
#
##
# User alias specification$
##
User_Alias FULLTIMERS = millert, mikef, dowdy
User_Alias PARTTIMERS = bostley, jwfox, crawl
User_Alias WEBMASTERS = will, wendy, wim
##
# Runas alias specification
##
Runas_Alias OP = root, operator
Runas_Alias DB = oracle, sybase
##
# Host alias specification
##
```

Listing 24-5 Sample /etc/sudoers file #1

```
SPARC = bigtime, eclipse, moet, anchor:\
Host_Alias
             SGI = grolsch, dandelion, black:\
             ALPHA = widget, thalamus, foobar:\
             HPPA = boa, nag, python
Host_Alias CUNETS = 128.138.0.0/255.255.0.0
Host_Alias CSNETS = 128.138.243.0, 128.138.204.0/24, 128.138.242.0
Host_Alias SERVERS = master, mail, www, ns
Host_Alias CDROM = orion, perseus, hercules
##
# Cmnd alias specification
##
Cmnd Alias
            DUMPS = /usr/sbin/dump, /usr/sbin/rdump, /usr/sbin/
restore, \setminus
                     /usr/sbin/rrestore, /usr/bin/mt
Cmnd_Alias KILL = /usr/bin/kill
Cmnd_Alias PRINTING = /usr/sbin/lpc, /usr/bin/lprm
Cmnd_Alias SHUTDOWN = /usr/sbin/shutdown
Cmnd_Alias HALT = /usr/sbin/halt, /usr/sbin/fasthalt
Cmnd_Alias REBOOT = /usr/sbin/reboot, /usr/sbin/fastboot
Cmnd_Alias SHELLS = /usr/bin/sh, /usr/bin/csh, /usr/bin/ksh, \
                      /usr/local/bin/tcsh, /usr/bin/rsh, \
                      /usr/local/bin/zsh
Cmnd_Alias SU = /usr/bin/su
Cmnd_Alias VIPW = /usr/sbin/vipw, /usr/bin/passwd, /usr/bin/chsh, \
                    /usr/bin/chfn
##
# Override builtin defaults
##
Defaults
                       syslog=auth
Defaults:FULLTIMERS
                       !lecture
Defaults:millert
                       !authenticate
Defaults@SERVERS
                      log_year, logfile=/var/log/sudo.log
##
# User specification
##
# root and users in group wheel can run anything on any machine
# as any user
root
             ALL = (ALL) ALL
%wheel
             ALL = (ALL) ALL
# full time sysadmins can run anything on any machine without a password
FULLTIMERS
             ALL = NOPASSWD: ALL
```

Listing 24-5 (continued)

```
# part time sysadmins may run anything but need a password
PARTTIMERS
           ALL = ALL
# jack may run anything on machines in CSNETS
jack
            CSNETS = ALL
# lisa may run any command on any host in CUNETS (a class B network)
             CUNETS = ALL
lisa
# operator may run maintenance commands and anything in /usr/oper/bin/
            ALL = DUMPS, KILL, PRINTING, SHUTDOWN, HALT, REBOOT, \
operator
             /usr/oper/bin/
# joe may su only to operator
joe
            ALL = /usr/bin/su operator
# pete may change passwords for anyone but root on the hp snakes
pete
            HPPA = /usr/bin/passwd [A-z]*, !/usr/bin/passwd root
# bob may run anything on the sparc and sgi machines as any user
# listed in the Runas_Alias "OP" (ie: root and operator)
bob
            SPARC = (OP) ALL : SGI = (OP) ALL
# jim may run anything on machines in the biglab netgroup
            +biglab = ALL
iim
# users in the secretaries netgroup need to help manage the printers
# as well as add and remove users
+secretaries ALL = PRINTING, /usr/bin/adduser, /usr/bin/rmuser
# fred can run commands as oracle or sybase without a password
            ALL = (DB) NOPASSWD: ALL
fred
# on the alphas, john may su to anyone but root and flags are not allowed
            ALPHA = /usr/bin/su [!-]*, !/usr/bin/su *root*
john
# jen can run anything on all machines except the ones
# in the "SERVERS" Host Alias
            ALL, !SERVERS = ALL
jen
# jill can run any commands in the directory /usr/bin/, except for
# those in the SU and SHELLS aliases.
             SERVERS = /usr/bin/, !SU, !SHELLS
jill
# steve can run any command in the directory /usr/local/op_commands/
# as user operator.
steve
            CSNETS = (operator) /usr/local/op_commands/
```

```
# matt needs to be able to kill things on his workstation when
# they get hung.
matt valkyrie = KILL
# users in the WEBMASTERS User_Alias (will, wendy, and wim)
# may run any command as user www (which owns the web pages)
# or simply su to www.
WEBMASTERS www = (www) ALL, (root) /usr/bin/su www
# anyone can mount/unmount a CD-ROM on the machines in the CDROM alias
ALL CDROM = NOPASSWD: /sbin/umount /CDROM,\
/sbin/mount -o nosuid\,nodev /dev/cd0a /CDROM
```

I want to point out how we set up specific commands to be executed without a password. The NOPASSWD: specification tells sudo to not prompt for a password for the specified commands. I extracted the following definition for the User_Alias FULLTIMERS:

```
# full time sysadmins can run anything on any machine without a password
FULLTIMERS ALL = NOPASSWD: ALL
```

The statement says to allow all the users defined in the FULLTIMERS sudo User_Alias to execute ALL commands without being prompted by sudo for a password. To allow a user to execute the vi command with root authority without a password, use the following syntax in the /etc/sudoers file:

```
# root_editors can edit any file with vi on the machine without a password
ROOTEDITOR VI = NOPASSWD: /usr/bin/vi
```

Let's look at the second sample /etc/sudoers file, shown in Listing 24-6.

```
# sudoers file.
#
# This file MUST be edited with the 'visudo' command as root.
#
# See the sudoers man page for the details on how to write a sudoers file.
#
# Users Identification:
#
# All ROOT access:
#
# d7742 - Michael
#
# Restricted Access to: mount umount and exportfs
#
#
```

Listing 24-6 Sample /etc/sudoers file #2

```
# Restricted Access to: Start and stop Fasttrack Web Server
#
# d3920 - Park
# d7525 - Brinker
# d7794 - Doan
# Restricted OPERATIONS access
#
# d6331 - Sutter
# d6814 - Martin
# d8422 - Smith
# d9226 - Milando
# d9443 - Summers
# d0640 - Lawson
# d2105 - Fanchin
# d2188 - Grizzle
# d3408 - Foster
# d3551 - Dennis
# d3883 - Nations
# d6290 - Alexander
# d2749 - Mayo
# d6635 - Wright
# d3916 - Chatman
# d6782 - Scott
# d6810 - Duckery
# d6811 - Wells
# d6817 - Gilliam
# d5123 - Crynick
# d7504 - Davis
# d7505 - McCaskey
# d7723 - Rivers
# Host alias specification
Host_Alias LOCAL=yogi
# User alias specification
User_Alias NORMAL=d7742,d7537,d7526,d6029,d7204,d1076,d7764,d7808
User_Alias ADMIN=e17742,d7211,d6895,d8665,d7539,b003
User_Alias ORACLE=d7742
User Alias SAP=d7742
User_Alias OPERATOR=d7742,d6895,d6331,d6814,d8422,d9226,d9443,d0640,
d2105, d2188, d3408, d3551, d3883, d6290, d2749, d6635, d3916, d6782, d6810,
d6811,d6817,d5123,d7504,d7505,d7723
User_Alias FASTTRACK=d3920,d7525,d7794
# Cmnd alias specification
```

www.it-ebooks.info

```
Cmnd_Alias MNT=/usr/bin/mount
Cmnd_Alias UMNT=/usr/bin/umount
Cmnd_Alias EXP_FS=/usr/bin/exportfs
Cmnd_Alias KILL=/usr/bin/kill
Cmnd_Alias ROOT_SU=/usr/bin/su -
Cmnd_Alias SU_ROOT=/usr/bin/su - root
Cmnd_Alias SUROOT=/usr/bin/su root
Cmnd_Alias ORACLE_SU=/usr/bin/su - oracle
Cmnd_Alias SAP_SU=/usr/bin/su - sap
Cmnd_Alias TCPDUMP=/usr/sbin/tcpdump
Cmnd_Alias ERRPT=/usr/bin/errpt
Cmnd Alias SVRMGRL=/oracle/product/8.0.5/bin/svrmgrl
Cmnd_Alias RSH_UPDATE=/usr/local/bin/rsh_update.ksh
Cmnd_Alias START_FT_YOGI=/usr/netscape/httpd-yogi/start
Cmnd_Alias STOP_FT_YOGI=/usr/netscape/httpd-yogi/stop
Cmnd_Alias START_FT_DINO=/usr/netscape/httpd-dino/start
Cmnd_Alias STOP_FT_DINO=/usr/netscape/httpd-dino/stop
Cmnd_Alias START_WSADM=/usr/netscape/start-admin
Cmnd_Alias STOP_WSADM=/usr/netscape/stop-admin
# User privilege specification
# FULL ROOT ACCESS!!!!!! (BE CAREFUL GRANTING FULL ROOT!!!!!!)
root ALL=(ALL) ALL
# This next user is not prompted for a password by sudo
d7742 ALL= NOPASSWD: ALL # Michael
# Only mount, umount and exportfs
NORMAL
            LOCAL=MNT, UMNT, EXP_FS
# Some Limited Sys Admin Functions
ADMIN
            LOCAL=MNT, UMNT, KILL, ORACLE_SU, SAP_SU, TCPDUMP, ERRPT,
ROOT_SU: \
      LOCAL=SU_ROOT, SUROOT, EXP_FS
# Some Operator Functions
OPERATOR LOCAL=RSH UPDATE
# Some FastTrack/WebAdm Functions
FASTTRACK LOCAL=START_FT_E1, STOP_FT_E1, START_FT_E2, STOP_FT_E2,
START WSADM,
STOP_WSADM
# Override Defaults
# Change the default location of the SUDO log file
Defaults
            logfile=/var/adm/sudo.log
```

Listing 24-6 (continued)

As you can see by the two sample /etc/sudoers files, you can get as detailed as you want. As you look at these files, notice that there are four kinds of aliases: User_Alias, Runas_Alias, Host_Alias, and Cmnd_Alias. The use of each alias type is listed next.

A User_Alias is a list that can contain any combination of usernames, UID (with a "#" prefix), system groups (with a "%" prefix), netgroups (with a "+" prefix), and other user-defined aliases. Any of these can be prefixed with the NOT operator, "!", to negate the entry.

A Runas_Alias can contain any of the same elements as the User_Alias; the only difference is that you use Runas_Alias instead of User_Alias in the configuration. The Runas_Alias allows execution of a command as a user other than root.

A Host_Alias is a list of hostnames, IP addresses, or netgroups (with a "+" prefix). The Host_Alias also supports the NOT operator, "!", to negate an entry. You will need to use the fully qualified DNS name if the hostname command on any machine returns the name of the machine in a fully qualified DNS format. The visudo editor will not catch this "error."

A Cmnd_Alias is list of one or more commands specified by a *full pathname*, not just the filename. You can also specify directories and other aliases to commands. The command alone will allow command arguments to the command, but you can disable command arguments using double quotes (""). If a directory is specified a user can execute any command within that directory, but *not* any subdirectories. Wildcards are allowed, but be very careful to ensure that the wildcard is working as expected.

I am not going to discuss every piece of sudo because very detailed documentation is included with the sudo distribution. Our next step is to look at how to use sudo and how to use sudo in a shell script.

Using sudo

We use sudo by preceding the command that we want to run with the word sudo. As an example, if my user ID is rmichael and I want to gain root access for the first time, I will follow the steps illustrated in Listing 24-7.

Listing 24-7 Using sudo for the first time

Notice the short lecture that is displayed in Listing 24-7. This lecture message is displayed only the first time sudo is used by each user. In the password field the user responds with his or her normal user-account password, *not* the root password. You should be careful granting full root permission like this. Allowing a user to su to root via the sudo program does not leave an audit trail of what the user did as root! You should still have the root history file if the user did not delete or edit the file. Also notice in Listing 24-7 that I added /usr/local/bin to my \$PATH. By default the sudo command is located in the /usr/local/bin directory, but most shops do not add this directory to the \$PATH environment variable as a normal path when setting up user accounts. Just make sure that *all sudo users* have the sudo command in the \$PATH or that they need to provide the full pathname to the sudo command.

Using sudo in a Shell Script

You can also use sudo in a shell script. As you create the shell script, add the sudo command as a prefix to each command that you want to execute as root. The script in Listing 24-8 uses the sudo command to allow the Operations Team to reset passwords.

On an AIX system you can manage user passwords with the pwdadm command. In this particular shell script we want our Operations Team to be able to change a user's password from a menu selection in a shell script. The bold text shown in Listing 24-8 points out the use of sudo and also the use of the tput command for reverse video, which we will study further in Chapter 15, "hgrep: Highlighted grep Script."

```
#!/usr/bin/ksh
# SCRIPT: chpwd_menu.ksh
# AUTHOR: Randy Michael
# DATE: 11/05/2007
# PLATFORM: AIX
# REV: 1.1.P
# PURPOSE: This script was created for the Operations Team
         to change user passwords. This shell script uses
#
#
          "sudo" to execute the "pwdadm" command as root.
#
         Each member of the Operations Team needs to be
         added to the /etc/sudoers file. CAUTION: When
#
         editing the /etc/sudoers file always use the
         /usr/local/sbin/visudo program editor!!!
         NEVER DIRECTLY EDIT THE sudoers FILE !!!
# REV LIST:
#
# set -x # Uncomment to debug this script
# set -n # Uncomment to check syntax without any execution
```

Listing 24-8 chpwd_menu.ksh shell script

```
****
#
        DEFINE FUNCTIONS HERE
****
function chg_pwd
{
USER_NAME="$1"
echo "\nThe next password prompt is for YOUR NORMAL PASSWORD"
echo "NOT the new password..."
# The next command turns off the checking of the password history
/usr/local/bin/sudo /usr/bin/pwdadm -f NOCHECK $USER_NAME
if [ $? -ne 0 ]
then
      echo "\nERROR: Turning off password history failed ... "
      usage
      exit 1
fi
# The next command changes the user's password
/usr/local/bin/sudo /usr/bin/pwdadm $USER_NAME
if [ $? -ne 0 ]
then
      echo "\nERROR: Changing $USER_NAME password failed..."
      usage
      exit 1
fi
# The next command forces the user to change his or her password
# at the next login.
/usr/local/bin/sudo /usr/bin/pwdadm -f ADMCHG $USER_NAME
return 0
}
*****
#
                START OF MAIN
**********
OPT=0
       # Initialize to zero
clear
        # Clear the screen
```

```
Listing 24-8 (continued)
```

```
while < $OPT != 99 > # Start a loop
do
# Draw reverse image bar across the top of the screen
# with the system name.
clear
tput smso
echo "
                             $(hostname)
tput sgr0
echo ""
# Draw menu options.
print "10. Change Password"
print "99. Exit Menu"
# Draw reverse image bar across bottom of screen,
# with error message, if any.
tput smso
echo "
                  $MSG
                                         п
tput sgr0
# Prompt for menu option.
read OPT
# Assume invalid selection was taken. Message is always
# displayed, so blank it out when a valid option is selected.
MSG="
         Invalid option selected
                                        # Option 10 - Change Password
if [ $OPT -eq 10 ]
then
       echo "\nUsername for password change? \c"
       read USERNAME
       grep $USERNAME /etc/passwd >/dev/null 2>&1
       if [ $? -eq 0 ]
       then
             chg_pwd $USERNAME
            if [ $? -eq 0 ]
```

п

Listing 24-8 (continued)

Listing 24-8 (continued)

The chpwd_menu.ksh shell script in Listing 24-8 displays a menu on the screen that has only two options: change a user's password or exit. This shell script uses the sudo program to execute the pwdadm command as the root user. The pwdadm command is used for password administration in AIX and has options to turn password history checking off and to force password changes on the next login attempt. The pwdadm command is executed three times in the chg_pwd function within the shell script. The first time pwdadm is executed as root we turn off the checking of the password history. Notice that I added a comment to the staff that the next password prompt is for *their normal user password*, not the root user password. I turn off the history checking because the password that the Operation Team is going to enter is a temporary password. The next time the user logs in, the system will prompt for a new password, and at this stage the password history will be checked. The second time that pwdadm is executed, the password is actually changed by the Operations Team member. The third time pwdadm is executed, the user is forced to change his or her password the next time they log in. Each time sudo is used to execute pwdadm as root.

Also notice the tput commands. The tput command has many options to control the cursor and the terminal. In this script we are using reverse video to display the hostname of the machine in the menu title bar at the top and to display messages at the bottom of the menu. There is much more on the tput command options in Chapter 15.

Logging to the syslog with sudo

System messages on UNIX can be logged using syslogd. The logged messages consist of at least a time and a hostname field, and normally a program name field, too, but the amount of information is determined by the configuration file /etc/syslog.conf.

When a system message is logged, syslogd determines where to log the message by referencing the /etc/syslog.conf file. We can set up sudo to log messages using syslogd by adding a line or two to the /etc/syslog.conf file.

The /etc/syslog.conf file consists of a list of rules that determine the *facility* to log, the detail level of logging, and the file or remote log host to send the message. Every rule consists of two fields: a selector field and the action field. Any type of white space, one or more spaces or tabs, can separate these fields. The selector field specifies the facility and priority for the specified action. The facility can be any of the following keywords: auth, authpriv, cron, daemon, kern, lpr, mail, mark, news, security (same as auth), syslog, user, uucp, and local0 through local7. By default, sudo logs to the syslogd facility local2. However, the facility is changeable, so we need to query sudo to see how it is configured on the system. We can check the facility sudo is configured to use by executing sudo -V (uppercase V). The -V (version) switch tells sudo to print the version number and exit, if the user is not root. If the user is root, the -V switch will print the defaults sudo was compiled with, as well as the machine's local network addresses. An example is shown in Listing 24-9.

```
[root@yogi ~] # sudo -V
Sudo version 1.6.9p9
Sudoers path: /etc/sudoers
Authentication methods: 'pam'
Syslog facility if syslog is being used for logging: local2
Syslog priority to use when user authenticates successfully: notice
Syslog priority to use when user authenticates unsuccessfully: alert
Send mail if the user is not in sudoers
Lecture user the first time they run sudo
Require users to authenticate by default
Root may run sudo
Allow some information gathering to give useful error messages
Set the LOGNAME and USER environment variables
Length at which to wrap log file lines (0 for no wrap): 80
Authentication timestamp timeout: 5 minutes
Password prompt timeout: 5 minutes
Number of tries to enter a password: 3
Umask to use or 0777 to use user's: 022
Path to mail program: /usr/sbin/sendmail
Flags for mail program: -t
Address to send mail to: root
Subject line for mail messages: *** SECURITY information for %h ***
Incorrect password message: Sorry, try again.
Path to authentication timestamp dir: /var/run/sudo
Default password prompt: Password:
Default user to run commands as: root
Path to the editor for use by visudo: /bin/vi
When to require a password for 'list' pseudocommand: any
```

Listing 24-9 Example sudo -V output executing with root authority

When to require a password for 'verify' pseudocommand: all File containing dummy exec functions: /usr/local/libexec/sudo_noexec.so Reset the environment to a default set of variables Environment variables to check for sanity: TERM LINGUAS LC_* LANGUAGE LANG COLORTERM Environment variables to remove: RUBYOPT RUBYLIB PYTHONINSPECT PYTHONPATH PYTHONHOME TMPPREFIX ZDOTDIR READNULLCMD NULLCMD FPATH PERL5DB PERL50PT PERL5LIB PERLLIB PERLIO DEBUG JAVA_TOOL_OPTIONS SHELLOPTS GLOBIGNORE PS4 BASH_ENV ENV TERMCAP TERMPATH TERMINFO_DIRS TERMINFO _RLD* LD_* PATH_LOCALE NLSPATH HOSTALIASES RES OPTIONS LOCALDOMAIN CDPATH IFS Environment variables to preserve: XAUTHORIZATION XAUTHORITY

Listing 24-9 (continued)

r	P7.
1	PS2
1	PS1
1	PATH
I	MAIL
]	LS_COLORS
1	KRB5CCNAME
I	HOSTNAME
1	DISPLAY
(COLORS
Local IP	address and netmask pairs:
-	192.168.100.38 / 255.255.255.0
:	fe80::299:25aa:fe7a:818c / ffff:ffff:ffff:ffff::

Listing 24-9 (continued)

There is a lot information here, but the only lines we are interested in are the syslog facility and priorities that sudo is configured to use:

Syslog facility if syslog is being used for logging: local2 Syslog priority to use when user authenticates successfully: notice Syslog priority to use when user authenticates unsuccessfully: **alert**

To quickly get this data, run the following command:

```
[root@yogi ~]# sudo -V | grep Syslog
Syslog facility if syslog is being used for logging: local2
Syslog priority to use when user authenticates successfully: notice
Syslog priority to use when user authenticates unsuccessfully: alert
```

With the syslog facility for sudo known, we can add an entry to the /etc/ syslog.conf to set up sudo logging. We can log to both a local log file and to a remote syslog server. If you have had any SOX or PCI audits, you already know the requirement for having a log history, and a central syslog repository is an excellent way to meet these logging requirements. The file shown in Listing 24-10 is called sample.syslog.conf and comes with the source code of the sudo distribution.

```
# This is a sample syslog.conf fragment for use with Sudo.
#
# Sudo logs to local2 by default, but this is changeable via the
# --with-logfac configure option. To see what syslog facility
# a sudo binary uses, run `sudo -V' as *root*. You may have
# to check /usr/include/syslog.h to map the facility number to
# a name.
#
# NOTES:
# The whitespace in the following line is made up of <TAB>
```

Listing 24-10 Sample syslog configuration file – sample.syslog.conf

```
characters, *not* spaces. You cannot just cut and paste!
     If you edit syslog.conf you need to send syslogd a HUP signal.
#
     Ie: kill -HUP process_id
#
#
     Syslogd will not create new log files for you, you must first
     create the file before syslogd will log to it. Eg.
      'touch /var/log/sudo'
# This logs successful and failed sudo attempts to the file /var/log/sudo
local2.debug
                 /var/log/sudo
# To log to a remote machine, use something like the following,
# where "loghost" is the name of the remote machine.
local2.debug
                  @loghost
```

Listing 24-10 (continued)

In this syslog.conf file example, we define both a local log file as well as a remote log host. Notice that we set the logging priority to debug so that both successful and failed sudo attempts are logged to the same file, /var/log/sudo.

After modifying the /etc/syslog.conf file, we need to have syslogd reread the configuration file. To do this, we need the process ID (PID) of the syslogd process so that we can send a HUP signal to syslogd, as shown in Listing 24-11.

```
[root@yogi ~]# ps aux | grep syslogd
root 1765 0.0 0.0 1652 600 ? Ss Dec03 0:00 syslogd -m 0
root 32349 0.0 0.0 3916 656 pts/1 R+ 13:15 0:00 grep
syslogd
[root@yogi ~]# kill -HUP 1765
```

Listing 24-11 Telling syslogd to reread the /etc/syslog.conf file

Listing 24-11 queries the system for the process named syslogd using the ps aux command. From this output, we find that the syslogd process ID is 1765. Then we use the kill -HUP 1765 command to tell syslogd to reread /etc/syslog.conf.

NOTE Understand that adding sudo logging to syslogd does not remove the logging we configured in the /etc/sudoers file, as shown here:

```
# Override Defaults
# Change the default location of the SUDO log file
Defaults logfile=/var/adm/sudo.log
```

Just be aware where you want the logs to go. It is entirely up to you.

The sudo Log File

Before we end this chapter I want to show you what the sudo log file looks like. Each time sudo is executed, an entry is made in the specified log. Logging can be to a file or to the system syslog. I specify a log file in the /etc/sudoers, but you may prefer the syslog. A short version of my sudo log file is shown in Listing 24-12.

```
Nov 9 10:07:44 : d7742 : TTY=pts/2 ; PWD=/usr/local ; USER=root ;
   COMMAND=/usr/bin/ftp bambam
Nov 9 10:09:13 : d7742 : TTY=pts/2 ; PWD=/usr/local ; USER=root ;
    COMMAND=/usr/bin/ftp dino
Nov 13 10:10:48 : d7742 : TTY=pts/0 ; PWD=/home/guest ; USER=root ;
    COMMAND=/usr/bin/whoami
Jul 23 17:35:47 : d7996 : TTY=pts/3 ; PWD=/home/guest ; USER=root ;
    COMMAND=/usr/sbin/mount /usr/local/common
Oct 2 09:29:33 : d7742 : TTY=pts/1 ; PWD=/home/d7742 ; USER=root ;
   COMMAND=/usr/bin/su -
Nov 14 16:01:31 : d7742 : TTY=pts/0 ; PWD=/home/d7742 ; USER=root ;
   COMMAND=/usr/bin/su - root
Nov 14 16:03:58 : rmichael : TTY=pts/0 ; PWD=/home/rmichael ; USER=root ;
   COMMAND=/usr/bin/su - root
Nov 15 11:31:32 : d7742 : TTY=pts/0 ; PWD=/scripts ; USER=root ;
   COMMAND=/usr/bin/pwdadm -f NOCHECK rmichael
Nov 15 11:31:32 : d7742 : TTY=pts/0 ; PWD=/scripts ; USER=root ;
    COMMAND=/usr/bin/pwdadm rmichael
Nov 15 11:31:32 : d7742 : TTY=pts/0 ; PWD=/scripts ; USER=root ;
    COMMAND=/usr/bin/pwdadm -f ADMCHG rmichael
Nov 15 14:58:49 : root : TTY=pts/0 ; PWD=/usr/local/sudo-1.6.3p7 ;
    USER=root ; COMMAND=/usr/bin/errpt
Nov 15 14:59:50 : d7742 : 3 incorrect password attempts ; TTY=pts/0 ;
    PWD=/home/d7742 ; USER=root ; COMMAND=/usr/bin/errpt
```

Listing 24-12 Sample sudo log file

In Listing 24-12 notice the last line of output. This line shows three incorrect password attempts. You can set up sudo to send an email on each password failure if you want immediate notification of misuse. The shell script in Listing 24-8 produced three log entries on each password change. I have highlighted several other entries for ftp and su to root to show you how the log entries look.

Summary

Through this chapter we have shown how to compile, install, configure, and use the sudo program. We all know that protecting the root password is one of our main tasks as a Systems Administrator, and sudo makes the job a little less difficult. When you use sudo in a shell script, it is important that each user is familiar with sudo

and has used it at least once from the command line. Remember that on the first use the lecture message is displayed, and you do not want a lecture in the middle of a menu! In the sudo distribution there are several files that you should review. The readme file has valuable information on installation and a lot of OS-specific problems and workarounds. The FAQ file answers the most frequently asked questions. The *Sudoer's Manual* is a must-read! This manual describes the many options in configuring your sudoers file. Finally, we have the *Visudo Manual* that explains how to use the visudo editor and lists the command options and possible error conditions. Again, I want to thank Todd Miller at http://sudo.ws for allowing me to use his material in this chapter.

In the next chapter, we are going to cover print-queue hell. If the printers are not printing, the queue may be down. We will look at techniques to keep the printers printing with a shell script.

Lab Assignments

- 1. Write a shell script that will uncompress and untar the sudo distribution files, and then run the configure, make, and make install commands to automatically install sudo for you.
- 2. Modify the shell script that you created in Lab Assignment 1 so that the script will NFS mount a remote directory containing the uncompressed and untarred sudo source code, and then run the configure, make, and make install commands to install sudo remotely.
- 3. Configure the /etc/sudoers file to allow the user ID e112233 to execute the /usr/bin/whoami, /usr/bin/mount, and /usr/bin/umount commands with root authority without the requirement to enter a password. Additionally, allow this same e112233 user to execute /usr/sbin/ifconfig with root authority with a valid password.

CHAPTER 25

Print-Queue Hell: Keeping the Printers Printing

If you have worked in a large systems environment for very long, you already know how frustrating it can be to keep the printer farm happy. In my contracting days I worked in several shops that consistently had problems with the printers. In most cases, the print queues went down because of network timeouts and extended device waits. In this kind of environment you have two choices: keep answering the calls from the help desk or write a shell script to monitor the printer queues and re-enable the queues as they drop offline.

I prefer the second method. Like every other Systems Administrator, I like to be proactive in my approach to solving the little problems as well as the big ones. The shop I remember the best is a hospital. This hospital has more than 30 satellite clinics around town and only one 100 MB/Sec pipe coming in to the hospital from the outside world. Most of the clinics have between three and five printers, with at least one printer active most of the day. When I came on board, the first problem I encountered was the huge volume of calls to the help desk about printer problems. What caught my eye was the fact that all of the calls came from the clinics, not from inside the hospital. I knew immediately that a shell script was in order! In this chapter we are going to look at two methods of bringing up the print queues, enabling individual queues and bringing up the whole lot. Because UNIX flavors vary on handling printers and queues, we first will look at the differences between the UNIX flavors.

System V versus BSD versus CUPS Printer Systems

Depending on the UNIX flavor, the commands vary to control the printers and queues, because some use the System V subsystem and others use BSD, also known as *Line Printer Protocol* (lpd), and then we have the option of the *Common UNIX Printer System* (CUPS). With AIX you have an ever more confusing situation beginning with AIX 5L, version 5.1. Starting with this release, AIX began supporting both the "classic" AIX

printer subsystem *and* the System V printer service. Another problem is that some commands do not provide the full print-queue name if the queue name exceeds seven characters. I have come up with some ways to get around the long queue names, and on most systems you do not have to worry about long queue names too much if you want to control *all* of the printers at once.

In this book we are covering AIX, HP-UX, Linux, OpenBSD, and Solaris operating systems, as well as the Common UNIX Printer System (CUPS). For no other reason that I can think of, let's cover the printer systems in alphabetical order.

AIX Print-Control Commands

AIX is the most interesting of the bunch with its new support for the System V printer service starting with AIX 5 L. Although the AIX classic printer subsystem will still be supported for many years, the move seems to be going to System V for printing service.

Classic AIX Printer Subsystem

Most AIX Systems Administrators still prefer to use the classic AIX printer subsystem. This is the primary printing that I have supported for years. With the AIX printer subsystem you do not have the detailed control that either CUPS or the System V service offers. For example, you do not control forms and user priorities at a granular level, and you cannot manage the printers independently of the print queues easily. With this printer subsystem anyone can print on any printer, and the print queue is either UP, allowing you to print, or DOWN, disabling all printing. The shell scripts we are going to write for the classic AIX printer subsystem work at the print-queue level.

The two commands we are going to use are **lpstat** and **enq** -**A**. Both commands produce the same output, but some administrators seem to like one over the other. As I stated earlier, we need to be aware that sometimes print queues are created with queue names longer than seven characters, which is the default that can be displayed with both of these commands. I guess IBM noticed this little problem and added the -W switch to give a wide character output. Look at Listings 25-1 and 25-2 to see the different outputs.

```
# lpstat
Queue Dev Status Job Files User PP % Blks Cp Rnk
hp4 lp0 READY
hp4-g1 lp0 READY
yogi_hp lp0 DOWN
yogi_hp lp0 DOWN
```

Listing 25-1 Output using lpstat or enq -A

# lpstat -W Queue	Dev	Status	Job	Files	User	PP	8	Blks Cp	Rnk
hp4	lp0	READY							
hp4-ps	lp0	READY							
hp4-gl	lp0	READY							
yogi_hp4_1	lp0	DOWN							
yogi_hp4_1ps	lp0	DOWN							

Listing 25-2 Output using lpstat -W or enq -AW

As you can see in Listing 25-1, the long queue names are cut off at the seventh character when using the lpstat or enq -A commands. By adding the -W switch to these commands we see the entire long queue name. This is important because you cannot control a print queue if you do not have the exact, and full, queue name.

There are two methods to script using either lpstat -W or enq -AW. One method is to loop through each queue that is reported DOWN; the other is to use one long compound command. We are first going to look at the looping method.

A little for loop can be used to extract the queue names out of the printers list in a DOWN state. The list used for the for loop comes from either of the following command statements:

```
lpstat -W | tail +3 | grep DOWN | awk '{print $1}'
```

or

```
enq -AW | tail +3 | grep DOWN | awk '{print $1}'
```

Both of these statements produce the same output. Notice that **tail** +3 is the second command in the pipe, just after the lpstat and enq commands. We use tail +3 in this statement to remove the two lines of header information. This method is much cleaner than trying to grep out some unique character in both of the header lines.

Notice that the number of lines, specified by +3, is one larger than the actual number of lines that we want to remove. Using the tail command this way, we are telling tail to start listing at the third line, so two lines are removed at the top of the output.

The third command in the pipe is where we grep for DOWN, looking for disabled printers, as shown in Listing 25-2. The output from this stage of the command is only the lines of the enq and lpstat output that contain the word DOWN. Using these lines as input for the next command in the pipe, we are ready to extract the actual queue name(s) of the disabled printers, as shown in the output here:

```
yogi_hp4_1 lp0 DOWN
yogi_hp4_1ps lp0 DOWN
```

The **awk** command, as we use it, is used to extract the field that we want to work with, which is the first field, the queue name. Using the preceding output as input to our awk statement we extract out the first field using the following syntax:

```
command | awk '{print $1}'
```

You can extract any valid field using awk as well as different fields at the same time. For example, if we want to extract fields 1 and 3, specified by \$1 and \$3, the following awk statement will take care of the task:

```
command | awk '{print $1, $3}'
```

Notice that I added a comma between \$1 and \$3. If the comma is omitted, there will *not* be a space between the two strings. Instead the output will be two strings appended together without a space.

For our for loop we can first send the lpstat and eng command output to a file and process the file in a loop, or we can use command substitution to add the statement directly into the for loop to create the list of objects to loop through. Let's look at our for loop structure:

Using this loop command statement, the for loop will loop through yogi_hp4_1 and yogi_hp4_1ps print-queue names, which is equivalent to the following for loop structure:

Because we never know which queues may be down, we need to parse through the output of the actual queue names of the printers in a disabled state. The shell script in its entirety is shown in Listing 25-3.

```
#!/bin/ksh
#
# SCRIPT: enable_AIX_classic.ksh
#
# AUTHOR: Randy Michael
# DATE: 03/14/2007
# REV: 1.1.P
#
# PLATFORM: AIX Only
#
# PURPOSE: This script is used to enable print queues on AIX systems.
#
# REV LIST:
#
```

Listing 25-3 for loop to enable "classic" AIX print queues

```
# set -x # Uncomment to debug this script
# set -n # Uncomment to check syntax without any execution
#
for Q in $( eng -AW | tail +3 | grep DOWN | awk '{print $1}')
do
        enable $Q
        (( $? == 0 )) || echo "\n$Q print queue FAILED to enable.\n"
done
```

Listing 25-3 (continued)

Inside the for loop we attempt to enable each print queue individually. If the return code of the **enable** command is not zero we echo an error message indicating that the queue could not be enabled. Notice the highlighted lines in Listing 25-3. We use the mathematical test, specified by the double parentheses, ((math test)). Using this math test you normally do not add a dollar sign, \$, in front of a numeric variable. When the variable is produced by the system, such as \$?, the dollar sign is required. Testing for equality also requires using the double equal signs, ==, because the single equal sign, =, is meant as an assignment, not a test.

After the test to check for a zero return code, we use a logical OR, specified by the double pipes, ||. This logical OR will execute the next command only if the return code of the enable \$Q command is nonzero, which means that the command failed. There is also a logical AND that is used by placing double ampersands, &&, in a command statement. A logical AND does just the opposite; it would execute the succeeding command if the test is true, instead of false. Both the logical OR and logical AND are used as replacements for if..then..else.. statements.

We can also accomplish this task by using a single compound command statement. Just as we used command substitution in the for loop, we can use command substitution to produce command parameters. For example, we can use our for loop command to create command parameters to the enable command. To see this more clearly, look at the following two commands:

```
enable $(enq -AW | tail +3 | grep DOWN | awk '{print $1}') 2>/dev/null
```

or

enable \$(lpstat -W | tail +3 | grep DOWN | awk '{print \$1}') 2>/dev/null

Both of these compound command statements produce the same result, enabling all of the print queues on the system. The only problem with using this technique is that if you execute this command and all of the printers are already enabled, you will get the following output from standard error:

usage: enable PrinterName ... Enables or activates printers.

As you can see, I sent this output to the bit bucket by adding 2>/dev/null to the end of the statement, but the return code is still nonzero if all of the printers are already enabled. This should not be a problem unless you want to create some notification that a printer failed to enable. In our for loop in Listing 25-3 we used the return code from the enable command to produce notification. I will leave the technique that you use up to you. If you do not want to see any output, you could add the single compound statement as a cron table entry; or use the for loop technique in a shell script to redirect the failure notification to a log file. If you use a log file you may want to add a date stamp.

System V Printing on AIX

Beginning with AIX 5L, IBM supports System V printing. I find that Solaris has the closest command usage and output. With only a few differences between AIX and Solaris System V printing in the output produced, you could use the shell scripts interchangeably. Because people tend to read only the parts of a technical book that they need to, I will devote this entire section to AIX System V printing.

To switch your AIX system from the "classic" AIX printer subsystem to System V printing, refer to your AIX reference manual. This section expects that you are already running System V printing.

Like Solaris, AIX uses the System V **lpc** (line printer control) command to control the printers and print queues. The nice thing about this print service is that you can control the queues and the printers independently. The main commands that we are interested in for AIX queuing and printing include the following options and parameters to the lpc command, as shown in Table 25-1.

As you can see in Table 25-1, the granularity of printer control is excellent, which gives us several options when creating shell scripts. To control *all* of the printing and queuing at one time you really do not need a shell script. The following two commands can start and stop *all* printing and queuing on *all* print queues at the same time:

lpc down all	#	Disable	all	printing	and	queuing
lpc up all	#	Enable a	all	printing	and (queuing

LPC COMMAND	COMMAND RESULT
disable (printer[@host] all)	Disables queuing
<pre>stop (printer[@host] all)</pre>	Disables printing
down (printer[@host] all)	Disables printing and queuing
enable (printer[@host] all)	Enables queuing
<pre>start (printer[@host] all)</pre>	Enables printing
up (printer[@host] all)	Enables printing and queuing

Table 25-1 AIX lpc Command Options

To keep all of the printers printing and queuing you only need the lpc up all command entered into a cron table. I placed an entry in my root cron table to execute this lpc command every 10 minutes, as shown here:

5,15,25,35,45,55 * * * * /usr/sbin/lpc up all >/dev/null 2>&1

This cron table entry enables all printing and queuing on all printers on the 5s, 24 hours a day, 7 days a week. With AIX System V printing, the data we are interested in is separated on three lines of output when we use the lpc status all command to monitor the printer service. The same BSD originating command executed on AIX, Linux, OpenBSD, and Solaris is shown here.

AIX System V output:

lpc status all
hp4V:
 queueing is enabled
 printing is disabled
 5 entries in spool area

Linux System V output:

# lpc status						
Printer	Printing	Spooling	Jobs	Server	Subserver	
Redirect Status/(Debug)						
hp4@localhost	enabled	disabled	0	none	none	

OpenBSD output:

```
# lpc status all
```

hp4V:

queueing is enabled printing is disabled 5 entries in spool area

Solaris System V output:

```
# lpc status all
bambam_hp4:
    queueing is enabled
    printing is enabled
    no entries
```

Of these four outputs Linux is the one that differs. With the data we are interested in for AIX residing on three separate lines for each print queue, we need a different strategy to get the exact data that we want. First notice that at the beginning of each stanza a queue name has a colon, :, appended to the name of the queue. Because this

character occurs only in the queue name, we can use the colon character as a tag for a grep statement. Following the queue-name entry, the next two lines contain the data that we are interested in pertaining to the status of the queuing and printing.

Because we have some unique tag for each entry, it is easy to extract the lines of data that we are interested in by using an *extended* **grep**, or **egrep**, statement, as shown here:

```
lpc status all | egrep ': | printing | queueing' | while read LINE
```

The egrep command works the same way as the grep command, except that you can specify multiple patterns to match. Each pattern is separated by a pipe *without any spaces*! If you add spaces on either side of the search pattern, the egrep statement will fail to make a match. The entire list of patterns is then enclosed within single forward tic marks, 'pattern1|pattern2|pattern3'. The output produced has the queue name on the first line, the printing status on the second line, and the queuing status on the third line.

The last part of the preceding command is where the output is piped to a while loop. On each read the entire line of data is loaded into the variable LINE. Inside of the while loop we use the following case statement to assign the data to the appropriate variable:

```
case $LINE in
 *:) Q=$(echo $LINE | cut -d ':' -f1)
 ;;
 printing*)
    PSTATUS=$(echo $LINE | awk '{print $3}')
    ;;
 queueing*)
    QSTATUS=$(echo $LINE | awk '{print $3}')
    ;;
esac
```

Notice that if \$LINE begins with *: we load the Q variable. If \$LINE begins with printing* we load the PSTATUS variable with the third field, which should be either enabled or disabled. We do the same thing in loading the QSTATUS variable with the third field of the value that the \$LINE variable points to.

The trick in this script is how to load and process three lines of data and then load and process three more lines of data, and so on. The most intuitive approach is to have a loop counter. Each time the loop counter reaches three we process the data and reset the loop counter back to zero. Take a look at the entire script in Listing 25-4 to see how this loop count works. Pay close attention to the bold type.

```
#!/bin/ksh
#
# SCRIPT: print_UP_SYSV_AIX.ksh
#
```

Listing 25-4 print_UP_AIX.ksh shell script

```
# AUTHOR: Randy Michael
# DATE: 03/14/2007
# REV: 1.1.P
#
# PLATFORM: AIX System V Printing
#
# PURPOSE: This script is used to enable printing and queuing separately
         on each print queue on AIX and Solaris systems.
#
#
# REV LIST:
#
# set -x # Uncomment to debug this script
# set -n # Uncomment to check syntax without any execution
****
LOOP=0
          # Loop Counter - To grab three lines at a time
lpc status all | egrep ': |printing|queueing' | while read LINE
do
    # Load three unique lines at a time
    case $LINE in
    *:) Q=$(echo $LINE | cut -d ':' -f1)
        ;;
    printing*)
        PSTATUS=$(echo $LINE | awk '{print $3}')
        ;;
    queueing*)
        QSTATUS=$(echo $LINE | awk '{print $3}')
        ;;
    esac
     # Increment the LOOP counter
     (( LOOP = LOOP + 1))
     if ((LOOP == 3)) # Do we have all three lines of data?
     then
         # Check printing status
         case $PSTATUS in
         disabled) 1pc start $Q >/dev/null
                   (($? == 0)) && echo "\n$Q printing re-started\n"
                   ;;
         enabled | *) : # No-Op - Do Nothing
                   ;;
         esac
         # Check queuing status
         case $QSTATUS in
         disabled) 1pc enable $Q >/dev/null
```

Listing 25-4 (continued)

```
(($? == 0)) && echo "\n$Q queueing re-enabled\n"
;;
enabled|*) : # No-Op - Do Nothing
;;
esac
LOOP=0 # Reset the loop counter to zero
fi
done
```

Listing 25-4 (continued)

Notice that we grab three lines at a time. The reason that I say that we are grabbing three lines at a time is because I use the case statement to specify unique tags for each line of data. I know that the queue name will have a colon, :, as a suffix. I know that the printing-status line will begin with printing*, and I know that the queuing line will begin with queueing*. We load only one variable on each loop iteration. So, to get three pieces of data (queue name, printing status, and queuing status), we need to go through the while loop three times for each printer queue. Once we pass the initial case statement, we increment the LOOP counter by one. If the \$LOOP variable is equal to 3 we have all of the data that we need to process a single printer queue. After processing the data for this printer queue, we reset the LOOP variable to zero, 0, and start gathering data for the next printer queue.

Sound simple enough? This same technique works for any fixed set of lines of data in command output or in a file. The only changes that are needed to use this method include creating unique tags for the data you are interested in and setting the \$LOOP equality statement to reflect the number of lines in each set of data.

More System V Printer Commands

We have been looking at only the lpc command thus far. We also need to look at two command parameters to the lpstat command in this section. The -a parameter lists the status of queuing, and the -p command parameter lists the status of printing. The nice thing about these two command options is that the output for each queue is on a single line, which makes the data easier to parse through. See Table 25-2.

COMMAND	DESCRIPTION
lpstat -a	Show status of queuing on all printers
lpstat -p	Show status of printing on all printers

Table 25-2 Syste	m V lpstat	Command	Options
------------------	------------	---------	---------

Other than having to query the printer service twice, having to use separate commands for monitoring printing and queuing is not so bad. The separation is built in because the -a and -p command parameters are mutually exclusive, which means that you cannot use -a and -p at the same time. Output from each command option is shown in Listing 25-5.

```
# lpstat -a
hp4 accepting requests since May 07 07:02 2007
yogi_hp4_lps accepting requests since May 07 07:02 2007
long_queue not accepting requests since Tue May 7 07:02:23 EDT 2007 -
s_q_nam not accepting requests since Tue May 7 07:02:23 EDT 2007 -
# lpstat -p
printer long_queue disabled since Tue May 7 07:02:01 EDT 2007.
available.
    stopped by user
printer s_q_nam disabled since Tue May 7 07:02:01 EDT 2007. available.
    stopped by user
printer hp4 unknown state. enabled since May 07 07:30 2007. available.
printer yogi_hp4_lps unknown state. enabled since May 07 07:30 2007. available.
```

Listing 25-5 lpstat -a and lpstat -p command output

Notice in Listing 25-5 that the output from each command option has a unique set of status information for each printer on each line of output. We want to use the uniqueness of the status information as tags in a grep statement. The terms make sense, too. A queue is either *accepting* new requests or is *not accepting* new requests, and a printer is either *enabled* for printing or is *disabled* from printing. Because we are interested only in the disabled and not-accepting states, we can create a simple script or a one-liner.

We need to know two things to enable printing and to bring up a print queue to accept new requests: the printer/queue name and the state of the queue or printer. The first step is to grep out the lines of output that contain our tag. The second step is to extract the printer/queue name from each line of output. Let's first look at using a while loop to bring everything up, as shown in Listing 25-6.

```
lpstat -a | grep 'not accepting' | while read LINE
do
        Q=$(echo $LINE | awk '{print $1}')
        lpc enable $Q
done
lpstat -p | grep disabled | while LINE
do
        P=$(echo $LINE | awk '{print $2}')
        lpc start $P
done
```

Listing 25-6 Scripting the lpstat command using -a and -p

Notice in Listing 25-6 that we have to work on the print queues and printers separately, by using two separate loops. In the first while loop all of the queuing is started. In the second loop we enable printing for each of the printers. The down side to this method occurs when you have hundreds of printers, and scanning through all of the printers twice can take quite a while. Of course, if you have hundreds of printers you should use lpc up all to bring everything up at once.

As I said before, we can also make a one-liner out of the two loops in Listing 25-6. We can combine the grep and awk commands on the same line and use command substitution to execute the lpc command. The following two commands replace the two while loops:

```
lpc enable $(lpstat -a | grep 'not accepting' | awk '{print $1}')
lpc start $(lpstat -p | grep disabled | awk '{print $2}')
```

The first command enables queuing, and the second command starts printing. The command substitution, specified by the \$(command) notation, executes the appropriate lpstat command, then greps on the tag and extracts the printer/queue name. The resulting output is used as the parameter to the lpc commands.

CUPS – Common UNIX Printing System

The following statement comes from www.cups.org:

"CUPS provides a portable printing layer for UNIX-based operating systems. It was developed by Easy Software Products and is now owned and maintained by Apple Inc. to promote a standard printing solution. It is the standard printing system in Mac OS X and most Linux distributions.

CUPS uses the Internet Printing Protocol ("IPP") as the basis for managing print jobs and queues and adds network printer browsing and PostScript Printer Description ("PPD") based printing options to support real-world printing."

I use CUPS on my Fedora 7×64 system, booboo. If you are not familiar with CUPS printing, it would be a good idea to look at the manual at the URL shown here:

http://www.cups.org/doc-1.1/sum.html

The amount of control you have in printing a document in CUPS is amazing. The details of print options are beyond the scope of this book, so let's look at controlling printing and queuing.

Speaking of "queuing," we must be careful scripting our solutions on different OS and printing systems because some spell the word "queuing" and AIX spells it "queueing."

CUPS uses the BSD command lpc to look at the queues' status, but controlling printing and queuing is done with the commands shown in Table 25-3.

We are interested in bringing the queues and printing up, so accept and cupsenable are the commands we use to start printing queued jobs, and to accept new print jobs for queuing. Now we just need to check the queue status and re-enable anything that is down.

CUPS COMMANDS	COMMAND RESULT
accept	Starts printing jobs in the queue
reject	Stops printing jobs in the queue
cupsenable	Allows new print jobs to be queued
cupsdisable	Rejects new print-queue requests

Table 25-3 CUPS Commands Table

A view of the print-queue status is shown in Listing 25-7.

Listing 25-7 Command output checking CUPS printing and queuing

The lpc BSD command implementation of CUPS does not allow for controlling either printing or queuing. Check out the script in Listing 25-8 to see how we re-enable CUPs printing and queuing.

```
#!/bin/ksh
#
#
SCRIPT: print_UP_CUPS.ksh
#
# AUTHOR: Randy Michael
# DATE: 08/27/2007
# REV: 2.1.P
#
#
# PLATFORM: ANY RUNNING CUPS DAEMON
#
# PURPOSE: This script is used to enable printing and queuing separately
# on each print queue for CUPS printing.
```

Listing 25-8 print_UP_CUPS.ksh shell script

```
# REV LIST:
# set -x # Uncomment to debug this script
# set -n # Uncomment to check syntax without any execution
*****
          # Loop Counter - To grab three lines at a time
LOOP=0
lpc status all | egrep ': printing queuing' | while read LINE
do
     # Load three unique lines at a time
    case $LINE in
    *:) Q=$(echo $LINE | cut -d ':' -f1)
        ;;
    printing*)
        PSTATUS=$(echo $LINE | awk '{print $3}')
        ;;
     queuing*)
        QSTATUS=$(echo $LINE | awk '{print $3}')
         ;;
     esac
     # Increment the LOOP counter
     ((LOOP = LOOP + 1))
     if ((LOOP == 3)) # Do we have all three lines of data?
     then
         # Check printing status
         case $PSTATUS in
         disabled) cupsenable $Q >/dev/null
                   ((\$? == 0)) \&\& echo -e "\n\$Q printing re-started\n"
                   sleep 1
                   ;;
         enabled | *) : # No-Op - Do Nothing
                   ;;
         esac
         # Check queuing status
         case $QSTATUS in
         disabled) accept $Q # >/dev/null
                   ((\$? == 0)) \&\& echo -e "\n\$Q queueing re-enabled\n"
                   ;;
         enabled *) : # No-Op - Do Nothing
                   ;;
         esac
```

Listing 25-8 (continued)

```
LOOP=0 # Reset the loop counter to zero fi done
```

Listing 25-8 (continued)

This shell script is almost identical to the shell script in Listing 25-4. The only differences we have in Listing 25-8 are the commands that CUPS uses to control UNIX printing and queuing. We still use a case statement to select the desired action, or non-action when everything is already enabled.

The trick to using this technique is how we are processing only the first three lines of the output of the lpc status all command. We use the egrep statement on only select lines containing, as a group, a : (colon), or the words printing* and queuing*. If we find a printer or queue disabled, the script will run the accept and/or cupsenable commands.

HP-UX Print-Control Commands

Of the UNIX operating systems, HP-UX has a unique **lpstat** command output. We do not have to do anything special to see the full print-queue names, and if queuing is disabled or printing is stopped, we get a *Warning*: message. With a warning message for each printer on a single line we can use grep and awk to find the printer/queue name and the status in a case statement. Let's first look at the lpstat output when both printing and queuing is up, as shown here:

```
# lpstat
printer queue for hp4_yogi_1
printer queue for yogi_hp4_1ps
```

If print requests were queued up they would be listed below the queue name. Now let's disable printing on the hp4_yogi_1 print queue:

```
# disable hp4_yogi_1
printer "hp4_yogi_1" now disabled
```

Now look at the output of the lpstat command:

lpstat

```
printer queue for hp4_yogi_1
dino: Warning: hp4_yogi_1 is down
```

printer queue for yogi_hp4_1ps

The warning message tells us that the printer is down; however, notice that the queue status is not listed here. Now let's bring down the hp4_yogi_1 print queue and see what this does:

```
# reject hp4_yogi_1
destination "hp4_yogi_1" will no longer accept requests
```

To see only queuing status we use the lpstat -a command, as shown here:

Because hp4_yogi_1 now has printing disabled and queuing stopped, I would expect that we should see some queue status output in the lpstat command output for the first time:

```
# lpstat
printer queue for hp4_yogi_1
dino: Warning: hp4_yogi_1 queue is turned off
dino: Warning: hp4_yogi_1 is down
printer queue for yogi_hp4_1ps
```

Just what we expected. From this little exercise we have determined that queuing is reported only when the queuing is stopped on the queue using the lpstat command alone. For our scripting effort let's stick to the lpstat output. We want to use the word Warning as a tag for our grep statement. Then we can further grep this extracted line to check printing and queuing status. If the string 'queue is turned off' is present we know that queuing is turned off, and if the string 'is down' appears on the line we know that printing is disabled. The only thing left to extract is the printer/queue name, which is always located in the third field.

To script this we can use the code in Listing 25-9. Pay attention to the bold type, and we will cover the script at the end.

```
#!/bin/ksh
#
# SCRIPT: print_UP_HP-UX.ksh
#
# AUTHOR: Randy Michael
# DATE: 03/14/2007
# REV: 1.1.P
#
```

Listing 25-9 print_UP_HP-UX.ksh shell script

```
# PLATFORM: HP-UX Only
#
# PURPOSE: This script is used to enable printing and queuing separately
           on each print queue on an HP-UX system.
#
 REV LIST:
#
# set -x # Uncomment to debug this script
# set -n # Uncomment to check syntax without any execution
lpstat | grep Warning: | while read LINE
do
      if (echo $LINE | grep 'is down') > /dev/null
      then
           enable $(echo $LINE | awk '{print $3}')
      fi
      if (echo $LINE | grep 'queue is turned off') >/dev/null
      then
           accept $(echo $LINE | awk '{print $3}')
      fi
done
```

Listing 25-9 (continued)

I want to point out a nice little trick in the shell script in Listing 25-9. In both of the if...then...fi statements, notice that we execute a command inside parentheses. What this technique allows us to do is execute a command in a *sub-shell* and use the command's resulting return code directly in the if...then...fi structure. We really could not care less about seeing the line that we are grepping on; however, if the return code from the command is zero, the pattern is present.

In the first half of the script in Listing 25-9, we check the status of printing. If a printer is found to be disabled, we use command substitution to produce the printer name for the enable command. Likewise, we check for the status of queuing in the second half of the script. Again, using command substitution we have the queue name to provide as a parameter to the accept command. Notice that I added the redirection to the bit bucket, specified by >/dev/null, after the command in the if statement. I add this redirection to /dev/null to suppress the output of the grep statement.

That is it for HP-UX printing. HP did a good job of keeping everything pretty straightforward in the printing arena.

Linux Print-Control Commands

Linux uses either the System V **lpc** (line printer control) command to control the printers and print queues, or the Common UNIX Printer System (CUPS). For our scripts it does not matter which underlying printer subsystem is used because the Linux lpc command works the same as other System V UNIX printer subsystems do. The nice thing about this print service is that you can control the queues and

the printers independently. The main commands that we are interested in for Linux queuing and printing include the options to the lpc command listed in Table 25-4.

As you can see in Table 25-4, the granularity of printer control is excellent, which gives us several options when creating shell scripts. To control *all* of the printing and queuing at one time you really do not need a shell script. The following two commands can start and stop all printing and queuing on all print queues at the same time:

```
lpc down all # Disable all printing and queuing
lpc up all # Enable all printing and queuing
```

To keep all of the printers printing and queuing you need just the lpc up all command entered into a cron table. I placed an entry in my root cron table to execute this command every 10 minutes. My cron table entry is shown here:

5,15,25,35,45,55 * * * * /usr/sbin/lpc up all >/dev/null 2>&1

This cron table entry enables all printing and queuing on all printers on the 5s, 24 hours a day, 7 days a week.

If we do want a little more control and if we keep a log of what is going on on a per queue/printer basis, we have to do a little scripting. The script that follows searches all of the queues and reports on the individual status of printing and queuing and then enables each one independently.

For this script we are going to use *arrays* to load the variables on each loop iteration. Array can be created and elements assigned values in two ways. The first technique is to use **set -A** to define the array and all of its elements. For example, if we want an array called QUEUE to contain the values for printing and queuing for a specified queue, we can set it up this way:

```
PQueue=yogi_hp4
Print_val=enabled
Queue_val=disabled
```

set -A QUEUE \$PQueue \$Print_val \$Queue_val

LPC COMMAND	COMMAND RESULT
disable (printer[@host] all)	Disables queuing
stop (printer[@host] all)	Disables printing
down (printer[@host] all)	Disables printing and queuing
enable (printer[@host] all)	Enables queuing
start (printer[@host] all)	Enables printing
up (printer[@host] all)	Enables printing and queuing

Table 25-4 Linux lpc Command Options

We could have assigned the values directly in the set -A statement, but this time we used variables for the assignments. This statement defines an array named QUEUE that contains three array elements. The elements loaded into the array are the values that the variables \$PQueue, \$Print_val, and \$Queue_val point to. For example, PQueue is assigned the value yogi_hp4, Print_val is assigned the value enabled, and Queue_val is assigned the value disabled. The result is that the first array element, 0 (zero), contains the value yogi_hp4, the second array element, 1 (one), has the value enabled, and the third array element, 2 (two), contains the value disabled, which is what the \$Queue_val variable points to. Using this technique requires that we access the array elements starting with 0, zero.

To address the array elements we use the following syntax:

```
${QUEUE[0]} # Points to value assigned to the first array element,
yogi_hp4
${QUEUE[1]} # Points to value assigned to the second array element,
enabled
${QUEUE[2]} # Points to the value assigned to the third array element,
disabled
```

To address all of the array's elements at the same time we use the following syntax:

Now, before I lose you, let's take a look at a more intuitive way of working with arrays and array elements. Instead of using the set -A command to define and load an array, we can define an array and load its elements at the same time using the following syntax:

```
QUEUE[1]=yogi_hp4
QUEUE[2]=enabled
QUEUE[3]=disabled
```

Notice that the first array element is now referenced by 1, one. These commands create an array named QUEUE and load the first three array elements, referenced by 1, 2, and 3, into array QUEUE. Now we can use the array directly in a command statement by pointing to the array element that we want to use. For example, if we want to print the *printing* status of the yogi_hp4 print queue, we use the following syntax:

echo "\nPrinter $\{QUEUE[1]\}\$ has print status $\{QUEUE[2]\}\$ "

This command produces the following output:

Printer yogi_hp4 has print status enabled

Now that we have seen the basics of working with arrays, let's look at a shell script to handle keeping the printing and queuing enabled on all of the printers individually. The first step is to load an array in a while loop. This is a little different from what we did before with arrays. In this case I want to use the lpc status all command to find printers that have either printing or queuing disabled. The output of the lpc status all command is shown here:

lpc status all
Printer Printing Spooling Jobs Server Subserver Redirect
Status/(Debug)
hp4@localhost enabled disabled 0 none none

This is an easy output to deal with, because all of the data for each queue is on a single line. The output that we are interested in is the printer name, the printing status, and the spooling status — the first three fields on the second line. We are not interested in the first line at all so we can get rid of it with a pipe to the **tail** command. When we add to our command we get the following output:

lpc status all | tail +2
yogi_hp4@localhost enabled disabled 0 none none

I currently have only one printer defined on this system, so the output is the status of a single printer. Now we want to load the first three fields into an array using a while loop. Look at the next command line to see how we are directly loading an array called pqstat with array elements of the first three fields on each line:

lpc status all | tail +2 | while read pqstat[1] pqstat[2] pqstat[3] junk

Because we want just the first three fields in the output, notice that the fourth variable in the read part of the while statement is junk. The junk variable is a catchall variable to capture any remaining strings on the line of output in a single variable. It is a *requirement* that you take care of this remaining text because if you neglect adding a variable to catch any remaining characters on the line, you will read the characters in as strings on the next loop iteration! This type of error produces some strange output that is hard to find and troubleshoot.

Notice that in the output of the lpc status all command the printer has queuing disabled, which is the third field. The easiest way to handle the two status fields is to use two case statements, with each tagging on a separate field. Look at the full script code in Listing 25-10, and we will cover the technique at the end.

```
#!/bin/ksh
#
# SCRIPT: print_UP_Linux.ksh
#
# AUTHOR: Randy Michael
# DATE: 03/14/2007
# REV: 1.1.P
#
```

Listing 25-10 print_UP_Linux.ksh shell script

```
# PLATFORM: Linux Only
#
# PURPOSE: This script is used to enable printing and queuing separately
#
         on each print queue on a Linux system. Logging can be
#
          enabled.
# REV LIST:
# set -x # Uncomment to debug this script
# set -n # Uncomment to check syntax without any execution
*****
# Initial Variables Here
*****
LOGILE=/usr/local/log/PQlog.log
[ -f $LOGFILE ] || echo /dev/null > $LOGFILE
*****
lpc status | tail +2 | while read pqstat[1] pqstat[2] pqstat[3] junk
do
    # First check the status of printing for each printer
    case ${pqstat[2]} in
    disabled)
            # Printing is disabled - print status and restart printing
           echo "${pqstat[1]} Printing is ${pqstat[2]}" \
                | tee -a$LOGFILE
            lpc start ${pqstat[1]} | tee -a $LOGFILE
            (($? == 0)) && echo "${pgstat[1]} Printing Restarted" \
                           | tee -a $LOGFILE
           ;;
    enabled | *) : # No-Op - Do Nothing
            ;;
    esac
    # Next check the status of queueing for each printer
    case ${pqstat[3]} in
    disabled)
             echo "${pqstat[1]} Queueing is ${pqstat[3]}" \
                  | tee -a $LOGFILE
             lpc enable ${pqstat[1]} | tee -a $LOGFILE
              (($? == 0)) && echo "${pqstat[1]} Printing Restarted" \
                           | tee -a $LOGFILE
            ;;
    enabled *) : # No-Op - Do Nothing
            ;;
    esac
done
```

Listing 25-10 (continued)

We start off this script in Listing 25-10 by defining the *\$LOGFILE*. Notice that the following command, after the log file definition, checks to see if the log file exists. If the *\$LOGFILE* does not exist, the result of the test is a nonzero return code. We use a logical OR, specified by the double pipes, ||, to execute the succeeding command to create a zero-length *\$LOGFILE* because it does not exist if the return code of the test is nonzero.

Next, we start our while loop to load the pqstat array on each loop iteration, which in our case is a single loop iteration for a single printer. This means that we load a one-dimensional array with new data on each loop iteration (one-dimensional arrays are all that the shells support). Again, notice the junk variable that is added as the last variable in the while loop statement. This extra variable is required to catch the remaining text in a single variable.

With the array loaded we proceed with two case statements to test for the status of printing and queuing on each print queue. Notice that we use the array element directly in the case statement, as shown here:

```
case ${pqstat[2]} in
```

We use the same technique with the print-queuing array element in a separate case statement. We have only two possible results for the array elements: enabled and disabled. The only result we are concerned about is any disabled value. If we receive any disabled values we attempt to re-enable the printing or queuing on the printer. Notice that the second option in both case statements includes enabled and anything else, specified by the wildcard, *, as shown here:

enabled | *)

We could have just used the wildcard to cover everything, but it is clearer to the reader of the script to see actual expected results in a case statement than just a catchall asterisk.

When a re-enabling task is completed successfully, notice the use of the logical AND to test the return code and give notification on a zero return code value, as shown here:

((\$? == 0)) && echo "\${pqstat[1]} Printing Restarted"

The second part of the command will execute only if the test for a zero return code is true. Otherwise, the system will report an error, so there is no need for us to add any failure notification.

To see everything that is happening on the screen and to log everything at the same time we use the **tee -a** command. This command works with a pipe and prints all of the output to the screen; at the same time it sends the exact same output to the file specified after tee -a. An example is shown here:

```
lpc start ${pqstat[1]} | tee -a $LOGFILE
```

This command attempts to restart printing on the print queue specified by the array element pqstat[1] and sends any resulting output to the screen and to the \$LOGFILE simultaneously.

Controlling Queuing and Printing Individually

Depending on the situation, you may not always want to enable printing and queuing at the same time. We can break up the shell script in Listing 25-10 and pull out the individual case statements to start either printing or queuing. Because printing is controlled by array element 2 we can extract the first case statement to create a new shell script. Let's call this shell script printing_only_UP_Linux.ksh. You can see the modifications in Listing 25-11.

```
#!/bin/ksh
#
# SCRIPT: printing_only_UP_Linux.ksh
# AUTHOR: Randy Michael
# DATE: 03/14/2007
# REV: 1.1.P
#
# PLATFORM: Linux Only
#
# PURPOSE: This script is used to enable printing on each printer
        on a Linux system. Logging is enabled.
#
# REV LIST:
# set -x # Uncomment to debug this script
# set -n # Uncomment to check syntax without any execution
****
# Initial Variables Here
******
LOGILE=/usr/local/log/PQlog.log
[ -f $LOGFILE ] || echo /dev/null > $LOGFILE
******
lpc status | tail +2 | while read pqstat[1] pqstat[2] pqstat[3] junk
do
    # Check the status of printing for each printer
    case ${pqstat[2]} in
    disabled)
            # Printing is disabled - print status and restart printing
             echo "${pgstat[1]} Printing is ${pgstat[2]}" \
                 | tee -a$LOGFILE
             lpc start ${pqstat[1]} | tee -a $LOGFILE
             (($? == 0)) && echo "${pqstat[1]} Printing Restarted" \
                          tee -a $LOGFILE
```

Listing 25-11 printing_only_UP_Linux.ksh shell script

```
;;
enabled|*) : # No-Op - Do Nothing
;;
esac
done
```

Notice that the only thing that was changed is that the second case statement structure was removed from the script and the name was changed. We can do the same thing to create a shell script that only enables queuing, as shown in Listing 25-12.

```
#!/bin/ksh
#
# SCRIPT: queuing_only_UP_Linux.ksh
#
# AUTHOR: Randy Michael
# DATE: 03/14/2007
# REV: 1.1.P
#
# PLATFORM: Linux Only
#
# PURPOSE: This script is used to enable printing and queuing separately
        on each print queue on a Linux system. Logging can be
#
#
        enabled.
# REV LIST:
# set -x # Uncomment to debug this script
# set -n # Uncomment to check syntax without any execution
****
# Initial Variables Here
****
LOGILE=/usr/local/log/PQlog.log
[ -f $LOGFILE ] || echo /dev/null > $LOGFILE
*****
lpc status | tail +2 | while read pqstat[1] pqstat[2] pqstat[3] junk
do
    # check the status of queueing for each printer
    case ${pqstat[3]} in
    disabled)
```

Listing 25-12 queuing_only_UP_Linux.ksh shell script

Notice that the only thing that was changed this time is that the first case statement structure was removed from the script and the name of the shell script was changed. You could also modify the shell script in Listing 25-10 to add a command-line parameter to let you control queuing and printing individually from the same shell script. I am going to leave this as an exercise for you to complete.

NOTE A hint for this exercise: expect only zero or one command-line parameters. If \$# is equal to zero, then enable both queuing and printing. If there is one parameter and the value of \$1 is "all," then enable both printing and queuing. If the \$1 parameter is equal to "printing," then enable only printing. If \$1 is equal to "queuing," then enable only queuing. You need to add a usage function to show how to use the shell script if the given value does not match what you are expecting.

Arrays are good to use in a lot of situations where you want to address certain output fields directly and randomly. All shell arrays are one-dimensional arrays, but using the array in a loop gives the *appearance* of a two-dimensional array.

Solaris Print-Control Commands

Solaris uses the System V **lpc** (line printer control) command to control the printers and print queues, as most System V UNIX systems do. The nice thing about this print service is that you can control the queues and the printers independently. The main commands that we are interested in for Solaris queuing and printing include the following options and parameters to the lpc command, as shown in Table 25-5.

As you can see in Table 25-5, the granularity of printer control is excellent, which gives several options when creating shell scripts. To control *all* of the printing and queuing at one time you really do not need a shell script. The following two commands can start and stop all printing and queuing on all print queues at the same time:

```
lpc down all # Disable all printing and queuing
lpc up all # Enable all printing and queuing
```

COMMAND RESULT
Disables queuing
Disables printing
Disables printing and queuing
Enables queuing
Enables printing
Enables printing and queuing

Table 25-5 Solaris lpc Command Options

To keep all of the printers printing and queuing you need only the lpc up all command entered into a cron table. I placed an entry in my root cron table to execute this command every 10 minutes. My cron table entry is shown here:

5,15,25,35,45,55 * * * * /usr/sbin/lpc up all >/dev/null 2>&1

This cron table entry enables all printing and queuing on all printers on the 5s, 24 hours a day, 7 days a week.

We have a nice situation here because we can use the same shell script that we used for the AIX System V printing on Solaris. Unlike Linux, where all of the data that we want is on a single line of output, with Solaris and AIX System V printing, the data we are interested in is separated on three lines of output. You can see the difference in the output here.

AIX System V output:

```
# lpc status all
hp4V:
    queueing is enabled
    printing is disabled
    5 entries in spool area
```

Linux System V output:

```
# lpc status
Printer Printing Spooling Jobs Server Subserver Redirect
Status/(Debug)
hp4@localhost enabled disabled 0 none none
```

Solaris System V output:

```
# lpc status all
```

```
bambam_hp4:
    queueing is enabled
```

```
printing is enabled no entries
```

Of these three outputs, Linux is the one that differs. With the data we are interested in for Solaris residing on three separate lines for each print queue, we need a different strategy to get the exact data that we want. First notice that at the beginning of the stanza for the queue name there is a colon, :, appended to the name of the queue. Because this character occurs only in the queue name, we can use the colon character as a tag for a grep statement. Following the queue-name entry the next two lines contain the data pertaining to the status of the queuing and printing.

Because we have some unique tag for each entry, it is easy to extract the lines of data that we are interested in by using an *extended* **grep**, or **egrep**, statement, as shown here:

```
lpc status all | egrep ': printing queueing' | while read LINE
```

The egrep command works the same way as the grep command, except that you can specify multiple patterns to match. Each pattern is separated by a pipe *without any spaces*! If you add spaces on either side of the search pattern the egrep statement will fail to make a match. The entire list of patterns is then enclosed within single forward tic marks, 'pattern1|pattern2|pattern3'. The output produced has the queue name on the first line, the printing status on the second line, and the queuing status on the third line.

The last part of the preceding command is where the output is piped to a while loop. On each read, the entire line of data is loaded into the variable LINE. Inside of the while loop we use the following case statement to assign the data to the appropriate variable:

```
case $LINE in
 *:) Q=$(echo $LINE | cut -d ':' -f1)
 ;;
 printing*)
    PSTATUS=$(echo $LINE | awk '{print $3}')
    ;;
 queueing*)
    QSTATUS=$(echo $LINE | awk '{print $3}')
    ;;
esac
```

Notice that if \$LINE begins with *: we load the Q variable. If \$LINE begins with printing* we load the PSTATUS variable with the third field, which should be either enabled or disabled. We do the same thing in loading the QSTATUS variable with the third field of the value that the \$LINE variable points to.

The trick in this script is how to load and process three lines of data and then load and process three more lines of data, and so on. The most intuitive approach is to have a loop counter. Each time the loop counter reaches three we process the data and reset the loop counter back to zero. Take a look at the entire script in Listing 25-13 to see how this loop count works. Pay close attention to the bold type.

```
#!/bin/ksh
#
# SCRIPT: print_UP_Solaris.ksh
# AUTHOR: Randy Michael
# DATE: 03/14/2007
# REV: 1.1.P
# PLATFORM: Solaris Only
#
# PURPOSE: This script is used to enable printing and queuing separately
          on each print queue on Solaris systems.
#
#
# REV LIST:
# set -x # Uncomment to debug this script
# set -n # Uncomment to check syntax without any execution
****
          # Loop Counter - To grab three lines at a time
LOOP=0
lpc status all | egrep ': printing queueing' | while read LINE
do
    # Load three unique lines at a time
    case $LINE in
     *:) Q=$(echo $LINE | cut -d ':' -f1)
         ;;
    printing*)
        PSTATUS=$(echo $LINE | awk '{print $3}')
        ;;
     queueing*)
        QSTATUS=$(echo $LINE | awk '{print $3}')
        ;;
     esac
     # Increment the LOOP counter
     (( LOOP = LOOP + 1))
     if ((LOOP == 3)) # Do we have all three lines of data?
     then
         # Check printing status
         case SPSTATUS in
         disabled) 1pc start $Q >/dev/null
                   ((\$? == 0)) \&\& echo "\n\$Q printing re-started\n"
                   ;;
         enabled *) : # No-Op - Do Nothing
```

Listing 25-13 print_UP_SUN.ksh shell script

```
;;
esac
# Check queuing status
case $QSTATUS in
disabled) lpc enable $Q >/dev/null
        (($? == 0)) && echo "\n$Q queueing re-enabled\n"
        ;;
enabled[*) : # No-Op - Do Nothing
        ;;
esac
LOOP=0 # Reset the loop counter to zero
fi
done
```

Within this while loop we are grabbing three lines of data at a time to process. I say that we are grabbing three lines at a time in Listing 25-13 because I use the case statement to specify unique tags for each line of data. I know that the queue name will have a colon, :, as a suffix. I know that the printing status line will begin with printing*, and I know that the queuing line will begin with queueing*. We load only one variable on each loop iteration, though. To get three pieces of data (queue name, printing status, and queuing status), we need to go through the while loop three times for each printer queue. Once we pass the initial case statement, we increment the LOOP counter by one. If the \$LOOP variable is equal to 3, we have all the data that we need to process a single printer queue. After processing the data for this printer queue.

Sound simple enough? This same technique works for any fixed set of lines of data in command output or in a file. The only changes that are needed to use this method include creating unique tags for the data you are interested in and setting the \$LOOP equality statement to reflect the number of lines that are in each set of data.

More System V Printer Commands

We have been looking only at the lpc command thus far. We also need to look at two command parameters to the lpstat command in this section. The -a parameter lists the status of queuing, and the -p command parameter lists the status of printing. The nice thing about these two command options is that the output for each queue is on a single line, which makes the data easier to parse through. The lpstat command options are shown in Table 25-6.

Other than having to query the printer subsystem twice, having to use separate commands for monitoring printing and queuing is not so bad. The separation is built in because the -a and -p command parameters are mutually exclusive, which means that you cannot use -a and -p at the same time. Listing 25-14 shows the output from each command option.

	-
COMMAND	DESCRIPTION
lpstat -a	Show status of queuing on all printers
lpstat -p	Show status of printing on all printers

Table 25-6 System V lpstat Command Options

```
# lpstat -a
hp4 accepting requests since May 07 07:02 2007
yogi_hp4_lps accepting requests since May 07 07:02 2007
long_queue not accepting requests since Tue May 7 07:02:23 EDT 2007 -
s_q_nam not accepting requests since Tue May 7 07:02:23 EDT 2007 -
# lpstat -p
printer long_queue disabled since Tue May 7 07:02:01 EDT 2007.
available.
    stopped by user
printer s_q_nam disabled since Tue May 7 07:02:01 EDT 2007. available.
    stopped by user
printer hp4 unknown state. enabled since May 07 07:30 2007. available.
printer yogi_hp4_lps unknown state. enabled since May 07 07:30 2007. available.
```

Listing 25-14 lpstat -a and lpstat -p command output

Notice in Listing 25-14 that the output from each command option has a unique set of status information for each printer on each line of output. We want to use the uniqueness of the status information as tags in a grep statement. The terms make sense, too. A queue is either *accepting* new requests or *not accepting* new requests, and a printer is either *enabled* for printing or *disabled* from printing. Because we are interested in only the disabled and not-accepting states, we can create a simple script or a one-liner.

We need to know two things to enable printing and to bring up a print queue to accept new requests: the printer/queue name and the state of the queue or printer. The first step is to grep out the lines of output that contain our tag. The second step is to extract the printer/queue name from each line of output. Let's first look at using a while loop to bring everything up, as shown in Listing 25-15.

```
lpstat -a | grep 'not accepting' | while read LINE
do
        Q=$(echo $LINE | awk '{print $1}')
        lpc enable $Q
done
```

Listing 25-15 Scripting the lpstat command using -a and -p

```
lpstat -p | grep disabled | while LINE
do
        P=$(echo $LINE | awk '{print $2}')
        lpc start $P
done
```

Notice in Listing 25-15 that we have to work on the print queues and printers separately, by using two separate loops. In the first while loop all of the queuing is started. In the second loop we enable printing for each of the printers. The down side to this method is where you have hundreds of printers. The time it takes to scan through all of the printers once and then rescan the printer service can be quite long. Of course, if you have hundreds of printers, you should use lpc up all to bring everything up at once.

As I said before, we can also make a one-liner out of the two loops in Listing 25-15. We can combine the grep and awk commands on the same line and use command substitution to execute the lpc command. The following two commands replace the two while loops:

```
lpc enable $(lpstat -a | grep 'not accepting' | awk '{print $1}')
lpc start $( lpstat -p | grep disabled | awk '{print $2}')
```

The first command enables queuing, and the second command starts printing. The command substitution, specified by the \$(command) notation, executes the appropriate lpstat command, then greps on the tag and extracts the printer/queue name. The resulting output is used as the parameter to the lpc commands.

Putting It All Together

Now we need to combine the shell scripts for each of the different UNIX flavors so that one script does it all. Please do not think that taking several shell scripts, making functions out of them, and combining the new functions into a new script are difficult tasks. To make one script out of this chapter we are going to take the best of our scripts and extract the code. For each shell script we make a new function, which requires only the word function, a function name, and the code block surrounded by curly braces, function function_name { code stuff here }. Let's take a look at the entire combined shell script in Listing 25-16 and cover the functions at the end.

```
#!/bin/ksh
#
# SCRIPT: PQ_UP_manager.ksh
#
# AUTHOR: Randy Michael
```

Listing 25-16 PQ_UP_manager.ksh shell script

```
# DATE: 08/14/2007
# REV: 2.1.P
# PLATFORM/SYSTEMS: AIX, CUPS, HP-UX, Linux, OpenBSD, and Solaris
#
# PURPOSE: This script is used to enable printing and queuing on
        AIX, CUPS, HP-UX, Linux, OpenBDS, and Solaris
#
# REV LIST:
# set -x # Uncomment to debug this script
# set -n # Uncomment to check syntax without any execution
*****
# DEFINE FUNCTIONS HERE
*****
function AIX_classic_printing
{
for Q in $( enq -AW | tail +3 | grep DOWN | awk '{print $1}')
do
    enable $Q
    (( $? == 0 )) || echo "\n$Q print queue FAILED to enable.\n"
done
}
****
function AIX_SYSV_printing
{
        # Loop Counter - To grab three lines at a time
LOOP=0
lpc status all | egrep ': |printing|queueing' | while read LINE
do
    # Load three unique lines at a time
    case $LINE in
    *:) Q=$(echo $LINE | cut -d ':' -f1)
       ;;
    printing*)
       PSTATUS=$(echo $LINE | awk '{print $3}')
       ;;
    queueing*)
       QSTATUS=$(echo $LINE | awk '{print $3}')
        ;;
    esac
    # Increment the LOOP counter
    ((LOOP = LOOP + 1))
```

Listing 25-16 (continued)

```
if ((LOOP == 3)) # Do we have all three lines of data?
     then
         # Check printing status
         case $PSTATUS in
         disabled) lpc start $Q # >/dev/null
                   ((\$? == 0)) \&\& echo "\n\$Q printing re-started\n"
                   ;;
         enabled |*) : # No-Op - Do Nothing
                  ;;
         esac
         # Check queuing status
         case $QSTATUS in
         disabled) lpc enable $Q # >/dev/null
                   ((\$? == 0)) \&\& echo "\n\$Q queueing re-enabled\n"
                   ;;
         enabled |*) : # No-Op - Do Nothing
                  ;;
         esac
         LOOP=0 # Reset the loop counter to zero
   fi
done
}
****
function CUPS_printing
{
LOOP=0
         # Loop Counter - To grab three lines at a time
lpc status all | egrep ': printing queuing' | while read LINE
do
    # Load three unique lines at a time
    case $LINE in
    *:) Q=$(echo $LINE | cut -d ':' -f1)
        ;;
    printing*)
        PSTATUS=$(echo $LINE | awk '{print $3}')
        ;;
    queuing*)
        QSTATUS=$(echo $LINE | awk '{print $3}')
        ;;
     esac
     # Increment the LOOP counter
     ((LOOP = LOOP + 1))
     if ((LOOP == 3)) # Do we have all three lines of data?
```

```
then
         # Check printing status
         case $PSTATUS in
         disabled) cupsenable $Q >/dev/null
                 ((\$? == 0)) & echo -e "\n$Q printing re-started\n"
                 sleep 1
                 ;;
         enabled |*) : # No-Op - Do Nothing
                 ;;
         esac
         # Check queuing status
         case $QSTATUS in
         disabled) accept $Q # >/dev/null
                 (($? == 0)) && echo -e "\n$Q queueing re-enabled\n"
                 ;;
         enabled | *) : # No-Op - Do Nothing
                 ;;
         esac
        LOOP=0 # Reset the loop counter to zero
   fi
done
}
****
function HP_UX_printing
{
lpstat | grep Warning: | while read LINE
do
     if (echo $LINE | grep 'is down') > /dev/null
     then
         enable $(echo $LINE | awk '{print $3}')
     fi
     if (echo $LINE | grep 'queue is turned off') >/dev/null
     then
         accept $(echo $LINE | awk '{print $3}')
     fi
done
}
****
function Linux_printing
{
```

```
lpc status | tail -n +2 | while read pqstat[1] pqstat[2] pqstat[3] junk
do
     # First check the status of printing for each printer
    case ${pqstat[2]} in
    disabled)
              # Printing is disabled - print status and restart printing
              echo "${pgstat[1]} Printing is ${pgstat[2]}"
              lpc start ${pqstat[1]}
              (($? == 0)) && echo "${pqstat[1]} Printing Restarted"
            : :
     enabled *) : # No-Op - Do Nothing
            ;;
     esac
     # Next check the status of queueing for each printer
    case ${pqstat[3]} in
     disabled)
              echo "${pqstat[1]} Queueing is ${pqstat[3]}"
              lpc enable ${pqstat[1]}
              (($? == 0)) && echo "${pqstat[1]} Printing Restarted"
             ;;
     enabled | *) : # No-Op - Do Nothing
            ;;
     esac
done
}
****
function OpenBSD_printing
{
LOOP=0
          # Loop Counter - To grab three lines at a time
lpc status all | egrep ': printing queuing' | while read LINE
do
     # Load three unique lines at a time
    case $LINE in
     *:) Q=$(echo $LINE | cut -d ':' -f1)
        ;;
    printing*)
        PSTATUS=$(echo $LINE | awk '{print $3}')
        ;;
     queuing*)
        QSTATUS=$(echo $LINE | awk '{print $3}')
        ;;
     esac
     # Increment the LOOP counter
     (( LOOP = LOOP + 1))
```

```
if ((LOOP == 3)) # Do we have all three lines of data?
     then
         # Check queuing status
         case $QSTATUS in
         disabled) lpc enable $Q >/dev/null
                   ((\$? == 0)) \&\& echo "\n\$Q queueing re-enabled\n"
                   sleep 1
                   ;;
         enabled |*) : # No-Op - Do Nothing
                  ;;
         esac
         # Check printing status
         case $PSTATUS in
         disabled) lpc up $Q >/dev/null
                   (($? == 0)) && echo "\n$Q printing re-started\n"
                   ;;
         enabled | *) : # No-Op - Do Nothing
                   ;;
         esac
         LOOP=0 # Reset the loop counter to zero
    fi
done
}
*****
function Solaris_printing
{
LOOP=0
          # Loop Counter - To grab three lines at a time
lpc status all | egrep ': printing queueing' | while read LINE
do
    # Load three unique lines at a time
    case $LINE in
     *:) Q=$(echo $LINE | cut -d ':' -f1)
        ;;
    printing*)
        PSTATUS=$(echo $LINE | awk '{print $3}')
        ;;
    queueing*)
        QSTATUS=$(echo $LINE | awk '{print $3}')
        ;;
     esac
     # Increment the LOOP counter
     ((LOOP = LOOP + 1))
     if ((LOOP == 3)) # Do we have all three lines of data?
```

```
then
        # Check printing status
        case $PSTATUS in
        disabled) lpc start $Q >/dev/null
                 (($? == 0)) && echo "\n$Q printing re-started\n"
                 ; ;
        enabled |*) : # No-Op - Do Nothing
                ;;
        esac
        # Check queuing status
        case $QSTATUS in
        disabled) lpc enable $Q >/dev/null
                 ((\$? == 0)) \&\& echo "\n\$Q gueueing re-enabled\n"
                 ;;
        enabled |*) : # No-Op - Do Nothing
                 ;;
        esac
        LOOP=0 # Reset the loop counter to zero
   fi
done
}
*****
****
# Is CUPS Running? If CUPS is running we can just
# run the CUPS standard commands.
ps auxw | grep -q [c]upsd
if (( $? == 0 ))
then
  CUPS_printing
   exit $?
fi
# What OS are we running?
# To start with we need to know the UNIX flavor.
# This case statement runs the uname command to
# determine the OS name. Different functions are
# used for each OS to restart printing and queuing.
case $(uname) in
AIX) # AIX okay...Which printer subsystem?
    # Starting with AIX 5L we support System V printing also!
```

```
# Check for an active qdaemon using the SRC lssrc command
     if (ps -ef | grep '/usr/sbin/qdaemon' | grep -v grep) \
         >/dev/null 2>&1
     then
           # Standard AIX printer subsystem found
           AIX_PSS=CLASSIC
     elif (ps -ef | grep '/usr/lib/lp/lpsched' | grep -v grep) \
           >/dev/null 2>&1
     then
           # AIX System V printer service is running
           AIX_PSS=SYSTEMV
     fi
     # Call the correct function for Classic AIX or SysV printing
     case $AIX_PSS in
     CLASSIC) # Call the classic AIX printing function
              AIX_classic_printing
              ;;
     SYSTEMV) # Call the AIX SysV printing function
              AIX_SYSV_printing
              ;;
     esac
      ;;
HP-UX) # Call the HP-UX printing function
       HP_UX_printing
      ;;
Linux) # Call the Linux printing function
       Linux_printing
      ;;
OpenBSD) # Call the OpenBSD printing function
        OpenBSD_printing
      ;;
SunOS) # Call the Solaris printing function
        Solaris_printing
      ;;
*)
        # Anything else is unsupported.
        echo "\nERROR: Unsupported Operating System: $(uname)\n"
        echo "\n\t\...\n"
      ;;
esac
```

For each of the operating systems and, in the case of AIX, each printer service, we took the previously created shell scripts, extracted the code, and placed it between function function_name { and the function-ending character }. We now have the following functions:

```
AIX_classic_printing
AIX_SYSV_printing
CUPS_printing
HP_UX_printing
Linux_printing
OpenBSD_printing
Solaris_printing
```

To execute the correct function for a specific operating system, we need to know the UNIX flavor. The **uname** command returns the following output for each of our target operating systems:

OS	uname Output
AIX	AIX
HP-UX	HP-UX
Linux	Linux
OpenBSD	OpenBSD
Solaris	SunOS

With the exception of AIX, this information is all that is needed to execute the correct function. But with AIX we have to determine which printer service is running on the server. Both types of print services have a process controlling them, so we can grep for each of the processes using the ps -ef command to find the currently running printer service. When the classic AIX printer subsystem is running, there is a /usr/sbin/qdaemon process running. When the System V printer service is running, there is a /usr/lib/lp/lpsched process running. With this information we have everything needed to make a decision on the correct function to run.

We added at the end of the script all of the function execution control in the case statement that is shown in Listing 25-17.

Listing 25-17 Controlling case statement listing to pick the OS

```
elif (ps -ef | grep '/usr/lib/lp/lpsched' | grep -v grep) \
           >/dev/null 2>&1
     then
           # AIX System V printer service is running
           AIX PSS=SYSTEMV
     fi
     # Call the correct function for Classic AIX or SysV printing
     case $AIX_PSS in
     CLASSIC) # Call the classic AIX printing function
               AIX_classic_printing
              ;;
     SYSTEMV) # Call the AIX SysV printing function
              AIX_SYSV_printing
              ;;
     esac
      ;;
HP-UX) # Call the HP-UX printing function
        HP_UX_printing
      ;;
Linux) # Call the Linux printing function
        Linux_printing
      ;;
OpenBSD) # Call the OpenBSD printing function
         OpenBSD_printing
      ;;
SunOS) # Call the Solaris printing function
        Solaris_printing
      ::
*)
        # Anything else is unsupported.
        echo "\nERROR: Unsupported Operating System: $(uname)\n"
        echo "\n\t\t...EXITING...\n"
      ;;
esac
```

I hope by now that the code in the case statement is intuitively obvious to read and understand. If not, the first line of the case block of code is the uname command. At this point we know what the OS flavor is. For HP-UX, Linux, OpenBSD, and Solaris we execute the target OS printing function. For AIX we make an additional test to figure out which one of the supported printing services is running. The two options are System V and the Classic AIX printer subsystem. Notice that I removed all of the logging functionality from the functions. With this type of setup, where you have the functions doing the work, you can move the logging out to the main body of the shell script. This means that you can capture all of the output data of the function to save to a log file, use the **tee** command to view the data while logging at the same time, or just point it to the bit bucket by redirection to /dev/null.

Other Options to Consider

As usual, we can always improve on a shell script, and these shell scripts are no exception. Some options that you may want to consider are listed next.

Logging

You may want to add logging with date/time stamps. If you are having a lot of trouble keeping certain print queues up, studying the log may give you a trend that can help you find the cause of the problem. Some queues may drop in a particular location more than others. This can indicate network problems to the site. Any time you start logging do not forget to keep an eye on the log files! I often see that a script is added to a production machine, and the next thing you know, the log file has grown so large that it has filled up the filesystem. Don't forget to prune the log files. Trimming the log files is another little shell script for you to write. *Hint: tail command*.

Exceptions Capability

In a lot of shops you do not want to enable every single printer and print queue. In this case you can create an *exceptions* file, which contains the queue/printer names that you want to exclude from enabling. You also may have special considerations if your shop uses specific forms at different times on some of the *floating* printers. Some shops are just print-queue hell! Having the capability to keep the majority of the printers active all of the time and exclude a few is a nice thing to have.

Maintenance

During maintenance windows and other times when you want to stop all printing, you may want to comment out any cron table entries that are executing the enabling scripts. You usually find this out after the fact.

Scheduling

I keep a script running 24×7 to keep all of the printers available. You may want to tailor the monitoring scheduling to fit business hours (my requirement is 24×7). Users' loading up on print jobs during the middle of the day is always a problem, so we try to hold big jobs for times of low activity. Low-activity times are the times when you want to be at home, so make sure you are keeping the printers printing during these hours, or the next morning you will have the same problem.

Summary

In this chapter we covered some unique techniques to handle the data from command output. In many scripts we used arrays to hold the data as array elements. In other cases we read in a line at a time and used tags to grab the data we needed. We learned how to process a specific number of lines of data in groups by using a loop counter within a while loop.

The techniques in this chapter are varied, but the solutions are readable and can be easily maintained. Someone will follow in your footsteps and try to figure out what you did when you wrote the shell script. Do not play the "job security" game because you are you own worst enemy when it comes to documenting your shell scripts. If you comment when you write the script and make a note in the REV section when you edit it, you will have a long, happy life using your shell script.

In the next chapter we are going to move into the world of the government-induced audits you will have to endure if you are working at a "public" company, SOX! What's SOX, you ask? Keep reading to find out about your worst nightmare coming to a cubicle near you.

Lab Assignments

- 1. Using the techniques from Chapter 17, "Filesystem Monitoring," modify the PQ_UP_manager.ksh script to allow *exceptions*. Sometimes we do not want to enable all of the printers or queues when we do printer maintenance and when a printer is not working. Hint: see Listing 17-14 and Listing 17-15.
- 2. Rewrite the PQ_UP_manager.ksh shell script in Listing 25-16 to log all printer and print queue "down" events with a date/time stamp. The new log file should reside in the /usr/local/log/ directory.
- 3. Write a new Bash shell script that will "prune" the log file to the last 3000 lines of data. Name this new script prune_printer_log.bash. Schedule the prune_printer_log.bash shell script to execute every day at 12:00 midnight using a cron table entry.

CHAPTER 26

Those Pesky Sarbanes-Oxley (SOX) Audits

If you work for any publicly traded company registered with the United States Securities and Exchange Commission (SEC), you have either already had an IT audit, or you are preparing for an upcoming IT audit. You can thank the Sarbanes-Oxley (SOX) act of 2002 for this extra headache. The SOX act was a result of the corporate financial scandals of Enron and the like. The purpose of SOX is to require the chief executive officer (CEO) and the chief financial officer (CFO) of all publicly traded companies to personally validate the accuracy of its financial records, and to ensure there are internal controls in place to protect all financial data. That last part is where all the IT staff entered the picture. You have probably noticed that the system change-request process has become burdensome with all the detailed plans for any system change and the back-out plans if something goes wrong. This is a good thing to do, but some organizations have gone overboard with the amount of detail required. As Systems Administrators, it falls on us to try to make sense of the new rules and to comply with the auditors. I am not going to cover the law, but rather tactics for UNIX Administrators to be prepared for an audit. Just remember that we are not the only department that will have to pass the audits. The Mainframe Team, the DBA Team, the Win-Tel Team, and the Network Team also have to pass the audits. So, be prepared to work with all these teams as the audit date approaches.

In this chapter, I am going to walk you through a typical audit of the UNIX machines in your landscape and give you some hints on making the process easy. Most people who have to work with the auditors hate spending time answering the same questions over and over again. Well, think about it; the auditors have their "script" to go by. This is not a shell script, but a set of criteria that they want to verify for a select set of UNIX machines. They want to look at the /etc/passwd and /etc/shadow files. Here they want to make sure that everyone has a password. They want to ensure that all non-essential network services are turned off. They want telnet, ftp, rsh, rcp, and all other non-encrypted communications disabled and their Open Secure Shell counterparts used instead. So, as you can see, because the questions they ask are pretty

limited in scope, you get asked the same questions by all of the different auditors. This makes the audit process a bit more predictable.

What to Expect

You will either be in a staff meeting or get an email one day talking about the upcoming audit. This will make every manager turn into a yes-man/yes-woman to any request upper management makes to get ready for this audit. It is upper management's butt that is on the line and everything runs downhill from there. There will be a lot of pressure and this audit will be granted emergency status, and you may even open up the "war room." You may be thinking that I'm making this up, but I'm not! This is serious stuff and they do not want to fail an audit; "Failure is Not an Option!"

The first audit is an internal audit. A different auditing firm will perform the internal audit that the "real" audit uses. Oh yes, there is more than one auditor to deal with. Internal auditors are on your side, so to speak. They will scan some systems and want to sit down with a Systems Administrator and query a representative set of machines. They will want printouts of the query outputs and other data. Just expect it. The auditors will analyze the data and let you know where the systems are not compliant. You go back and fix the issues raised in the first round of audits, and then they scan the machines again to ensure the issues are corrected.

After you go through all that, the real auditors arrive on site. They will want to run some system scans and sit down with a Systems Administrator to query a representative set of machines to ensure they are compliant. Sound familiar? They ask the same questions that the first set of auditors asked. You just repeat the same process again, and you should pass the audit with flying colors.

How to Work with the Auditors

If you are the lucky soul who "gets to" work with the auditors, I have a simple way to start the automation process for the audits. You and the auditors sit at your terminal. Ask the auditors which machine they want to see first. Log in to that server and get root access. You will need root access to complete an audit. When you get logged in, create a directory somewhere called "auditstuff". I sometimes use /tmp/auditstuff. Then we just create a script session, so we log and save every step as we audit this first machine:

script /tmp/auditstuff/audit-hostname.log

Once you start the script session, ask the auditors what they want to see. Then just show them anything they want to look at. When finished with that machine, press Ctrl+D to exit the script session, which saves the /tmp/auditstuff/audithostname.log file. Now that we know what they want to see, we can just make a copy of the script file and edit it to extract all the commands we used to answer the audit questions, and now we have a shell script to do the system queries for us. Now you can just copy this new script, maybe named /tmp/auditstuff/sys-audit.sh, to the other machines and execute it, saving the data in a log file for the auditors to review.

It may not be as quick as that, because when you find out what the auditors want to look at, you need to make your queries more precise. Then you can make the output in more of a report form by parsing through the data. You definitely need to brush up on the **find** command, because you're going to have to search the system for different sets of files. Files include those with the sticky bit set, world-writable files, and many others. You're also going to need to parse out the data of interest from some queries, so also brush up on awk, sed, cut, and the various looping-control structures.

What the Auditors Want to See

The full scope of a SOX audit encompasses the entire network infrastructure, but we are limiting our discussion to the UNIX environment and auditing specific UNIX machines. To pass a SOX audit, there is a basic set of criteria that must be met at the machine level. Some of these include file and directory permissions to system-critical and system-configuration files, world-writable files, all non-encrypted communications must be disabled, and all user accounts have a password and that password expires on a periodic basis. The following is a broader list of system-specific tasks to harden a system to pass a SOX audit:

- Disable remote login for all service or application accounts, such as root, oracle, and dbadm. Users must log in with their personal account and then switch-user (su) to the service account.
- All user accounts must have a password.
- All users' passwords must expire on a regular basis (although there are always exceptions for service and application system accounts).
- Ensure that passwords are of a minimum length, and are a mix of alphanumeric and special characters.
- Disable all accounts that have not been accessed in 30 days.
- Remove or disable all unused default system and application accounts, such as uucp.
- Review all user accounts that are a member of any administrative group (system, root, and so on). Each user assigned to a service group will need to be verified and documented.
- Disable telnet, ftp, rsh, remsh, rlogin, rstatd, tftpd, talkd, bootps, fingerd, uucp and any other non-encrypted communications.
- Install the latest version of Secure Shell and use ssh in place of telnet and rsh, use scp in place of rcp, and use sftp in place of ftp. You can download OpenSSH at www.openssh.org.
- Ensure that the file permissions to all system-critical and system-configuration files and directories are set correctly. Files and directories include /etc/passed, /etc/shadow, /etc/security/passwd (AIX), /etc/hosts, /etc/services, /etc, /usr/sbin, /opt, /etc/security, and /usr/lib. There are many more, and every UNIX flavor has a different set of files and directories.

- Ensure that all UNIX servers have the latest operating system patches and release, if possible. This is always a hard one, and you really have to work with the business unit that owns the applications to work in a maintenance window for patches and upgrades. You really need the latest patches because most likely the auditors will also probe network ports looking for vulnerabilities.
- You should have a documented plan for scheduled upgrades and patches for both the operating system and applications.
- You may have a requirement to install Tripwire to send alerts when critical system files are modified.
- You may have a requirement to set up a central audit server to hold all of the UNIX system's syslog files.
- Search the system for all files that have the sticky bit set, allowing users extended privileges.

This list should give you a pretty good idea about the topics the auditors will look at.

Some Handy Commands

You need to get familiar with a set of commands that will make your life easier when dealing with auditor questions. Most commands you use weekly, but you might not see some of the extra command switches to limit data output.

Using the id Command

Because we started with user access to the system, let's start with the **id** command. The id command can tell us about various aspects of a user-account definition. We can list the group memberships, both real and effective, GID, group name, and UID. The little script in Listing 26-1 queries the /etc/passwd file and lists the group membership of each user account.

Listing 26-1 search_group_id.Bash shell script

NOTE Solaris does not support awk -F :. For Solaris we must use nawk (new awk). This case statement will alias awk to nawk if the UNIX flavor is SunOS:

```
case $(uname) in
SunOS) alias awk=nawk
;;
esac
```

Notice in Listing 26-1 that we specify the : (colon) character as a field separator with the syntax awk -F :. This allows us to use '{print \$1}' to extract the username as the first field of every line in the /etc/passwd file. With this output listing the usernames, we pipe this data to a while read ID loop. Then for each \$ID, we echo the username to the screen, continuing on the same line specified by echo -e ''\$(ID)\c''. The -e switch on the echo command enables the backslash operator \c in Bash shell. Then we use the id command with the -g and -n switches. This syntax tells the id command to limit output to the group *name*, as opposed to the group ID (GID) number.

Using the find Command

Next let's look at the find command, one of the most powerful commands on any UNIX system. You can search a system for any type of file, including files, directories, links, sockets, block special files, character special files, and many others. You can search the system for files that are world-readable, world-writable, files with the sticky bit set, as well as perform an action on each file found. For example, you might want to find all the files modified in the past seven days to back up to tape:

find . -mtime -7 | xargs tar -cvf /dev/rmt0

This find command will search for files modified within the past seven days, starting in the current directory and then searching all subdirectories. This output is piped to xargs, which adds the previous output as arguments to the tar command.

The find command can also search for files that have the sticky bit set by using the -perm -1000 -type f command switches:

find / -perm -1000 -type f

This find command searches the entire system, starting in the root directory, for files of type file that have minimum permissions of the sticky bit set.

To find all files on the system that are world-writable, use the following find command syntax:

find / -perm -2 -exec ls -1 {} ;

This find command searches the entire system, beginning in the root directory, for any type of file (including directories) that is world-writable. For each file found, the -exec switch specifies to perform a long listing. Notice that we ended the -exec switch option with a blank space followed by a backslash-semicolon, \;. This is a requirement to terminate the -exec switch option.

To find all the .rhosts files on the system and show the contents of each file, use the following syntax:

find / -name .rhosts -print -exec cat {} \;

This find command searches the entire system, beginning in the root directory, for any file named .rhosts. For each file found, the -exec switch lists the filename followed by the contents of each file. Again, notice that we ended the -exec switch option with a blank space followed by a backslash-semicolon, \;. This is a requirement to terminate the -exec switch option.

Using the awk and cut Commands

We can parse fields out of data by using the awk and cut commands. Both of these commands work on rows, or lines of data. The cut command can also cut out specific characters in a line of data by specifying the specific character placement range in the string.

To extract the first field of data from an entire file using the awk command, we use the following syntax:

awk '{print \$1}' filename

or

cat filename | awk '{print \$1}'

Both of these commands will display the entire file while limiting the output to the first field of each line in the file, using a space for the field delimiter. As an example, we want to extract all the IP addresses from the /etc/hosts file. Because the IP address is the first field in the /etc/hosts file, the following command will work:

awk '{print \$1}' /etc/hosts

The original /etc/hosts file is shown in Listing 26-2.

```
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1 localhost.localdomain localhost
::1 localhost6.localdomain6 localhost6
192.168.1.100 yogi # AIX
192.168.1.102 booboo # Linux
192.168.1.103 fred # OpenBSD
```

Listing 26-2 /etc/hosts file

If we run the previous awk statement, we get the following output:

```
# awk '{print $1}' /etc/hosts
#
#
127.0.0.1
::1
192.168.1.100
192.168.1.102
192.168.1.103
```

If we want to extract more than one field of data, we add the desired positional parameter, separating each field with a comma (,) and a space, if we want the field separated. If we want both strings concatenated together, we leave out the space, as shown here:

```
awk '{print $1,$2}' /etc/hosts
```

To extract the IP address and hostname fields, \$1, and \$2, respectively, while leaving white space between the values, we can use either of the following commands:

```
awk '{print $1, $2}' /etc/hosts
```

or

cat /etc/hosts | awk '{print \$1, \$2}'

The resulting output of this awk statement is shown here:

```
# awk '{print $1, $2}' /etc/hosts
# Do
# that
127.0.0.1 localhost.localdomain
::1 localhost6.localdomain6
192.168.1.100 yogi
```

```
192.168.1.102 booboo
192.168.1.103 fred
```

We still have a little extra output we are not interested in. We can omit the comment lines, those beginning with a hash mark (#), by adding a pipe grep -v '^#' to the end of the previous command statement. The caret (^) specifies *begins with*, so this grep statement matches any line of data that begins with a hash mark (#). The -v switch tells grep to show everything *except* what grep pattern matched on, thus deleting the unwanted comment lines. The result of this command is shown here:

```
# awk '{print $1, $2}' /etc/hosts | grep -v '^#'
127.0.0.1 localhost.localdomain
::1 localhost6.localdomain6
192.168.1.100 yogi
192.168.1.102 booboo
192.168.1.103 fred
```

This is better, but we still have the loopback address, 127.0.0.1, and an entry for IPv6 protocol (::1) that we are not interested in. To remove these, we add a second pipe to egrep -v '127.0.0.1|::1' (extended grep) to the statement to match multiple patterns. The result of this addition is shown here:

```
# awk '{print $1, $2}' /etc/hosts | grep -v '^#' | egrep -v '127.0.0.1|::1'
```

192.168.1.100 yogi 192.168.1.102 booboo 192.168.1.103 fred

Now we have only the IP addresses and hostnames listed without the unwanted data.

The cut command can also be used to extract one or more fields of data. For example, to cut the IP address from the /etc/hosts file, we use either of the following commands:

```
# cut -f1 /etc/hosts
#
#
127.0.0.1
::1
192.168.1.100
192.168.1.102
192.168.1.103
or
# cat /etc/hosts | cut -f1
#
#
127.0.0.1
::1
192.168.1.100
```

```
192.168.1.102
192.168.1.103
```

However, some Linux distributions require us to specify the field separator for the cut command even if it is white space. To specify the field delimiter, we add -d followed by the field delimiter character, as follows:

```
# cut -d ' ' -f1 /etc/hosts
#
#
127.0.0.1
::1
192.168.1.100
192.168.1.102
192.168.1.103
```

The previous example specifies a blank space as the field delimiter. We can also extract multiple fields with the cut command by adding the additional field numbers separated by a comma (,), but *no space*. The following cut command extracts the first and second fields from the /etc/hosts file:

```
[root@booboo scripts]# cut -d ' ' -f1,2 /etc/hosts
# Do
# that
127.0.0.1 localhost.localdomain localhost
::1 localhost6.localdomain6 localhost6
192.168.1.100 yogi
192.168.1.102 booboo
192.168.1.103 fred
```

We can again strip out the unwanted lines of data with a single egrep statement, egrep -v '^# | 127.0.0.1 | ::1', as shown here:

```
# cut -d ' ' -f1,2 /etc/hosts | egrep -v '^#|127.0.0.1|::1'
192.168.1.100 yogi
192.168.1.102 booboo
192.168.1.103 fred
```

Now let's look at parsing through the /etc/passwd file. The /etc/passwd file is a field-delimited file with the colon (:) character being the field delimiter, as shown here:

```
root:x:0:0:root:/root:/bin/Bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
oracle:x:205:205::/home/oracle:/bin/Bash
sterling:x:513:514::/usr2/sterling:/bin/Bash
kingjon:x:0:0::/home/knudsepe:/bin/Bash
```

This is part of a Linux /etc/passwd file. Let's look at an example of looking for all users in the /etc/passwd file who are members of the system, or root, group. The group ID (GID) is located in the third field on each line of the file, and the system/root groups are specified by the number 0. The example shown here uses an awk statement to extract the first and third fields, and then we check to see if the GID is equal to 0. If so, we display a warning to the user. This time we need to tell awk what to use for a field delimiter by specifying awk -F : to declare a colon (:) as a field delimiter, overriding the awk command default of white space.

NOTE The Solaris implementation of awk does not support declaring a field separator with awk -F :. For Solaris, use nawk (new awk) instead. You can add the following case statement in the variable declaration of the section to correct this problem:

```
case $(uname) in
SunOS) alias awk=nawk
;;
esac
```

This case statement will alias awk to nawk if the UNIX flavor is Solaris, specified by SunOS.

Take a look at the chk_passwd_gid_0.Bash script shown in Listing 26-3, and we will look at the details at the end.

```
#!/bin/Bash
#
# SCRIPT: chk_passwd_gid_0.Bash
# PURPOSE: This script searches the /etc/passwd
#
  for all non-root users who are a member of
#
   the system/root group, GID=0
#
*****
# DECLARE FILES AND VARIABLES HERE
*****
case $(uname) in
SunOS) alias awk=nawk
         ;;
esac
**********
# BEGINNING OF MAIN
*****
awk -F ':' '{print $1, $3}' /etc/passwd | while read U G
```

Listing 26-3 chk_passwd_gid_0.Bash shell script

```
do
    # If the user is root skip the test
    if [ $U != 'root' ]
    then
        # Test for GID=0
        if (( G == 0 ))
        then
            echo "WARNING: $U is a member of the root/system group"
        fi
    fi
    done
```

Listing 26-3 (continued)

This chk_passwd_gid_0.Bash shell script in Listing 26-3 begins with an awk statement to extract data fields \$1 and \$3, and then sends this output to a while read U G loop. Next we test to see if the user ID for each line of data read in is the root user. If root, we skip the test because root is supposed to be part of the system/root group. Otherwise, we test to see if the GID captured from the data field is 0. If so, we display a warning message to the user and continue looping. The result of running this script with one user being a member of the system/root group is shown here:

```
# ./passwd_file_test.Bash
WARNING: kingjon is a member of the root/system group
```

If we tried to use the cut command for this task, it would not work as well because the cut command displays the field separator along with the data if more than one field of data is extracted, as shown here:

```
cut -d ':' -f1,3 /etc/passwd
root:0
bin:1
daemon:2
adm:3
lp:4
oracle:205
sterling:513
kingjon:0
```

If we had piped this cut command output to a while read U G loop, as we did with the awk statement, it would fail. To make it work, we would have to pipe it to a sed command and substitute a blank space for each colon (:). Extracting a single field using cut will result in only the data, without an additional field separator, being displayed, but multi-field extractions using the cut command will include the field separator in the output.

So, depending on what we are doing, sometimes we want to use awk and sometimes cut is better. Just be aware that each has its limitations.

Using the sed Command

Now let's look at sed. The sed program is a stream editor that we use for character substitution in a file, or other output data string. Use the following syntax to substitute data with sed:

```
sed s/'current_string'/'new_string'/g filename
```

or

```
cat filename | sed s/'current_string'/'new_string'/g
```

Both commands will display the entire file with every occurrence of current_string replaced with new_string. In the sed part of the statement, the s specifies substitute, and the g tells sed to do a global substitution.

As an example, suppose we have a file named fruit.txt that contains the following data:

apple orange peach banana pineapple grape **bluebarry blackbarry**

Notice that the last two entries are spelled incorrectly. We can use a sed statement to substitute the correct spelling for "berry", which is misspelled "barry", as shown here:

```
# sed s/barry/berry/g fruit
apple
orange
peach
banana
pineapple
grape
blueberry
blackberry
```

That fixed the problem. We can also cat the file and pipe the output to sed, as shown here:

```
# cat fruit | sed s/barry/berry/g
apple
orange
peach
banana
pineapple
grape
```

blueberry blackberry

We can also use sed to delete all the blank lines in a file using the following syntax:

```
sed /^$/d filename
```

or

```
cat filename | sed /^$/d
```

This should be intuitively obvious. If the caret ($\hat{}$) means *begins with*, and a dollar sign (\$) means *ends with*, we are saying *if it begins with the end of line, delete it*. The trailing /d in the sed statement means delete.

We can do the same thing with a command using the following syntax:

some_command | sed /^\$/d

If you want to save the output, redirect it to a file with either > to create a new file or overwrite an existing file, or >> to create a new file or append text to the end of an existing file. We can also pipe the output to tee -a *outputfile* to both display the output to the screen and append the output to the *outputfile* at the same time, as shown here:

```
some_command | sed /^$/d | tee -a outputfile
```

This command will create the *outputfile* if it does not already exist, and will append it to an already existing file.

Using the dirname and basename Commands

When we need to separate a fully qualified filename into the directory path and the filename, we can use the **dirname** and **basename** commands, respectively. Both commands require a single command-line argument.

The dirname command will return the directory part of a full-path filename, as follows:

```
# dirname /usr/local/bin/somescript.Bash
/usr/local/bin
```

The basename command will return the filename part of a full-path filename, as shown here:

basename /usr/local/bin/somescript.Bash
somescript.Bash

These are handy tools for a lot of scripts. We have seen throughout this book a technique to capture the name of an executing shell script, or function, using the following command:

basename \$0

This basename command will return the name of the shell script, if executed in the main body of the shell script, or the name of the function, if executed inside a function from within a shell script. In many shell scripts we capture the name of the shell script in the variable declaration section of the shell script using the following variable assignment in association with command substitution, as shown here:

THIS_SCRIPT=\$(basename \$0)

The \$() specifies command substitution. Enclosing a command in back tics, `command`, is also a method of command substitution. The result of the basename \$0 command is assigned to the variable THIS_SCRIPT. We can do the same thing with a function with this variable assignment in association with command substitution:

```
THIS_FUNCTION=$(basename $0)
```

Same command, but the scope is different depending on where the variable assignment is made. These are also handy tools when writing scripts to create reports.

Other Things to Consider

As you get into the audit process, you will have a better idea of what your particular auditor wants to see. The find command will quickly become your friend, so study the find manual page, **man find.** There are so many options to the find command that it may take you several days to play with them all.

For everything an auditor wants to see you can write a shell script to query the system to extract the desired data. The trick is to limit and parse the output to get the desired data in a usable format. This is where awk, sed, and cut help you out. Once you have the scripts set up, you can have anybody, well just about anybody, run the scripts for you. Then you can get back to some real work.

Summary

I added this chapter at the request of several Systems Administrators I have worked with over the past few years. I was always the lucky soul who "got to" work with the auditors. I found that if you automate the process, you can respond to auditor requests very quickly. And believe me, management will see this!

The most important thing to remember with any audit, whether SOX or PCI (financial institutions processing more than six million credit cards per year), is that it's upper management's butt on the line, and everything goes downhill from there.

In the next chapter we are going to study using rsync with Dirvish to perform snapshot-type backups. I hope your audits go well. See you in the next chapter.

Lab Assignments

- 1. Write a shell script that will list all user accounts that have membership in the root, system, sys, adm, admin, printq, dba, or nobody groups.
- 2. Write a shell script that will query the entire system for all directories that have the sticky bit set. Limit the query to not search /opt and /proc.
- 3. Write a shell script that will search the system for all regular files that are world-writable that have been created in the past 30 days.
- 4. Write a shell script that will search the system for all directories that have been modified in the past 10 minutes.
- 5. Write a shell script that will search the system for all regular files that have been accessed between 5 and 10 minutes ago.

CHAPTER 27

Using Dirvish with rsync to Create Snapshot-Type Backups

Six months ago I hadn't heard of Dirvish and I had used **rsync** fewer than 10 times. Now when I need to copy files either locally or over the network, I like to use rsync for its compression capabilities and, after an initial copy, the ability to replicate only changes *in a file* as opposed to *changed files*. Usually when a file is modified it is only a small percentage of the file that has changed, so sending only the updates greatly reduces the amount of data transmitted over the network. We are able to do this feat because of hard links and inodes. Read on; this is a very beneficial chapter for any shop, and it's free!

Dirvish is a backup program, originally written by J.W. Schultz, that you can download from www.dirvish.org. Dirvish uses rsync to transfer files over the network, and thus uses the rsync *remote-update protocol*, which allows us to do an incremental backup by transmitting only the *changed parts of the changed files*, as opposed to a tape incremental backup that requires transmitting the entire contents of *all changed files*. We studied rsync in detail in Chapter 7, "Using rsync to Efficiently Replicate Data."

The Dirvish backup program uses one or more backup servers with the image data stored on disk. Ideally, this backup server should be located in another building, or location, and then we have offsite backup storage. You may still want to make periodic backup tapes for legal purposes on a monthly and yearly basis.

In this chapter I want to accomplish two things:

- Familiarize you with the details of Dirvish configuration and how Dirvish works on the command line.
- Write a shell script to do all the hard work for us. This is a menu-driven user interface that will allow us to interact with Dirvish to configure a new server to back up, delete a server backup, restore files, and all the other things we need to do with backup images.

How Does Dirvish Work?

Dirvish is a backup program that produces *snapshot-style* backups to disk storage by utilizing the rsync remote-update protocol. Traditionally, backups have always been stored on tapes, and more recently CD-ROM and DVD optical storage. Dirvish is different in that it stores the *image filesystem trees on disk* on the backup server.

Typical backups consist of a periodic full system backup, followed by a set of incremental backups to archive only the files that have changed since the last backup. This process continues until the next full system backup, when the process repeats. Then when we have to restore a system we have to first extract the file(s) from the full backup, and then extract data from each incremental backup until we have the data current for that date. With Dirvish, after we perform the initial backup, all backups are incremental. Additionally, only the *changes* in the files, *not the entire files*, are transferred over the network. This magic is performed with the help of rsync under the covers using hard links associated with inodes. The idea is that after we make an initial backup as a set of reference files, we only need to transfer the changes to a file over the network with the remaining data replicated with pointers to the reference image files. Even though each backup is "incremental," each backup image is completely intact. To restore, we just copy the files to wherever we need them. We will write a shell script that will take care of these details for us.

How Much Disk Storage Will I Need?

So, how does this "incremental backup" work and how much disk space do we need? To understand the guts of Dirvish, we need to get down to the filesystem level and understand the concept of an *inode*. Each separate file and directory in a filesystem has a unique number associated with it, called an inode. The operating system references files on the system by an inode number. As we know, directories are just special files in UNIX. A directory is a special file with a list of inodes that map to each of the files that reside in the directory. So, as with any pointer, you can have a directory that has in its list more than one filename pointing to the same inode. Using this technique, Dirvish is able to represent multiple files by pointing to the same inode, just as we create soft and hard links to point to files or directories to save disk space and to set links for application files and directories. This is nothing more than a hard link! Therefore, the disk space requirements for Dirvish will be much less than you first realize. Depending on the data you are backing up and time of retention before the data expires, you can start with 1.5–3 times the source data. This is just an estimate to get you started. Ideally, we expire the oldest backup just prior to performing the new backup to help conserve space. Separate filesystems should be created on the Dirvish backup server(s) to store the backup image data.

Configuring Dirvish

There is a great guide to using Dirvish, *Configuring and Using Dirvish for Snapshot Backups* written by Jason Boxman, at http://edseek.com/~jasonb/articles/dirvish_backup/index.html. You should reference this study guide for more advanced

details on Dirvish. I have referenced this document to write this chapter because it is the best document I have found. You can find another source of good information on Dirvish in the wiki at http://www.dirvish.org/wiki?VaultBranch (be sure the *V* and *B* are capped as shown here), and, of course, the official Dirvish web site, http://www.dirvish.org. As we go through this chapter, I will also point out specific manual pages you can look at for more detail.

We need to start with the Dirvish lingo and structure. Dirvish uses the analogy of a big banking firm. We use terms like *bank*, *branch*, and *vault* to describe locations where backup images are stored. It is always a good idea to create a separate filesystem for Dirvish backup images, such as /prod_backups or /backup_bank1. A bank is a directory (such as /backup_bank1) that contains vaults (subdirectories of server names to back up specifying a top-level directory tree) only. Do not store anything else in a bank directory. A bank is defined in the master configuration file (master.conf), allowing us to configure more than one bank if we desire. A vault is where the backup image is stored for a particular filesystem. There is one vault for every *filesystem* we back up with Dirvish. The best way to use Dirvish is to specify the root directory (/) as the backup tree to archive, and create a list of root subdirectories and files to exclude from the backup. An *image* is the entire directory tree structure that resulted from a Dirvish backup operation. We also have the concept of a branch. A *branch* is a way to archive multiple clients having very similar data by sharing the same vault. For example, if the yogi and booboo servers contain almost the same data, then when we perform a backup, two directory tree structures are created in the *shared* vault with the file and date format yogi-YYMMDDHHMMSS and booboo-YYMMDDHHMMSS.

Installing Dirvish

On my Linux machine, I installed Dirvish using yum, although apt works, too. Using yum takes care of installing the Dirvish dependencies. For details on setting up yum or apt refer the respective manual pages, **man yum** and **map apt**. Dirvish requires two Perl modules: Time::ParseDate and Time::Period. Installing Dirvish with yum is the simplest method, as shown in Listing 27-1.

Listing 27-1 Installing Dirvish on Linux with yum

--> Processing Dependency: perl(Time::ParseDate) for package: dirvish --> Processing Dependency: perl(Time::Period) for package: dirvish --> Restarting Dependency Resolution with new changes. --> Running transaction check ---> Package perl-Time-Period.noarch 0:1.20-1.fc6 set to be updated ---> Package perl-Time-modules.noarch 0:2003.1126-4.fc6 set to be updated ---> Package dirvish.noarch 0:1.2.1-2.fc6 set to be updated Dependencies Resolved _____ Arch Version Repository Size Package _____ Installing: noarch 1.2.1-2.fc6 fedora 51 k dirvish Installing for dependencies: perl-Time-Period noarch 1.20-1.fc6 fedora 14 k perl-Time-modules noarch 2003.1126-4.fc6 fedora 36 k Transaction Summary _____ Install 3 Package(s) 0 Package(s) Update Remove 0 Package(s) Total download size: 101 k Is this ok [y/N]: y Downloading Packages: 00:00 00:00 00:00 Running Transaction Test Finished Transaction Test Transaction Test Succeeded Running Transaction Installing: perl-Time-modules Installing: perl-Time-Period Installing: dirvish Installed: dirvish.noarch 0:1.2.1-2.fc6 Dependency Installed: perl-Time-Period.noarch 0:1.20-1.fc6 perl-Time-modules.noarch 0:2003.1126-4.fc6 Complete!

Listing 27-1 (continued)

[root@booboo scripts]# dirvish
vault undefined
[root@booboo scripts]#

Listing 27-1 (continued)

You can also download the latest version of Dirvish from http://www.dirvish .org. Then extract the tarball files to a directory somewhere. Make sure to read the INSTALL document. Now we just execute the install.sh script and answer a few questions, as shown in Listing 27-2.

```
[root@booboo Dirvish-1.2]# sh install.sh
perl to use (/usr/bin/perl)
What installation prefix should be used? () /usr/local
Directory to install executables? (/usr/local/bin)
Directory to install MANPAGES? (/usr/local/man)
Configuration directory (/etc/dirvish) /usr/local/etc
Perl executable to use is /usr/bin/perl
Dirvish executables to be installed in /usr/local/bin
Dirvish manpages to be installed in /usr/local/man
Dirvish will expect its configuration files in /usr/local/dirvish
Is this correct? (no/yes/quit) yes
Executables created.
Install executables and manpages? (no/yes) yes
installing /usr/local/bin/dirvish
installing /usr/local/bin/dirvish-runall
installing /usr/local/bin/dirvish-expire
installing /usr/local/bin/dirvish-locate
installing /usr/local/man/man8/dirvish.8
installing /usr/local/man/man8/dirvish-runall.8
installing /usr/local/man/man8/dirvish-expire.8
installing /usr/local/man/man8/dirvish-locate.8
installing /usr/local/man/man5/dirvish.conf.5
Installation complete
Clean installation directory? (no/yes) yes
Install directory cleaned.
```

Listing 27-2 Installing Dirvish with install.sh

With Dirvish installed, we are now ready to do a little configuration.

Modifying the master.conf Dirvish Configuration File

On startup Dirvish expects to find its master configuration file in /etc/dirvish.conf or /etc/dirvish/master.conf. This is not a typo! Let's just call it master.conf to not get confused. The master.conf file contains all the configuration information and default definitions and values for Dirvish. This master.conf file uses the *stanza* file format. Using the stanza format we can specify single values or lists of values. Single values have the form option:value. Lists of options must be specified on multiple lines, as shown here:

option:

value1 value1 value3 . . . valuen

Every value must reside on a different line and must be indented by any type of white space (spaces or tabs); one space is sufficient to define a value. If you need to add a comment to the master.conf file, a hash mark (#) specifies a comment to the end of the line. The initial master.conf from a fresh install is shown in Listing 27-3.

```
bank:
 exclude:
                lost+found/
                *~
                .nfs*
 Runall:
                        22:00
                yogi
                booboo 22:00
                dino 22:00
                wilma
                      22:00
                fred
                       22:00
 expire-default: +15 days
 expire-rule:
        MIN HR DOM MON
                           DOW STRFTIME_FMT
         * *
                * *
                            1 +3 months
                1-7 *
                             1 +1 year
               1-7 1,4,7,10 1
```

Listing 27-3 Initial installation master.conf file

The first thing we need to define in the master.conf file is one or more banks. Within a bank we have one or more vaults. A vault contains the image trees of

individual filesystems, and we must create a vault for each filesystem we want to back up. I will show you how to back up the entire system by just excluding the subdirectories you do not wish to back up, and thus using only one vault per system.

To define two banks, /backup_bank1 and /prod_backups, in the master.conf file, we use the following stanza format:

bank:

/backup_bank1 /prod_backups

To define a vault, we need only create a subdirectory in one of the banks. Dirvish knows a subdirectory of a bank is a vault if the vault contains a subdirectory named dirvish. This dirvish subdirectory contains a file named default.conf. If a subdirectory in a bank does not meet these criteria, it is just a subdirectory and Dirvish ignores it. We will cover creating default.conf files later in this chapter.

Let's assume we want to create a vault to store backup images of the yogi server in the /backup_bank1 bank. The directory path to define this vault is shown here with the required default.conf file:

```
/backup_bank1/yogi/dirvish/default.conf
```

Just the presence of this directory path and default.conf file makes this a Dirvish vault; there is not any configuration other than creating the default.conf file.

Creating the default.conf File for Each Filesystem Backup

Each filesystem we back up has its own default.conf file that defines that particular backup. The default.conf file specifies the name of the client to back up, the bank to use for the backup, the vault to store the images, the image name, the backup server name, days before the backup expires, and any files and directories we want to exclude from the backup image. A simple default.conf file is shown in Listing 27-4.

Listing 27-4 Simple default.conf file

The default.conf file is straightforward. It specifies that we back up the /scripts filesystem on the booboo server. The Dirvish backup server is yogi; the bank assigned for this backup is /backup_bank1 and the vault is booboo, with an image directory having the Base name booboo with a filename/time stamp format booboo-%y%m%d%H%M%S. So, a Dirvish server backup image directory may be /backup_bank1/booboo/booboo-071210143304, using the definition from our default.conf file. Notice that we did exclude all files and subdirectories in /scripts/test/ and /scripts/tmp/ from the backup. If you look closely at the exclude: stanza, you will see that we are specifying *relative pathnames*. Because we start in the root directory, /, all pathnames are *relative to the root directory*. Also, when defining a subdirectory to exclude, we need to add a trailing forward slash — for example, subdirectory/.

Performing a Full System Backup

In the example in Listing 27-4, we configured the default.conf file to back up the /scripts directory and all subdirectories, with the exception of /scripts/test/ and /scripts/tmp/. We can expand on this idea to back up the entire system, but exclude any temporary and other data that is not really needed for a backup. Additionally, because we are doing a full system backup, we can use a branch to save space on the backup server. Using a branch allows us to back up entire systems using a fraction of the space normally required. Of course, when you set up a branch, you need to set one up for each flavor of UNIX in your landscape. For more information on using a branch, see the manual page for the dirvish command, **man dirvish**.

The default.conf file shown in Listing 27-5 uses the branch root and excludes filesystems like /tmp, /proc, and /cdrom from the backup image.

```
client: booboo
expire: 5 day
index: gzip
server: yogi
tree: /
vault: booboo
branch: root
image: root-%y%m%d%H%M%S
exclude:
        proc/
        sys/
        tmp/
        dev/shm
        dev/.SRC-unix/
        dev/SRC
        usr/HTTPServer/logs/siddport
        usr/IBMIHS/logs/cgisock
        usr/IBMIHS/logs/siddport
        var/ct/
```

Listing 27-5 default.conf defining a full system backup

```
install/
dev/log
opt/ISS/
usr/HTTPServer/logs/tmp.sock
cdrom
```

Listing 27-5 (continued)

A default.conf file must be created for each system or filesystem, depending on how you define the tree:, exclude:, and branch: stanza directives. Now let's look at using Dirvish on the command line.

Using Dirvish on the Command Line

Dirvish has many commands to perform specific tasks for executing backups. We can back up a single server/filesystem or run all the backups at once. We can list backup images to find the correct file and image date for the data we want to restore, and we can delete expired backup images.

The first step after creating a new default.conf file is to do the initial backup reference image. We do this with the following syntax:

/usr/local/sbin/dirvish --vault vaultname [--branch branch] --init

The --init switch tells Dirvish to perform an initial backup. Once this initial backup is done, we execute all future backups without the --init switch. To perform the initial backup of the booboo vault, use the following syntax:

/usr/local/sbin/dirvish --vault booboo --init

For all new backups after the initial backup, we use the following syntax:

/usr/local/sbin/dirvish --vault vaultname [--branch branch]

To run a Dirvish backup of booboo after the initial Dirvish backup, use the following syntax:

/usr/local/sbin/dirvish -vault booboo

We can also do the backup using the dirvish-runall command. This command runs all the jobs specified in the Runall: stanza of the Dirvish master configuration file, master.conf. In this example, we specify yogi, booboo, dino, wilma, and fred as servers to back up when the dirvish-runall command is executed. For more information on the dirvish-runall command, see **man dirvish-runall**.

Runall:

yogi	22:00
booboo	22:00
dino	22:00
wilma	22:00
fred	22:00

To search a particular vault for a file or directory, we use the dirvish-locate command. The basic syntax is shown here:

/usr/local/sbin/dirvish-locate vault[:branch] patternfile

The patternfile is the file or directory we are searching for. To search for the file /scripts/dirvish_ctrl in the yogi vault, we use the syntax shown in Listing 27-6.

```
dirvish-locate yogi /scripts/dirvish_ctrl
2 matches in 16 images
/scripts/dirvish_ctrl
   Dec 11 14:03 yogi-071211140359
   Dec 11 12:56 yogi-071211125815
   Dec 11 10:04 yogi-071211115107, yogi-071211115013
   Dec 10 12:31 yogi-071210123131, yogi-071210123128
   Dec 10 12:30 yogi-071210123049
   Dec 10 12:28 yogi-071210122858
   Dec 10 12:19 yogi-071210122226, yogi-071210122214,
                yogi-071210122030
   May 25 09:28 yogi-070719153227, yogi-070627130342,
                yogi-070626093920
                yogi-070529152028
   May 21 14:30 scripts-070521144312
/scripts/dirvish_ctrl.log
   Dec 11 14:03 yogi-071211140359
   Dec 11 12:56 yogi-071211125815
```

Listing 27-6 Using the dirvish-locate command

This search returns a list of all the matching files and directories, along with the date/time stamp of each backup image. The backup archives are shown as the yogi-071211140359 files in this example. For more information on using the dirvish-locate command, see **man dirvish-locate**.

Now that we have all the commands, we can build a shell script to do all the hard work for us.

A Menu-Interface Shell Script to Control Dirvish

In this section we are going to build a shell script that is a simple menu-driven interface for Dirvish. In any menu-type shell script, we have to do a lot of cursor control to manipulate the data on the screen. For this script we are going to use a lot of sed and awk, parse through stanza format files, edit files using sed, restore files to a *restore area* on the Dirvish backup server, and a whole boatload of other things. The goal is to have a shell script that we can use right after a fresh Dirvish installation to do the initial configuration and then use on a daily basis to maintain the backups. I call this script dirvish_ctrl. I wrote this as a full backup script, in that I define the vaults as the hostname of the server we are backing up. Here we will use the terms *vault, server*, and *host* interchangeably. The first step we will cover is creating the main menu. The easiest way to do this is with a function. The function in Listing 27-7 displays the main menu on the screen.

```
display_main_menu ()
{
  clear # Clear the screen
  echo -e "\n\n\tWELCOME TO DIRVISH BACKUP COMMAND-CENTER
  \t1) Dirvish RunAll Backups
  \t2) Dirvish Run Backup
  \t3) Dirvish Locate/Restore Image
  \t4) Dirvish Expire/Delete Backup(s)
  \t5) Dirvish Add a New Backup
  \t6) Dirvish Remove a Backup
  \t7) Dirvish Manage Backup Banks
  \t8) EXIT
  "
  echo -e "\tSelect an Option: \c"
}
```

Listing 27-7 Main menu function

Now any time we need to display the main menu we just call the display_main_ menu function. The main menu in action is shown in Listing 27-8.

```
WELCOME TO DIRVISH BACKUP COMMAND-CENTER
1) Dirvish Run All Backups
2) Dirvish Run a Backup
3) Dirvish Locate/Restore Image
```

Listing 27-8 dervish_ctrl shell script main menu

```
    4) Dirvish Expire/Delete Backup(s)
    5) Dirvish Add a New Backup
    6) Dirvish Remove a Backup
    7) Dirvish Manage Backup Banks
    8) EXIT
    Select an Option:
```

Listing 27-8 (continued)

From the main menu shown in Listing 27-8 a user can control the Dirvish backup program without really knowing any of the Dirvish commands we have been studying. Let's go through each of the seven Dirvish options.

NOTE You can also use the select shell command to create a menu. For details, see **man select**.

Running All Backups

In this section we are going to use the dirvish-runall command. Just understand that this Dirvish command option will run a backup only if the vault has been defined in the RunAll: stanza in the master.conf file, as shown here:

```
Runall:
```

22:00
22:00
22:00
22:00
22:00

Executing the dirvish-runall command will only run backups for yogi, booboo, dino, wilma, and fred. The run_all function is shown in Listing 27-9.

```
run_all ()
{
    clear
    echo -e "\n\n"
    /usr/local/sbin/dirvish-runall
    if (( $? == 0 ))
    then
        echo -e "\n\tBackups Complete...Press Enter to Continue...\c"
```

Listing 27-9 run_all function

```
else
echo -e "\n\tDirvish Backup ERROR...Press Enter to Continue...\c"
fi
read KEY
}
```

Listing 27-9 (continued)

This run_all function in Listing 27-9 is pretty simple. We clear the screen, and then run the dirvish-runall command and check the return code. We give the end user feedback on the success or failure and wait for him or her to press Enter. Executing the run_all function produces the following output:

```
14:03:58 dirvish --vault yogi
14:03:59 dirvish --vault booboo
14:04:04 dirvish --vault dino
14:04:22 dirvish --vault wilma
14:04:51 dirvish --vault fred
14:05:04 done
```

Backups Complete... Press Enter to Continue...

Running a Particular Backup

The next step is to write a function to ask the user which backup to run, and then run that particular backup. Check out the run_backup function in Listing 27-10, and we will cover the details at the end.

```
run_backup ()
{
# set -x
clear # Clear the screen
echo -e "\n\n\t\tRUN A PARTICULAR BACKUP\n"
echo -e "\tEnter a Hostname to Back up: \c"
read HTBU # Host to Back up
echo "Searching for default.conf in ${HTBU}'s Vault"
BANK_LIST=$(parse_conf)
for P in $BANK_LIST
do
    # Find the default config file for $HTBU
   CF=$(find ${P}/${HTBU} -type f -name default.conf)
    if [[ ! -z $CF ]]
    then
        echo -e "\nFound Configuration File...Starting Backup..."
        /usr/local/sbin/dirvish --vault $HTBU
```

Listing 27-10 run_backup function

```
RC=$?
        echo -e "\nDirvish Exit Code: $RC"
        echo -e "\nBackup Complete..."
        echo -e "\nPress Enter to Continue...\c"
        read KEY
        break
    else
        echo -e "\nERROR: Could not Locate the Configuration File for
$HTBU"
        echo -e "\n...You Need to Configure $HTBU for Dirvish Backup
First"
        echo -e "\nPress Enter to Continue...\c"
        read KEY
    fi
done
if [[ -z "$CF" ]]
then
    echo -e "\nERROR: Could not Locate the Configuration File for $HTBU"
    echo -e "\n...You Need to Configure $HTBU for Dirvish Backup First"
    echo -e "\nPress Enter to Continue...\c"
    read KEY
fi
}
```

Listing 27-10 (continued)

In the run_backup function in Listing 27-10 we begin by clearing the screen and displaying a screen header. Next we ask the user for a server to back up. Remember; we are using the terms *server* and *vault* and *filesystem* interchangeably for this script. With the server name we search for the default.conf file to ensure it is an already configured server. If we find the default.conf file, we execute the backup of that server using the following command:

```
/usr/local/sbin/dirvish --vault $HTBU
```

We give the user feedback on the success or failure of the backup if we find the default.conf file. If the default.conf file is not found in any banks, we inform the user that this server is not configured in Dirvish and wait for the user to press Enter.

Locating and Restoring Images

In this section we want to interact with the user by asking for a file or directory to search for in the image archives. With a pattern to search for, we present the query data

and ask the users if they found what they are looking for. If the users found the files, we ask if they want the file or directory restored to a separate restore-area filesystem on the local Dirvish backup server. The locate_restore_image function is too long to repeat here. Please check out this function in the dirvish_ctrl shell script shown in Listing 27-24.

In the locate_restore_image function we first clear the screen and display the screen heading. The first question we ask the users is the host where the file or directory resides. Then we ask the users to enter as much of the file or directory name as they know. We use this information to search the backup images for the desired file or directory:

```
/usr/local/sbin/dirvish-locate $H ${DP} | more
```

This command searches the \$H vault for the pattern \${DP} (for data path) and lists the results one page at a time by piping this output to more. After the full output is displayed, we ask the users if they found what they are looking for. If they did, we ask whether they want to restore the data. If the answer is yes (y or Y), we ask for the *full pathname* of the file/directory to restore. Using this new value, we do another search using the dirvish-locate command. This lists all matching files and directories, along with each of the archive image names, with the date/time stamp.

From this output, we ask the user to enter the exact file or directory name to restore, and then which archive image to extract the file(s) from. We then use the find command to search the defined bank, vault, and specified archive image tree, and copy all files found recursively to the local restore area, which is defined as /dirvish_restores in this script.

Expiring and Deleting Backup Images

In this section our goal is to write a function to delete expired backup images, and expire backups that are not currently expired. To delete expired backups, we execute the dirvish-expire command. To expire a backup image that is not currently expired, we have to edit the summary file for that particular backup image to change the expiration date/time stamp.

The expire_backup function is too long to repeat here. Please look at the dirvish_ctrl shell script in Listing 27-24 for details.

We start the expire_backup function by clearing the screen and displaying a new sub-menu. For this shell script option, we need the ability to

- 1. Delete all expired backups in all vaults/hosts.
- 2. Delete expired backups for one vault/host.
- 3. Expire a single non-expired backup for one vault/host.
- 4. Expire all backups for one vault/host.

Listing 27-11 shows the code to create our expire backups menu.

Listing 27-11 Code to display the expire backups menu

The resulting menu is shown in Listing 27-12.

```
DIRVISH EXPIRE BACKUP(S)

1) Delete Expired Backups for ALL Hosts

2) Delete Expired Backups for One Host

3) Expire One Backup for One Host

4) Expire All Backups for One Host

5) Previous Menu

Select an Option:
```

Listing 27-12 Dirvish expire backup menu

The first two options use the dirvish-expire command to delete already-expired backups, but the last two options to expire backups that are not already expired require more work.

The command to *expire all backups in all vaults* is shown here:

/usr/local/sbin/dirvish-expire --tree

The command to expire *all backups in a single vault* is shown here:

/usr/local/sbin/dirvish-expire --tree --vault \$H

The variable \$H defines the vault to delete all expired backups in.

Using sed to Modify the summary File

The next part gets kind of involved-with a deep sed statement to modify the particular backup image or all backups in the vault. If we have a backup that is current and has not yet expired, the only way to expire the backup is to modify the summary file located in the image directory. A sample summary file is shown in Listing 27-13 (the expiration date is highlighted in bold).

```
[root@yogi booboo-071211080741]# cat summary
client: booboo
tree: /scripts
rsh: ssh
Server: yogi
Bank: /prod_backups
vault: booboo
branch: default
Image: booboo-071211080741
Reference: default
Image-now: 2007-12-11 08:07:41
Expire: 180 day == 2008-06-08 08:07:41
exclude:
       lost+found/
       *~
       .nfs*
       tmp/
       test/
SET permissions devices init numeric-ids stats
UNSET checksum sparse whole-file xdev zxfer
ACTION: rsync -vrltH --delete -pgo --stats -D --numeric-ids
--exclude-from=/prod_backups/fred/fred-071211080741/exclude
fred:/scripts/ /prod_backups/fred/fred-071211080741/tree
Backup-begin: 2007-12-11 08:07:41
Backup-complete: 2007-12-11 08:07:42
Status: success
[root@booboo fred-071211080741]#
```

Listing 27-13 Sample image summary file

If you have problems with a backup, you should look in the image directory for the summary and log files. They can help you troubleshoot a problem. The only thing we are interested in here is the expiration date shown here:

```
Expire: 180 day == 2008-06-08 08:07:41
```

The part of this line of data we are interested in is 2008-06-08 08:07:41, the actual date/time stamp. We need to create a date/time stamp that has the current date and time. We can to this with the **date** command. If we use the date syntax date "+%Y-%m-%d %H:%M:%S" we will produce a similar date and time stamp:

```
[root@booboo /]# date "+%Y-%m-%d %H:%M:%S"
2007-12-11 08:22:07
[root@booboo /]#
```

Now, how do we modify the summary file with this new date/time stamp? We can do character substitution with a sed statement. Take a look at this sed statement in Listing 27-14, and we will cover the details at the end.

Listing 27-14 Modify the summary file with a sed statement

The sed statement is one long statement that continues on four lines. This sed statement first greps on Expire in the summary file and uses this result as the value to substitute. The second part of the sed statement extracts the same line out of the summary file but uses the cut command to extract the first four fields, followed by our new date/time stamp. This resulting substitution is redirected to the new filename summary2. If this sed statement has a zero return code, we move the summary2 file to replace the summary file. With this change the backup is now expired, but not yet deleted.

Adding a New Backup

To add a new backup, all we need to do is create a new vault in an existing bank and create a default.conf file to define the backup. To create this new default.conf file, we need to query the user for some information, and some other information we already know. Check out the Dirvish Add a New Backup menu in Listing 27-15.

```
DIRVISH ADD A NEW BACKUP
Select Each Option to Add a New Backup
1) Enter Hostname
```

Listing 27-15 Dirvish Add a New Backup menu

```
    2) Select a Bank for this Backup
    3) Enter Directory Tree to Back up
    4) Enter Files and Directories to Ignore
    5) Enter Days to Retain Each Backup
    6) Add the New Backup to Dirvish
    7) Create the Initial Backup
    8) Main Menu
    Select Each Option to Add a New Backup
    Enter Option:
```

Listing 27-15 (continued)

To create a new backup, the user selects each option. Options 1–5 can be selected in any order, but adding the new backup definition and running the initial backup (Options 6 and 7) are to be completed last in the series.

Let's look at these one at a time. In Option 1 we ask the user to enter the hostname of the server we want to add to Dirvish. Remember; I wrote this script to back up a bunch of servers, so the hostname here is the name of the new vault we are creating. We search the bank for the vault/host entered to ensure that it does not already exist. The code for this option is shown in Listing 27-16.

Listing 27-16 Code to enter a hostname and ensure it does not exist in a bank

Note that we use the find command to search all the banks for this new vault/hostname. We can have only one vault for each host in this script.

Option 2 displays all the banks defined in the Dirvish master configuration file, master.conf. Then we ask the user to enter the bank for this backup. This code is shown in Listing 27-17.

```
FINISH=0
until (( FINISH == 1 ))
do
   clear
   BANK=
   echo -e "\n\n\t\tSELECT A BANK TO STORE THIS BACKUP\n"
   echo -e "\nAvailable Banks:\n"
   for B in $BANK_LIST
   do
       echo -e "\t$B"
   done
   echo -e "\nPlease Enter a Bank: \c:"
   read BANK
   if $(echo $BANK_LIST | grep -q ${BANK})
   then
       echo "$BANK selected for $HN"
      FINISH=1
   else
       echo -e "\nERROR: $BANK is not a Valid Bank"
       sleep 2
       continue
   fi
done
```

Listing 27-17 Code to display all banks and ask for a selection

Notice in Listing 27-17 that we verify the bank entered is a valid bank.

In Option 3 we just ask for a top-level directory tree that we want to back up. In Option 4 we will be able to exclude files and subdirectories we do not want to back up. The code to query the user for a directory tree to back up is shown here:

```
echo -e "\n\tEnter the Directory Tree to Back up: \c" read TREE
```

If Option 4 is selected, we display all the contents of the tree specified in Option 3 and ask the users if they want to exclude anything from the backups. This is the data for the exclude: stanza directive in the default.conf file. We read and save each entry the user wants to exclude from the backup using the read_input function shown in Listing 27-18.

```
read_input ()
{
LINE=
```

Listing 27-18 read_input function

```
while true
do
    read "ENTRY"
    if [[ $ENTRY = 99 ]]
    then
        echo "$LINE"
        break
    fi
    LINE="$LINE
$ENTRY"
done
}
```

Listing 27-18 (continued)

The idea with the read_input function in Listing 27-18 is to keep looping and reading input and saving each value until 99 is entered as a value. This signals the loop to exit, specified by the break command. The entire code section to enter items to exclude is shown in Listing 27-19.

```
echo -e "\n\n\tDIRECTORY TREE(S) and FILES TO IGNORE\n"
echo -e "\nThe Specified Directory Tree Contains the Following Files/
Directories:\n\n"
if [ $HN = $(hostname) ]
then
   ls $TREE | more
else
   ssh $HN 1s $TREE | more
fi
echo -e "\nDo you want to Exclude any Files or Directories from the
Backups? (y/n): \c"
read ANS3
case $ANS3 in
y Y) echo -e "\n\tEnter Directories and Files to Ignore One per Line\n"
    echo -e "Enter 99 when finished\n"
     IGNORE_LIST=$(read_input)
    continue
     ; ;
n|N) IGNORE_LIST=$(echo)
    continue
    ;;
  *) echo -e "\nInvalid Entry..."
     ;;
esac
```

Listing 27-19 Code to read in and save exclude data

After displaying all the files located in the specified tree, we ask the users if they would like to exclude anything. If the answer is yes (y or Y), we use read_input to

gather the data. Notice that we call the read_input function and assign the resulting list to the IGNORE_LIST variable.

Option 5 is where we ask for the number of days to retain the backups. The query code is shown here:

```
echo -e "\n\tDays before each Backup Expires: \c" read DAYS
```

Simple enough. We just assign the variable to DAYS variable.

Option 6, to add a new backup, is where we actually create the new default.conf file. Check out the code in Listing 27-20, and we will cover the details at the end.

```
build_default_conf ()
{
# Define files and variables here
HN="$1" # Hostname of server to add
BANK="$2" # Bank to use for backup
DAYS="$3" # Days before backups expire
TREE="$4" # Directory tree to back up
IGNORE_LIST="$5" # Files and directories to ignore
OUTFILE=$DEFAULT_CONF # Name of the output file
# All of the following output is used to build the
# new default.conf file for this backup
{
echo "client: $HN"
echo "bank: $BANK"
echo "vault: $HN"
echo "server: $DIRVISH_SERVER"
echo "expire: $DAYS day"
echo "index: gzip"
echo "log: gzip"
echo "image: ${HN}-%y%m%d%H%M%S"
echo "tree: $TREE"
echo -e "\nexclude:"
echo "$IGNORE_LIST" | sed /^$/d | while read Q
do
   echo -e "\t$Q"
done
} >$OUTFILE
}
```

Listing 27-20 build_default_conf function

Notice in the build_default_conf function in Listing 27-20 that we are expecting five arguments: the hostname, bank, days before the backup expires, the directory tree to back up, and the list of objects to exclude from the backups. The remaining data we already know, such as the Dirvish backup server (this script assumes it is executing on the Dirvish backup server), image name with the date format defined, and using gzip

to compress the index. See how we put all the echo statements within curly braces, { }, and then use a single output redirection to create the \$OUTFILE.

The file that the build_default_conf function creates is a temporary file, not the real default.conf file we need to create in the new vault. The last step is to create the new vault and copy the newly created default.conf file to the newly created vault. The rest of the code is shown in Listing 27-21.

```
echo -e "\n\nCreating default.conf Dirvish Configuration File for $HN"
build_default_conf $HN "$BANK" $DAYS "$TREE" "$IGNORE_LIST"
echo -e "\nCopying file to: ${BANK}/${HN}/dirvish/default.conf"
cp -f $DEFAULT_CONF "${BANK}/${HN}/dirvish/default.conf"
echo -e "\n...Press ENTER to Continue...\c"
read ANS
```

Listing 27-21 Code to create a new default.conf file

In Listing 27-21 we inform the user that we are creating the new default.conf file, and then call the build_default_conf function, passing the hostname, bank, days before expiration, the tree to back up, and the list of files and directories to ignore. Notice that "\$BANK", "\$TREE", and "\$IGNORE_LIST" are all enclosed in double quotes. These double quotes are required because there may be white space in the data assigned to each of these variables.

The last step is to copy the new default.conf file to the new vault to complete the vault configuration.

Removing a Backup

By now you are probably thinking, "Okay, when is this script going to end?" Well, one more topic after this. In this section, we want to remove a vault/hostname from the Dirvish backup server. The only way to do this is to remove the vault's directory and all its contents. The work in the delete_backup function is done by one command:

```
rm -fr "$VAULT_ID"
```

Before we start deleting directories and all their contents, we need to do a little sanity checking to ensure we do not break something. Check out the delete_backup function in Listing 27-22, and we will cover the details at the end.

```
delete_backup ()
{
  clear
  echo -e "\n\n\tREMOVE A SERVER FROM DIRVISH BACKUPS\n"
  echo -e "\n\tEnter the Host to Remove from Dirvish Backups: \c"
  read H
BANK_LIST=$(parse_conf)
```

Listing 27-22 delete_backup function

```
for P in $BANK_LIST
do
   VAULT_ID=$(find $P -type d -name "$H")
   if [[ ! -z "$VAULT_ID" ]]
   then
       STOP=0
       until (( STOP == 1 ))
       do
          echo -e "\n\tRemove Backup Vault $H from the Dirvish Backup
Server? (y/n): \c"
          read ANS
          case $ANS in
          y Y) echo -e "\n\tRemoving Backup Vault for $H from Dirvish
Backups...\c"
               rm -fr "$VAULT_ID"
               echo -e "\n\tBackup Vault Removed...Press Enter to
Continue...\c"
               read KEY
               STOP=1
               : :
          n N) echo -e "\n\tVault Removal Canceled...Press Enter to
Continue...\c"
               read KEY
               STOP=1
               continue
               ;;
            *) echo -e "\n\tInvalid Entry..."
               ;;
          esac
       done
  fi
done
}
```

Listing 27-22 (continued)

We start the delete_backup function in Listing 27-22 by clearing the screen and displaying the screen header. We ask the user for the hostname/vault to delete, then we search all the banks for the specified hostname/vault. If we find the vault, we ask the user to confirm deleting it ("Are you sure...?"). If the user responds with y or Y we run the rm -fr "\$VAULT_ID", which deletes the vault from the Dirvish backup server.

Managing the Dirvish Backup Banks

I wrote this script with the ability to modify the master.conf file in much the same way we modified the summary files to expire a backup image. The manage_banks function is too long to repeat here. For details see the dirvish_ctrl shell script in Listing 27-24.

We begin the manage_banks function by clearing the screen and displaying the screen header. The next step is to display all the banks defined in the Dirvish master configuration file, master.conf. If no banks are currently defined, such as in a fresh Dirvish installation, we inform the user. We then display the menu to manage the Dirvish backup banks. This menu code is shown here:

echo -e "\n\n1) Add a New Backup Bank" echo -e "\n2) Delete a Current Backup Bank" echo -e "\n3) Return to the Previous Menu" echo -e "\n\nSelect an Option: \c"

Our options are to add a new Dirvish backup bank or to delete a currently defined backup bank. Adding a new Dirvish backup bank is a two-step process. First we define the new bank in the Dirvish master configuration file, master.conf. Then we need to create the directory for the new bank on the system.

NOTE It is always a good idea to create a separate filesystem for each new bank. You can also create a large filesystem and create different banks/ directories in the same filesystem. Either way, do *not* just add the new bank's directory to the root filesystem. If you do, you could fill the up root filesystem and disable the system.

Adding a New Dirvish Backup Bank

To add a new bank, we need to parse through the master.conf file to find the last bank defined in the list. As we parse through the file, we assign the values to the LAST_ENTRY variable on each loop iteration. When the loop completes executing, we have the last bank defined under the bank: stanza directive in the master.conf file. This for loop finds the last entry and saves the list of banks in a file, as shown here:

```
for B in $BANK_LIST
do
  ((COUNT == COUNT + 1))
  LAST_ENTRY=$B
  echo -e "\t$B" | tee -a $BANK_FILE
done
```

Now we can add a new bank definition to the master.conf file with character substitution using a sed statement, as follows:

```
sed "s!$LAST_ENTRY!& \n\t${A_BANK}!g" $M_CONFIG > ${M_CONFIG}.
modified
if (( $? == 0 ))
then
    cp ${D_CONFIG} ${D_CONFIG}.$(date +%m%d%Y)
    cp ${D_CONFIG}.modified ${D_CONFIG}
    echo -e "\n$A_BANK Successfully Added to Dirvish Master
Config File"
    chk_create_dir $A_BANK
```

```
else
echo -e "\nERROR: Adding $A_BANK Failed...See Administrator..."
fi
```

This sed statement adds the new bank after the last defined bank in the list, or adds it as the first bank in the list if the bank: stanza directive has no other banks defined. This step creates the master.conf.modified file. If the sed statement completes with a return code of 0, we copy the new file to master.conf.

Deleting a Dirvish Backup Bank

Deleting a bank is a two-step process if you want to delete all the backup images that reside in that bank. The code to delete a bank from the master.conf file and to remove the directory structure, upon verification, is shown in Listing 27-23.

```
cat $D_CONFIG | grep -v $R_BANK > ${D_CONFIG}.modified
cp -p $D_CONFIG ${D_CONFIG}.bak.$(date +%m%d%y)
cp ${D_CONFIG}.modified $D_CONFIG
echo -e "\n$R_BANK Removed from the Dirvish Configuration File..."
if [ -d $R_BANK ]
then
    echo -e "\nDo You Want to Remove the $R_BANK Directory? (y/n): \c"
   read ANS
    case $ANS in
    y Y) rm -r $R_BANK
         if (($? == 0))
         then
            echo -e "\n$R_BANK Directory Removed Successfully"
         else
              echo -e "\nERROR: Remove $R_BANK Directory Failed"
         fi
         ;;
    n|N) echo -e "\nSkipping $R_BANK Directory Removal"
         ;;
      *) echo -e "\nInvalid Input..."
         ;;
         esac
fi
```

Listing 27-23 Code to delete a Dirvish bank

The first step is to remove the bank definition from the master.conf file. As far as Dirvish is concerned, this bank no longer exists, but the directory structure is still in place. Before we delete the actual directory structure containing the bank and all the archived images, we ask the user. If the user confirms deleting the contents, we proceed with deleting the bank directory and all subdirectories.

To delete a bank from the master.conf file, we use grep with the -v pattern switch. The -v switch tells grep to list everything *except* the pattern. Then we copy

the newly modified file to master.conf. Finally, we ask the user before we delete the bank directory structure and all the archived images.

Putting It All Together

The preceding sections described each of the pieces that make up the dirvish_ctrl shell script. Now let's put it all together. There are too many individual little details for me to describe everything in minute detail. Study the dirvish_ctrl shell script in Listing 27-24 in detail, and you will surely pick a lot of tips and tricks.

```
#!/bin/Bash
#
# SCRIPT: dirvish_ctrl
# AUTHOR: Randy Michael
# DATE: 5/14/2007
# REV: 1.0
# PLATFORM: Most any UNIX platform
#
# PURPOSE: This script is used to interface with the Dirvish
#
  backup utilities.
#
# set -x # Uncomment to debug this script
#
# set -n # Uncomment to check the script's syntax
      # without any execution. Do not forget to
#
#
       # recomment this line!
# REVISION LIST:
*****
# IF YOU MODIFY THIS SCRIPT DOCUMENT THE CHANGE(S) !!!
*****
#
# Revised by:
# Revision Date:
# Revision:
#
****
#
# Revised by:
# Revision Date:
# Revision:
#
```

Listing 27-24 dirvish_ctrl menu interface shell script

```
*****
# DEFINE FILES AND VARIABLES HERE
RESTORE_AREA=/dirvish_restores
DIRVISH_SERVER=$(hostname)
DEFAULT_CONF=/tmp/dirvish_build_conf.out
LOGFILE=$(basename $0).log
echo -e "\n\t\tDirvish Log Created: $(date) \n" >$LOGFILE
DONE=0
****
# DEFINE FUNCTIONS HERE
****
display_main_menu ()
{
# This function displays the main menu
clear # Clear the screen
# Display the menu header and menu
echo -e "\n\n\tWELCOME TO DIRVISH BACKUP COMMAND-CENTER
\t1) Dirvish Run All Backups
\t2) Dirvish Run a Backup
\t3) Dirvish Locate/Restore Image
\t4) Dirvish Expire/Delete Backup(s)
\t5) Dirvish Add a New Backup
\t6) Dirvish Remove a Backup
\t7) Dirvish Manage Backup Banks
\t8) EXIT
echo -e "\tSelect an Option: \c"
}
****
```

Listing 27-24 (continued)

```
read_input ()
{
# This function is used to save user input
LINE= # Initialize LINE to NULL
while true
do
   read "ENTRY"
   if [[ $ENTRY = 99 ]]
   then
       echo "$LINE"
       break
   fi
   LINE="$LINE
$ENTRY"
done
}
****
parse_conf ()
{
# This function parses through the Dirvish Master
# Configuration File, specified by $M_CONFIG, to
# gather a list of all of the defined Dirvish banks.
# set -x # Uncomment to debug this function
# Initial local variables
BANK_LIST=
START=0
# Loop through the $M_CONFIG file until we find
# the 'bank:' stanza, then add each bank to a list
# until we reach another stanza, 'stanza:'
while read DATA
do
   [[ $DATA = 'bank:' ]] && START=1 && continue
   if (( START == 1 ))
   then
      if $(echo "$DATA" | grep -q ':')
      then
          break
      else
          if [[ -z "$BANK_LIST" ]]
```

```
Listing 27-24 (continued)
```

```
then
              BANK_LIST="$DATA"
          else
              BANK_LIST="$BANK_LIST $DATA"
          fi
      fi
   fi
done < $M_CONFIG # Feed the while loop from the bottom
echo "$BANK_LIST" # Return the list
}
*****
build_default_conf ()
{
# Define files and variables here
HN="$1"
           # Hostname of server to add
BANK="$2"
           # Bank to use for backup
DAYS="$3" # Days before backups expire
         # Directory tree to back up
TREE="$4"
IGNORE_LIST="$5" # Files and directories to ignore
OUTFILE=$DEFAULT_CONF # Name of the output file
# All of the following output is used to build the
# new default.conf file for this backup
{
echo "client: $HN"
echo "bank: $BANK"
echo "vault: $HN"
echo "server: $DIRVISH_SERVER"
echo "expire: $DAYS day"
echo "index: gzip"
echo "log: gzip"
echo "image: ${HN}-%y%m%d%H%M%S"
echo "tree: $TREE"
echo -e "\nexclude:"
echo "$IGNORE_LIST" | sed /^$/d | while read Q
do
   echo -e "\t$Q"
done
} >$OUTFILE
}
*****
run_all ()
```

Listing 27-24 (continued)

```
# This function runs all of the backups defined in
# the Dirvish master configuration file stanza "Runall:"
clear # Clear the screen
echo -e "\n\n"
# Execute all master.conf defined backups
/usr/local/sbin/dirvish-runall
if (( $? == 0 )) # Check the return code!
then
   echo -e "\n\tBackups Complete...Press Enter to Continue...\c"
else
   echo -e "\n\tDirvish Backup ERROR...Press Enter to Continue...\c"
fi
read KEY
}
*****
run_backup ()
# This function runs one particular backup
# set -x # Uncomment to debug this function
clear # Clear the screen
# Display the screen heading and query the user
echo -e "\n\n\t\tRUN A PARTICULAR BACKUP\n"
echo -e "\tEnter a Hostname to Back Up: \c"
read HTBU # Host to Back Up
echo -e "\nSearching for default.conf in ${HTBU}'s Vault"
BANK_LIST=$(parse_conf)
for P in $BANK_LIST
do
    # Find the Default Config File (default.conf) for $HTBU
    DCF=$(find ${P}/${HTBU} -type f -name default.conf)
    if [[ ! -z $DCF ]]
    then
        echo -e "\nFound Configuration File...Starting Backup..."
```

```
Listing 27-24 (continued)
```

```
/usr/local/sbin/dirvish --vault $HTBU
       RC=$?
       echo -e "\nDirvish Exit Code: $RC"
       echo -e "\nBackup Complete..."
        echo -e "\nPress Enter to Continue...\c"
       read KEY
       break
    else
       echo -e "\nERROR: Could not Locate the Configuration File for
 $HTBU"
       echo -e "\n...You Need to Configure $HTBU for Dirvish Backup
First"
       echo -e "\nPress Enter to Continue...\c"
       read KEY
    fi
done
if [[ -z "$DCF" ]]
then
    echo -e "\nERROR: Could not Locate the Configuration File
for $HTBU"
    echo -e "\n...You Need to Configure $HTBU for Dirvish
Backup First"
    echo -e "\nPress Enter to Continue...\c"
    read KEY
fi
}
*****
chk_create_dir ()
{
# This function is used to add a new Dirvish "bank" to
# the Dirvish backup server, which should be this machine
D=$1 # Directory name to add
if [ ! -d $D ] # Check if the directory already exists
then
    echo -e "\n$D Directory Does Not Exist...Create $D Directory?
(y/n): \c"
    read ANS
    case $ANS in
    y Y) echo -e "\nCreating $D Directory..."
        mkdir $D
        if (( $? == 0 ))
         then
            echo -e "\n$D Directory Created"
```

Listing 27-24 (continued)

```
else
            echo -e "\n$D Directory Creation Failed...See
Administrator..."
         fi
         ;;
   n N) echo -e "\nSkipping Creating the $D Directory"
         echo -e "\nWARNING: Until the $D Directory is Created
Dirvish Cannot use this Bank\n"
         ::
      *) echo -e "\nInvalid Input..."
        ;;
    esac
fi
}
*****
locate_restore_image ()
{
# set -x # Uncomment to debug this function
clear # Clear the screen
# Display the screen heading, query the user, display the
# files matching the search, and ask if the user wants to
# restore the files. If a restore is requested we ensure
# there is a directory for the target server in the $RESTORE_AREA,
# if not create it, and copy the files into this subdirectory in
# the $RESTORE AREA.
echo -e "\n\n\t\tLOCATE/RESTORE A BACKUP IMAGE\n"
echo -e "Enter the Hostname where the File/Directory Resides: \c"
read H
echo -e "\nEnter as Much of the Directory Path and Filename as
You Know:\n"
read "DP"
if [ "$DP" = '/' ]
then
    echo -e "\nERROR: This Program is Not Intended for Full System
Restores...\n"
   sleep 2
   return
fi
echo -e "\nSearching for Archive Images..."
# Search and display all matching files one page at a time
/usr/local/sbin/dirvish-locate $H ${DP} | more
```

Listing 27-24 (continued)

```
QUIT=0
until (( QUIT == 1 ))
do
   echo -e "\nDid You Find What You Want? (y/n): \c"
   read ANS1
   case $ANS1 in
   y Y) echo -e "\nDo You Want to Perform a Restore? (y/n): \c"
        read ANS2
        while true
        do
           case $ANS2 in
           y Y) QUIT=1
                break
                ;;
           n N) echo -e "\nPress Enter to Continue...\c"
                read KEY
                return 0
                ;;
             *) echo -e "\nInvalid Input..."
                ;;
           esac
        done
        ;;
   n|N) return 0
        ;;
     *) echo -e "\nInvalid Input..."
        ;;
   esac
done
echo -e "\nEnter the FULL PATHNAME of the File/Directory You Want to
Restore:\n"
read TARGET
if [ -z "$TARGET" ]
then
   echo -e "\nInvalid Entry..."
   sleep 2
   return
fi
if [[ $(echo "$TARGET" | cut -c1) != '/' ]]
then
    echo -e "\nERROR: $TARGET is Not a Valid Full Pathname...\n"
   sleep 2
   break
fi
```

Listing 27-24 (continued)

```
echo -e "\nSearching Images..."
/usr/local/sbin/dirvish-locate "$H" "${TARGET}" | more
echo -e "\nEnter the Image to Restore From: \c"
read IMAGE_NAME
echo
# Check to ensure that the defined $RESTORE_AREA
# directory has been created on the system.
if [[ ! -d $RESTORE_AREA ]]
then
   mkdir $RESTORE_AREA
fi
# Ensure that a subdirectory exists for this host
if [[ ! -d ${RESTORE_AREA}/${H} ]]
then
   mkdir ${RESTORE_AREA}/${H}
fi
BANK_LIST=$(parse_conf)
# This is where we restore the data
for BANK in $BANK_LIST
do
   if [ -d "$TARGET" ]
   then
       # If $TARGET is a directory we only want the last
       # part of the path, not the entire path.
       TARGET=$(basename "$TARGET")
       find ${BANK}/${H}/${IMAGE_NAME}/tree -name "${TARGET}" -print \
            -exec cp -rp {} "${RESTORE_AREA}/${H}" \; 2>/dev/null
       if (($? == 0))
       then
           echo -e "\nFiles Restored to the Following Restore Area:"
           echo -e "\n${RESTORE_AREA}/${H}/${TARGET}\n"
          break
       else
           echo -e "\nRestore Failed...See Administrator...\n"
           break
```

```
Listing 27-24 (continued)
```

```
fi
   else
       # We are working with a file
      D_NAME=$(dirname "$TARGET")
      F_NAME=$(basename "$TARGET")
      echo "DIRNAME is $D_NAME"
       # Find and copy the file(s) to the $RESTORE_AREA
      find ${BANK}/${H}/${IMAGE_NAME}/tree/ -name "${F_NAME}" \
           -exec cp -rp {} "${RESTORE_AREA}/${H}" \; 2>/dev/null
      if (($? == 0))
      then
          echo -e "\nFiles Restored to the Following Restore Area:"
          echo -e "\n${RESTORE_AREA}/${H}/${F_NAME}\n"
          break
      else
          echo -e "\nRestore Failed...See Administrator...\n"
          break
      fi
  fi
done
echo -e "\nPress Enter to Continue...\c"
read KEY
}
****
expire_backup ()
{
# This function is used to expire backups
# set -x # Uncomment to debug this function
clear # Clear the screen
# Display the screen header and menu
echo -e "
   \n\n\t\tDIRVISH EXPIRE BACKUP(S)\n
  \n\t1) Delete Expired Backups for ALL Hosts
  n\t2) Delete Expired Backups for One Host
   \n\t3) Expire One Backup for One Host
   \n\t4) Expire All Backups for One Host
   \n\t5) Previous Menu
```

Listing 27-24 (continued)

```
\n\n\tSelect an Option: \c"
read ANS
case $ANS in
1) echo -e "\n\tDeleting Expired Backups for ALL Hosts...\n"
   /usr/local/sbin/dirvish-expire --tree
   echo -e "\n\tTasks Complete..."
   echo -e "\n\tPress Enter to Continue...\c"
   read KEY
   ;;
2) echo -e "\n\tEnter the Hostname to Delete Expired Backups: \c"
   read H
   /usr/local/sbin/dirvish-expire --tree --vault $H | more
   echo -e "\n\tTasks Complete..."
   echo -e "\n\tPress Enter to Continue...\c"
   read KEY
   ;;
3) EXPIRE_STAMP="$(date "+%Y-%m-%d %H:%M:%S")"
   echo -e "\n\tEnter the Hostname for the Backup Image: \c"
   read H
   BANK_LIST=$(parse_conf)
   MYPWD=$(pwd)
   # Parse through all of the banks to this host
   for B in $BANK_LIST
   do
     if [ -d ${B}/${H} ]
      then
          cd ${B}/${H}
         IMAGE_LIST=$(find . -type d -name "*-[0-9][0-9]*" |
cut -c3-)
      fi
   done
   echo -e "\nSelect an Image to Expire from the Following List:\n"
   echo -e "\n$IMAGE_LIST" | more
   echo -e "\n\nEnter the Image to Expire: \c"
   read IMAGE_NAME
   cd $MYPWD
   if [[ -r ${B}/${H}/${IMAGE_NAME}/summary ]]
   then
      # This is where we expire a backup that is not currently
      # expired. The 'summary' file contains the expire datestamp
      echo -e "\nExpiring Image: $IMAGE_NAME"
```

Listing 27-24 (continued)

```
sed s/"$(grep Expire ${B}/${H}/${IMAGE_NAME}/summary)"/
"$(grep Expire ${B}/${H}/${IMAGE_NAME}/summary | cut -f 1-4 -d")
$EXPIRE_STAMP"/g "${B}/${H}/${IMAGE_NAME}/summary" >
"${B}/${H}/${IMAGE_NAME}/summary2"
     mv "${B}/${H}/${IMAGE_NAME}/summary2" "${B}/${H}/${IMAGE_NAME}
/summary"
   fi
   # We expired the backups; now we ask the user if we should delete
   # the expired images
   STOP=0
  until (( STOP == 1 ))
   do
     echo -e "\nTasks Complete...Do You Want to Delete This Expired
Image? (y/n): \c"
    read ANS
    case $ANS in
    y Y) echo -e "\nDeleting Expired Image: $IMAGE_NAME"
          # This next command deletes all expired images for $H
          /usr/local/sbin/dirvish-expire --vault $H
          RC=$?
          echo -e "\n\tDirvish-expire Completed with Return Code: $RC"
          echo -e "\n\tTasks Complete..."
          echo -e "\n\tPress Enter to Continue...\c"
          read KEY
          STOP=1
          ;;
    n|N) STOP=1
         continue
         ;;
       *) echo -e "\n\tInvalid Entry..."
          ;;
     esac
   done
   ;;
4) EXPIRE_STAMP=$(date "+%Y-%m-%d %H:%M:%S")
  clear
   echo -e "\nEnter the Hostname for the Backup Image: \c"
   read H
   BANK_LIST=$(parse_conf)
  MYPWD=$(pwd)
   for P in $BANK_LIST
```

```
do
      if [ -d ${P}/${H} ]
      then
         cd ${P}/${H}
         IMAGE_LIST=$(find . -type d -name "*-[0-9][0-9]*" |
cut -c3-)
         B=$P
      fi
   done
   echo -e "\nSetting the Following Images to Expire:\n\n"
   echo "$IMAGE_LIST" | more
   cd $MYPWD
   echo -e "\nPress Enter to Continue...\c"
   echo "$IMAGE_LIST" | while read IMAGE_NAME
   do
     if [ -f ${B}/${H}/${IMAGE_NAME}/summary ]
     then
        echo "Expiring $IMAGE_NAME"
        sed s/"$(grep Expire ${B}/${H}/${IMAGE_NAME}/summary)"/
"$(grep Expire ${B}/${H}/${IMAGE_NAME}/summary | cut -f 1-4 -d ' ')
$EXPIRE_STAMP"/g "${B}/${H}/${IMAGE_NAME}/summary" >
"${B}/${H}/${IMAGE_NAME}/summary2"
        if (($? == 0))
        then
           mv ${B}/${H}/${IMAGE_NAME}/summary2 ${B}/${H}/
${IMAGE_NAME}/summary
        fi
     fi
   done
   STOP=0
   until (( STOP == 1 ))
   do
    echo -e "\nTasks Complete...Do You Want to Delete Expired
Images? (y/n): \c"
    read ANS
    case $ANS in
     y Y) echo -e "\nDeleting Expired Images...\n"
          /usr/local/sbin/dirvish-expire --vault $H
          RC=$?
          echo -e "\nDirvish-expire Completed with Return Code: $RC"
          echo -e "\nTasks Complete..."
          echo -e "\nPress Enter to Continue...\c"
          read KEY
          STOP=1
```

```
;;
    n|N) STOP=1
         continue
         ;;
      *) echo -e "\n\tInvalid Entry..."
         ;;
    esac
   done
   ;;
5) :
  ;;
*) echo -e "\nInvalid Entry"
   ;;
esac
}
*****
add_backup ()
{
# This function is used to define a new backup to Dirvish
# set -x # Uncomment to debug this function
# Get a list of available banks
BANK_LIST=$(parse_conf)
ESCAPE=0
until (( ESCAPE == 1 ))
do
 clear # Clear the screen
  # Display the screen heading and the menu
  echo -e "
 \n\t\t DIRVISH ADD A BACKUP
  \n\tSelect Each Option to Add a New Backup
 \n\t1) Enter Hostname
  \n\t2) Select a Bank for this Backup
 \n\t3) Enter Directory Tree to Back Up
  \n\t4) Enter Directory Trees to Ignore
 \n\t5) Enter Days to Retain Each Backup
  \n\t6) Add the New Backup to Dirvish
 \n\t7) Create the Initial Backup
  \n\t8) Main Menu
  \n\tSelect Each Option to Add a New Backup
  \n\tEnter Option: \c"
  read OPTION # Read in the user response
```

Listing 27-24 (continued)

```
# Perform the desired operation
  case $OPTION in
  1) echo -e "\n\tEnter the Hostname for this Backup: \c"
     read HN
     for B in $BANK_LIST
     do
         P=$(find $B -type d -name $HN)
         if [ ! -z $P ]
         then
             echo -e "\n\tWARNING: Vault $HN Already Exists in the
Following Bank:"
             echo -e "\n\t\P"
             echo -e "\n\tDo you want to create a new default.conf file
for HN? (y/n): \c"
             ANS=n # Set the default answer to 'n', No.
             read ANS
             case $ANS in
             y Y) continue
                  ;;
             n|N) return
                  ;;
             esac
         fi
     done
     ;;
  2) FINISH=0
     until (( FINISH == 1 ))
     do
       clear
       BANK=
       echo -e "\n\n\t\tSELECT A BANK TO STORE THIS BACKUP\n"
       echo -e "\nAvailable Banks:\n"
       for B in $BANK_LIST
       do
            echo -e "\t$B"
       done
       echo -e "\nPlease Enter a Bank: \c:"
       read BANK
       if $(echo $BANK_LIST | grep -q ${BANK})
       then
           echo "$BANK selected for $HN"
           FINISH=1
       else
           echo -e "\nERROR: $BANK is not a Valid Bank"
           sleep 2
           continue
       fi
     done
```

```
;;
  3) echo -e "\n\tEnter the Directory Tree to Back Up: \c"
    read TREE
    continue
     ;;
  4) clear
     echo -e "\n\n\tDIRECTORY TREE(S) and FILES TO IGNORE\n"
     echo -e "\nThe Specified Directory Tree Contains the Following
Files/Directories:\n\n"
     if [ $HN = $(hostname) ]
     then
         ls $TREE | more
     else
         ssh $HN 1s $TREE | more
     fi
     echo -e "\nDo you want to Exclude any Files or Directories from
the Backups? (y/n): \c"
    read ANS3
    case $ANS3 in
    y Y) echo -e "\n\tEnter Directories and Files to Ignore One per
Line\n"
          echo -e "Enter 99 when finished\n"
          IGNORE LIST=$ (read input)
          continue
          ;;
     n|N) IGNORE_LIST=$(echo)
          continue
          ;;
       *) echo -e "\nInvalid Entry..."
          ;;
     esac
     ;;
  5) echo -e "\n\tDays before each Backup Expires: \c"
    read DAYS
    continue
     : :
  6) clear
     echo -e "\n\tADD NEW BACKUP TO DIRVISH"
     if [[ -r "${BANK}/${HN}/dirvish/default.conf" ]]
     then
        echo -e "\nWARNING: default.conf File Exists...Rebuild default.
conf? (y/n): \c"
        read ANS
        case $ANS in
        y Y) echo -e "\n\nCreating default.conf Dirvish Configuration
File for $HN"
             build_default_conf $HN "$BANK" $DAYS "$TREE"
"$IGNORE_LIST"
```

Listing 27-24 (continued)

```
echo -e "\nCopying file to: ${BANK}/${HN}/dirvish
/default.conf"
             cp -f $DEFAULT_CONF "${BANK}/${HN}/dirvish/default.conf"
             echo -e "\n\n... Press ENTER to Continue... \c"
             read ANS
             ;;
          *) break
             ;;
        esac
     else
         echo -e "\n\nCreating default.conf Dirvish Configuration
File for $HN"
         build_default_conf $HN "$BANK" $DAYS "$TREE" "$IGNORE_LIST"
         echo -e "\nCopying file to: ${BANK}/${HN}/dirvish/default.
conf"
         cp -f $DEFAULT_CONF "${BANK}/${HN}/dirvish/default.conf"
         echo -e "\n... Press ENTER to Continue... \c"
         read ANS
     fi
     ;;
  7) clear
     echo -e "\n\n\tCREATE INITIAL DIRVISH BACKUP"
     echo -e "\n"
     if [[ ! -d "${BANK}/${HN}" ]]
     then
        echo -e "\nCreating Vault $HN for this Backup"
        mkdir "${BANK}/${HN}"
        mkdir "${BANK}/${HN}/dirvish"
     else
        echo -e "\nBackup Vault for $HN Exists...Continuing..."
     fi
     if [[ ! -r "${BANK}/${HN}/dirvish/default.conf" ]]
     then
        Now we have enough information to build the new default.conf
file
        echo -e "Creating default.conf Dirvish Configuration File for
$HN"
        build_default_conf $HN "$BANK" $DAYS "$TREE" "$IGNORE_LIST"
        echo -e "\nCopying file to: ${BANK}/${HN}/dirvish/default.
conf"
        cp -f $DEFAULT_CONF "${BANK}/${HN}/dirvish/default.conf"
        # Now we need to run an initial backup
        echo -e "\n\tRun an Initial Backup Now? (y/n): \c"
        read BACK_NOW
        case $BACK_NOW in
```

```
y|Y) echo -e "\nExecuting Initial Dirvish Backup for $HN..."
             /usr/local/sbin/dirvish --vault $HN --init
             RC=$?
             echo -e "\nInitial Backup for $HN Completed with a
Return Code: $RC"
             echo -e "\n... Press Enter to Continue... \c"
             read KEY
             ;;
        n N) echo -e "\nInitial Backup Skipped..."
             echo -e "\nDo Not Forget to Run an Initial Dirvish
Backup!"
             echo -e "\nTo Manually Run an Initial Backup Enter:\n"
             echo -e "\t/usr/local/sbin/dirvish --vault $HN --init\n"
             echo -e "\n... Press Enter to Continue... \c"
             read KEY
             ;;
         *) echo -e "\nInvalid Input..."
            ;;
        esac
     else
        FINISH=0
        until (( FINISH == 1 ))
        do
           echo -e "\nRun an Initial Backup Now? (y/n): \c"
           read BACK_NOW
           case $BACK_NOW in
           y Y) echo -e "\nExecuting Initial Dirvish Backup for $HN..."
                /usr/local/sbin/dirvish --vault $HN --init
                RC=$?
                echo -e "\nInitial Backup for $HN Completed with a
Return Code: $RC"
                echo -e "\nPress Enter to Continue...\c"
                read KEY
                FINISH=1
                ;;
           n N) echo -e "\nInitial Backup Skipped..."
                echo -e "\nDo Not Forget to Run an Initial Dirvish
Backup!"
                echo -e "\nTo Manually Run an Initial Backup Enter:\n"
                echo -e "\t/usr/local/sbin/dirvish --vault $HN
--init\n"
                echo -e "\nPress Enter to Continue...\c"
                read KEY
                FINISH=1
                ;;
            *) echo -e "Invalid Entry..."
               sleep 2
               ;;
```

```
esac
      done
    fi
    continue
    ;;
  8) break
    ;;
 esac
done
}
*****
delete_backup ()
{
# This function is used to delete backups from the Dirvish server
clear # Clear the screen
echo -e "\n\n\t\tREMOVE A SERVER FROM DIRVISH BACKUPS\n"
echo -e "\n\tEnter the Host to Remove from Dirvish Backups: \c"
read H
BANK_LIST=$(parse_conf)
for P in $BANK_LIST
do
   VAULT_ID=$(find $P -type d -name "$H")
   if [[ ! -z "$VAULT_ID" ]]
   then
       STOP=0
       until (( STOP == 1 ))
       do
          echo -e "\n\tRemove Backup Vault $H from the Dirvish Backup
Server? (y/n): \c"
          read ANS
          case $ANS in
          y|Y) echo -e "\n\tRemoving Backup Vault for $H from
Dirvish Backups...\c"
               rm -fr "$VAULT_ID"
               echo -e "\n\tBackup Vault Removed...Press Enter
to Continue...\c"
               read KEY
               STOP=1
               ;;
          n N) echo -e "\n\tVault Removal Canceled...Press Enter to
Continue...\c"
               read KEY
               STOP=1
```

```
Listing 27-24 (continued)
```

```
continue
               ; ;
            *) echo -e "\n\tInvalid Entry..."
               ;;
          esac
       done
   fi
done
}
manage_banks ()
{
# This function is used to add and delete Dirvish banks
# set -x # Uncomment to debug this function
clear # Clear the screen
# Get a list of currently defined Dirvish banks
BANK_LIST=$(parse_conf)
# Display the screen header information
echo -e "\n\n\tMANAGE DIRVISH BACKUP BANKS"
echo -e "\n\nCurrently Configured Backup Bank(s):\n"
NO_BANK=0
# If this is an initial installation there will not
# be any Dirvish banks defined.
if [ -z "$BANK_LIST" ]
then
   NO_BANK=1
   echo -e "\nNo Backup Banks Have Been Defined in Dirvish\n"
else
   BANK_FILE=/tmp/banklist.out
   >$BANK FILE
   COUNT=0
   for B in $BANK_LIST
   do
      ((COUNT == COUNT + 1))
      LAST_ENTRY=$B
      echo -e "\t$B" | tee -a $BANK_FILE
   done
fi
```

```
# Display the menu options
echo -e "\n\n1) Add a New Backup Bank"
echo -e "\n2) Delete a Current Backup Bank"
echo -e "\n3) Return to the Previous Menu"
echo -e "\n\nSelect an Option: \c"
# Read the user input
read OPT
case $OPT in
1) # Add a New Backup Bank
   echo -e "\nEnter the Bank to Add: \c"
   read A_BANK
   echo -e "\nAdding New Backup Bank: $A_BANK"
   if (( NO_BANK == 0 ))
   then
     sed "s!$LAST_ENTRY!& \n\t${A_BANK}!g" $M_CONFIG > ${M_CONFIG}.
modified
    if (( $? == 0 ))
     then
       # Save the old Dirvish master config file with today's
datestamp
        cp ${M_CONFIG} ${M_CONFIG}.$(date +%m%d%Y)
        cp ${M_CONFIG}.modified ${M_CONFIG}
        echo -e "\n$A_BANK Successfully Added to Dirvish Master
Config File"
        # Check to see if the $A_BANK directoey exists, if not
        # ask the user if it is okay to create it.
        chk_create_dir $A_BANK
     else
        echo -e "\nERROR: Adding $A_BANK Failed...See
Administrator..."
    fi
   else
     if $(grep -q "bank:" $M_CONFIG)
     then
        # NOTICE: It is important to note that sed does not "require"
        # us to use / as a field separator. Here we are using ! as a
        # sed field separator because we are working with UNIX
directory
        # paths, sed gets confused using / as a field separator.
        sed "s!bank:!& \n\t${A_BANK}!g" $M_CONFIG > ${M_CONFIG}.
modified
```

```
if (( $? == 0 ))
        then
            cp ${M_CONFIG} ${M_CONFIG}.$(date +%m%d%Y)
            cp ${M_CONFIG}.modified ${M_CONFIG}
            echo -e "\n$A_BANK Successfully Added to Dirvish Master
Config File"
            chk_create_dir $A_BANK
        else
            echo -e "\nERROR: Adding $A_BANK Failed...See
Administrator..."
        fi
     else
        echo -e "bank:\n\t$A_BANK" >> ${M_CONFIG}.modified
        if (($? == 0))
       then
           cp ${M_CONFIG} ${M_CONFIG}.$(date +%m%d%Y)
           cp ${M_CONFIG}.modified ${M_CONFIG}
           echo -e "\n$A_BANK Successfully Added to Dirvish Master
Config File"
           chk_create_dir $A_BANK
        else
           echo -e "\nERROR: Adding $A_BANK Failed...
See Administrator..."
        fi
     fi
   fi
   rm -f $BANK_FILE
   echo -e "\nPress Enter to Continue...\c"
   read KEY
   ;;
2) echo -e "\nEnter the Backup Bank to Remove: \c"
   read R_BANK
   if [ -d $R_BANK ]
   then
     POPULATED=$(ls $R_BANK | wc -1)
     if ((POPULATED > 0))
     then
        echo -e "\nWARNING: The Bank $R_BANK has the Following Backup
Images:\n"
        ls $R_BANK | more
        echo -e "\nAre you Sure you Want to Remove this Bank and
all of the Backup Images? (y/n): \c"
        read ANS
        case $ANS in
```

```
y|Y) continue
             ;;
        n N) break
             ;;
          *) echo -e "\nInvalid Input..."
             ;;
        esac
     fi
   fi
   if $(cat "$BANK_FILE" | grep -q "$R_BANK")
   then
       if (( COUNT == 1 ))
       then
           echo -e "\nWARNING: $R_BANK is the Only Backup Bank
Currently Configured!"
           echo -e "\nRemoving this Bank Will Cripple Dirvish..."
       fi
       echo -e "\nAre you Sure You Want to Remove the $R_BANK Bank?
(y/n): \c"
       read ANS4
       case $ANS4 in
       y|Y) cat $M_CONFIG | grep -v $R_BANK > ${M_CONFIG}.modified
            cp -p $M_CONFIG ${M_CONFIG}.bak.$(date +%m%d%y)
            cp ${M_CONFIG}.modified $M_CONFIG
            echo -e "\n$R_BANK Removed from the Dirvish Configuration
File..."
            if [ -d $R_BANK ]
            then
               echo -e "\nDo You Want to Remove the $R_BANK Directory?
(y/n): \c"
               read ANS
               case $ANS in
               y Y) rm -r $R_BANK
                   if (( $? == 0 ))
                   then
                       echo -e "\n$R_BANK Directory Removed
Successfully"
                   else
                       echo -e "\nERROR: Remove $R_BANK Directory
Failed"
                   fi
                   ;;
               n|N) echo -e "\nSkipping $R_BANK Directory Removal"
                   ;;
                 *) echo -e "\nInvalid Input..."
                   ;;
               esac
```

```
Listing 27-24 (continued)
```

```
fi
           ;;
      n|N) echo -e "\nSkipping Bank Removal\n"
           ;;
        *) echo -e "\nInvalid Entry..."
          ;;
      esac
  else
      echo -e "\nERROR: $R_BANK is Not a Valid Bank"
  fi
  echo -e "\nPress Enter to Continue...\c"
  read KEY
  ;;
3) continue
  ;;
*) echo -e "\nInvalid Entry...\n"
  ;;
esac
}
******
check_root_user ()
{
# This function ensures the user is "root"
if [[ $(whoami) != root ]]
then
   echo -e "\n\n\tERROR: Only the root User can Execute this
Program..."
   echo -e "\n\n\t...EXITING...\n"
   exit 1
fi
}
*****
check_config ()
{
# Find the Dirvish Master Configuration File
if [ -r /etc/dirvish.conf ]
then
   M_CONFIG=/etc/dirvish.conf
elif [ -r /etc/dirvish/master.conf ]
```

Listing 27-24 (continued)

```
then
   M_CONFIG=/etc/dirvish/master.conf
else
   echo -e "\n\n\tERROR: Dirvish is not installed on this system"
   echo -e "\tTo use this program install rsync and dirvish first."
   echo -e "\n\t...Exiting...\n"
   exit 2
fi
}
****
# BEGINNING OF MAIN
****
check_root_user
check_config
until (( DONE == 1 ))
do
   display_main_menu
   read OPTION
   case $OPTION in
   1) # Dirvish RunAll
     run_all
      ;;
   2) # Dirvish Run Backup
      run_backup
      ;;
   3) # Dirvish Locate Image
     locate_restore_image
      ;;
   4) # Dirvish Expire Backup(s)
      expire_backup
      ;;
   5) # Dirvish Add Backup
      add_backup
      ; ;
   6) # Dirvish Delete Backup
      delete_backup
      ;;
   7) # Manage Backup Banks
      manage_banks
      ;;
   8) clear
      DONE=1
      ;;
   *) echo -e "\n\tERROR: Invalid Entry..."
```

Listing 27-24 (continued)

```
sleep 2
;;
esac
done
```

I know this is a long shell script, but it takes care of a lot of different Dirvish tasks and is a good tool for any shop using Dirvish as a backup solution.

Using the dirvish_ctrl Shell Script

In this section we are going to show examples of using the dirvish_ctrl shell script to perform each task. Please follow through this section, and then play around with Dirvish on your own, I'm sure you will be impressed with the capabilities of Dirvish.

The dirvish_ctrl main menu is shown in Listing 27-25.

```
WELCOME TO DIRVISH BACKUP COMMAND-CENTER
1) Dirvish Run All Backups
2) Dirvish Run a Particular Backup
3) Dirvish Locate/Restore a Backup Image
4) Dirvish Expire/Delete Backup (s)
5) Dirvish Add a New Backup
6) Dirvish Remove a Backup
7) Dirvish Manage Backup Banks
8) EXIT
Select an Option:
```

Listing 27-25 dirvish_ctrl shell script main menu

We are going to step through using each option in the following sections.

Running All Backups Defined in the Runall: Stanza

The dirvish-runall command is used for this task and has the limitation of requiring each server to be defined in the Runall: stanza in the master.conf file.

Notice in Listing 27-26 that yogi, booboo, dino, wilma, and fred are the only backups defined in the Runall: stanza in the Dirvish master configuration file, master.conf.

```
14:03:58 dirvish --vault yogi
14:03:59 dirvish --vault booboo
14:04:04 dirvish --vault dino
14:04:22 dirvish --vault wilma
14:04:51 dirvish --vault fred
14:05:04 done
Backups Complete...Press Enter to Continue...
```

Listing 27-26 Using Option 1, Dirvish Run All Backups

Running One Particular Backup

We also have the ability to run individual backups anytime. Listing 27-27 shows a backup of booboo.

```
RUN A PARTICULAR BACKUP
Enter a Hostname to Back Up: booboo
Searching for default.conf in booboo's Vault
Found Configuration File...Starting Backup...
Dirvish Exit Code: 0
Backup Complete...
Press Enter to Continue...
```

Listing 27-27 Using Option 2, Dirvish Run a Particular Backup

When we run an individual backup, the script first checks to ensure the vault has a default.conf file. If the default.conf file is found, we execute the backup using the following syntax:

/usr/local/sbin/dirvish -vault \$HN

where \$HN is the name of the vault to back up.

Locating and Restoring Files

If you make backups, you need a way to restore the files when needed. The nice thing about Dirvish is that each snapshot backup is fully intact in the image tree

subdirectory. So, after we find the file we want in the individual image directory, we just copy the files/directories recursively to the local restore area. In this script the restore area is hard-coded to /dirvish_restores. If your Dirvish backup server is different, modify the RESTORE_AREA variable in the definition section at the top of the dirvish_ctrl shell script in Listing 27-24. Listing 27-28 shows the process of locating and restoring a file.

```
LOCATE/RESTORE A BACKUP IMAGE
Enter the Hostname where the File/Directory Resides: yogi
Enter as Much of the Directory Path and Filename as You Know:
dirvish_ctrl
Searching for Archive Images...
3 matches in 16 images
/scripts/DESTIN_FL/dirvish_ctrl
    May 25 2007 yogi-071211140359, yogi-071211125815
   May 25 2007 yogi-071211115107, yogi-071211115013,
                yogi-071210123131
                yogi-071210123128, yogi-071210123049,
                yogi-071210122858
                yogi-071210122226, yogi-071210122214,
                 yogi-071210122030
/scripts/dirvish_ctrl
    Dec 11 14:03 yogi-071211140359
    Dec 11 12:56 yogi-071211125815
    Dec 11 10:04 yogi-071211115107, yogi-071211115013
    Dec 10 12:31 yogi-071210123131, yogi-071210123128
    Dec 10 12:30 yogi-071210123049
    Dec 10 12:28 yogi-071210122858
    Dec 10 12:19 yogi-071210122226, yogi-071210122214,
                yogi-071210122030
    May 25 09:28 yogi-070719153227, yogi-070627130342,
                yogi-070626093920
                yogi-070529152028
    May 21 14:30 scripts-070521144312
/scripts/dirvish_ctrl.log
    Dec 11 14:03 yogi-071211140359
    Dec 11 12:56 yogi-071211125815
Did You Find What You Want? (y/n):
Do You Want to Perform a Restore? (y/n): y
```

Listing 27-28 Using Option 3, Dirvish Locate/Restore a Backup Image

```
Enter the FULL PATHNAME of the File/Directory You Want to Restore:
/scripts/dirvish_ctrl
Searching Images...
2 matches in 16 images
/scripts/dirvish_ctrl
   Dec 11 14:03 yogi-071211140359
   Dec 11 12:56 yogi-071211125815
   Dec 11 10:04 yogi-071211115107, yogi-071211115013
   Dec 10 12:31 yogi-071210123131, yogi-071210123128
    Dec 10 12:30 yogi-071210123049
    Dec 10 12:28 yogi-071210122858
    Dec 10 12:19 yogi-071210122226, yogi-071210122214,
                yogi-071210122030
   May 25 09:28 yogi-070719153227, yogi-070627130342,
                yogi-070626093920
                yogi-070529152028
   May 21 14:30 scripts-070521144312
/scripts/dirvish_ctrl.log
   Dec 11 14:03 yogi-071211140359
    Dec 11 12:56 yogi-071211125815
Enter the Image to Restore From: yogi-071211140359
DIRNAME is /scripts
Files Restored to the Following Restore Area:
/dirvish_restores/yogi/dirvish_ctrl
Press Enter to Continue ...
```

Notice in Listing 27-28 that we can search all the images in a specific vault for files and directories without knowing the entire file or directory name. The search does a grep-type search for patterns. The result is a list of all the matching files and directories and a list of all the images that contain files that match the search pattern. The user must select a file to restore and the image to retrieve the file from. As stated before, retrieving the file is nothing more than a cp - fpr command.

Deleting Expired Backups and Expiring Backups

This option lets us delete expired backups from all vaults, and from individual vaults. We can also expire images that are not currently expired. This is the method to use to delete a backup you no longer need but is not expired. Listing 27-29 shows the sub-menu to delete and expire backups.

```
DIRVISH EXPIRE BACKUP(S)

1) Delete Expired Backups for ALL Hosts

2) Delete Expired Backups for One Host

3) Expire One Backup for One Host

4) Expire All Backups for One Host

5) Previous Menu

Select an Option:
```

Listing 27-29 Using Option 4, Dirvish Expire/Delete Backup(s) Sub-Menu

Now let's look at each option individually. Listing 27-30 shows how to delete expired backups from all vaults.

```
Deleting Expired Backups for ALL Hosts ...
Expiring images as of 2007-12-11 14:56:19
VAULT:BRANCH IMAGE
                              CREATED
                                                EXPIRED
yogi:default yogi-070521143304 2007-05-21 14:33 15 day == 2007-05-25
09:27
yogi:default yogi-070521143444 2007-05-21 14:34 15 day == 2007-05-25
09:27
yogi:default yogi-070521143540 2007-05-21 14:35 15 day == 2007-05-25
09:27
yogi:default yogi-070521143806 2007-05-21 14:38 15 day == 2007-05-25
09:27
yogi:default yogi-070521143823 2007-05-21 14:38 15 day == 2007-05-25
09:27
yogi:default yogi-070521143841 2007-05-21 14:38 15 day == 2007-05-25
09:27
yogi:default yogi-070521143927 2007-05-21 14:39 15 day == 2007-05-25
09:27
yogi:default scripts-070521144235 2007-05-21 14:42 15 day
== 2007-05-25 09:28
yogi:default
              scripts-070521144312 2007-05-21 14:43 15 day
== 2007-05-25 09:27
```

Listing 27-30 Using Expire Option 1, Delete Expired Backups for ALL Hosts

```
Tasks Complete...
Press Enter to Continue...
```

As Dirvish deletes expired backups, it produces a list of activities. Listing 27-31 shows how to delete expired backups for a particular vault.

```
DIRVISH EXPIRE BACKUP(S)

1) Delete Expired Backups for ALL Hosts
2) Delete Expired Backups for One Host
3) Expire One Backup for One Host
4) Expire All Backups for One Host
5) Previous Menu
Select an Option: 2
Enter the Hostname to Delete Expired Backups: booboo
Tasks Complete...
Press Enter to Continue...
```

Listing 27-31 Using Expire Option 2, Delete Expired Backups for One Host

Notice that we did not go to a sub-menu for this option. Listing 27-32 shows how to expire one particular image in one particular vault.

```
Enter the Hostname for the Backup Image: yogi
Select an Image to Expire from the Following List:
yogi-071210123128
yogi-071210122858
```

Listing 27-32 Using Expire Option 3, Expire One Backup for One Host

```
yogi-071210123049
yogi-071210122030
yogi-071211125815
yogi-071211115013
yogi-071211115107
yogi-071210123131
yogi-071211140359
yogi-071210122226
yogi-071210122214
        Enter the Image to Expire: yogi-071211140359
       Tasks Complete...Do You Want to Delete This Expired Image?
(y/n): y
        Deleting Expired Image: yogi-071211140359
        Dirvish-expire Completed with Return Code: 0
        Tasks Complete ...
        Press Enter to Continue...
```

This option in Listing 27-32 asks the user for a vault. Then we show a list of current backup images. The script asks the user to enter an image name to expire, and modify the summary file located in the specified vault to change the time stamp to the current date/time. Listing 27-33 shows how to expire all the backup images in a particular vault.

```
Enter the Hostname for the Backup Image: booboo
Setting the Following Images to Expire:
yogi-071211100332
yogi-071211140359
yogi-07121114138
yogi-071211141309
yogi-071211141248
Press Enter to Continue...Expiring yogi-071211100332
Expiring yogi-071211140359
Expiring yogi-07121114138
Expiring yogi-07121114138
```

Listing 27-33 Using Expire Option 4, Expire All Backups for One Host

```
Expiring yogi-071211115107
Expiring yogi-071211141248
Tasks Complete...Do You Want to Delete Expired Images? (y/n): y
Deleting Expired Images...
Expiring images as of 2007-12-11 15:21:07
Restricted to vault yogi
VAULT:BRANCH IMAGE
                                         EXPIRED
                             CREATED
yogi:default yogi-071211115107 2007-12-11 11:51 30 day == 2007-12-11
15:20
yogi:default yogi-071211140359 2007-12-11 14:03 30 day == 2007-12-11
15:20
yogi:default yogi-071211141138 2007-12-11 14:11 30 day == 2007-12-11
15:20
yogi:default yogi-071211141248 2007-12-11 14:12 30 day == 2007-12-11
15:20
yogi:default yogi-071211141309 2007-12-11 14:13 30 day == 2007-12-11
15:20
Dirvish-expire Completed with Return Code: 0
Tasks Complete...
Press Enter to Continue...
```

For this option we modify the date/time stamp in the summary files in all image subdirectories for that particular vault.

Adding a New Dirvish Backup Vault

To add a new backup vault, we need only to create a new vault directory and create a default.conf file for it. To create the new default.conf file, we need some information. Listing 27-34 shows the sub-menu to add a new Dirvish vault.

```
DIRVISH ADD A BACKUP
Select Each Option to Add a New Backup
1) Enter Hostname
2) Select a Bank for this Backup
```

Listing 27-34 Main Menu Option 5, Add a New Backup Sub-Menu

3) Enter Directory Tree to Back Up
 4) Enter Files and Directories to Ignore
 5) Enter Days to Retain Each Backup
 6) Add the New Backup to Dirvish
 7) Create the Initial Backup
 8) Main Menu
 Select Each Option to Add a New Backup
 Enter Option:

Listing 27-34 (continued)

The process of adding a new backup is to select Options 1–7, one at a time, to configure the new backup in Dirvish, and then to run an initial backup. Let's look at each step individually.

Option 1 is where we get the name of the new vault. Remember that we refer to hostname and vault interchangeably for this shell script. Listing 27-35 shows adding a new hostname.

DIRVISH ADD A BACKUP Select Each Option to Add a New Backup 1) Enter Hostname 2) Select a Bank for this Backup 3) Enter Directory Tree to Back Up 4) Enter Files and Directories to Ignore 5) Enter Days to Retain Each Backup 6) Add the New Backup to Dirvish 7) Create the Initial Backup 8) Main Menu

Listing 27-35 Add a Backup Option 1, Enter Hostname

```
Select Each Option to Add a New Backup
Enter Option: 1
Enter the Hostname for this Backup: booboo
```

Listing 27-35 (continued)

Notice that this option did not create a sub-menu. Listing 27-36 shows how to select a bank to store this new backup in.

```
SELECT A BANK TO STORE THIS BACKUP
Available Banks:
/backup_bank1
/prod_backups
Please Enter a Bank: /backup_bank1
```

Listing 27-36 Add a Backup Option 2, Select a Bank for This Backup

Notice that we display all the banks that are currently defined in the master.conf file before we ask the user to enter the desired bank for this backup. Listing 27-37 shows how to add the directory tree to back up.

DIRVISH ADD A BACKUP Select Each Option to Add a New Backup 1) Enter Hostname 2) Select a Bank for this Backup 3) Enter Directory Tree to Back Up 4) Enter Files and Directories to Ignore 5) Enter Days to Retain Each Backup 6) Add the New Backup to Dirvish

Listing 27-37 Add a Backup Option 3, Enter Directory Tree to Back Up

7) Create the Initial Backup
 8) Main Menu
 Select Each Option to Add a New Backup
 Enter Option: 3
 Enter the Directory Tree to Back Up: /scripts

Listing 27-37 (continued)

This step does not create a sub-menu, either. In Listing 27-38 we list all the files and directories in the tree specified in the last step, and ask the users if they meant to exclude anything. If they do, we keep reading input until 99 is entered.

```
DIRECTORY TREE(S) and FILES TO IGNORE

The Specified Directory Tree Contains the Following Files/Directories:

mysqlbackup.sh

dsh

get-date.ksh

pluck.sh

tmp/shortlog.out

Do you want to Exclude any Files or Directories from the Backups? (y/n): y

Enter Directories and Files to Ignore One per Line

Enter 99 when finished

tmp/

test/

99
```

Listing 27-38 Add a Backup Option 4, Enter Files and Directories to Ignore

NOTE Notice that when we define sub-directories to exclude, we use relative pathnames (path does not start with a /) with a trailing forward slash — for example, tmp/. This is an important point to note because we must specify the relative pathname in the exclude: stanza of the default.conf file.

```
DIRVISH ADD A BACKUP
Select Each Option to Add a New Backup
1) Enter Hostname
2) Select a Bank for this Backup
3) Enter Directory Tree to Back Up
4) Enter Directory Trees to Ignore
5) Enter Days to Retain Each Backup
6) Add the New Backup to Dirvish
7) Create the Initial Backup
6) Main Menu
Select Each Option to Add a New Backup
Enter Option: 5
Days before each Backup Expires: 30
```

Listing 27-39 shows the prompt for the number of days before the backup expires.

Listing 27-39 Add a Backup Option 5, Enter Days to Retain Each Backup

Notice again that we did not create a sub-menu. Listing 27-40 creates the new vault directory, if it does not already exist, and the new default.conf file that described this backup.

```
ADD NEW BACKUP TO DIRVISH
WARNING: default.conf File Exists...Rebuild default.conf? (y/n): y
Creating default.conf Dirvish Configuration File for booboo
Copying file to: /backup_bank1/borg/dirvish/default.conf
...Press ENTER to Continue...
```

Listing 27-40 Add a Backup Option 6, Add the New Backup to Dirvish

The internal file we first create is a temporary file. Here we copy the file to the correct vault. Listing 27-41 shows the step where we create the initial Dirvish backup. This initial backup is required because it is the reference set for future backups to reduce the amount of data transmitted over the network. The unchanged data is not transmitted.

```
CREATE INITIAL DIRVISH BACKUP
Backup Vault for borg Exists...Continuing...
Run an Initial Backup Now? (y/n): y
Executing Initial Dirvish Backup for borg...
Initial Backup for borg Completed with a Return Code: 0
Press Enter to Continue...
```

Listing 27-41 Add a Backup Option 7, Create the Initial Backup

Removing a Dirvish Vault

To remove a Dirvish vault, we need only to remove the vault's subdirectory and all its contents. Listing 27-42 shows the removal of a vault.

```
REMOVE A SERVER FROM DIRVISH BACKUPS
Enter the Host to Remove from Dirvish Backups: booboo
Remove Backup Vault booboo from the Dirvish Backup Server? (y/n): y
Removing Backup Vault for booboo from Dirvish Backups...
Backup Vault Removed...Press Enter to Continue...
```

Listing 27-42 Main Menu Option 6, Dirvish Remove a Backup

Here we ask the user for the name of the vault/host to remove. Before removing a vault, we ask the user to confirm the removal.

Managing Dirvish Backup Banks

This task allows us to add and delete Dirvish backup banks. Listing 27-43 shows the sub-menu to manage the Dirvish backup banks.

```
MANAGE DIRVISH BACKUP BANKS
Currently Configured Backup Bank(s):
/backup_bank1
/prod_backups
/backup_bank2
1) Add a New Backup Bank
2) Delete a Current Backup Bank
3) Return to the Previous Menu
Select an Option:
```

Listing 27-43 Main Menu Option 7, Dirvish Manage Backup Banks Sub-Menu

Adding a New Dirvish Backup Bank

To add a new Dirvish backup bank, we select Option 3. Listing 27-44 shows how to add a new bank to the Dirvish configuration, and ask the user before we create the new directory for the bank.

```
MANAGE DIRVISH BACKUP BANKS

Currently Configured Backup Bank(s):

/backup_bank1

/prod_backups

1) Add a New Backup Bank

2) Delete a Current Backup Bank

3) Return to the Previous Menu

Select an Option: 1

Enter the Bank to Add: /qa_backups
```

Listing 27-44 Example of adding a new Dirvish backup bank

Adding New Backup Bank: /qa_backups /qa_backups Successfully Added to Dirvish Master Config File /qa_backups Directory Does Not Exist...Create /qa_backups Directory? (y/n): y Creating /qa_backups Directory... /qa_backups Directory Created Press Enter to Continue...

Listing 27-44 (continued)

Notice in Listing 27-44 that we ask the user before we create the directory for the new bank. This is to give the users a chance to build a new filesystem for the bank if they have not already done so.

Removing a Dirvish Backup Bank

To remove a Dirvish backup bank, we select Option 2. Listing 27-45 shows how to remove a bank from the Dirvish configuration, and then prompt the user to delete the bank from the system. This will also remove all the backup images stored in that directory tree.

```
Enter the Backup Bank to Remove: /backup_bank2
Are you Sure You Want to Remove the /backup_bank2 Bank? (y/n): y
/backup_bank2 Removed from the Dirvish Configuration File...
Do You Want to Remove the /backup_bank2 Directory? (y/n): y
/backup_bank2 Directory Removed Successfully
Press Enter to Continue...
```

Listing 27-45 Removing a Dirvish backup bank

Notice that before we delete a bank directory, we confirm the removal. Removing the bank directory deletes all the stored images, as well.

Other Things to Consider

I know this is a long chapter and we covered a lot of points in fine detail, but we can always improve. I did not cover using branches very much in this chapter, and you may want to add some functionality to support branches if you use them. If you are running backups on a daily basis, this menu program is really not what you want. A better bet is to add a cron entry to the root cron table to first delete all expired backups by running the dirvish-expire command, then execute the dirvish-runall command to kick off the backups. Of course, you need to add each vault to the Runall: stanza in the master.conf file. A sample cron entry is shown here:

0, 23, 0, 0, 0 /usr/local/sbin/dirvish-expire -quiet; /usr/local/sbin/dirvish-runall --quiet

This cron entry will start every night at 11:00 p.m. and first run the dirvish-expire command in quiet mode. When the dirvish-expire command completes, the dirvish-runall command is started in quiet mode.

Summary

I really enjoyed writing this chapter. You should play around with Dirvish on your own and modify the dirvish_ctrl shell script to meet the needs of your shop.

In the next chapter we will look at techniques to monitor a user's keystrokes using the script command.

Lab Assignments

- 1. Modify the dirvish_ctrl shell script to give the user the option when creating a new backup to add it to the Runall: stanza in the master.conf file.
- 2. Rewrite the dirvish_ctrl shell script to use the select command with a case statement to create all the menus. For more details on the select command, see the "Create a Menu with the select Command" section in Chapter 1, "Scripting Quick Start and Review," and the online manual page, **man select**.
- 3. Install Dirvish on a server, from scratch, using any method you choose. Using the dirvish_ctrl shell script, configure an initial bank, then configure Dirvish to back up a remote server while excluding all *.log, *.a, core, /proc, and /tmp files and directories. Run an initial backup and time its execution. After this initial backup completes, run a second backup (not another initial backup) and time this second backup. What is the percentage difference in execution times between the two backups?

CHAPTER **28**

Monitoring and Auditing User Keystrokes

In most large shops there is a need, at least occasionally, to monitor a user's actions. Thanks to Sarbanes-Oxley requirements on publicly traded United States companies for auditing, we are now required to audit the keystrokes of anyone with root access to the system or other administration type accounts, such as oracle. Contractors on site can pose a particular security risk. Typically when a new application comes into the environment, one or two contractors are on site for a period of time for installation, troubleshooting, and training personnel on the product. I always set up contractors in **sudo** (see Chapter 23, "Creating a System-Configuration Snapshot," for more details on sudo) to access the new application account, after I change the password. sudo tracks only the commands that were entered with a date/time stamp. The detail of the command output from stdout and stderr does not get logged so you do not have a complete audit trail of exactly what happened if a problem arises.

To get around this dilemma you can track a user's keystrokes from the time he or she accesses a user account until the time he or she exits the account, if you have the space for the log file. This little feat is accomplished using the **script** command. The idea is to use sudo to kick off a shell script that starts a script session. When the script session is running, all of the input and output on the terminal is captured in the log file. Of course, if the user goes into some menus or programs the log file gets a little hard to read, but we at least have an idea of what happened. This monitoring is not done surreptitiously because I always want everyone to know that the monitoring is taking place. When a script session starts, output from the script command informs the user that a session is running and gives the name of the session's log file. We can also set up monitoring to take place from the time a user logs in until the user logs out. For this monitoring we do not need sudo, but we do need to edit the \$HOME/.profile or other login-configuration file for the particular user.

Syntax

Using the script command is straightforward, but we want to do a few more things in the shell script. Giving a specific command prompt is one option. If you are auditing root access, you need to have a timeout set so that after about five minutes (see the TMOUT environment variable discussion after Listing 28-3 later in this chapter) the shell times out and the root access ends. On a shell timeout, the session is terminated and the user is either logged out or presented with a command prompt, but we can control this behavior. We have many options for this set of shell scripts. You are going to need to set up sudo, *Super User Do*, on your machine. The full details for installing and configuring sudo are in Chapter 23. We want sudo to be configured with the names of each of the shell scripts that are used for this monitoring effort, as well as the specific users that you will allow to execute them. We will get to these details later.

The script command works by making a typescript of everything that appears on the terminal. The script command is followed by a filename that will contain the captured typescript. If no filename is given the typescript is saved in the current directory in a file called typescript. For our scripting we will specify a filename to use. The script session ends when the forked shell is exited, which means that there are two exits required to completely log out of the system. The script command has the following syntax:

script [filename]

As the script session starts, notification is shown on the terminal and a time stamp is placed at the top of the file, indicating the start time of the session. Let's look at a short script session as used on the command line in Listing 28-1.

```
[root:yogi]@/# more /usr/local/logs/script/script_example.out
Script command is started on Wed May 8 21:35:27 EDT 2002.
[root:yogi]@/# cd /usr/spool/cron/crontabs
[root:yogi]@/usr/spool/cron/crontabs# ls
adm root sys uucp
[root:yogi]@/usr/spool/cron/crontabs# ls -al
total 13
drwxrwx--- 2 bin cron
                                              512 Feb 10 21:36 .
drwxr-xr-x 4 bin cron
-rw-r--r-- 1 adm cron
                                             512 Jul 26 2001 ..
                                            2027 Feb 10 21:36 adm

      -rw-----
      1 root
      cron
      1125 Feb 10 21:35 root

      -rw-r--r--
      1 sys
      cron
      864 Jul 26 2001 sys

      -rw-r--r--
      1 root
      cron
      703 Jul 26 2001 uucp

[root:yogi]@/usr/spool/cron/crontabs# cd ../..
[root:yogi]@/usr/spool# ls -1
total 12
drwxrwsrwt 2 daemon staff
                                              512 Sep 17 2000 calendar
                                              512 Jul 26 2001 cron
drwxr-xr-x 4 bin cron
drwxrwxr-x 7 lp
                          lp
                                              512 Mar 23 15:21 lp
```

Listing 28-1 Command-line script session

```
      drwxrwxr-x
      5 bin
      printq
      512 May 01 20:32 lpd

      drwxrwxr-x
      2 bin
      mail
      512 May 06 17:36 mail

      drwxrwxr---
      2 root
      system
      512 May 06 17:36 mqueue

      drwxrwxr-x
      2 bin
      printq
      512 Apr 29 11:52 qdaemon

      drwxr-xr-x
      2 root
      system
      512 Jul 26 2001 rwho

      drwxrwsrwx
      2 bin
      staff
      512 Jul 26 2001 secretmail

      drwxr-xr-x
      11 uucp
      uucp
      512 Mar 13 20:43 uucp

      drwxrwxrwx
      2 uucp
      uucp
      512 Apr 16 2001 writesrv

      drwxrwxr-x
      2 root
      system
      512 Apr 16 2001 writesrv

      groot:yogi]@/usr/spool#
      exit
      8 21:36:11 EDT 2002.
```

Listing 28-1 (continued)

Notice that every keystroke is logged as well as all of the command output. At the beginning and end of the log file a script command time stamp is produced. These lines of text are also displayed on the screen as the script session starts and stops. These are the user notifications given as the monitoring starts and stops.

Scripting the Solution

There are three different situations in which you want to use this type of monitoring/auditing. In this first instance we have users that you want to monitor the entire session. In the next situation you want to monitor activity only when a user wants root access to the system. Our systems have only direct root login enabled on the console. To gain access to the root user remotely, you must log in as yourself and su to root, or use sudo to gain root access. So, for our script, we can use sudo to switch to root using the broot script (shown later in this chapter). The third script is a catchall for other administrative user accounts that you want to audit. The first script is covering end-to-end monitoring with the script execution starting at login through the user's \$HOME/.profile.

Before we actually start the script session, there are some options to consider. Because we are executing a shell script from the user's .profile we need to ensure that the script is the last entry in the file. If you do not want the users to edit any .profile files, you need to set the ownership of the file to root and set the user to read-only access.

Logging User Activity

We are keeping log files so it is a good idea to have some kind of standard format for the log filenames. You have a lot of options for filenames, but I like to keep it simple. Our log files use the following naming convention:

```
[hostname].[user $LOGNAME].[Time Stamp]
```

We want the hostname because most likely you are monitoring users on multiple systems and using a central repository to hold all of the log files. When I write a shell script I do not want to execute a command more times than necessary. The hostname command is a good example. Assigning the system's hostname to a variable is a good idea because it is not going to change, or it should not change, during the execution of the script. To assign the hostname of the system to a variable use the following command-substitution syntax:

THISHOST=\$(hostname)

For the date/time stamp a simple integer representation is best. The following date command gives two digits for month, day, year, hour, minute, and second:

```
TS=$(date +%m%d%y%H%M%S)
```

Now we have to reference only the \$TS variable for the date/time stamp. Because the user may change we can find the active username with either of the following environment variables:

```
echo $LOGNAME
echo $USER
echo $LOGIN
```

As you change user IDs by using the switch user command (su), all of these environment variables change accordingly. However, if a user does a switch user using sudo, the \$LOGIN environment variable carries over to the new user while the \$LOGNAME and \$USER environment variables gain the new user ID. Now we have everything to build a log filename. A good variable name for a log file is LOGFILE, unless this variable is used by your system or another application. On my systems the LOGFILE variable is not used. Not only do we need to create the name of the \$LOGFILE, but we need to create the file and set the permissions on the file. The initial permissions on the file need to be set to read/write by the owner, chmod 600 \$LOGFILE. The following commands set up the log file:

```
TS=$(date +%m%d%y%H%M%S)  # Create a time stamp
THISHOST=$(hostname)  # Query the system for the hostname
LOGFILE=${THISHOST}.${LOGNAME}.$TS  # Name the log file
touch ${LOGDIR}/$LOGFILE  # Create an empty log file
chmod 600 ${LOGDIR}/${LOGFILE}  # Set the file permissions
```

A sample filename is shown here:

yogi.randy.05110214519

The filename is good, but where do we want to store the file on the system? I like to use a separate variable to hold the directory name. With two separate variables representing the directory and filename, you can move the log directory to another location and have to change just one entry in the script. I set up a log directory on my

system in /usr/local/logs. For these script log files I added a subdirectory called script. Then I set a LOGDIR variable to point to my logging directory, as shown here:

```
LOGDIR=/usr/local/logs/script
```

Starting the Monitoring Session

With the logging set up we are ready to start a script session. We start the session using the following syntax:

```
script ${LOGDIR}/${LOGFILE}
```

When the script session starts, a message is displayed on the screen that informs the user that a script session has started and lists the name of the script log file, as shown here:

```
Script command is started. The file is /usr/local/logs/script/yogi.randy. 051102174519.
```

If the user knows that monitoring is going on and also knows the name of the file, what is to keep the user from editing or deleting the log? Usually directory permissions will take care of this little problem. During the script session the actual log file is an open file — that is, actually a system temporary file that cannot be accessed directly by the user. But if the user is able to delete the *\$LOGFILE*, you have lost the audit trail. This is one problem that we will discuss later.

Where Is the Repository?

So far here is the scenario. A user has logged in to the system. As the user logs in, a monitoring session is started using the script command, which logs all of the terminal output in a log file that we specify. During the time that the session is active the log file is open as a system temporary file. When the session ends, by a user typing exit or Ctrl + D or by an exit signal, the log file is closed and the user is notified of the session ending, and again the name of the log file is displayed.

For security and auditing purposes we need to have a central repository for the logs. The method I like to use is email. When the session ends we want to set the file permissions on the log file to *read only* by the owner. Then we email the log to another machine, ideally, which is where the repository is located. Once the email is sent I compress the local file and exit the script.

With two copies of the user session existing on two different machines, an audit will easily detect any changes. In fact, if a user tries to change the log these commands will also be logged. You may have different ideas on handling the repository, but I set up a user on a remote machine that I use as a log file manager, with a name **logman**. The logman user's email is the repository on the audit/security machine. For simplicity in this shell script we are going to email the logs to the local logman user. Later in this chapter we will cover mutt and metamail to show techniques to send

files as attachments to an email. To send a simple email, I use the mailx command, as shown here:

mailx -s "\$TS - \$LOGNAME Audit Report" \$LOG_MANAGER < \${LOGDIR}/\${LOGFILE}</pre>

In the log_keystrokes.ksh shell script in Listing 28-2, the \$LOG_MANAGER is defined as logman. The nice thing about having a variable hold the mail recipients is that you can add a second repository of other people to receive email notifications. By using the local logman account you have other options. You can set up mail aliases; one of my favorites is to use the logman account as a *bounce account*. By adding a .forward file in the \$HOME directory for the logman user, you can redirect all of the email sent to the logman user to other destinations. If a .forward file exists in a user's home directory, the mail is not delivered to the user but instead is sent to each email address and alias listed in the .forward file. A sample .forward file is shown here:

```
yogibear@cave.com
booboo@cave.com
dino@flintstones.org
admin
```

With the preceding entries in the \$HOME/.forward file for the logman user, all mail directed to logman is instead sent to the three email addresses and all of the addresses pointed to by the admin email alias.

The Scripts

We have covered all of the basics for the shell scripts. We have three different shell scripts that are used in different ways. The first script is intended to be executed at login time by being the last entry in the user's \$HOME/.profile. The second shell script is used only when you want to gain root access, which is done through sudo, and the third script is a catchall for any other administration-type accounts that you want to audit, which also use sudo. Let's first look at the login script called log_keystrokes.ksh, shown in Listing 28-2.

```
#!/bin/ksh
#
#
SCRIPT: log_keystrokes.ksh
#
# AUTHOR: Randy Michael
# DATE: 05/08/2002
# REV: 1.0.P
# PLATFORM: Any Unix
#
#
# PURPOSE: This shell script is used to monitor a login session by
# capturing all of the terminal data in a log file using
# the script command. This shell script name should be
# the last entry in the user's $HOME/.profile. The log file
```

Listing 28-2 log_keystrokes.ksh shell script

```
is both kept locally and emailed to a log file
        administrative user either locally or on a remote machine.
# REV LIST:
# set -n # Uncomment to check syntax without any execution
# set -x # Uncomment to debug this shell script
# This user receives all of the audit logs by email. This
# Log Manager can have a local or remote email address. You
# can add more than one email address if you want by separating
# each address with a space.
LOG_MANAGER="logman"
                  # List to email audit log
**********
*****
cleanup_exit ()
# This function is executed on any type of exit except of course
# a kill -9, which cannot be trapped. The script log file is
# emailed either locally or remotely, and the log file is
# compressed. The last "exit" is needed so the user does not
# have the ability to get to the command line without logging.
if [[ -s ${LOGDIR}/${LOGFILE} ]]
then
   mailx -s "$TS - $LOGNAME Audit Report" $LOG_MANAGER < ${LOGDIR}/
${LOGFILE}
   compress ${LOGDIR}/${LOGFILE} 2>/dev/null
fi
exit
}
# Set a trap
trap 'cleanup_exit'1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 26
******
```

Listing 28-2 (continued)

```
TS=$(date +%m%d%v%H%M%S)
                            # File time stamp
THISHOST=$(hostname|cut -f1-2 -d.) # Host name of this machine
LOGFILE=${THISHOST}.${LOGNAME}.$TS # Creates the name of the log file
touch $LOGDIR/$LOGFILE  # Creates the actual file
set -o vi 2>/dev/null
                           # Previous commands recall
stty erase ^?
                           # Set the backspace key
# Set the command prompt
export PS1="[$LOGNAME:$THISHOST]@"'$PWD> '
chmod 600 ${LOGDIR}/${LOGFILE} # Change permission to RW for the owner
script ${LOGDIR}/${LOGFILE} # Start the script monitoring session
chmod 400 ${LOGDIR}/${LOGFILE} # Set permission to read-only for
                           # the owner
cleanup_exit
                           # Execute the cleanup and exit function
```

Listing 28-2 (continued)

The log_keystrokes.ksh script in Listing 28-2 is not difficult when you take a close look. At the top we define the cleanup_exit function that is used when the script exits to email and compress the log file. In the next section we set a trap and define and set some variables. Finally, we start the logging activity with a script session.

In the cleanup_exit function notice the list of exit codes that the trap command will exit on. This signal list ensures that the log file gets emailed and the file gets compressed. The only exit signal we cannot do anything about is a kill -9 signal because you cannot trap kill -9, because this particular kill signal tells the system to remove the process from the process table, *without* any clean-up! There are more exit signals if you want to add to the list in the trap statement, but I think the most captured are listed in the trap statement.

The last command executed in this shell script is exit because in every case the cleanup_exit function must execute. If exit is not the last command, the user will be placed back to a command prompt without any logging being done. The reason for this behavior is that the script session is really a fork of the original shell. Therefore, when the script command stops executing, one of the shells in the fork terminates, but not the original shell. This last exit logs out of the original shell. You may want to replace this last exit, located in the cleanup_exit function, with logout, which will guarantee the user is logged out of the system.

Logging root Activity

In some shops there is a need to log the activity of the root user. If you log root access activity, you have an audit trail and it is much easier to do root cause analysis

by analyzing the audit logs. You will also be ahead of the game if you are subject to Sarbanes-Oxley (SOX) audits. SOX audits are required only for United States publicly traded companies. We can use the same type of shell that we used in the previous sections, but this time we will use sudo instead of a .profile entry. I call this script broot because it is a short name for "I want to *be* root." In this section let's look at the shell script in Listing 28-3 and go through the details at the end.

```
#!/bin/ksh
# SCRIPT: broot
# AUTHOR: Randy Michael
# DATE: 05/08/2007
# REV: 1.0.P
# PLATFORM: Any Unix
#
# PURPOSE: This shell script is used to monitor all root access by
        capturing all of the terminal data in a log file using
#
        the script command. This shell script is executed from the
#
        command line using sudo (Super User Do). The log file
         is kept locally and emailed to a log file administrative
        user either locally or on a remote machine. Sudo must be
        configured for this shell script. Refer to your sudo notes.
# USAGE: sudo broot
# REV LIST:
# set -n # Uncomment to check syntax without any execution
# set -x # Uncomment to debug this shell script
# This user receives all of the audit logs by email. This
# Log Manager can have a local or remote email address. You
# can add more than one email address if you want by separating
# each address with a space.
LOG_MANAGER="logman"
                   # List to email audit log
************************
******
cleanup_exit ()
# This function is executed on any type of exit except of course
# a kill -9, which cannot be trapped. The script log file is
```

Listing 28-3 broot shell script listing

```
# emailed either locally or remotely, and the log file is
# compressed. The last "exit" is needed so the user does not
# have the ability to get to the command line without logging.
if [[ -s ${LOGDIR}/${LOGFILE} ]]
then
   mailx -s "$TS - $LOGNAME Audit Report" $LOG_MANAGER < ${LOGDIR}/
${LOGFILE}
   nohup compress ${LOGDIR}/${LOGFILE} 2>/dev/null &
fi
exit
}
# Set a trap
trap 'cleanup_exit'1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 26
TS=$(date +%m%d%y%H%M%S)
                             # File time stamp
THISHOST=$(hostname)
                             # Host name of this machine
LOGDIR=/usr/local/logs/script  # Directory to hold the logs
LOGFILE=${THISHOST}.${LOGNAME}.$TS # Creates the name of the log file
touch $LOGDIR/$LOGFILE
                              # Creates the actual file
TMOUT=300
                              # Set the root shell timeout!!!
export TMOUT
                              # Export the TMOUT variable
set -o vi
                              # To recall previous commands
stty erase _
                              # Set the backspace key
# Run root's .profile if one exists
if [[ -f $HOME/.profile ]]
then
    . $HOME/.profile
fi
# set path to include /usr/local/bin
echo $PATH|grep -q ':/usr/local/bin' || PATH=$PATH:/usr/local/bin
# Set the command prompt to override the /.profile default prompt
PS1="$THISHOST:broot> "
export PS1
```

Listing 28-3 (continued)

Listing 28-3 (continued)

There is one extremely important difference between this script and the script in Listing 28-2. In the broot script in Listing 28-3 we execute the .profile for root, if there is a .profile for root. You may ask why we did not execute the profile last time. The answer involves the recursive nature of running a file onto itself. In the previous case we had the following entry in the \$HOME/.profile file:

. /usr/local/bin/log_keystrokes.ksh

We add this entry beginning with a "dot," which means to execute the following file as the last entry in the \$HOME/.profile. If you add execution of \$HOME/.profile into the *shell script* you end up executing the log_keystrokes.ksh shell script recursively. When you run the script like this you fill up the buffers and you get an error message similar to the following output:

ksh: .: 0403-059 There cannot be more than 9 levels of recursion.

For monitoring root access with the broot script we are not executing from the .profile, but we use sudo to run this broot script, so we have no worries about recursion. At the top of the script in Listing 28-3 we define a LOG_MANAGER. This list of one or more email addresses is where the log files are going to be emailed. You may even want real-time notification of root activity. I like to send the log files off to my audit box for safekeeping using my logman user account.

The next step is to set a trap. If the script exits on signals 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 26, the cleanup_exit function is executed. This trap ensures that the log file gets emailed and the file gets compressed locally. In the next section we define and set the variables that we use. Notice that we added a shell timeout, specified by the TMOUT environment variable. If someone with root access is not typing for five minutes, the shell times out. You can set the TMOUT variable to anything you want or even comment it out if you do not want a shell timeout. To disable TMOUT, assign 0 to the variable, TMOUT=0. The measurement is in seconds. The default is 300 seconds, or 5 minutes, for this script.

After the variable definitions we execute the root .profile. We run the profile here because we are not running the broot script from a login \$HOME/.profile, as we did

with the log_keystrokes.ksh script in Listing 28-2. Next we add /usr/local/bin to root's \$PATH, if it is not already present. And, finally, before we are ready to execute the script command we set a command prompt.

The final four things we do are (1) set the permissions on the log file so we can write to it; (2) run the script command using the log filename as a parameter; (3) set the file permissions on the log file to read-only; and (4) execute the cleanup_exit function to email the log and compress the file locally.

Some sudo Stuff

I have inserted a short /etc/sudoers file for Listing 28-4 to show entries that need to be made. The entire task of setting up and using sudo is shown in Chapter 14, "Number Base Conversions." Pay attention to the bold type in Listing 28-4.

```
# sudoers file.
 # This file MUST be edited with the 'visudo' command as root.
 #
 # See the sudoers man page for the details on how to write a
 # sudoers file.
 #
# Users Identification:
# All access:
# randy - Randy Michael
 # terry - Admin
 # Restricted Access to: mount umount and exportfs
 # oracle - Oracle Admin
 # operator - operator access
 # Host alias specification
Host_Alias LOCAL=yogi
# User alias specification
User_AliasROOTADMIN=randy,terryUser_AliasNORMAL=randy,operator,terryUser_AliasADMIN=randy,terryUser_AliasORACLE=oracleUser_AliasDB2=db2admUser_AliasOPERATOR=operator
```

```
Listing 28-4 Example /etc/sudoers file
```

```
# Runas alias specification
Runas_Alias ORA=oracle
# Cmnd alias specification
Cmnd_Alias BROOT=/usr/local/bin/broot
Cmnd_Alias MNT=/usr/bin/mount
Cmnd_Alias UMNT=/usr/bin/umount
Cmnd_Alias EXP_FS=/usr/bin/exportfs
Cmnd_Alias KILL=/usr/bin/kill
Cmnd_Alias ORACLE_SU=/usr/bin/su - oracle
Cmnd Alias TCPDUMP=/usr/sbin/tcpdump
Cmnd_Alias ERRPT=/usr/bin/errpt
Cmnd_Alias SVRMGRL=/oracle/product/8.0.5/bin/svrmgrl
# User privilege specification
root ALL=(ALL) ALL
ROOTADMIN LOCAL=BROOT
NORMAL LOCAL=MNT, UMNT, EXP_FS
ADMIN
    LOCAL=BROOT, MNT, UMNT, KILL, ORACLE_SU, TCPDUMP, ERRPT: \
    LOCAL=EXP FS
ORACLE LOCAL=SVRMGRL
# Override Defaults
Defaults logfile=/var/adm/sudo.log
```

Listing 28-4 (continued)

Three entries need to be added to the /etc/sudoers file. Do *not* ever edit the sudoers file directly with **vi**. There is a special program called **visudo**, in the /usr/local/sbin directory, that has a wrapper around the vi editor that does a thorough check for mistakes in the file before the file is saved. If you make a mistake the visudo program will tell you where the error is located in the /etc/sudoers file.

The three entries that need to be added to /etc/sudoers are listed next and are highlighted in bold text in Listing 28-4.

Define the User_Alias, which is where we give a name to a group of users. For this file let's name the list of users who can get root access ROOTADMIN, as shown here:

User_Alias ROOTADMIN=randy,terry

Next we need to define the Cmnd_Alias, which is where we define the full pathname to the command, as shown here:

Cmnd_Alias BROOT=/usr/local/bin/broot

The last step is to define the exact commands that the User_Alias group of users can execute. In our case we have a separate User_Alias group only for the users who can use the broot script. Notice that the definition also specifies the machine where the command can be executed. I always let sudo execution take place only on a single machine at a time, specified by LOCAL here:

ROOTADMIN LOCAL=BROOT

Once the /etc/sudoers file is set up, you can change the root password and allow root access only by using the broot script. Using this method you have an audit trail of root access to the system.

Monitoring Other Administration Users

More often than not, you will want add to the list of auditing that can be done. This next script is rewritten to allow you to quickly set up a broot type shell script by changing only the username and the script name. The method that we use to execute the script command is what makes this script different — and easy to modify.

For ease of use we can use a lot of variables throughout the script. We have already been doing this to some extent. Now we will call the monitored user the *effective user*, which fits our new variable *SEFF_USER*. For this script, I have set the effective username to oracle. You can make it any user that you want to. Take a look at this shell script in Listing 28-5, and pay particular attention to the boldface type.

```
#!/bin/ksh
#
# SCRIPT: "Banybody" boracle - This time
# AUTHOR: Randy Michael
# DATE: 05/08/2007
# REV: 1.0.P
# PLATFORM: Any Unix
#
# PURPOSE: This shell script is used to capture all "$EFF_USER"
#
         access by capturing all of the terminal data in a log
#
          file using the script command. This shell script is
          executed from the command line using sudo (Super User Do).
#
#
          The log file is kept locally and emailed to a log file
          administrative user either locally or on a remote
#
          machine. Sudo must be configured for this shell script.
         Refer to your sudo notes. The effective user, currently
#
          oracle, can be changed by setting the "EFF USER" variable
#
          to another user, and changing the name of the script.
#
          This is why the original name of the script is called
#
          "Banybody".
# ORIGINAL USAGE: sudo Banybody
```

Listing 28-5 boracle shell script

```
# THIS TIME USAGE ==> USAGE: sudo boracle
# REV LIST:
         5/10/2007: Modified the script to replace the hard-coded
#
                  username with the variable $EFF_USER. This
#
                  allows flexibility to add auditing of more
#
                  accounts by just changing the EFF_USER variable
                  and the script name.
# set -n # Uncomment to check syntax without any execution
# set -x # Uncomment to debug this shell script
# This EFF_USER is the username you want to be to execute
# a shell in. An su command is used to switch to this user.
EFF_USER=oracle
# This user receives all of the audit logs by email. This
# Log Manager can have a local or remote email address. You
# can add more than one email address if you want by separating
# each address with a space.
LOG_SERVER=yogi
LOG_MANAGER="logman@$LOG_SERVER"
                              # List to email audit log
******
cleanup_exit ()
# This function is executed on any type of exit except of course
# a kill -9, which cannot be trapped. The script log file is
# emailed either locally or remotely, and the log file is
# compressed. The last "exit" is needed so that the user does not
# have the ability to get to the command line without logging.
if [[ -s ${LOGDIR}/${LOGFILE} ]] # Is it greater than zero bytes?
then
   mailx -s "$TS - $LOGNAME Audit Report" $LOG_MANAGER < ${LOGDIR}/
${LOGFILE}
```

Listing 28-5 (continued)

```
}
trap 'cleanup_exit' 1 2 3 5 15
******
***********
TS=$(date +%m%d%y%H%M%S)
                            # File time stamp
                            # Hostname of this machine
THISHOST=$(hostname)
LOGFILE=${THISHOST}.${EFF_USER}.$TS # Creates the name of the log file
touch $LOGDIR/$LOGFILE
                             # Creates the actual file
TMOUT=300
                              # Set the root shell timeout!!!
export TMOUT
                              # Export the TMOUT variable
set -o vi
                              # To recall previous commands
stty erase ^?
                              # Set the backspace key
# set path to include /usr/local/bin
echo $PATH|grep -q ':/usr/local/bin' || PATH=$PATH:/usr/local/bin
# Set the command prompt to override the /.profile default prompt
PS1="$THISHOST:b${EFF_USER}> "
export PS1
chmod 666 ${LOGDIR}/${LOGFILE}
                           # Set permission to read/write
# To get the script session to work we have to use the switch user (su)
# command with the -c flag, which means execute what follows. Sudo is
# also used just to ensure that root is executing the su command.
# We ARE executing now as root because this script was started with
# sudo. If a nonconfigured sudo user tries to execute this command
# then it will fail unless sudo was used to execute this script as root.
# Notice we are executing the script command as "$EFF USER". This
# variable is set at the top of the script. A value such as
# "EFF_USER=oracle" is expected.
sudo su - $EFF_USER -c "script ${LOGDIR}/${LOGFILE}"
chmod 400 ${LOGDIR}/${LOGFILE} # Set permission to read-only for
                          # the owner
cleanup_exit
                          # Execute the cleanup and exit function
```

Listing 28-5 (continued)

The most important line to study in Listing 28-5 is the third line from the bottom:

sudo su - \$EFF_USER -c "script \${LOGDIR}/\${LOGFILE}"

There are several points to make about this command. Notice that we start the command with sudo. Because you must use sudo to execute the boracle script, and you are already executing as root, why use sudo here? We use sudo here to ensure that the boracle script was indeed started with sudo. If any old user runs the boracle command we want it to fail if sudo was not used.

The second command in the previous statement is su - \$EFF_USER. The significance of the hyphen, -, is important here. Using the hyphen, -, with a space on both sides tells the su command to switch to the user pointed to by \$EFF_USER, oracle in our case, and run that user's .profile. If the hyphen is omitted or the spaces are not around the hyphen, the user .profile is not executed, which is a bad thing in this case.

The last part of this command is where we start our script session. When you switch users with su, you can specify that you want to run a command as this user by adding the -c switch followed by the command enclosed in single or double quotes. Do not forget the quotes around the command.

The only other real change is the use of the EFF_USER variable. This variable is set at the top of the script, and changing this variable changes who you want to "be." If you want to create more admin auditing scripts, copy the boracle file to a new filename and edit the file to change the name at the top of the script and modify the EFF_USER variable. That's it!

Other Options to Consider

Throughout this chapter we have covered some interesting concepts. You may have quite a few things that you want to add to these scripts. I have come up with a few myself.

Emailing the Audit Logs

Depending on the extent of monitoring and auditing you need to do, you may want to send the files to several different machines. I selected using email for the transport, but you may have some other techniques, such as automated FTP. You may also want to compress the files before you email, or whatever, the log files. To email a compressed file, you will need some type of mail tool, such as metamail's metasend, written by Nathaniel S. Brenstein (ftp://ftp.research.telcordia.com/pub/nsb/), or the open source mutt utility, written by Michael Elkins (http://www.mutt.org). This is needed sometimes because the mail reader will think that some of the characters in the log file are control characters to mail commands. This can cause some strange things to happen. You can download metamail from ftp://ftp.research.telcordia.com/pub/nsb/mm2.7.tar.Z.

The syntax to use metasend to send a binary file, such as a compressed log file, is shown here:

```
metasend -b -t $LOG_MANAGER -s "Audit log" \
    -m application/postscript -f ${LOGDIR}/${LOGFILE}.Z
```

Metamail requires that you specify the file type, or MIME type, so the mail reader on the recipient side knows what kind of file it is dealing with. MIME types include "text/plain," "image/gif," "application/postscript," and "audio/wav."

Mutt is an interactive program but we can supply input to the body of the email the same way we do with the mail command. The syntax to use mutt to send an attached file is shown here:

```
mutt -a ${LOGDIR}/${LOGFILE}.Z $LOG_MANAGER < /dev/null</pre>
```

Notice that we sent the email with a null body, specified by < /dev/null. So the only thing the email contains is the attached file. Of course, we need to modify our shell scripts to compress the log file before we email it out.

One last thing: watch the disk space! When you start logging user activity you need to keep a close check on disk space, both on the local server and the \$LOG_SERVER. Most systems store email in /var. If you fill up /var for an extended period of time you may crash the box. For my log files I create a large dedicated filesystem called /usr/local/logs. With a separate filesystem I do not have to worry about crashing the system if I fill up the filesystem. You can probably think of other methods to move the files around as the emails are received. If you do create a logging filesystem you need to override the MAIL variable in the /etc/profile for the logman user by adding this declaration to the logman's \$HOME/.profile:

MAIL=/usr/local/logs/\$LOGNAME

NOTE Heiner Steven wrote an excellent tutorial on sending files as email attachments. You can find his discussion at http://shelldorado.com/articles/mailattachments.html.

Compression

For all of these scripts we used the compress command. This compression algorithm is okay, but we can do better. I find that gzip has a much better compression algorithm, and the compression ratio is tunable for your needs. The tuning is done using numbers as a parameter to the gzip command, as shown here:

gzip -9 \$LOGFILE

The valid numbers are 1 to 9, with 9 indicating the highest compression. This extra compression does come at a price — time! The higher the number, the longer it takes to compress the file. By omitting the number you use gzip in default mode, which is -5. For our needs you will still see a big increase in compression over compress in about the same amount of time.

Need Better Security?

Another option for this keystroke auditing is to use Open Secure Shell and keep a real-time encrypted connection to the log server by creating a *named pipe*. This can be done but it, too, has some potential problems. This first major problem is that you introduce a dependency for the logging to work. If the connection is lost, the script session ends. For auditing root activities, and especially when all other root access has been disabled, you can have a real nightmare. I will leave this idea for you to play around with because it is beyond the scope of this book.

Inform the Users

I did not add this chapter to the book for everyone to start secretly monitoring everyone's keystrokes. Always be up front with the user community, and let them know that an audit is taking place. I know for a fact that Systems Administrators do not like to have the root password taken away from them. I know first hand about the reaction.

If you are going to change the user password, please place the root password in a safe place where, in case of emergency, you can get to the password without delay. Your group will have to work out how this is accomplished.

Sudoers File

If you start running these scripts and you have a problem, first check your sudo configuration by looking at the /etc/sudoers file. There are some things to look for that the visudo editor will not catch:

- Check the LOCAL line. This variable should have the hostname of your machine assigned.
- Check for exact pathnames of the files.
- Ensure that the correct users are assigned to the correct commands.

The visudo editor does catch most errors, but there are some things that are not so easy to test for.

Summary

I had a lot of fun writing this chapter and playing with these scripts. I hope you take these auditing scripts and use them in a constructive way. The information gathered can be immense if you do not have a mechanism for pruning the old log files. The following command works pretty well:

```
find /log_directory -mtime +30 -print -exec rm {} \;
```

This command will remove all the files in /log_directory that have not been modified in 30 days. You may want to add a -name parameter to limit what you delete. As with any type of monitoring activity that creates logs, you need to watch

the filesystem space very closely, especially at first, to see how quickly logs are being created and how large the log files grow.

Another topic that comes up a lot is the shell timeout. The only place I use the TMOUT environment variable is in the broot script. If you add a shell timeout to your other administrative accounts you may find that a logout happens during a long processing job. With these users I expect them to just lock the terminal when they leave.

Lab Assignments

- 1. Modify all three shell scripts to utilize mutt to email compressed log files as attachments.
- 2. Modify all three shell scripts to utilize metamail's metasend to email compressed log files as attachments.

A Closing Note from the Author

I sure hope that you enjoyed this chapter, and the whole book. The process of writing this book has been a thrill for me. Every time I started a new chapter, I had a firm idea of what I wanted to accomplish, but usually along the way I got these little brainstorms that helped me build on the basic idea that I started with. Some five-page chapters turned into some of the longest chapters in the book. In every case, though, I always tried to hit the scripting techniques from a different angle. Sometimes this resulted in a long script or roundabout way of accomplishing the task. I really did do this on purpose. There is always more than one way to solve a challenge in UNIX, and I always aimed to make each chapter different and interesting. I appreciate that you bought this book, and in return I hope I have given you valuable knowledge, and insight into solving any problem that comes along. Now you can really say that the solution to any challenge is *intuitively obvious*! Thank you for reading, and best regards.

APPENDIX **A**

What's on the Web Site

This appendix shows a list of the shell scripts and functions that are included on this book's companion web site, www.wiley.com/go/michael2e. Each of the shell scripts and functions has a brief description of the purpose.

Shell Scripts

Chapter 1

script.stub

This file has the basic framework to begin writing a shell script.

keyit.rsa

This script is used to set up RSA OpenSSH keys for no-password access to a remote machine. This script must be executed by the user who needs the keys set up.

keyit.dsa

This script is used to set up DSA OpenSSH keys for no-password access to a remote machine. This script must be executed by the user who needs the keys set up.

generic_rsync.Bash

This is a generic shell script to copy files using rsync.

```
select_system_info_menu.Bash
```

This shell script uses a select statement to create menu options for general system information.

test_string.ksh

This script is used to test and display the composition of a character string based on common criteria.

Chapter 2

```
24_ways_to_parse.ksh
```

This script shows the different ways of reading a file line-by-line. Again, there is not just one way to read a file line-by-line and some are faster than others and some are more intuitive than others.

random_file.Bash

This script is used to create a random-character text file that is the size, in MB, specified on the command line.

Chapter 3

There are no shell scripts to list in Chapter 3.

Chapter 4

timing_test_function.Bash

This script is used to test the elapsed_time function.

Chapter 5

parse_record_files.Bash

This shell script is used to merge and parse fixed- and variable-length record files.

Chapter 6

tst_ftp.ksh

Simple FTP automated file-transfer test script.

get_remote_dir_listing.ksh

Script to get a remote directory listing using FTP.

get_ftp_files.ksh

Shell script to retrieve files from a remote machine using FTP.

put_ftp_files.ksh

Shell script to upload files to a remote machine using FTP.

get_remote_dir_listing_pw_var.ksh

Script to get a directory listing from a remote machine using FTP. The passwords are stored in an environment file somewhere on the system, defined in the script.

```
sftp-scp-getfile.Bash
```

This shell script is a simple example of using scp and sftp without passwords.

```
ftp-ls-getfile.exp
```

This Expect script is used to do a long listing on a remote machine and FTP a file to the local server.

get_ftp_files_pw_var.ksh

Script to retrieve files from a remote machine using FTP. This script gets its password from an environment file somewhere on the system, defined in the script.

put_ftp_files_pw_var.ksh

Script to upload files to a remote machine using FTP. This script gets its password from an environment file somewhere on the system, defined in the script.

Chapter 7

```
generic_rsync.Bash
```

This is a generic shell script used to copy files with rsync.

generic_DIR_rsync_copy.Bash

This is a generic shell script used to replicate directory structures to one or more remote machines.

generic_FS_rsync_copy.Bash

This is a generic shell script used to replicate one or more filesystems to one or more remote machines.

```
rsync_daily_copy.ksh
```

This shell script is used to replicate an Oracle database to two remote machines.

ftp-get-file.exp

This Expect script is used to transfer a remote file to the local system using FTP.

ftp-get-file-cmd-line.exp

This Expect script is used to get files from a remote system by specifying the filenames on the command line.

etc-hosts-copy.exp

This Expect script is used to copy a master /etc/hosts file to a list of machines.

showplatform.exp

This Expect script is used with Sun Blade Servers to execute the Sun Blade Chassis command showplatform -v.

findslot

This shell script utilizes the showplatform.exp Expect script to locate the slot a Sun Blade server is installed in.

boot-net-install.exp

This Expect script is used with Sun Blades and Sun's JumpStart to execute the command: boot net - install at the "ok" prompt.

Chapter 9

findlarge.ksh

This shell script is used to find "large" files. The file-size limit is supplied on the command line and the search begins in the current directory.

Chapter 10

proc_mon.ksh

Process monitor that informs the user when the process ends.

proc_wait.ksh

Process monitor that informs the user when the process starts.

proc_watch.ksh

Process monitor that monitors a process as it starts and stops.

proc_watch_timed.ksh

Process monitor that monitors a process for a user-specified amount of time.

Chapter 11

random_number_testing.Bash

This shell script is a test of the /dev/urandom character special file used to create random numbers.

random_number.ksh

This shell script is used to create pseudo-random numbers.

mk_unique_filename.ksh

This shell script creates unique filenames.

random_file.Bash

This shell script is used to create a user-defined MB-size file filed with random characters using the /dev/random character special file to create the random numbers that point to array elements.

Chapter 12

mk_passwd.Bash

This shell script is used to create pseudo-random passwords.

Chapter 13

float_add.ksh

Adds a series of floating-point numbers together using the bc utility.

float_subtract.ksh

Subtracts floating-point numbers using the bc utility.

float_multiply.ksh

Multiplies a series of floating-point numbers together using the bc utility.

float_divide.ksh

Divides two floating-point numbers using the bc utility.

float_average.ksh

Averages a series of floating-point numbers using the bc utility.

chg_base.ksh

This shell script converts numbers between bases 2 and 36.

chg_base_bc.Bash

This shell script converts numbers between bases 2 and 16 using the bc utility.

mk_swkey.ksh

Shell script to create a software license key using the hexadecimal representation of the IP address.

equate_any_base.ksh

Converts numbers between any base.

equate_base_2_to_16.ksh

Converts numbers from base 2 to base 16.

equate_base_16_to_2.ksh

Converts numbers from base 16 to base 2.

equate_base_10_to_16.ksh

Converts numbers from base 10 to base 16.

equate_base_16_to_10.ksh

Converts numbers from base 16 to base 10.

equate_base_10_to_2.ksh

Converts numbers from base 10 to base 2.

equate_base_2_to_10.ksh

Converts numbers from base 2 to base 10.

equate_base_10_to_8.ksh

Converts numbers from base 10 to base 8.

equate_base_8_to_10.ksh

Converts numbers from base 8 to base 10.

hgrep.ksh

This shell script works similar to grep except that it shows the entire file with the pattern match highlighted in reverse video.

Chapter 16

keyit

This shell script is used to set up OpenSSH encryption keys for password-free access.

proc_mon.ksh

Process monitor that informs the user when the process ends.

proc_wait.ksh

Process monitor that informs the user when the process starts.

proc_watch.ksh

Process monitor that monitors a process as it starts and stops.

proc_watch_timed.ksh

Process monitor that monitors a process for a user-specified amount of time.

Chapter 17

fs_mon_AIX.ksh

This shell script is used to monitor an AIX system for full filesystems using the percentage method.

fs_mon_AIX_MBFREE.ksh

This shell script is used to monitor an AIX system for full filesystems using the MB-free method.

fs_mon_AIX_MBFREE_excep.ksh

This shell script is used to monitor an AIX system for full filesystems using the MB-free method with exceptions capability.

fs_mon_AIX_PC_MBFREE.ksh

This shell script is used to monitor an AIX system for full filesystems using the percentage method with exceptions capability.

fs_mon_AIX_excep.ksh

Basic AIX filesystem monitoring using the percent method with exceptions capability.

fs_mon_ALL_OS.ksh

This shell script autodetects the UNIX flavor and monitors the filesystems using both percent and MB-free techniques with an autodetection to switch between methods.

fs_mon_HPUX.ksh

This shell script is used to monitor an HP-UX system for full filesystems using the percentage method.

fs_mon_HPUX_MBFREE.ksh

This shell script is used to monitor an HP-UX system for full filesystems using the MB-free method.

fs_mon_HPUX_MBFREE_excep.ksh

This shell script is used to monitor an HP-UX system for full filesystems using the percentage method with exceptions capability.

fs_mon_HPUX_PC_MBFREE.ksh

This shell script is used to monitor an HP-UX system for full filesystems using the percentage method with exceptions capability.

fs_mon_HPUX_excep.ksh

Basic HP-UX filesystem monitoring using the percent method with exceptions capability.

fs_mon_LINUX.ksh

This shell script is used to monitor a Linux system for full filesystems using the percentage method.

fs_mon_LINUX_MBFREE.ksh

This shell script is used to monitor a Linux system for full filesystems using the MB-free method.

fs_mon_LINUX_MBFREE_excep.ksh

This shell script is used to monitor a Linux system for full filesystems using the percentage method with exceptions capability.

fs_mon_LINUX_PC_MBFREE.ksh

This shell script is used to monitor a Linux system for full filesystems using the percentage method with exceptions capability.

fs_mon_LINUX_excep.ksh

Basic Linux filesystem monitoring using the percentage method with exceptions capability.

fs_mon_SUNOS.ksh

This shell script is used to monitor a SunOS system for full filesystems using the percentage method.

fs_mon_SUNOS_MBFREE.ksh

This shell script is used to monitor a SunOS system for full filesystems using the MB-free method.

fs_mon_SUNOS_MBFREE_excep.ksh

This shell script is used to monitor a SunOS system for full filesystems using the percentage method with exceptions capability.

fs_mon_SUNOS_PC_MBFREE.ksh

This shell script is used to monitor a SunOS system for full filesystems using the percentage method with exceptions capability.

fs_mon_SUNOS_excep.ksh

Basic SunOS filesystem monitoring using the percent method with exceptions capability.

Chapter 18

AIX_paging_mon.ksh

Shell script to monitor AIX paging space.

HP-UX_swap_mon.ksh

Shell script to monitor HP-UX swap space.

Linux_swap_mon.ksh

Shell script to monitor Linux swap space.

SUN_swap_mon.ksh

Shell script to monitor SunOS swap space.

all-in-one_swapmon.ksh

Shell script to monitor AIX, HP-UX, Linux, and SunOS swap/paging space.

uptime_loadmon.ksh

System load monitor using the uptime command.

uptime_fieldtest.ksh

Script to test the location of the latest uptime load information as it changes based on time.

sar_loadmon.ksh

Load monitor using the sar command.

iostat_loadmon.ksh

Load monitor using the iostat command.

vmstat_loadmon.ksh

Load monitor using the vmstat command.

Chapter 20

stale_LV_mon.ksh

This shell script is used to monitor AIX stale Logical Volumes.

stale_PP_mon.ksh

This shell script is used to monitor AIX stale Physical Partitions.

stale_VG_PV_LV_PP_mon.ksh

This shell script is used to monitor AIX stale partitions in Volume Groups, Physical Volumes, Logical Volumes, and Physical Partitions.

Chapter 21

SSAidentify.ksh

Shell script to control SSA disk subsystem disk-identification lights.

Chapter 22

pingnodes.ksh

This shell script is used to ping nodes. The operating system can be AIX, HP-UX, Linux, or SunOS.

AIXsysconfig.ksh

This shell script is used to gather information about an AIX system's configuration.

Chapter 24

chpwd_menu.ksh

This shell script uses sudo to allow support personnel to change passwords.

Chapter 25

enable_AIX_classic.ksh

Enables all AIX "classic" print queues.

print_UP_AIX.ksh

Enables all AIX System V printers and queues.

print_UP_HP-UX.ksh

Enables all HP-UX System V printers and queues.

print_UP_Linux.ksh

Enables all Linux System V printers and queues.

printing_only_UP_Linux.ksh

Enables printing on Linux System V printers.

queuing_only_UP_Linux.ksh

Enables queuing on Linux System V printers.

print_UP_SUN.ksh

Enables all SunOS System V printers and queues.

PQ_all_in_one.ksh

Enables all printing and queuing on AIX, HP-UX, Linux, and SunOS by autodetecting the UNIX flavor.

Chapter 26

search_group_id.Bash

This shell script is used to parse the /etc/passwd file and use the id command to display the group name, as opposed to the GID.

chk_passwd_gid_0.Bash

This shell script searches the /etc/passwd file for users who are a member of group 0 (system/root group).

Chapter 27

```
dirvish_ctrl
```

This shell script is a menu interface to Dirvish snapshot backups.

Chapter 28

broot

Shell script to capture keystrokes of anyone gaining root access.

banybody

Shell script to capture keystrokes of any user defined in the shell script.

log_keystrokes.ksh

Shell script to log users' keystrokes as they type on the keyboard.

Functions

Chapter 1

rotate_line

This function is a progress indicator that appears as a twirling line.

elapsed_time

This function converts seconds into hours, minutes, and seconds format.

ping_host

This function detects the UNIX flavor, and then pings the host specified by the function's \$1 argument, using the correct syntax for that system.

Chapter 2

build_random_line

This function is uses a random number to pick an array element.

load_default_keyboard

This function loads a default keyboard layout into an array. All of the following 24 functions are different methods to process a file line-by-line:

```
cat_while_read_LINE
while_read_LINE_bottom
cat_while_LINE_line
while_LINE_line_bottom
cat_while_LINE_line_cmdsub2
while_LINE_line_bottom_cmdsub2
for_LINE_cat_FILE
for_LINE_cat_FILE_cmdsub2
while_line_outfile
while_read_LINE_FD_IN
cat_while_read_LINE_FD_OUT
while_read_LINE_bottom_FD_OUT
while_LINE_line_bottom_FD_OUT
while_LINE_line_bottom_cmdsub2_FD_OUT
for_LINE_cat_FILE_FD_OUT
for_LINE_cat_FILE_cmdsub2_FD_OUT
while_line_outfile_FD_IN
while_line_outfile_FD_OUT
while_line_outfile_FD_IN_AND_OUT
while_LINE_line_FD_IN
while_LINE_line_cmdsub2_FD_IN
while_read_LINE_FD_IN_AND_OUT
while_LINE_line_FD_IN_AND_OUT
while_LINE_line_cmdsub2_FD_IN_AND_OUT
```

Chapter 3

There are no functions to list in Chapter 3.

Chapter 4

dots

This function is used as a progress indicator showing a series of dots every 10 seconds or so.

rotate

This function is used as a progress indicator showing the appearance of a rotating line.

elapsed_time

This function converts seconds to hours, minutes, and seconds format.

parse_fixed_length_records

This function parses through a fixed-length record file using the cut command.

```
parse_variable_length_records
```

This function parses through a variable-length record file using the cut command.

merge_fixed_length_records

This function appends the filename of the record file to the end of each record in the file.

merge_variable_length_records

This function appends the filename of the record file to the end of each record in the file.

Chapter 6

pre_event

Function to allow for pre events before processing.

post_event

Function to allow for post events after processing.

Chapter 7

```
elapsed_time
```

This function converts seconds to hours, minutes, and seconds format.

```
verify_copy
```

This function verifies that that file sizes match between the local machine and one or more remote machine(s).

ready_to_run

This function starts at the beginning of the shell script and loops until the \$READY_TO_RUN file is found on the system.

Chapter 8

increment_by_1

This Expect proc (function) increments a number by 1.

There are no functions to list in Chapter 9.

Chapter 10

mon_proc_end

This function loops until the monitored process ends execution.

mon_proc_start

This function loops until the monitored process starts execution.

pre_event_script

This function executes as a pre event before processing starts.

startup_event_script

This function executes as a startup event.

post_event_script

This function executes after processing completes.

test_string

This function is used to test text strings.

proc_watch

This function watches as a monitored process starts and stops.

Chapter 11

get_random_number

This function produces a pseudo-random number between 1 and 32,767.

in_range_random_number

Creates a pseudo-random number less than or equal to the *\$UPPER_LIMIT* value, which is user-defined.

in_range_fixed_length_random_number_typeset

This function creates a right-justified fixed-length pseudo-random number using the typeset command.

get_random_number

This function creates a pseudo-random number using the shell's RANDOM variable.

get_date_time_stamp

This function displays a date/time stamp.

get_second

This function displays the current clock second.

my_program

This function is where the program to execute is defined.

build_random_line

This function creates a line of random text characters.

elapsed_time

This function converts seconds to hours, minutes, and seconds format.

load_default_keyboard

This function loads the KEYS array with a default U.S. QWERTY keyboard layout.

Chapter 12

in_range_random_number

This function creates pseudo-random numbers within one and a "max value."

load_default_keyboard

This function is used to load a U.S. 102-key board layout into a keyboard file.

check_for_and_create_keyboard_file

If the \$KEYBOARD_FILE does not exist, asks the user to load the "standard" keyboard layout, which is done with the load_default_keyboard function.

build_manager_password_report

Builds a file to print for the secure envelope.

Chapter 13

There are no functions to list in Chapter 13.

Chapter 14

There are no functions to list in Chapter 14.

There are no functions to list in Chapter 15.

Chapter 16

check_HTTP_server

This function is used to check an application web server and application URL pages.

Chapter 17

load_EXCEPTIONS_data

This function loads the data in the \$EXCEPTIONS file while ignoring the lines commented out with a hash mark (#).

load_FS_data

This function loads the current filesystem statistics for processing.

check_exceptions

This function tests if a filesystem is out of limits.

display_output

This function displays the *\$OUTFILE*.

get_OS_info

This function queries the system for the UNIX flavor.

???Author: Are descriptions to come for these?load_AIX_FS_data

This function loads AIX filesystem data.

load_HP_UX_FS_data

This function loads HP-UX filesystem data.

load_LINUX_FS_data

This function loads Linux filesystem data.

load_OpenBSD_FS_data

This function loads OpenBSD filesystem data.

load_Solaris_FS_data

This function loads Solaris filesystem data.

There are no functions to list in Chapter 18.

Chapter 19

There are no functions to list in Chapter 19.

Chapter 20

There are no functions to list in Chapter 20.

Chapter 21

man_page

Function to create man page type information about the proper usage of the SSAidentify.ksh shell script.

twirl

Progress indicator that looks like a "twirling" or rotating line.

all_defined_pdisks

Function that lights all disk-identification lights for all defined pdisks.

all_varied_on_pdisks

Function that lights all disk-identification lights that are in varied-on Volume Groups.

list_of_disks

Function that acts on each pdisk by turning on/off the SSA disk-identification lights.

Chapter 22

ping_host

This function executes the correct ping command based on UNIX, the UNIX flavor, AIX, HP-UX, Linux, or SunOS.

ping_nodes

This function is used to ping a list of nodes stored in a file. This function requires the ping_host function.

All of these functions are used in gathering system information from an AIX system. Refer to Chapter 13 for more details in the AIXsysconfig.ksh shell script.

get_host get_OS get_OS_level get_ML_for_AIX get_TZ get_real_mem get_arch get_devices get_long_devdir_listing get_tape_drives get_cdrom get_adapters get_routes get_netstats get_fs_stats get_VGs get_varied_on_VGs get_LV_info get_paging_space get_disk_info get_VG_disk_info get_HACMP_info get_printer_info get_process_info get_sna_info get_udp_x25_procs get_sys_cfg get_long_sys_config get_installed_filesets check_for_broken_filesets last_logins

Chapter 24

There are no functions to list in Chapter 24.

Chapter 25

AIX_classic_printing

This function enables all AIX "Classic" print queues in a down state.

AIX_SYSV_printing

This function enables printing and queuing on all printers and queues in a down state.

CUPS_printing

This function enables CUPS printing and queuing on all printers and queues in a down state.

HP_UX_printing

This function enables printing and queuing on all printers and queues in a down state.

Linux_printing

This function enables printing and queuing on all printers and queues in a down state.

OpenBSD_printing

This function enables printing and queuing on all printers and queues in a down state.

Solaris_printing

This function enables printing and queuing on all printers and queues in a down state.

Chapter 26

There are no functions to list in Chapter 26.

Chapter 27

display_main_menu

This function displays the main menu of the dirvish_ctrl shell script.

read_input

This function will read in input until 99 is entered.

parse_conf

This function is used to parse through the Dirvish master.conf file.

build_default_conf

This function creates a new Dirvish backup server definition.

run_all

This function executes the dirvish-runall command.

run_backup

This function executes one particular Dirvish backup.

chk_create_dir

This function creates a new Dirvish bank.

locate_restore_image

This function is used to locate Dirvish archives and restore specific files to a local "restore area."

expire_backup

This function is used to delete expired Dirvish backups.

add_backup

This function is used to add a new Dirvish backup.

delete_backup

This function is used to delete Dirvish backup definitions (deletes a Dirvish vault).

manage_banks

This function is used to create and delete Dirvish banks.

check_root_user

This function checks to ensure the user executing the script is root.

check_config

This function queries the system for the specified Dirvish default.conf file for a particular backup.

Chapter 28

cleanup_exit

This function emails the captured keystroke script file before exiting the shell script.

Index

SYMBOLS

() (parentheses) command, 15 double-parentheses mathematical test, 117, 337, 338, 346, 363, 414, 575, 609, 682, 734 & (ampersand) & (piping to background), 36, 351, 364 && (logical AND) operator, 13, 20, 570, 813 background execution, 13 bitwise AND operator, 20 escaped, 3, 4 * (asterisk) multiplication operator, 20 wildcard, 31, 830 ^(caret) ^\$ (remove blank lines), 669 ^# (remove commented out lines), 669, 858 bitwise exclusive OR operator, 20 escaped, 3, 4 start of line, 170, 669, 733, 858, 863 : (colon) as field delimiter, 157, 159, 167, 536, 537, 569 getops command and, 33, 34, 349, 450, 496 NFS check and, 569, 570 no-op and, 74, 96, 195, 348, 380, 416 \$ (dollar sign) \$? (check return code), 29-30, 183-184, 217 \$\$ (current PID), RANDOM shell variable and, 269, 311, 371, 379, 389 ^\$ (remove blank line), 669 "\$*" (special parameter), 17 "\$@" (special parameter), 17 \$* (special parameter), 17-18 \$@ (special parameter), 17 \$ (()) command, 15 elapsed time and, 148 \$! operator, 44, 144, 145, 147, 546, 547 blank line, 669 command substitution, 19, 80, 88, 97, 127, 128, 331, 484. See also command substitution end of line, 170, 863

escaped, 3, 4, 404, 664 variable name and, 15, 28, 304 . (dots) escaped, 3, 4 series of, 43, 143-145, 967 " (double quotes) around variables, 18, 50, 160, 169, 247, 516, 517, 570 command substitution and, 18, 50 command-line switches in, 450 disable command arguments with, 797 escaped, 3, 4 special parameters with, 17 usage for, 18 = (equal sign) == equality operator, 20, 337 != inequality operator, 20 escaped, 3, 4 ! (exclamation point) != inequality operator, 20 "!" NOT operator, 20, 797 escaped, 3, 4 / (forward slash) /\$ regular expression, 231 division operator, 20 escaped, 3, 4relative pathnames and trailing, 928 trailing, rsync and, 51, 52, 53, 220, 221, 222.231 \ (backslash) escaping with, 3-4, 402, 404, 408, 496 operators, -e switch and, 147, 148, 431, 503, 518, 617,855 n, t, b, c, v, 19# (hash mark/pound sign), 4 ^# (remove commented out lines), 669, 858 commented out lines, 4, 7, 129, 669, 726 as field separator, 4, 485 as padding, 158, 373 for separating report sections, 757

- (hyphen) -- auto-decrement operator, 20 escaped, 3, 4 ps auxw command without, 675 su command and, 951 subtraction operator, 20 unary minus operator, 20 < (left angle bracket) << bitwise shift left operator, 20 << command, 41, 128, 129 <= (less than or equal to) operator, 20 escaped, 3, 4 less than operator, 20 % (percent sign) escaped, 558 modulo arithmetic operator, 20, 372, 406 printf command and, 477 as text character, 558 | (pipe character) & (piping to background), 36, 351, 364 || (logical OR) operator, 13, 20, 570, 687, 705, 813 bitwise OR operator, 20 escaped, 3, 4 + (plus sign) ++ auto-increment operator, 20 addition operator, 20 escaped, 3, 4 as "greater than" specifier, 326 unary operator, 20 ? (question mark) \$? (return code checking), 29-30, 183-184, 217 escaped, 3, 4 > (right angle bracket) >> (appends to end of file), 13 >> bitwise shift right operator, 20 >= (greater than or equal to) operator, 20 escaped, 3, 4 greater than operator, 20 ; (semicolon) array element, 404 escaped, 3, 4 (single/forward quotes), 18 around square brackets, 28, 510 escaping special characters with, 4 \sim (tilde) binary inversion, 20 escaped, 3, 4 <> (angle brackets) left. See < right. See > string comparison, 15 { } curly braces Expect control structures and, 306, 313, 314, 315 if...then...else statement and, 313 shell script in, 230 variable separated from character with, 389, 559 [] square brackets mathematical expression in, 316 single quotes around, 28, 510 special character, 3, 4 test command, 15 ' back tic ('command'), 19, 77, 80, 88, 97, 127, 128, 331, 484. See also command substitution , comma (special character), 3, 4

 (()) command, 15, 20. See also double-parentheses mathematical test
 \$ (()) command, 15, 148

+ ([0–9]) -type regular expressions, Bash shell and, 65, 327

A

-a rsync switch, 51, 220, 232 abs function, 21 accept command, 820, 821 acos function, 21 ADD variable, 442, 469 add backup function, 975 addition, in shell script, 434-443 addition operator (+), 20 AIX classic printer subsystem, 50, 810-814 for loop in, 812–813 df -k command output, 586 output columns of interest, 586 iostat command output, 646 lsps command, 604–605 more command and, 516, 523 paging monitor, 607-613 pg/page commands and, 516, 523 ping command, 724 print-control commands, 810-820 sar command output, 649 scripts for filesystem monitoring, 961-962. See also monitoring filesystems stale disk partitions (monitoring for), 48-49, 677-696 system information, functions (list) for, 973 System V printing, 50, 814-820 system-configuration snapshot, 741-775 topas command, 675 AIX_classic_printing function, 847, 973 AIX_paging_mon.ksh shell script, 610-611, 973 in action, 612 calculations for, 607-610 exceeding 5% paging limit, 612-613 PC LIMIT variable, 611–612 AIXsysconfig.ksh shell script, 745-756, 965 in action, 757-774 analysis, 756-757 database/application-level statistics and, 774 AIX_SYSV_printing function, 847, 973 aliases file, 134, 137 all defined pdisks function, 699, 703-705, 972 all-in-one_swapmon.ksh shell script, 630-636, 963 analysis, 636-637 error message, 637, 638 all varied on pdisks function, 699, 705-707, 972 alpha OLTP server, 252. See also replicating Oracle databases alphanumeric page, 131, 139 ampersand (&) |& (piping to background), 36, 351, 364 && (logical AND) operator, 13, 20, 570, 813 background execution, 13 bitwise AND operator, 20 escaped, 3, 4 angle brackets (<>), 15 left. See <

right. See > string comparison, 15 APIs. See application programming interfaces application monitoring. See monitoring processes/applications application programming interfaces (APIs), 548. See also SNMP apt manual pages, 293 systat package installation with, 642, 643 Tcl, Expect installation with, 291, 293 yum v., 643 arbitrary-precision arithmetic, 54, 433. See also floating-point math arithmetic operators (list), 20-21 array elements, 60, 403 ; (semicolon), 404 KEYS array and, 404 array pointer, 402 arrays, 60-61, 403-404, 432, 833 KEYS, 403–404 loading, 60-61, 403-404 ASCII text, 4, 67 asin function, 21 asterisk (*) multiplication operator, 20 wildcard, 31, 830 at command, 31-32 audit server, for syslog files, 854 auditing/monitoring user's keystrokes, 53–54, 935 - 954audits, PCI, 804, 864 audits, SOX. See Sarbanes-Oxley audits auto-decrement operator (--), 20 autoexpect, 213-216, 296-303. See also Expect Expect scripts created with, 291, 297-299, 324, 539 - 545ftp-get-file-cmd-line.exp script, 304-305, 958 in action, 305-306 ftp-get-file.exp script, 299-300, 958 new, 302-303 unneeded code in, 300-302 ftp-ls-getfile.exp script, 213-215, 957 in action, 216 script command v., 213, 296-297 showplatform.exp Expect script, 318-320, 958 in action, 320-321 auto-increment operator (++), 20 automated event notification, 131-141 email notification techniques, 41-42, 132-138 filesystem monitoring and, 600 paging and swap space, 638 paging/modem dialing products, 139 proactive approach, 131 SNMP traps and, 139–140 stale disk partition monitoring and, 695–696 automated execution, of filesystem monitoring, 600-601 automated FTP file transfer, 51, 187-218. See also rcp; rsync; scp autoexpect program and, 213-216 debug mode, 217 encryption and, 209-210 expect scripts and, 212-216 get_ftp_files.ksh shell script, 192-195, 956

in action, 196 as upload script, 196-199 get_ftp_files_pw_var.ksh shell script, 204-207, 957 get_remote_dir_listing.ksh shell script, 191–192 get_remote_dir_listing_pw_var.ksh script, 203-204 goal of, 190 hard-coded passwords, 199 variable-replacement technique, 199-203 here document for, 51, 188-189 password variables in, 203-209 put_ftp_files.ksh shell script, 196-199, 957 put_ftp_files_pw_var.ksh shell script, 207-209, 957 remote directory listings and, 190-192 remote files downloading from remote system, 192-196 uploading to remote system, 196-199 shell script password, 217 tst_ftp.ksh shell script, 189-190, 956 automated hosts pinging (with notification of failure), 49, 723-740 command syntax, 723-724 notification of unknown host, 738 pinging /etc/hosts file, 737 \$PINGLIST variable-length-limit problem, 736 pingnodes.ksh shell script, 728-733, 964 in action, 735-736 analysis, 733–735 automated execution, with cron table entry, 739 creation, 725-728 functions in, 733-734 logging capability, 737–738 notification method, 738-739 setting trap, 728 variables in, 725-727 automating interactive programs. See autoexpect; Expect average of list of numbers, in shell script, 467-472 awk command, 14, 485, 488, 489, 811, 812 chg_base_bc.Bash shell script, 509 chg base.ksh script and, 503 cut command v., 168-169 Solaris and. See nawk; nawk command SOX audits and, 856-861 variable-length record files and, 164

B

\b (backslash operator), 19, 700 back tics ('command'), 19, 77, 80, 88, 97, 127, 128, 331, 484. *See also* command substitution background function, 145 co-process with, 34–36, 348, 349–351 backslash (\) escaping with, 3–4, 402, 404, 408, 496 operators, -e switch and, 147, 148, 431, 503, 518, 617, 855 \n, \t, \b, \c, \v, 19 backups incremental, 868 programs, 867. *See also* Dirvish; rsync remote-update protocol and, 51, 219, 220, 232, 233, 235, 237, 247, 251, 867 backups (continued) snapshot-style, 868 tape incremental, 867 Backus, John, 413 Backus-Naur form (BNF), 413 bad interpreter error, 295 banks, 869 backup, 890-893, 930-932 adding, 891-892, 931-932 deleting, 892-893, 932 banybody shell script. See boracle shell script base conversions. See number base conversions base filename, 388, 389 basename \$0 command, 364, 379, 439, 484, 494, 864 basename command, 14, 45, 171 SOX audits and, 863-864 base#number notation, 503, 504, 506, 509 Bash shell (Bourne-Again shell), 4 + ([0–9]) -type regular expressions and, 65, 327, 380, 416 declaration statement for, 6 echo command, correct usage of, 548 Linux execution in, 19, 195, 255 RANDOM shell variable, 370 batch-processing errors, 157, 169. See also merge process bc utility, 20, 54-55 here documents float.add.ksh shell script, 442-443 float.divide.ksh shell script, 466-467 float.subtract.ksh shell script, 451-452 syntax, 433-434 Linux_swap_mon.ksh shell script and, 618–620 manual page, 473 for number base conversions, 506-512 scale and, 433, 434, 443, 450, 452, 458, 467, 626 shell scripts float add.ksh, 434-443 float_average.ksh, 467-472 float_divide.ksh, 460-467 float multiply.ksh, 452-460 float_subtract.ksh, 443-452 online, 959 testing data integrity, 440-441, 448-450, 458, 460-461 bdf command (HP-UX), 585 command output, 588 output columns of interest, 589 Beginning-of-Line, 170 bin directory, 559 binary inversion (\sim), 20 bit bucket, 32, 33 bitwise AND operator (&), 20 bitwise exclusive OR operator (^), 20 bitwise OR operator (|), 20 bitwise shift left operator (<<), 20 bitwise shift right operator (>>), 20 blank lines, removal of, 58 blank screen, 153 BMC Patrol, 140 BNF. See Backus-Naur form boot logical volume, 680 boot net - install command, 322, 323 boot-net-install.exp Expect script, 322-323, 958 bootps, 853

boracle shell script, 948-951, 966 analysis, 951 bounce account, 136-137, 940 Bourne shell, 4 declaration statement for, 6 echo command, correct usage of, 548 RANDOM shell variable, 370 Bourne-again shell. See Bash shell Boxman, Jason, 868 braces/brackets. See angle brackets; curly braces; parentheses; square brackets branch01 fixed.dat, 173-174 branch01 variable.dat, 174 branch02_variable.dat, 174 branches, 869 using, 874 BRANCH_RECORDS_FIXED.LST, 181 BRANCH_RECORDS_VARIABLE.LST, 181 bravo OLTP server, 252. See also replicating Oracle databases break command, 10, 11, 887 Brenstein, Nathaniel, 951 broot shell script, 943-948, 966 /etc/sudoers file, 946-948, 953 log keystrokes.ksh script v., 945 root password and, 948 build_default_config function, 888, 889, 974 build_manager_password_report function, 421, 970 description of, 410-413 build random line function, 398, 966, 967

C

\c (backslash operator), 19, 700 C++, 756 C programming language, 20 arithmetic operators, 20-21 compiler, 779 interpreted shell scripts/functions v., 4, 756 POSIX C type notation, 637, 744 C shell. See csh cal command, 13 capturing large files. See large files user's keystrokes. See user's keystrokes caret (^) ^\$ (remove blank line), 669 ^# (remove commented out lines), 669, 858 bitwise exclusive OR operator, 20 escaped, 3, 4 start of line, 170, 669, 733, 858, 863 case sensitivity, 3 case statement, 10 Expect's version of, 306-313 curly braces in, 306 inside while loop, 146 iostat fields of data, 648 nested, 338 parsing with, 338-341, 415-416 sar fields of data, 651 vmstat fields of data, 653-654 cat command, 12, 74 catching delayed command output, 36-37 cat_while_LINE_line, 76-77 cat_while_LINE_line_cmdsub2, 78-79 cat_while_read_LINE, 74-75

cat_while_read_LINE_FD_OUT, 83-84 cc, 779 cd (change directory) command, 12, 188, 210 cfgmgr (Configuration Manager), 678 cfgmgr -V command, 678 change directory command. See cd command changing passwords, 203. See also variable-replacement technique check config function, 975 check exceptions function, 561, 569-570, 971 error message, 600 modified, 569-570, 576 check_filesystems function, 133, 134 check_for_and_create_keyboard_file function, 419, 425, 429, 970 description of, 409-410 check_for_broken_filesets function, 752, 973 check HTTP server function, 545, 971 checking return code (\$?), 29-30, 183-184, 217 check root user function, 975 checksum search algorithm, rsync, 219, 289 chg_base_bc.Bash shell script, 506-512, 960 in action, 511-512 analysis, 509-511 awk/nawk usage, 509 chg_base.ksh shell script, 500-506, 960 in action, 505-506 analysis, 503-504 awk/nawk command, 503 case statement for echo usage, 504 chg pwd function, 801 chgrp command, 12 chk_create_dir function, 975 chk_passwd_gid_0.Bash shell script, 860-861, 966 chmod command, 12 file permission options, 21–23 syntax, 22-23 chown command, 12 chpwd_menu.ksh shell script, 798-801, 965 cleanup function, 699, 702-703 cleanup_exit function, 942, 975 clear command, 12 Cmnd_Alias, 797 full pathnames of commands and, 797, 947 colon (:) command-line switches and, 450 as field delimiter, 157, 159, 167, 536, 537, 569 getops command and, 33, 34, 349, 450, 496 NFS check and, 569, 570 no-op and, 74, 96, 195, 348, 380, 416 column headings in command output, removal of, 59 - 60, comma (special character), 3, 4 'command'. See command substitution command(s). See also specific commands for declaring shells, 6 Expect. See Expect output. See command output print (list), 13 readability/style of, 6-7 on remote host, 23-25 select, 56-58 shell script (list), 12-14 stringing together. See pipes style/readability, 6-7

substitution. See command substitution symbol (list), 14-15 user-information, 25-26 wildcards, 12, 31, 214, 298, 299, 319 command output column headings in, removal of, 59-60 delayed, catching, 36-37 in loop, 40-41, 128 command substitution, 77 'command', 77, 80, 88, 97, 127, 128, 331, 484 \$ (), 19, 80, 88, 97, 127, 128, 331, 484 double quotes (") and, 18, 50 no. See single quotes within sed statement, 515, 516-517, 525 timing of, 127-128 command-line arguments, 15 in mk passwd.ksh shell script, 414-418 parsing with getopts, 33-34, 217, 338, 348-349, 417-418, 495-497 with nested case statement, 338-341, 415-416 testing, 414-416 command-line script session, 936-937 command-line switches. See also specific switches colon placement in list of, 450 double quotes around list of, 450 comments, 6-8 # and, 4, 7, 129, 669, 726, 858 importance of, 6-8, 693 script.stub shell script and, 7-8, 955 Common UNIX Printer System. See CUPS communicating with users, 27. See also rwall command; talk command; wall command; write command COMPLETE_FILE, 255 compress command, 952 gzip v., 952 compression of files, rsync and, 51, 53, 187, 219, 220, 221, 232, 289 conditional tests, Expect and, 306-318 Configuration Manager. See cfgmgr configure command, 782, 787 example of running, 782-787 Configuring and Using Dirvish for Snapshot Backups (Boxman), 868 continue command, 10, 11 control structures, 8-10 case statement, 10 Expect, curly braces and, 306, 313, 314, 315 for...in statement, 9 if...then statement, 8 if...then...elif...(else) statement, 9 if...then...else statement, 8, 313 in...then statement, 8 until statement, 9 while statement, 9 converting numbers between bases. See number base conversions co-process, 34-36, 348, 349-351 example of usage, 350-351 as pipe to background, 36, 351, 364 copy command, 12. See also cp; rsync cos function, 21 cosh function, 21 COUNT variable, 314, 315, 316, 317, 318, 438, 458, 459

cp (copy), 12. See also rsync CPU hogs, 675. See also top-like monitoring tools CPU load, monitoring. See monitoring system load crashing system, 133, 601, 603, 641, 952 cron tables, 30-31 automated execution of filesystem monitoring, 600-601 entry syntax, 31 rsync_daily_copy.ksh shell script and, 254 crontab command, 30-31 csh (C shell), 6 CUPS (Common UNIX Printer System), 50, 809, 810, 820-823 command output checking printing/queuing, 821 commands, 821 online manual, 820 cupsdisable command, 821 cupsenable command, 820, 821 CÚPS_printing function, 847, 974 curly braces ({ }) Expect control structures and, 306, 313, 314, 315 if...then...else statement and, 313 shell script in, 230 variable separated from character with, 389, 559 current working directory, 328 cursor control commands, for echo command, 700 example, 700-701 cut command, 14, 485, 488, 555 awk command v., 168-169 fixed-length record files and, 164 SOX audits and, 856-861

D

data movement techniques. See automated FTP file transfer; rcp; rsync; scp date command, 13, 328, 347, 938 switches, 388 date/time stamp, and unique filenames, 371, 384, 388, 389 DB2, 603 dd command, 48, 370, 376, 377, 379, 402, 407. See also /dev/random debug mode, 217 declaring functions (before using), 5, 537, 575, 698 shells, 6 default.conf file, 873-874 code to create new, 889 defining full system backup, 874-875 delete_backup function, 889-890, 975 /dev/random, 48, 369, 370, 376-379 dd command and, 48, 370, 376, 377, 379, 402, 407 /dev/urandom v., 376, 402 od command and, 48, 370, 376, 377, 407 /dev/urandom, 48, 370, 376–379 /dev/random v., 376, 402 df command, 14, 74, 237, 238 df -g, 566 df -k, 554, 565, 566 command output AIX, 586 Linux, 586 OpenBSD, 587 SÛN/Solaris, 587-588

output columns of interest AIX, 586 Linux, 587 OpenBSD, 587 Sun/Solaris, 588 df -m, 566 diff command, 13 directories, replicating. See replicating multiple directories dirname command, SOX audits and, 863-864 Dirvish, 289, 867-933 on command line, 875-876 configuring, 868-869 Configuring and Using Dirvish for Snapshot Backups, 868 default.conf file, 873-874 code to create new, 889 defining full system backup, 874-875 dirvish_ctrl shell script. See dirvish_ctrl shell script disk space requirements, 868 download, 871 image summary file, sed and, 883-884 installing, 869–871 with install.sh, 871 with yum, 869-871 master.config file, 872-873 menu-driven interface for, 876-918 usage examples, 918-932 overview, 868 Perl modules and, 869 rsync and, 289. See also rsync dirvish_ctrl shell script, 893-918, 966 adding new backup vault, 884-889 examples, 925-930 backup banks (managing), 890-893, 930-932 adding, 891-892, 931-932 deleting, 892-893, 932 building, 876-893 expiring/deleting backup images, 881-884 examples, 921-925 sed for summary file modification, 883-884 locating/restoring images, 880-881 example, 919-921 main menu, 877-878 removing backup vault, 889-890 example, 930 running all backups, 878-879 examples, 918-919 running particular backup, 879-880 example, 919 usage examples, 918-933 dirvish-expire command, 881, 882 dirvish-locate command, 876, 881 dirvish-runall command, 875, 878, 918, 919 disable command, 13 display_main_menu function, 974 display_output function, 575, 600, 971 dividend, 460-461 division, in shell script, 460-467 division operator (/), 20 divisor, 460-461 DMZ, 527 dollar sign (\$) \$? (check return code), 29-30, 183-184, 217

\$\$ (current PID), RANDOM shell variable and, 269, 311, 371, 379, 389 ^\$ (remove blank line), 669 "\$*" (special parameter), 17 "\$@" (special parameter), 17 \$* (special parameter), 17–18 \$@ (special parameter), 17 \$ (()) command, 15 elapsed time and, 148 \$! operator, 44, 144, 145, 147, 546, 547 as blank line, 669 command substitution, 19, 80, 88, 97, 127, 128, 331, 484. See also command substitution end of line, 170, 863 escaped, 3, 4, 403, 664 variable name and, 15, 28, 304 dots (.) escaped, 3, 4 series of, 43, 143-145, 967 double ampersand && (logical AND operator), 13, 20, 570, 813 double equal sign == (equality operator), 20, 337 double pipes || (logical OR operator), 13, 20, 570, 705, 813 double quotes (") around variables, 18, 50, 160, 169, 247, 516, 517, 570 command substitution and, 18, 50 command-line switches in, 450 disable command arguments with, 797 escaped, 3, 4 special parameters with, 17 usage for, 18 double-parentheses mathematical test ((math test)), 117, 337, 338, 346, 363, 414, 575, 609, 682, 734 downloading remote files from remote system, 192 - 196get_ftp_files.ksh shell script, 192-196, 956 get_ftp_files_pw_var.ksh shell script, 204-207, 957 post-FTP events, 192, 193, 195 pre-FTP events, 192, 193, 195 DSA keys, 24, 210

Ε

-e switch, 19, 147, 148, 405, 431, 503, 518, 617, 855 ECHO, 148 echo command, 13, 19-20, 86, 88 correct usage of, 19-20, 548, 590 Bourne/Bash shell, 548 Korn shell, 548 cursor control commands for, 700 example, 700-701 -e switch added to, 19, 147, 148, 405, 431, 503, 518, 617,855 proc watch.ksh script and, 346 progress indicators and, 143, 144, 145, 148 EcoTools, 140 effective user, 948 egrep (extended grep) statement, 13, 232 grep v., 528, 816 monitoring filesystems and, 555, 601 monitoring processes and, 527, 546-547 EGREP_LIST variable, 227, 228, 232

elapsed time method, 44-45, 148-150 \$ (()) command and, 148 series-of-dots method with, 151-153 shell script in action, 150 elapsed_time function, 44-45, 148-149, 398, 966, 967,968 expanded to days, 153 in generic_DIR_rsync_copy.Bash shell script, 230 in random_file.bash shell script, 398 in rsync daily copy.ksh shell script, 262, 275 Elkins, Michael, 951 emacs command, 14 emacs editor, 14, 30. See also vi editor email notification techniques, 41-42, 132-138 outbound email, 41-42 bounce account and, 136-137 .forward file and, 136–137, 940 problems with, 134-138 sendmail command and, 137-138 paging/modem dialing products, 139 EMAIL_FROM, 255 emailing audit logs, 951-952 enable command, 13, 813 enable_AIX_classic.ksh shell script, 812, 965 enabled printers. See print-queue management encrypted connections, password-free, 210-211, 530. See also ssh encryption, FTP and, 209-210. See also scp; sftp encryption keys, 210, 530. See also sftp creation of, 210, 530 DSA, 24, 210 mk_passwd.ksh script and, 432 RSA, 24, 210 enq command, 13 -A, 810 output with, 810 Enron, 851 entropy pool, 376, 402 equal sign (=) == equality operator, 20, 337 != inequality operator, 20 escaped, 3, 4 equate_any_base.ksh shell script, 490-500, 960 beginning of main, 498-500 code segment to parse command line, 499 code segment to verify number base variables, 497 - 498correct/incorrect usage of, 495 finding number to convert, 498–500 parse command line, getopts, 495–497 switches in, 490 testing in, 494-495 usage function in, 494 variables in, 494 equate_base_2_to_10.ksh shell script, 960 equate_base_2_to_16.ksh shell script, 478-481, 960 tests in, 480-481 equate_base_8_to_10.ksh shell script, 960 equate_base_10_to_2.ksh shell script, 960 equate_base_10_to_8.ksh shell script, 960 equate_base_10_to_16.ksh shell script, 481-485, 960 analysis, 484–485 equate_base_16_to_2.ksh shell script, 960 equate_base_16_to_10.ksh shell script, 960 error log, SSA disk identification and, 721

errors all-in-one_swapmon.ksh shell script, 637, 638 bad interpreter, 295 batch-processing, 157, 169. See also merge process check_exceptions function, 600 file permissions, 327 standard, 68, 81-82 usage, 180, 331, 364, 383, 384, 413 escaping special characters, 3-4 with backslash, 3, 4, 402, 404, 408, 496 with single quotes, 4 /etc/hosts file, pinging, 737 etc-hosts-copy.exp Expect script, 308-310, 958 in action, 312-313 analysis, 310-312 /etc/motd file, 27 /etc/sudoers file, 946-948, 953 sample # 1, 791-794 sample # 2, 794-797 /etc/syslog.conf file, 801, 802, 804, 805 EtherPage, 139 exclamation point (!) != inequality operator, 20 "!" NOT operator, 20, 797 escaped, 3, 4 -exec command switch, 326 exit command, 10, 11 exit signals, 25, 414, 418. See also traps complete list, 25 exit_trap function, 439 exp function, 21 Expect (programming language), 216 Bash script to talk with, 294 case statement and, 306-313 commands, 293-294 conditional tests, 306-318 example syntax, 307-308 control structures, curly braces and, 306, 313, 314, 315 downloading/installing, 291-293 ftp automation, 212-216 functions and, 317-318 if...then...else loop and, 313 incr operator, 315, 316 location (path) of, 295 for loop and, 315-317 man page, 216 proc operator, 317-318 set command, 304, 305 source code, 291 SSH login to known host, 307 SSH login to unknown host, 306-307 Tcl and, 291-293 web site, 291, 323 while loop and, 314-315 expect command, 293, 294 expect eof, 295 Expect scripts. See also autoexpect autoexpect and creation of, 291, 297-299, 324, $5\overline{3}9 - 545$ automated ftp and, 212-216 Bash shell script interacting with, 296 boot-net-install.exp, 322-323, 958 etc-hosts-copy.exp, 308-310, 958 in action, 312-313 analysis, 310-312

failure/hanging, 299 for text2.exp, using expression to increment, 317 for_text.exp, 315-316 in action, 316 ftp-get-file-cmd-line.exp, 304-305, 958 in action, 305-306 ftp-get-file.exp, 299-300, 958 new, 302-303 unneeded code in, 300-302 ftp-ls-getfile.exp, 213-215, 957 in action, 216 function_test.exp, proc operator in, 317-318 if-then-else.exp, 313 with JumpStart, 322-323 passmass, 323-324 remotely execute who/w commands with, 543-545 scp and, 210 sftp and, 210 showplatform.exp, 318-320, 958 in action, 320-321 with Sun Blade Chassis, 318-323 that starts and talks to Bash script, 295 variables and, 304-306 while_test.exp, 314 in action, 314-315 expire backups menu, 882 code to display, 882 expire backup function, 881, 975 exponential notation, numbers in, 56. See also floating-point numbers extended grep statement. See egrep statement

F

f command, 14 failures and event notification, 131 Expect script, 299 notification of automated hosts ping with, 49, 732-740 with logical OR, 705 Oracle Data Guard server and, 255 during rsync process, 257 script, 5, 6 during sudo installation, 790 system crashes, 133, 601, 603, 641, 952 tape drive, 641 traps and, 269 from unexpected output/input, 12 feedback, user. See progress indicators field delimited lines of data. See variable-length record files field delimiter (:), 157, 159, 167, 536, 537, 569 field separator, hash marks as, 485 fields (in records), 157. See also record files file command, 12 file descriptors, 68, 81-82. See also parsing files line-by-line; stderr; stdin; stdout range of values, 68 file permissions, 21-23 chmod command, 21-22 syntax, 22-23 errors, 327 SOX audits and, 853 suid/sgid and, 21-23

File Transfer Protocol. See FTP filenames, unique, 384-392 filesystem alerts, 325, 333 filesystems monitoring, 553-602. See also monitoring filesystems replicating. See replicating multiple filesystems vaults as, 880 find command, 13, 325, 855-856 executing as root, 327 manual page, 325 power of, 325 SOX audits and, 855-856 switches -amin, 333 -amin n, 333 -atime n, 333 -mmin n, 333 -mtime n, 333 -print, 326 -size, 326 -xdev, 333 syntax, 326-327 find \$SEARCH_PATH -amin 10 --type f, 333 find \$SEARCH_PATH -mmin 10 -type f, 333 finding large/specific files, 53, 325-334. See also large files findlarge.ksh shell script, 328-331, 958 in action, 332 analysis, 331–332 improved/customized, 333-334 findslot Bash script, 321-322, 958 finger command, 14 FirstPAGE, 139 fixed-length pseudo-random numbers between 1 and user-defined maximum, 373-376 in_range_fixed_length_random_number function and, 373-374 padded with leading zeros, 374 typeset command, 375 sample output for, 374 fixed-length record files cut command and, 164 description/definition, 157 example, 158, 165 data in each field, 158 fields in, 158 merge script for, 45-46, 170-171 merge/process, 181 parsing, while_read_LINE_bottom_FD_OUT and, 160, 164-166 placement of data field within record, 164, 166, 184 post_processing_fixed_records.dat file listing, 181 - 182string length of, 46-47, 171-172 typeset command and, 47 flat files, 157, 251 float_add.ksh shell script, 434-443 in action, 443 analysis, 439-440 building math statement for bc utility, 441-442 functions in, 439-440 here document, 442-443

testing for integers/floating-point numbers, 440 - 441variables in, 439 float average.ksh shell script, 467-472 in action, 469 average of list of numbers (code segment), 468 running total of numbers (code segment), 467 - 468float divide.ksh shell script, 460-467 in action, 467 code to extract dividend/divisor, 461 here document, 466-467 floating-point math (in shell script), 20-21, 54-55, 433-473. See also bc utility floating-point numbers (testing for), 440-441 float_multiply.ksh shell script, 452-460 in action, 460 parsing command line for valid numbers, 458 - 460float_subtract.ksh shell script, 443-452 in action, 452 building math statement string for bc, 450-451 here document, 451-452 parsing command line, with getopts, 449-450 for loop, 68, 74. See also parsing files line-by-line; select command to enable classic AIX print queues, 812-813 Expect's version of, 315-317 force_conservative mode, 299 for...in statement, 9 fork() system call, 675 for_LINE_cat_FILE, 79-80 for_LINE_cat_FILE_cmdsub2, 80-81 for_LINE_cat_FILE_cmdsub2_FD_OUT, 39, 88-89, 162 for_LINE_cat_FILE_FD_OUT v., timing difference, 163 second place in timing tests, 39, 162 for_LINE_cat_FILE_FD_OUT, 39-40, 87-88, 162-163 for_LINE_cat_FILE_cmdsub2_FD_OUT v., timing difference, 163 third place in timing tests, 39, 162-163 for_text2.exp Expect script, 317 for_text.exp Expect script, 315-316 in action, 316 forward file, bounce account with, 136-137, 940. forward slash (/) /\$ regular expression, 231 division operator, 20 escaped, 3, 4 relative pathnames with trailing, 928 trailing, rsync and, 51, 52, 53, 220, 221, 222, 231 forward tics/quotes. See single quotes free command (Linux), 606 -m output, 618-619 frequency variations. See also pseudo-random numbers of radioactive decay events, 369 of white noise, 369 from field, 41-42, 137 mail command and, 42 frozen screen, 153 fs mon AIX except.ksh shell script, 562-564, 961 fs mon AIX.ksh shell script, 556-559, 961

fs_mon_AIX_MBFREE_except.ksh shell script, 570-573,961 fs_mon_AIX_MBFREE.ksh shell script, 567-568, 961 fs_mon_AIX_PC_MBFREE.ksh shell script, 576-583,961 fs_mon_ALL_OS.ksh shell script, 591-600, 962 fs_mon_HPUX_except.ksh shell script, 962 fs mon HPUX.ksh shell script, 962 fs_mon_HPUX_MBFREE_except.ksh shell script, 962 fs mon HPUX MBFREE.ksh shell script, 962 fs_mon_HPUX_PC_MBFREE.ksh shell script, 962 fs_mon_LINUX_except.ksh shell script, 962 fs mon LINUX.ksh shell script, 962 fs_mon_LINUX_MBFREE_except.ksh shell script, 962 fs mon LINUX MBFREE.ksh shell script, 962 fs_mon_LINUX_PC_MBFREE.ksh shell script, 962 fs_mon_SUNOS_except.ksh shell script, 963 fs_mon_SUNOS.ksh shell script, 963 fs_mon_SUNOS_MBFREE_except.ksh shell script, 963 fs_mon_SUNOS_MBFREE.ksh shell script, 963 fs_mon_SUNOS_PC_MBFREE.ksh shell script, 963 FS_PATTERN variable, 238, 247 FTP (File Transfer Protocol). See also automated FTP file transfer; OpenSSH connections, syntax for, 187-190 debug mode, 217 disable, 209, 530, 851, 853 post-FTP event processing, 187, 190, 192, 193, 195 pre-FTP event processing, 187, 190, 192, 193, 195 for remote directory listings, 190-192 sftp v., 209. See also sftp typical file download, 188 ftp command, 187–190. See also rsync -d switch, 217 debug option, 217 -i command switch, 189 -n switch, 189 subcommands. See get; mget; mput; nlist; put -v switch, 189 ftp-get-file-cmd-line.exp script, 304-305, 958 in action, 305-306 ftp-get-file.exp Expect script, 299-300, 958 new, 302-303 unneeded code in, 300-302 ftp-ls-getfile.exp script, 213-215, 957 in action, 216 full pathnames Cmnd_ALIAS and, 797, 947 pwd command and, 326, 331 functions, 4–5, 966–975. See also specific functions (complete list), 966–975 bc. See bc utility declaring/defining, before using, 5, 537, 575, 698 Expect and, 317-318 form of, 5 (list) in AIXsysconfig.ksh shell script, 973 interpreted, 4, 756 (list) mathematical, 21 for parsing files line-by-line, 73-98 (list), 967 proc operator as. See proc

written at system level, 5 written in shell scripts, 4–5 function_test.exp Expect script, 317–318

G

gamma master database server (Oracle), 252. See also replicating Oracle databases gcc, 779 generic_DIR_rsync_copy.Bash shell script, 223–233 analysis, 230-233 elapsed time function in, 230 with files existing at target, 235-237 verify copy function in, 230 when files do not exist on target, 233-235 generic_FS_rsync_copy.Bash script, 238-247 performing initial copy, 247-249 when files exist on target, 249-251 generic rsync.Bash script, 52-53, 221-222, 955 get subcommand, 190 get_adapters function, 749, 973 get_arch function, 747, 973 get cdrom function, 748, 973 get_devices function, 748, 973 get disk info function, 750, 973 get fs stats function, 749, 973 get_ftp_files.ksh shell script, 192-195, 956 in action, 196 as upload script, 196-199 get_ftp_files_pw_var.ksh shell script, 204-207, 957 get_HACMP_info function, 951, 973 get_host function, 746, 973 get_installed_filesets function, 752, 973 get_long_devdir_listing function, 748, 973 get_long_sys_config function, 752, 973 get_LV_info function, 750, 973 get_ML_for_AIX function, 747, 973 get_netstats function, 749, 973 getopts command colon (:) and, 33, 34, 349, 450, 496 example of usage, 348-349 limitations, 365 parse command-line arguments with, 33-34, 217, 338, 348-349, 417-418, 495-497 put_ftp_files_pw_var.ksh shell script and, 218 get_OS function, 973 get_OS_info function, 584, 590, 971 get_OS_level function, 584, 590, 973 get_paging_space function, 750, 973 get_printer_info function, 751, 973 get process info function, 751, 973 get_random_number function, 372 get_real_mem function, 747, 973 get_remote_dir_listing.ksh shell script, 191-192, 956 get_remote_dir_listing_pw_var.ksh script, 203-204, 957 get routes function, 749, 973 get_sna_info function, 951, 973 get_sys_cfg function, 752, 973 get_tape_drives function, 748, 973 get_TZ function, 747, 973 get_udp_x25_procs function, 752, 973 get_varied_on_VGs function, 750, 973 get_VG_disk_info function, 750, 973 get_VGs function, 749, 973

GID (group ID), 854, 855, 860, 861 global variables, 255, 473, 576 GNU function method, 75, 574 more command. See more command greater than operator (>), 20 greater than or equal to operator (>=), 20 grep command egrep v., 528, 816 highlighted. See hgrep process monitoring and, 335, 528. See also monitoring processes/applications with ps aux command, 336 unique filenames and, 384 wrong way to use, 560 gunzip command, 779 gzip, 779, 888, 952 compress command v., 952

Н

Hamilton, Bruce, 775 hanging, Expect script, 299 hard-coded passwords, 199 variable-replacement technique, 199-203 hash mark # (pound sign), 4 ^# (remove commented out lines), 669, 858 commented out lines, 4, 7, 129, 669, 726 as field separator, 485 as padding, 158, 373 for separating report sections, 757 hdisks, 694, 695, 697, 698 cross-reference script, pdisks and, 721 translating to pdisks, 698 head command, 13 here documents, 11, 188, 433-434 automated FTP file transfer, 51, 188-189 bc utility and float.add.ksh shell script, 442-443 float.divide.ksh shell script, 466-467 float.subtract.ksh shell script, 451-452 Linux_swap_mon.ksh shell script, 619 indentation and, 622 requirements for, 443 sftp and, 188, 210 syntax, 11, 433-434 hexadecimal representation of IP address, 475, 477, 485 as license key, 485-489 hgrep (highlighted grep), 512, 515-525 more command, 515-516, 523 reverse video control, 49, 50, 515, 516-517 sed command, 515, 516-517 tput command, 50, 516 options (list), 516, 524-525 rmso, 516, 525 sgr0, 516, 525 smso, 516, 524, 525 hgrep.Bash shell script, 519-524, 961 analysis, 523-524 in Bourne/Korn shells, 518 building, 517-518 hgrep.ksh shell script, 961 higher-calling script/function, 576 highlighted grep. See hgrep highlighting text in file, 49-50

Host Alias, 797 hostname command, 328 hosts pinging. See automated hosts pinging hosts.equiv file, 23 HP-UX bdf command output, 588 output columns of interest, 589 iostat command output, 646 line command and, 76, 80 more command and, 516 pg/page commands and, 516, 523 ping command, 724 ping_host function and, 49 print-control commands, 823-825 sar command output, 649-650 scripts for filesystem monitoring, 962. See also monitoring filesystems sendmail command, location of, 42, 137, 138 swapinfo command, 605-606 -tm output, 613-614 swap-space monitor, 613-618 swap-space report, 617-618 HP_UX_printing function, 847, 974 HP-UX_swap_mon.ksh shell script, 615-616, 963 building, 613-615 swap-space report, 617-618 HUP signal, 136, 805 HylaFAX+, 139 hyphen (-) -- auto-decrement operator, 20 escaped, 3, 4 ps auxw command without, 675 su command and, 951 subtraction operator, 20 unary minus operator, 20

-i command switch (ftp), 189 -i switch, 47, 172 id command, 854-855 SOX audits and, 854-855 identification lights, SSA. See SSA disk identification if...then statement, 8 if...then...elif...(else) statement, 9 if...then...else statement, 8 Expect's version of, curly braces and, 313 if-then-else.exp Expect script, 313 image, 869 summary file, sed and, 883-884 incr Expect operator, 315, 316 incremental backups, remote-update protocol and, 51, 219, 220, 232, 233, 235, 237, 247, 251, 867 increment_by_1 (Expect proc), 317, 318, 968 indentation, here documents and, 622 indicating progress. See progress indicators inequality operator (!=), 20 infinite loop proc_watch function, 34, 36, 351, 364 while loop, 43, 144, 147 inodes, 868 input, standard. See stdin in_range_fixed_length_random_number function, 373-374

in_range_fixed_length_random_number_typeset function, 379, 384, 388 in_range_random_number function, 372-373, 379, 420, 970 description, 406-407 install.sh, Dirvish installation with, 871 int function, 21 interact command, 294 interactive programs, automation of. See autoexpect; Expect Internet Printing Protocol (IPP), 820 interpreted shell scripts/functions, 4, 756 iostat command, 14 case statement (fields of data), 648 output (system load monitoring) AIX, 646 common denominator for data, 648-649 HP-UX, 646 Linux, 647 OpenBSD, 647 Solaris, 647 script for system load monitoring, 665-670 sysstat package and, 642 iostat_loadmon.ksh shell script, 665-668, 964 in action, 670 command statement, 668-670 IP address (hexadecimal representation), 475, 477, 485 as license key, 475, 477 mk swkey.ksh shell script, 485-489 IPP. See Internet Printing Protocol

J

JumpStart, 322–323. *See also* Expect scripts junk variable, 489, 614, 622, 828, 830 junk2 variable, 614, 622, 624

K

keeping printers printing. See print-queue management kernel pseudo-random number generator, 369, 370, 376 keyboard file, 402 checking for, 419 keyit shell script, 530-534, 961 keyit.dsa script, 24, 955 keyit.rsa script, 24, 955 KEYS array, loading one array element at a time, 404 in one step, 403-404 keys, license. See license keys kill -9 command, 25, 36, 269, 351, 418, 438, 440, 702, 942 kill command, 25, 145 Korn shell, 4 declaration statement for, 6 echo command, correct usage of, 548 RANDOM shell variable, 370 regular expressions for testing integers/floating-point numbers, 441 shift command. See shift command

L

-L switch, 47, 172 -l switch, 47, 172 labels, 188, 189, 433. See also here documents large files, 53, 325-334 creation of, timing test and, 68-73 findlarge.ksh shell script, 328-331, 958 in action, 332 analysis, 331-332 improved/customized, 333-334 hgrep and, 515. See also hgrep very, 327 large filesystems, 553, 565, 573. See also monitoring filesystems last command, 26 last logins function, 753, 973 lcd (local change directory) command, 188 < (left angle bracket) << bitwise shift left operator, 20 << command, 41, 128, 129 <= (less than or equal to) operator, 20 escaped, 3, 4 less than operator, 20 \$LENGTH variable, test for, 416 less than operator (<), 20 less than or equal to operator (<=), 20 let command, 20 license keys, 512 hexadecimal representation of IP address as, 475, 477, 485-489, 512 mk_passwd.ksh script and, 432 mk_swkey.ksh script and, 485-489 line command, 67, 68, 76, 77, 80 Linux and, 68, 76, 80 line printer control command. See lpc command line-by-line parsing techniques. See parsing files line-by-line lines blank, removal of, 58 repeating, removal of, 58 rotating, 43–44, 145–148 Linux apt, 293 systat package installation with, 642, 643 Tcl and Expect installation with, 291, 293 yum v., 643 df -k command output, 586 df -k output columns of interest, 587 execution of, in Bash shell, 19, 195, 255 free command, 606 -m output, 618-619 iostat command output, 647 line command and, 68, 76, 80 ping command, 724 print-control commands, 825-833 sar command output, 650 scripts for filesystem monitoring, 962. See also monitoring filesystems swap-space monitor, 618-622 sysstat package, 642-643 top command, 641, 675 example output, 641

yum, 291 apt v., 643 manual pages, 293 systat package installation with, 642–643 Tcl and Expect installation with, 291–293 Linux_printing function, 847, 974 Linux_swap_mon.ksh shell script, 620-621, 963 in action, 621 bc utility and, 618-620 calculations for, 618-620 linx command-line browser, 545-546 list paging space command. See lsps command list of disks function, 699, 707-708, 972 literal text, 18 load_AIX_FS_data function, 589, 971 load_default_keyboard function, 398, 410, 418, 967, 970 description of, 407-409 load EXCEPTIONS data function, 561, 575, 971 load_FS_data functions, 575, 584, 590, 971 load_HP_UX_FS_data function, 589, 971 loading arrays, 60-61, 403-404 load_LINUX_FS_data function, 589, 971 load OpenBSD FS data function, 589, 971 load_Solaris_FS_data function, 590, 971 local change directory command. See lcd command local processes, monitoring, 527-530. See also monitoring processes/applications locate_restore_image function, 975 log files (importance of), 184, 217, 231, 695 paging/swap space monitoring, 638 log function, 21 LÕGFILE, 255 logic code, for large and small filesystem free-space script, 574-575 logical AND operator (&&), 13, 20, 570, 813 logical map, 678 logical negation (!), 20 logical OR operator (||), 13, 20, 570, 687, 705, 813 logical partitions (LP), 678 Logical Unit Number (LUN), 677 Logical Volume Manager (LVM), 677–678 logical volumes (LVs), 678 boot, 680 loop to show number of stale PPs, 682 raw, 678 remp temp01, LV statistics for, 681 log_keystrokes.ksh shell script, 940-943, 966 analysis, 942-943 broot script v., 945 logging root activity, 942-943 logman account, 939, 940 long-running processes. See progress indicators loops, 68 for. See for loop in background, 144 break command and, 10-11 command output in, 40-41, 128 infinite, proc_watch function, 34, 36, 351, 364 Oracle database replication and. See replicating Oracle databases for parsing files. See parsing files line-by-line

until, 36-37 while. See while loop lowercase tr/typeset commands for, 28–29, 510 variables in, 15 LP. See logical partitions lp command, 13 lpc (line printer control) command, 814 command options (AIX), 814-815 command options (Linux), 826-827 command options (Solaris), 834-835 lpr command, 13 lpstat command, 13 command output (AIX), 810 -a, 818, 819 -p, 818, 819 -Ŵ, 811 command output (HP-UX), 823-824 command output (System V) -a, 838 -p,838 scripting, with -a and -p, 838-839 ls command, 12 lslv command, 678-679 lsps command (AIX), 604-605 lspv command, 679 lsvg command, 678 -l appvg2 rootvg (output), 679-680 -0,679 LUN. See Logical Unit Number LV level, monitoring for stale partitions at, 677, 679 - 684LV statistics, for remp_temp01 logical volume, 681 LVM. See Logical Volume Manager LVs. See logical volumes

Μ

-m switch, 403, 412, 413, 415, 416, 418, 421 MACHINE LIST, 230, 231, 247, 256 mail code segment, 133 mail command, 41, 132-134 - s switch, 132 from field and, 42 sendmail command v., 734 -v switch, 134, 135 mail notification techniques, 41-42, 132-136 MAIL FILE variable, 132, 133 MAIL_LIST variable, 133 MAILMESSAGEFILE, 255 MAILOUT LIST variable, 132 mailx command, 41, 132-134 from field and, 42 maintenance window, 27 make command, 782, 787 command output, 788-790 make install, 790 command output, 790 Makefile, 779, 782, 787 man command, 14 manage_banks function, 890, 891, 975 managed objects, 139

managed variables, 139 Management Information Base. See MIB manager's report, 421-422 disable, 432 man_page function, 699, 701, 720, 972 master.config Dirvish configuration file, 872-873 math, in shell script, 20-21. See also bc utility addition, 434-443 average of list of numbers, 467-472 division, 460-467 multiplication, 452-460 subtraction, 443-452 math operators (list), 20-21 mathematical expression, in square brackets, 316 mathematical functions (list), 21 mathematical tests. See double-parentheses mathematical test MAX COUNT variables, 439, 458, 494, 499 memory (physical), 603. See also paging space; swapping space memory leak, 612 menu creation, with select command, 56-58, 955-956 merge function, 180 merge process, 169–172 based on record-format type, 173-183 merge script fixed-length record files, 45-46, 170-171 variable-length record files, 46, 171 merge_variable_length_records function, 177, 180, 181,968 metamail, 939, 951, 952 metasend, 951, 952 mget subcommand, 190 MIB (Management Information Base), 139, 140, 548 Miller, Todd, 778, 791, 807 MIME types, 952 minus sign. See hyphen mirrors, 678 mkdir command, 12 mk_passwd.Bash shell script, 405, 406, 959 mk_passwd.ksh shell script, 422-431 analysis of, 418–422 building blocks/order of execution, 405-422 building new pseudo-random password code, 420checking for keyboard file, 419 command-line arguments, testing/parsing of, 414 - 418functions in, 406-414 loading KEYS array, 419-420 password selection, 432 printing manager's report, 421-422 disable, 432 pseudo-random list of keyboard characters with, 432 license keys/encryption keys, 432 setting trap, 418 variables in, 405 mk_swkey.ksh shell script, 485-489, 960 analysis, 488-489 mk_unique_filename.ksh shell script, 385-392 in action, 390-392 analysis, 388-390 functions in, 388-389

modem (dial-out only), 131, 139, 601, 738 software, 139 modulo arithmetic operator (%), 20, 372, 406 modulo N arithmetic, 372, 398, 406 monitoring filesystems, 553-602 automated execution, 600-601 command syntax, 553-556 cron table and, 600-601 egrep statement and, 601 event notification, 600 full filesystem script, 558 MB-of-free-space method, 565-568 with exceptions, 568-573 percentage of space method, 556-559 with exceptions, 559-565 percentage used-MB free method (with auto-detect), 573-583 proactive approach, 600 script on UNIX flavors (AIX, Linux, HP-UX, OpenBSD, Solaris), 583-600 shell scripts (list), on wiley website, 961-963 monitoring for stale disk partitions (AIX-specific), 48-49,677-696 automated execution and, 695 disk subsystem commands, 678-679 event notification and, 695-696 log files and, 695 LVM and, 677-678 methods at LV level, 677, 679-684 at PV level, 677, 684-687 at VG/LV/PV levels, 677, 687-694 scripts stale LV mon.ksh, 682-683, 964 stale_PP_mon.ksh, 685-686, 964 stale_VG_PV_LV_PP mon.ksh, 688-692, 688-694,964 SSA disks and, 694-695 monitoring paging/swap space, 603-639 command syntax AIX lsps command, 604-605 HP-UX swapinfo command, 605-606 Linux free command, 606 OpenBSD swapctl command, 606 Solaris swap command, 607 event notification, 638 log file, 638 proactive approach, 638 on scheduled basis, 638 shell scripts AIX paging monitor, 607-613 all-in-one paging/swap-space monitor, 630-637 HP-UX swap-space monitor, 613–618 Linux swap-space monitor, 618–622 Solaris swap-space monitor, 625-630 swap-space report (output), 604 HP-UX, 617-618 monitoring processes/applications, 335-368, 527-549 checking HTTP server/application, 545-546 command syntax, 336 egrep for, 527, 546-547 end of process, 338-342 grep and, 335, 528

levels/stages of, 527, 547-548 local processes, 527-530 code segment, 529 Oracle databases (status of), 536-539 ping command, 49, 217, 218, 247, 256, 527, 547, 548 proactive approach, 549 remote with expect script, 539-545 with OpenSSH and rsh, 530-536 scripts online, 958 proc mon.ksh, 338-342, 958, 961 proc wait.ksh, 338, 958, 961 proc_watch.ksh, 343-347, 958, 961 proc watch timed.ksh, 347-367, 958, 961 startup and endtime, 342-347 startup loop, 336–338 timed execution, 347-367 timing in, 335 waiting on processes to finish, 546-547 monitoring system load, 641-675 CPU hogs, 675 iostat command, 665 common denominator for output data, 648-649 output on UNIX flavors, 646-647 scripting with, 665-670 syntax, 645-646 proactive approach, 641 problem detection, 674 sar command, 659–660 common denominator for output data, 650-651 graphing data, 675 output on UNIX flavors, 649-650 scripting with, 660-665 syntax, 649 top-like monitoring tools, 641, 675 uptime command, 655 common denominator for output, 645 output, 644-645 scripting with, 655-659 syntax, 644 vmstat command, 670 common denominator for output data, 653-654 output on UNIX flavors, 651-653 scripting with, 670-674 syntax, 651 monitoring/auditing user's keystrokes, 53-54, 935-954 basics for script, 937-940 logging user activity, 937-939 repository, 939-940 starting monitor session, 939 command-line script session, 936-937 emailing audit logs, 951-952 informing users of, 953 monitoring other administration users, 948-951 OpenSSH and named pipe, 953 script command, 935 syntax, 936 scripts boracle, 948-951, 966 broot, 943-948, 966 compression in, 952 log keystrokes.ksh, 940-943

mon_proc_end function, 344, 969 mon_proc_start function, 344, 969 more command, 12, 74. *See also* pg command hgrep and, 515–516, 523 HP-UX/AIX systems and, 516, 523 mput subcommand, 190, 196 mrranger node, 735, 736 multiplication, in shell script, 452–460 multiplication operator (*), 20 mutt utility, 939, 951, 952 mv command, 12 my_program function, 389, 390 my_sql_query.sql SQL script, 538

Ν

\n (backslash operator), 19, 700 -n switch ftp, 189 id command and, 855 pseudo-random passwords and, 102, 407, 413, 415, 416, 418 Naur, Peter, 413 nawk command, 19, 168, 503, 509, 537 netstat command, 14 newaliases command, 134, 137, 411, 740 new-awk. See nawk command nlist subcommand, 190-191 no-op, 74, 96, 195, 348, 380, 416. See also colon NOPASSWD: specification, 794 NOT operator "!", 20, 797 notification, automated. See automated event notification nslookup command, 738 NULL data, 180, 379 NULL variable, 232 testing for, 58-59 number base conversions, 55-56, 475-513 base 2 to base 16, 478-481 base 8 to base 16, 476-477 base 10 to base 16, 476, 481-485 base 10 to hexadecimal, 477 base 10 to octal, 477 in Bash shell, 506-512. See also bc utility bc utility for, 506–512 common, 477 interactive script for, 500-506 printf command and, 56-57, 476-477 shell scripts, 477-512 (list), 477-478, 960 chg_base_bc.Bash, 506-512 chg base.ksh, 500-506 equate_any_base.ksh, 490-500 equate_base_2_to_16.ksh, 478-481 equate_base_10_to_16.ksh, 481-485 mk_swkey.ksh, 485-489 typeset command and, 55, 475-476 numbers in exponential notation, 56. See also floating-point numbers pseudo-random. See pseudo-random numbers numeric tests. See double-parentheses mathematical test NUM LIST variable, 439, 440, 441, 442, 451, 453, 462

0

octal dump command. See od command od (octal dump) command, 48, 370, 376, 377, 407. See also /dev/random creating different-size random numbers with, 377 OLTP (online transaction processing) database servers, 251. See also replicating Oracle databases filesystem layout, 252-253 one-dimensional arrays. See arrays online transaction processing. See OLTP Open Secure Shell. See OpenSSH OpenBSD df -k command output, 587 output columns of interest, 587 iostat command output, 647 line command and, 68, 76, 80 ping command, 724 ping_host function and, 49 ps aux command and, 336 sendmail command, location of, 42, 137, 138 swapctl command, 606 -lk output, 622 swap-space monitor, 622-625 OpenBSD_printing function, 847, 974 OpenBSD_swap_mon.ksh shell script, 623-624 in action, 624-625 creation of, 622-623 OpenSSH (Open Secure Shell), 24-25, 530. See also SSH DSA keys, 24, 210 encrypted file transfers, 209-210 named pipe and, 953 remote monitoring with, 530-536 RSA keys, 24, 210 rsh v., 24, 274, 289 scp. See scp sftp. See sftp SOX audits and, 851 operating systems. See AIX; HP-UX; Linux; OpenBSD; Solaris operators, arithmetic (list), 20-21 /opt, 558, 559 OPTARG variable, 33, 348, 350, 496 oracle administration account, 53, 538, 853, 935 Oracle Data Guard server, 255 Oracle databases, 251 checking for status, 536-539 master database server filesystem layout, 252 replication of, 219, 251-289 Oracle DBA Team's shell script, 251, 256, 257, 269, 275Oracle Listener, 538 Oracle+SQL query, 539 outbound email, 41-42 bounce account and, 136-137 .forward file and, 136-137, 940 problems with, 134-138 sendmail command and, 137-138 testing, 134-135 output control, 32-36 silent running, 32-33 output, standard. See stdout

Ρ

-p switch, 214, 298, 299, 301, 319 page command, 12, 516, 523. See also more command AIX/HP-UX systems and, 516, 523 page fault, 603 paging space, 603. See also monitoring paging/swap space swapping space v., 603 paging/modem dialing products, 139 parameters positional, 15, 458 shift command and, 16-17, 458-459 special, 17-18 parentheses () command, 15 double-parentheses mathematical test, 117, 337, 338, 346, 363, 414, 575, 609, 682, 734 parse conf function, 974 parse_fixed_length_records function, 165-166, 968 parse_records_files.Bash script, 956 /data directory, 174 /data/branch_records_fixed.1st, 174 /data/branch_records_variables.1st, 174 listing, 175-180 parse_variable_length_records function, 167-168, 968 parsing command-line arguments with getopts, 33-34, 217, 338, 348-349, 417-418, 449-450, 495-497 with nested case statement, 338-341, 415-416 parsing files line-by-line, 37-40, 67-129 command syntax, 67-68 file descriptors, 68, 81-82 input redirection in, 82 functions for, 73-98 list (on wiley web site), 967 methods, 73-98 fastest, 37-40, 122-126 timing test code to test command input, 116-117 large file creation, 68-73 pipe's impact in, 129 shell script, 99-116 sorted timing data by method, 121-122 timing data for each loop method, 117-121 passmass Expect script, 323-324 passwd command, 12 password page, 403, 411–412 password report, 421-422 disable, 432 password variables, ftp scripts with, 203-209 password-free encrypted connections, 210-211, 530. See also ssh passwords changing, 203 hard-coded, 199 passmass Expect script and, 323 pseudo-random. See pseudo-random passwords root, 410, 777 broot script and, 948 changing, 948, 953 sudo and, 778, 798, 806 rotating, 401 shell script, 217

SOX audits and, 853 variable-replacement technique and, 199-203 weak, 401, 403 patches/upgrades, SOX audits and, 854 pathnames full Cmnd_Alias and, 797, 947 pwd command and, 326, 331 relative, 874, 928 PCI audits, 804, 864. See also Sarbanes-Oxley audits PC_LIMIT variable, 611-612 pdisks, 695, 697, 698 cross-reference script, hdisks and, 721 translating hdisks to, 698 percent sign (%) escaped, 558 modulo arithmetic operator, 20, 372, 406 printf command and, 477 as text character, 558 periods (.). See dots Perl modules, Dirvish and, 869 scripts, 67, 295, 515 permissions. See file permissions pg command, 12, 516, 523. See also more command AIX/HP-UX systems and, 516, 523 physical partitions (PP), 677 loop to show number of stale, 682 physical volumes (PVs), 677 hdisk5, PV statistics for, 684 statistics, for hdisk5 PV, 684 PID (process ID), 144 current (\$\$), RANDOM shell variable and, 269, 311, 371, 379, 389, 401 of last background process, \$! operator and, 44, 144, 145, 147, 546, 547 proc_watch.ksh shell script and, 346 pseudo-random numbers and, 371 ping command, 49, 217, 218, 247, 256, 527, 547, 548. See also automated hosts pinging; monitoring processes/applications AIX, 724 /etc/hosts file and, 737 HP-UX, 724 Linux, 724 OpenBSD, 724 Solaris, 724 ping_host function, 49, 733, 966, 972 \$PINGLIST, 733, 734 variable-length-limit problem, 736 ping_nodes function, 733-734, 972 pingnodes.ksh shell script, 728-733, 964 in action, 735-736 analysis, 733-735 automated execution, with cron table entry, 739 creation, 725-728 functions in, 733-734 logging capability, 737-738 notification method, 738-739 setting trap, 728 variables in, 725-727 pipe character (|) & (piping to background), 36, 351, 364 || (logical OR) operator, 13, 20, 570, 687, 705, 813

bitwise OR operator, 20 escaped, 3, 4 pipes, 13 cat while read LINE, 74 character limit, 128, 129 style in shell script v., 6 plus sign (+)++ auto-increment operator, 20 addition operator, 20 escaped, 3, 4 as "greater than" specifier, 326 unary operator, 20 positional parameters, 15, 458 access value of, 59 shift command and, 16-17, 458-459 POSIX C type notation, 637, 744 post_event function, 195, 968 post event script function, 364, 969 post-FTP event processing, 187, 190, 192, 193, 195 post_processing_fixed_records.dat file listing, 181-182 post_processing_variable_records.dat file listing, 182 - 183PostScript Printer Description (PPD), 820 pound sign (#). See hash mark PP. See physical partitions PPD. See PostScript Printer Description PQ all in one.ksh shell script, 965 PQ_UP_manager.ksh shell script, 839-846 analysis, 847-849 pr command, 12 pre_event function, 195, 968 pre_event_script function, 364, 969 pre-FTP event processing, 187, 190, 192, 193, 195 print commands (list), 13 print working directory command. See pwd command printf command number base conversions, 56-57, 476-477 percent sign (%) and, 477 typeset command v., 477 printing_only_UP_Linux.ksh shell script, 831-832, 965 print-queue management, 50, 809-850 AIX classic printer subsystem, 50, 810-814 print-control commands, 810-820 System V printing, 814-820 CUPS, 50, 809, 810, 820-823 HP-UX print-control commands, 823-825 Linux print-control commands, 825-833 PQ UP manager.ksh shell script, 839-846 analysis, 847-849 proactive approach, 809 scheduling and, 849 script improvements exceptions capability, 849 logging with date/time stamps, 849 Solaris print-control commands, 833-839 print_UP_AIX.ksh shell script, 816-818, 965 print_UP_CUPS.ksh shell script, 821-823 print_UP_HP-UX.ksh shell script, 824-825, 965 print UP Linux.ksh shell script, 828-830, 965 print UP SUN.ksh shell script, 836-837, 965

proactive approaches automated event notification, 131. See also automated event notification filesystem monitoring, 600. See also monitoring filesystems paging/swap space monitoring, 638. See also monitoring paging/swap space print-queue management, 809. See also print-queue management process/application monitoring, 549. See also monitoring processes/applications secure password creation, 403. See also pseudo-random passwords system load monitoring, 641. See also monitoring system load proc (Expect operator), 317-318 increment_by_1, 317, 318, 968 process ID. See PID process monitoring. See monitoring processes/applications process_data function, 166, 169, 180 processing files line-by-line. See parsing files line-by-line proc_mon_end function, 346 proc_mon.ksh shell script, 338-342, 958, 961 in action, 342 analysis, 341-342 trap exit function, 342 usage function, 341 proc_mon_start function, 346 proc_wait.ksh script, 338, 958, 961 on web site, 337 proc_watch function, 34, 36, 351, 364, 969 infinite loop, 34, 36, 351, 364 proc_watch.ksh shell script, 343-347, 958, 961 in action, 347 analysis, 346-347 echo command in, 346 PID and, 346 proc mon end function, 346 proc_mon_start function, 346 timestamps and, 346 proc_watch_timed.ksh script, 347-367, 958, 961 in action, 365-366 analysis, 363-365 functions in, 364 modification, 367 proc_watch function in, 364 progress indicators, 43-45, 143-154 echo command and, 143, 144, 145, 148 elapsed time, 44-45, 148-150 improving, 153 rotating line, 43-44, 145-148 rsync and, 143 series of dots, 43, 143-145 prstat command (Solaris), 675 ps -A command, 27 ps auxw command, 27, 642, 675 ps command, 14, 27, 232 switches aux, 336 -ef, 27 -f, 27 -Kf, 27

pseudo-random characters, 384 file filled with, 392-399 pseudo-random number generator (kernel), 369, 370, 376 pseudo-random numbers, 47-48, 369-400, 401-402 as array pointer, 402 creation of 0-32767 range, 371-372 between 1 and user-defined maximum, 372-373 /dev/random, 48, 369, 370, 376-379 /dev/urandom, 48, 370, 376-379 file (filled with random characters), 392-399 fixed-length numbers between 1 and user-defined maximum, 373-376 with od command, 377 functions get random number, 372 in_range_fixed_length_random_number, 373 - 374in_range_fixed_length_random_number _typeset, 379, 384, 388 in range random number, 372-373, 379 in_range_random_number (description of), 372-373, 379, 406-407, 420 list (on wiley web site), 969-970 PID and RANDOM shell variable, 269, 311, 371, 379, 389, 401 shell scripts, 379-384 mk_unique_filename.ksh, 385–392 online, 959 random_file.bash, 69-73, 392-396 random_number.ksh, 379-384 random_number_testing.bash, 377-379 unique filenames, 384-392 pseudo-random passwords, 401–432 arrays, syntax for, 403-404 -m switch, 403, 412, 413, 415, 416, 418, 421 mk_passwd.ksh shell script, 422-431, 959 analysis of, 418-422 building new pseudo-random password code, 420building of/order of execution, 405-422 checking for keyboard file, 419 command-line arguments, testing/parsing of, 414 - 418functions in, 406-414 loading KEYS array, 419-420 overview of ideas for, 402-403 printing manager's report, 421-422 disable, 432 setting trap, 418 variables in, 405 -n switch, 102, 407, 413, 415, 416, 418 proactive approach, 403 pseudo-terminal (pty), 343 pty, 343 put subcommand, 190, 196 put_ftp_files.ksh shell script, 196-199, 957 put_ftp_files_pw_var.ksh shell script, 207–209, 957 getopts and, 218 puts command, 313, 314, 315, 318 PV level, monitoring for stale partitions at, 677, 684 - 687

PVs. See physical volumes pwd (print working directory) command, 12, 326, 328 full pathnames and, 326, 331 pwdadm command, 798, 801

Q

qprt command, 13 question mark (?) \$? (return code checking), 29–30, 183–184, 217 escaped, 3, 4 queuing v. queueing (spelling difference), 820 queuing_only_UP_Linux.ksh shell script, 832–833, 965 QuickPage, 139 quotes. See back tics; double quotes; single quotes

R

-R switch, 47, 172 radioactive decay events, frequency variations of, 369 RAID configurations, 678 random number generator. See pseudo-random number generator random numbers, 369-370. See also pseudo-random numbers true, 369, 401 random passwords. See pseudo-random passwords RANDOM shell variable, 47, 370, 401 current PID (\$\$) and, 269, 311, 371, 379, 389, 401 random file.bash shell script, 69-73, 392-396, 956 in action, 397 analysis, 398 functions in, 398 randomness, 370, 401-402 random_number.ksh shell script, 379-384 functions in get random number, 379 in_range_fixed_length_random_number _typeset, 379, 384, 388 in range random number, 372-373, 379 random number testing.bash shell script, 377-379 raw LVs, 678 RCOUNT, 147 rcp (remote copy), 187. See also rsync disable, 851 read command, 67, 77 read input function, 886-887, 888, 974 ready_to_run function, 968 in rsync daily copy.ksh shell script, 262, 270 READYTORUN_FILE, 255 real memory, 603. See also paging space; swapping space record files, 45-47, 157-185 definition of, 157 fixed-length cut command and, 164 description/definition, 157 example, 158, 165 merge script for, 45-46, 170-171 merge/process, 181 parsing, 164–166 placement of data field within record, 164, 166, 184

post_processing_fixed_records.dat file listing, 181 - 182string length of, 46-47, 171-172 typeset command and, 47 merge/process, based on record-format type, 173 - 183names, added to end of records, 157, 169-170 sed statement character substitution in, 169-170 parsing/processing, 160-164 variable-length awk command and, 164, 167 data fields and their values, 164, 166, 184 description/definition, 157 example, 159-160 field delimiter, knowledge of, 164, 166, 184 merge script for, 46, 171 merge/process, 182 parsing, 166-169 post_processing_variable_records.dat file listing, 182-183 white space in data fields, 169 record-file parsing function, 180 record-processing function, 180 records, 157 fixed-length (sample), 158 regular expressions. See also specific regular expressions + ([0-9])-type, 65, 327, 380, 416 from df command output, 237, 238 extended, egrep and, 13 FS_PATTERN and, 247 string tests and, 61, 364, 380 reject command, 821 relative pathnames, 874, 928 remainder operator (%), 20, 372, 406. See also modulo N arithmetic remote copy. See rcp remote directory listings, ftp and, 190-192. See also automated ftp file transfer get_remote_dir_listing.ksh shell script, 191-192 nlist subcommand, 190–191 remote files downloading from remote system, 192-196 uploading to remote system, 196-199 remote shell. See rsh remote-update protocol, rsync, 51, 219, 220, 232, 233, 235, 237, 247, 251, 867 removing blank lines from file, 58 column headings in command output, 59-60 repeated lines in file, 58 remsh command, 14 repeated lines, removal of, 58 replicating data, 51-53. See also rsync replicating multiple directories (rsync), 223-237 generic_DIR_rsync_copy.Bash shell script, 223-233 analysis, 230-233 elapsed time function in, 230 with files existing at target, 235-237 verify_copy function in, 230 when files do not exist on target, 233-235 replicating multiple filesystems (rsync), 237–251 generic_FS_rsync_copy.Bash script, 238-247 performing initial copy, 247-249 when files exist on target, 249-251

replicating Oracle databases (rsync), 219, 251 - 289master database server filesystem layout, 252 OLTP database server filesystem layout, 252-253 Oracle DBA Team's script and, 251, 256, 257, 269, 275 rsync_daily_copy.ksh shell script, 257-269, 957 in action, 276-289 analysis of, in logical execution order, 269-275 code to check for remaining rsync sessions, 266, 271-272 code to verify completion of remote rsync sessions, 268, 273 code to verify remote rsync sessions completed, 268, 273 copy internal script log file (\$LOGFILE), 269 elapsed_time function in, 262, 275 example code, 151–152 executed via root cron table, 254 file and variable definitions, 254-255 final verification and notification code, 268, 274 loop to monitor remaining rsync sessions, 267, 272 - 273loop to start all rsync sessions, 266, 271 loop to wait for startup file, 264, 270 ready_to_run function in, 262, 270 script code to check for other sessions, 263, 269 setting trap for exit signals, 269 verify_copy function in, 274-275 restart command, 530 resyncing disks, 677, 687-694. See also monitoring for stale disk partitions return code (\$?), checking, 29-30, 183-184, 217 return command, 10, 11 reverse video, 49, 50, 515, 516-517. See also hgrep soft, 516 .rhosts file, 23 > (right angle bracket) >> (appends to end of file), 13 >> bitwise shift right operator, 20 >= (greater than or equal to) operator, 20 escaped, 3, 4 greater than operator, 20 rm command, 12, 13 RN variable, 48, 376, 389, 406 root access, sudo and, 721 root password, 410, 777 broot script and, 948 changing, 948, 953 sudo and, 778, 798, 806 Rosetta Stone for UNIX, 775 rotate function, 145-146, 967 in shell script, 147 rotate_line function, 43-44, 966 in shell script, 44 rotating line method, 43-44, 145-148 rotating passwords, 401 RPM packages, 643. See also apt; yum RSA keys, 24, 210 rsh (remote shell), 14, 23-24. See also OpenSSH disable, 530, 851, 853 OpenSSH v., 24, 274, 289 remote monitoring with, 530-536 rsync data transfer and, 274, 289 rstatd, 853

rsync, 51-53, 219-290 -a switch, 51, 220, 232 checksum search algorithm, 219, 289 compression of files with, 51, 53, 187, 219, 220, 221, 232, 289 Dirvish and, 289. See also Dirvish generic_DIR_rsync_copy.Bash shell script, 223-233, 957 analysis, 230-233 elapsed time function in, 230 with files existing at target, 235-237 verify_copy function in, 230 when files do not exist on target, 233-235 generic_FS_rsync_copy.Bash script, 238-247, 957 performing initial copy, 247-249 when files exist on target, 249-251 generic_rsync.Bash script, 52–53, 221–222, 955, 957 manual page, 289 progress indicators and, 143 remote-update protocol, 51, 219, 220, 232, 233, 235, 237, 247, 251, 867 replicating data, 51–53 multiple directories, 222-237 multiple filesystems, 237-251 Oracle databases, 219, 251-289 rsh and, 274, 289 rsync_directory_list.lst file (example), 222 shell script output, with end-user feedback, 152, simple generic shell script, 52–53 syntax, 219-220 temporary files, 235 / (trailing forward slash) and, 51, 52, 53, 220, 221, 222 -v switch, 51, 220, 232 -z switch, 51, 219, 220, 232 rsync_daily_copy.ksh shell script, 257-269, 957 in action, 276-289 analysis of, in logical execution order, 269-275 code to check for remaining rsync sessions, 266, 271-272 code to verify remote rsync sessions completed, 268, 273 copy internal script log file (\$LOGFILE), 269 elapsed_time function in, 262, 275 example code, 151-152 executed via root cron table, 254 file and variable definitions, 254-255 final verification and notification code, 268, 274 loop to monitor remaining rsync sessions, 267, 272-273 loop to start all rsync sessions, 266, 271 loop to wait for startup file, 264, 270 ready_to_run function in, 262, 270 script code to check for other sessions, 263, 269 setting trap for exit signals, 269 verify_copy function in, 274-275 rsync_directory_list.lst file (example), 222 RSYNCFAILED_FILE, 255 run_all function, 878-879, 974 Runas_Alias, 797 runaway process, 641, 675 run backup function, 879-880, 975

www.it-ebooks.info

rwall command, 14, 27

S

sample /etc/sudoers file #1,791-794 # 2, 794-797 sample.syslog.conf, 804-805 SAP, 603, 641 sar (system activity report) command, 14 case statement (fields of data), 651 command statement, 662-664 output (system load monitoring) AIX, 649 common denominator for data, 650-651 HP-UX, 649-650 Linux, 650 Solaris, 650 script for system load monitoring, 660-665 sysstat package and, 642 Sarbanes-Oxlev (SOX) Act of 2002, 209, 851 Sarbanes-Oxley (SOX) audits, 209, 530, 851-865, 943. See also monitoring/auditing user's keystrokes internal, 852 OpenSSH and, 851 "real," 852 shell scripts chk_passwd_gid_0.Bash, 860-861, 966 search group id.Bash, 854-855, 965, 966 system-specific tasks, 853-854 administrative accounts, review of, 853 central audit server for syslog files, 854 disable non-encrypted communications, 853 disable old accounts, 853 disable remote login, 853 file permissions (correct settings), 853 password security, 853 patches/upgrades, 854 sticky bit, 12, 21, 854 Tripwire, 854 useful commands awk/cut commands, 856-861 basename command, 863-864 dirname command, 863-864 find command, 855-856 id command, 854-855 sed command, 862-863 what to expect, 852 working with auditors, 852-853 sar_loadmon.ksh shell script, 660-662, 964 in action, 665–666 sar command statement, 662-664 scale, 433, 434, 443, 450, 452, 458, 467, 626. See also bc utility SCALE variable, 439, 450, 458 Schultz, J.W., 867 scope, of variables, 15, 199, 473, 576 scp (Secure Copy), 187. See also rsync expect script and, 210 script command, 53, 935 autoexpect v., 213, 296-297 syntax, 936 script session, command-line, 936-937 script.exp, 540-543 SCRIPT_NAME variable, 379, 380, 439 script.stub shell script, 7-8, 955 sdiff command, 13

SEARCH DIR, 255 search group id.Bash shell script, 854-855, 965, 966 SEC. See Securities and Exchange Commission SECONDS shell variable, 148, 150, 367, 398 Secure Copy. See scp secure ftp. See sftp Secure Shell. See ssh Securities and Exchange Commission (SEC), 209, 851 sed statement, 14, 50, 469, 556, 862-863 character substitution, record file names, added to end of records. 169-170 command substitution within, 515, 516-517, 525 hgrep and, 515, 516-517 image summary file and, 883 removing blank lines with, 58 SOX audits and, 862-863 syntax, 516 seed, 370, 401. See also PID select command, 56-58 select shell command, 878 select system info menu.Bash, 56, 955-956 semicolon (;) array element, 404 escaped, 3, 4 send command, 294 sendmail alias, 132, 134 sendmail command, 41-42, 734, 735 location of HP-UX, 42, 137, 138 OpenBSD, 42, 137, 138 SunOS, 42, 137, 138 mail command v., 734 outbound email and, 137-138 sendmail.cf file, 135, 136 send_notification function, 42, 138, 734, 735 Serial Storage Architecture. See SSA series-of-dots method, 43, 143-145 background function and, 145 elapsed-time method with, 151-153 looping in background, 144 servers, vaults as, 877 set -A command, 60, 403, 404 set command (Expect), 304, 305 sftp (secure ftp), 187, 210. See also rsync encryption keys, creation of, 210 expect script and, 210 ftp v., 209 here document and, 188, 210 no-password Secure Shell access, 210-211 syntax, 211 scp syntax v., 211-212 sftp-scp-getfile.Bash shell script, 211-212, 957 sgid, 12 file permissions and, 21-23 shared vault, 869 shell scripts , 4, 955-966. See also specific shell scripts (list), 955–966 basic concept, 4 case sensitive, 3 commands (list), 12-14 comments in, 6-8, 693 in curly braces, 230 declaring shells in, 6

shell scripts (continued) executable, 21. See also file permissions functions written in, 4-5 interpreted, 4, 756 math in. See bc utility notification in. See automated event notification password for, 217 running, 5 starter file, 7-8, 955 style in, 6-8 user feedback and, 494, 693. See also progress indicators shells, 4 arrays. See arrays Bash, 4 Bourne, 4 C, 6 declaring, in shell script, 6 Korn, 4 location of, 4 shell_script_name, 5 shift command, 16-17, 458-459, 499-500 show all instances status function, 537-538 show_oratab_instances function, 536-537 showplatform -v Blade Chassis command, 318, 319, 320, 321 showplatform.exp Expect script, 318-320, 958 in action, 320-321 findslot Bash script and, 321-322, 958 SIGHUP, 25 SIGINT, 25 SIGKILL, 25 SIGQUIT, 25 SIGSTOP, 25 SIGTERM, 25 silent mode, 32 silent running, 32-33 Simple Network Management Protocol. See SNMP simple_SQL_query function, 538 sin function, 21 single quotes ('), 18 around square brackets, 28, 510 escaping special characters with, 4 sinh function, 21 sleep command, 13, 144 SMS Client, 139 snapshots. See Dirvish; system-configuration snapshot sniffer, 530 SNMP (Simple Network Management Protocol), 140 traps, 139-140 APIs and, 548 soft reverse video, 516 software keys. See license keys Solaris configuring sudo on, 778 df -k command output, 587-588 output columns of interest, 588 iostat command output, 647 nawk command, 19, 168, 503, 509, 537 ping command, 724 ping_host function and, 49 print-control commands, 835-839

prstat command, 675 sar command output, 650 scripts for filesystem monitoring, 963. See also monitoring filesystems sendmail command, location of, 42, 137, 138 swap command, 607 -s output, 625 swap-space monitor, 625-630 wilma machine, 196 Solaris printing function, 847, 974 some_command 2>&1,68 SOX. See Sarbanes-Oxley spawn command, 294, 295 switch option, 323 telnet and, 294, 295 special characters (list), 3-4 escaping, 3–4 special parameters, 17-18 ''\$*'', 17 ''\$@'', 17 \$*, 17-18 \$@, 17 specific files. See large files spell command, 13 SQL query function, 538 SQL script, my_sql_query.sql, 538 SQL+Oracle query, 539 sqlplus command, 538, 539 sqrt function, 21 square brackets ([]) escaped, 3, 4 mathematical expression in, 316 single quotes around, 28, 510 test command, 15 SSA (Serial Storage Architecture) disk identification, 694-695, 697-722 command syntax, 698 error log, 721 functions (list), 699 control, 703-708 usage/user feedback, 699-703 hdisks/pdisks, 694, 695, 697, 698 cross-reference script, 721 translating, 698 root access and sudo, 721 shell script, 709-719, 964 analysis, 720-721 SSAidentify.ksh shell script, 709-719, 964 analysis, 720-721 functions in, 699-708 ssaidentity command, 698 ssaxlate command, 694, 695, 698, 705, 720 ssh (Secure Shell), 210. See also OpenSSH example of running remote command, 535–536 importance of, 530 no-password access, 210-211, 530 sample secure shell login, 535 tunnel, 530, 535 SSH login to known host (Expect), 307 SSH login to unknown host (Expect), 306-307 ssh-keygen, 24, 25, 210 stale disk partitions, 48-49, 677-696 stale LV mon.ksh shell script, 682-683, 964 stale PP mon.ksh shell script, 685-686, 964

stale VG PV LV PP mon.ksh shell script, 688-692,964 analysis of, 692-694 command summary for, 688 standard error. See stderr standard input. See stdin standard output. See stdout startup_event_script function, 364, 969 stderr (standard error), 68, 81-82 stdin (standard input), 68, 81-82 stdout (standard output), 68, 81-82 Steven, Heiner, 952 sticky bit, 12, 21, 854 stringing together commands. See pipes strings comparison (<>), 15 length, 46-47, 157, 171, 172. See also fixed-length record files testing, 61-65 test string function and, 364, 969 style, in shell scripts, 6-8 su command, 14, 24, 311, 327, 530, 798, 853, 937, 951 hyphen and, 951 sudo and, 798, 937 subtraction, in shell script, 443–452 subtraction operator (-), 20 sudo (superuser do), 327, 721, 775, 777-807 adding, to syslogd, 805 broot shell script, /etc/sudoers file, 946-948, 953 compiling, 779-790 configure command (command output), 782-787 make command (command output), 787-790 untarring the sudo distribution, 779-782 configuring, 790-797 on Solaris, 778 downloading, 778-779 log file, 806 logging to syslog with, 801-805 need for, 777-778 root access and, 721 sample.syslog.conf, 804-805 SSA disk identification and, 721 su command and, 798, 937 syslogd, /etc/syslog.conf file and, 805 using, 797-798 in shell script, 798-801 -V output (executing with root authority), $\hat{802} - 804$ sudoers file, /etc/sudoers file, 946-948, 953 sample # 1, 791-794 sample # 2, 794-797 Sudoer's Manual, 807 suid, 12 file permissions and, 21–23 Sun Blade Chassis Expect scripts with, 318-323 showplatform -v Blade Chassis command, 318, 319, 320, 321 Sun Microsystems JumpStart, 322-323. See also Expect scripts SunOS. See Solaris SUN_swap_mon.ksh shell script, 627-629, 963 in action, 629–630 creation of, 625-627

superuser do. See sudo swap command (Solaris), 607 -s output, 625 swapctl command (OpenBSD), 606 -lk command output, 622 swapinfo command (HP-UX), 605-606 -tm command, 613 output, 613-614 swapping space, 603. See also monitoring paging/swap space paging space v., 603 switches (command-line). See also specific switches colon placement in list of, 450 double quotes around list of, 450 symbol commands (list), 14-15 syntax. See specific syntaxes syslog, 777 files, central audit server for, 854 logging to, with sudo, 801-805 syslogd, 801, 802, 805 adding sudo to, 805 reread /etc/syslog.conf file, 805 sysstat package, 642–643 system activity report command. See sar command system crashes, 133, 601, 603, 641, 952 system load, monitoring. See monitoring system load system noise, 376, 401 System V printing (AIX), 50, 814-820 system-configuration snapshot (AIX), 741-775 AIXsysconfig.ksh shell script, 745-756, 965 in action, 757-774 analysis, 756-757 database/application-level statistics and, 774 functions (list) in, 973 command syntax, 742-744 commands for AIX (list), 742-744

Т

\t (backslash operator), 19, 700 tail command, 13, 59, 608, 811, 828, 849 talk command, 14, 27 talkd, 853 tan function, 21 tanh function, 21 Tcl. See also Expect Expect and, 293 installation of, 291-293 tcpdump, 530 tee command, 133, 343, 849 -a, 275, 638, 694, 830 TelAlert, 139 telnet, 24, 320, 323, 530. See also OpenSSH disable, 530, 851, 853 spawn and, 294, 295 temporary files, rsync, 235 terabyte disk drives, 600, 601 test command [], 15 testing command-line arguments, 414-416 data integrity in floating-point math shell scripts. See bc utility for floating-point numbers/integers, 440-441 for NULL variables, 58-59

outbound email, 134-135

testing (continued) strings, 61-65 uniq command and, 58 uppercase/lowercase text and, 28–29 test_string function, 364, 969 test_string.ksh shell script, 61-65, 956 tftpd, 853 thrashing condition, 603 three-dimensional arrays, 60, 403. See also arrays tics. See back tics; double quotes; single quotes tilde (\sim) binary inversion, 20 escaped, 3, 4 time command (built-in shell), 74, 99, 232, 271, 399 /usr/bin/time command v., 232, 271, 399 time method, elapsed, 44-45, 148-150 time-based script execution, 30-32 timestamps, proc_watch.ksh shell script and, 346 timing, in process monitoring, 335. See also monitoring processes/applications timing_test_function.bash script, 149-150, 956 Tivoli NetView, 140 TMOUT environment variable, 936, 945, 954 TOKEN variable, 458 tokens, 16. See also shift command top command (Linux), 641, 675 example output, 641 top level down, scope and, 576 topas command (AIX), 675 top-like monitoring tools, 641, 675 prstat command (Solaris), 675 ps auxw command, 27, 642, 675 top command (Linux), 641, 675 example output, 641 topas command (AIX), 675 TOTAL_NUMBERS variable, 461, 467 touch command, 192 tput command, 50, 516, 798, 801 options, 516 (list), 524-525 rmso, 516, 525, 612 sgr0, 516, 525 smso, 516, 524, 525, 612 tr command, 28-29, 510 trailing forward slash (/) relative pathnames and, 928 rsync and, 51, 52, 53, 220, 221, 222, 231 transfer of files. See automated FTP file transfer trap command, 364-365 trap_exit function, 36, 342, 418, 427, 693 description of, 414 in mk_passwd.ksh shell script, 418, 427 in proc mon.ksh shell script, 342 in proc_watch_timed.ksh script, 364 traps, 25, 418, 439 kill -9 command and, 25, 36, 269, 351, 418, 438, 440, 702, 942 rsync_daily_copy.ksh shell script and, 269 setting, 418 SNMP, 139-140 APIs and, 548 trigger threshold, 139, 327, 328, 332, 556, 566, 574, 612, 613, 617, 655 Tripwire, 854 true random numbers, 369, 401. See also pseudo-random numbers

tst_ftp.ksh shell script, 189-190, 956 tty command, 343 tty device, 343, 346 tunnel, SSH, 530, 535 turning on/off SSA identification lights, 697–722 24_ways_to_parse.ksh shell script, 99–116, 956 twirl function, 699, 701-702, 972 two-dimensional arrays, 60, 403. See also arrays appearance of, 833 typeset command, 28–29, 172 -i DAY, 255 number base conversions and, 55, 475-476 options for, 47, 172 pad number with leading zeros, 375 printf command v., 477 in random-number function, 375 string length of record files and, 47, 172 uppercase/lowercase text and, 28-29, 510

U

-u switch, 47, 172 uname command, 14, 584, 637, 847 all-in-one paging/swap space monitor and, 630, 637 command result, 584 function result, 584 usefulness of, 739 unary minus operator (-), 20 unary operator (+), 20 uniq command, 58 unique filenames, 384–392 date/time stamp and, 371, 384, 388, 389 grep command and, 384 UNIX, flavors of. See AIX; HP-UX; Linux; OpenBSD; Solaris unknown host message, 306, 734, 738 untarring sudo distribution, 779-782 until loop, 36-37 until statement, 9 upgrades/patches, SOX audits and, 854 uploading remote files to remote system, 196-199 put_ftp_files.ksh shell script, 196-199, 957 put_ftp_files_pw_var.ksh shell script, 207-209, 957 uppercase tr/typeset commands for, 28–29, 510 variables in, 15 uptime command, 26, 337 script for system load monitoring, 655-659 system load monitoring common denominator for output, 645 output, 644-645 syntax, 644 uptime fieldtest.ksh shell script, 964 uptime_loadmon.Bash shell script, 655-658 in action, 659 analysis, 658-659 uptime_loadmon.ksh shell script, 964 usage error, 180, 331, 364, 383, 384, 413 usage function, 34, 117, 195, 331, 341, 364, 388, 413, 439, 699 description of, 413 in proc_watch_timed.ksh script, 364 with single echo command, 699-700

usage_error function, 195

user, effective, 948 user feedback. *See also* progress indicators shell scripts and, 494, 693 User_Alias, 797, 947, 948 user-communication commands, 27–28 user-information commands, 25–26. *See also* last command; w command; who command user's keystrokes (monitoring/auditing), 53–54, 935–954 /usr, 559 /usr/bin/time command, 232, 271, 399. *See also* time command uucp, 853

V

\v (backslash operator), 19 -v rsync switch, 51, 220, 232 -v switch ftp command, 189 mail command, 134, 135 verbose mail mode with, 135 verbose mode with, 189, 338 VAR, 46, 171, 183, 304 variable-length record files awk command and, 164, 167 data fields and their values, 164, 166, 184 description/definition, 157 example, 159-160 data in fields, 159 fields in, 159 field delimiter, knowledge of, 164, 166, 184 merge script for, 46, 171 merge/process, 182 parsing, 166-169 while_read_LINE_bottom_FD_OUT and, 160, 167 - 168post_processing_variable_records.dat file listing, 182 - 183variable-replacement technique, 199-203 variables, 15. See also specific variables \$ and name of, 15, 28, 304 defined, 15 double quotes around, 18, 50, 160, 169, 247, 516, 517, 570 Expect scripts and, 304-306 global, 255, 473, 576 in lowercase, 15 managed, 139 null, 58-59 scope of, 15, 199, 473 in uppercase, 15 with white space, 18, 169, 247, 516, 517 varyonvg command, 692 vaults, 869 adding new backup, 884-889 examples, 925-930 removing backup, 889-890 example, 930 as servers/filesystems, 877, 880 shared, 869 verbose mail mode, with -v switch, 135 verbose mode, with -v switch, 189, 338 verify_copy function, 968 in generic_DIR_rsync_copy.Bash shell script, 230 in rsync_daily_copy.ksh shell script, 274-275

Veritas filesystem, 697 VG/LV/PV levels, , monitoring for stale partitions at, 677, 687-694 VGs. See volume groups vi command, 14, 794 vi editor, 14, 30, 516, 721, 791, 947 visudo and, 721, 791, 947 virtual memory statistics command. See vmstat command visudo, 721, 791, 797, 807, 947 Visudo Manual, 807 vmstat command case statement (fields of data), 653 - 654script for system load monitoring, 670-674 system load monitoring common denominator for output data, 653-654 output on UNIX flavors, 651-653 syntax, 651 vmstat (virtual memory statistics) command, 14, 337 vmstat loadmon.ksh shell script, 671-673, 964 in action, 674 analysis, 673-674 volume groups (VGs), 678

W

w command, 13, 26 remotely execute, with expect script, 543-545 wall command, 14, 27 wc command, 13 weak passwords, 401, 403. See also pseudo-random passwords while loop, 68, 74. See also parsing files line-by-line in background, 144 case statement inside, 146 Expect's version of, 314-315 curly braces in, 314 getopts and. See getopts command infinite, 43, 144, 147 while statement, 9 while_LINE_line_bottom, 77-78 while LINE line_bottom_cmdsub2, 79 while LINE line bottom cmdsub2 FD OUT, 87 while_LINE_line_bottom_FD_OUT, 86 while_LINE_line_cmdsub2_FD_IN, 93-94 while_LINE_line_cmdsub2_FD_IN_AND_OUT, 97-98 while_LINE_line_FD_IN, 92-93 while LINE line FD IN AND OUT, 96-97 while line outfile, 81 while_line_outfile_FD_IN, 89-90 while_line_outfile_FD_IN_AND_OUT, 91-92 while line outfile FD OUT, 90-91 while_read_LINE_bottom, 40, 75-76, 164 fastest method (not using file descriptors), 40, 164while read LINE bottom FD_OUT, 37-38, 85, 160 easiness of understanding, 39, 162 first place (tie) in timing tests, 37, 160 parsing fixed-length record files, 160, 164-166 variable-length record files, 160, 167-168 while read LINE FD IN, 81-83

while_read_LINE_FD_IN_AND_OUT, 38-39, 94-96, 161 difficulty of understanding, 39, 161 first place (tie) in timing tests, 38, 161 while_test.exp Expect script, 314 in action, 314–315 white noise, frequency variations of, 369 white space in data fields, 169 variables with, 18, 169, 247, 516, 517 who am I command, 14 who command, 13, 26 remotely execute, with expect script, 543-545 whoami command, 14 wildcards, 12, 31, 214, 298, 299, 319, 797, 830 *, 31, 830 ?, 31 wiley.com/go/michael2e,7 functions (list), 966-975 shells scripts (list), 955-966

WORK_DIR, 255 world-writable files, 853 write command, 14, 27

X

-x switch, 47, 172

Y

yum, 291 apt v., 643 Dirvish installation with, 869–871 manual pages, 293 sysstat package installation with, 642–643 Tcl and Expect installation with, 291–293

Z

-z rsync switch, 51, 219, 220, 232 -z switch, 47, 172 -Zn switch, 47, 172