

The Linux[®] Command Line

William E. Shotts, Jr.

Copyright 2008-2009, William E. Shotts, Jr.

This work is licensed under the Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/3.0/us/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Linux[®] is the registered trademark of Linus Torvalds. All other trademarks belong to their respective owners.

This book is part of the LinuxCommand.org project, a site for Linux education and advocacy devoted to helping users of legacy operating systems migrate into the future. You may contact the LinuxCommand.org project at <http://linuxcommand.org>.

Release History

Version	Date	Description
09.07	July 18, 2009	Completed first draft

Table of Contents

Part 1 - Introduction.....	1
1 - Introduction.....	2
Why Use The Command Line?.....	2
What This Book Is About.....	3
Who Should Read This Book.....	3
What's In This Book.....	4
How To Read This Book.....	5
Prerequisites.....	5
Why I Don't Call It "GNU/Linux".....	5
Acknowledgments.....	6
Further Reading.....	6
Part 2 - Learning The Shell.....	7
2 - What Is The Shell?.....	8
Terminal Emulators.....	8
Your First Keystrokes.....	8
Command History.....	9
Cursor Movement.....	9
A Few Words About Mice And Focus.....	9
Try Some Simple Commands.....	10
Ending A Terminal Session.....	11
The Console Behind The Curtain.....	11
Further Reading.....	11
3 - Navigation.....	12
Understanding The File System Tree.....	12
The Current Working Directory.....	13
Listing The Contents Of A Directory.....	14
Changing The Current Working Directory.....	14

Absolute Pathnames.....	14
Relative Pathnames.....	15
Some Helpful Shortcuts.....	16
Important facts about file names.....	16
4 - Exploring The System.....	18
More Fun With ls.....	18
Options And Arguments.....	19
A Longer Look At Long Format.....	20
Determining A File's Type With file.....	21
Viewing File Contents With less.....	22
What Is "Text".....	22
Less Is More.....	24
A Guided Tour.....	24
Symbolic Links.....	27
Hard Links.....	28
5 - Manipulating Files And Directories.....	29
Wildcards.....	29
Character Ranges.....	31
Wildcards Work In The GUI Too.....	31
mkdir – Create Directories.....	31
cp – Copy Files And Directories.....	32
Useful Options.....	32
Examples.....	33
mv – Move And Rename Files.....	33
Useful Options.....	34
Examples.....	34
rm – Remove Files And Directories.....	35
Useful Options.....	35
Examples.....	35
Be Careful With rm!.....	36
ln – Create Links.....	36
Hard Links.....	37
Symbolic Links.....	37
Let's Build A Playground.....	38
Creating Directories.....	38
Copying Files.....	38
Moving And Renaming Files.....	39
Creating Hard Links.....	40
Creating Symbolic Links.....	41
Removing Files And Directories.....	43
Summing Up.....	44

6 - Working With Commands.....	45
What Exactly Are Commands?.....	45
Identifying Commands.....	46
type – Display A Command's Type.....	46
which – Display An Executable's Location.....	46
Getting A Command's Documentation.....	47
help – Get Help For Shell Builtins.....	47
--help – Display Usage Information.....	48
man – Display A Program's Manual Page.....	48
apropos – Display Appropriate Commands.....	50
whatis – Display A Very Brief Description Of A Command.....	50
The Most Brutal Man Page Of Them All.....	50
info – Display A Program's Info Entry.....	51
README And Other Program Documentation Files.....	52
Creating Your Own Commands With alias.....	52
Revisiting Old Friends.....	55
7 – Redirection.....	56
Standard Input, Output, And Error.....	56
Redirecting Standard Output.....	57
Redirecting Standard Error.....	59
Redirecting Standard Output And Standard Error To One File.....	59
Disposing Of Unwanted Output.....	60
/dev/null In Unix Culture.....	60
Redirecting Standard Input.....	60
cat – Concatenate Files.....	60
Pipelines.....	62
Filters.....	63
uniq - Report Or Omit Repeated Lines.....	63
wc – Print Line, Word, And Byte Counts.....	63
grep – Print Lines Matching A Pattern.....	64
head / tail – Print First/Last Part Of Files.....	65
tee – Read From Stdin And Output To Stdout And Files.....	66
Summing Up.....	67
Linux Is About Imagination.....	67
8 - Seeing The World As The Shell Sees It.....	68
Expansion.....	68
Pathname Expansion.....	69
Tilde Expansion.....	70
Arithmetic Expansion.....	70
Brace Expansion.....	71
Parameter Expansion.....	72

Command Substitution.....	73
Quoting.....	74
Double Quotes.....	74
Single Quotes.....	76
Escaping Characters.....	77
Backslash Escape Sequences.....	77
Summing Up.....	78
9 - Advanced Keyboard Tricks.....	79
Command Line Editing.....	79
Cursor Movement.....	79
Modifying Text.....	80
Cutting And Pasting (Killing And Yanking) Text.....	80
The Meta Key.....	81
Completion.....	81
Programmable Completion.....	83
Using History.....	83
Searching History.....	83
History Expansion.....	85
Summing Up.....	86
10 – Permissions.....	87
Owners, Group Members, And Everybody Else.....	88
Reading, Writing, And Executing.....	89
chmod – Change file mode.....	91
What The Heck Is Octal?.....	91
Setting File Mode With The GUI.....	94
umask – Set Default Permissions.....	95
Some Special Permissions.....	96
Changing Who You Are.....	97
su – Run A Shell With Substitute User And Group IDs.....	98
sudo – Execute A Command As Another User.....	99
Ubuntu And sudo.....	100
chown – Change File Owner And Group.....	101
chgrp – Change Group Ownership.....	102
Exercising Our Privileges.....	102
Changing Your Password.....	105
11 - Processes.....	106
How A Process Works.....	106
Viewing Processes.....	107
Viewing Processes Dynamically With top.....	109
Controlling Processes.....	111
Interrupting A Process.....	112

Putting A Process In The Background.....	112
Returning A Process To The Foreground.....	113
Stopping (Pausing) A Process.....	113
Signals.....	114
Sending Signals To Processes With kill.....	115
Sending Signals To Multiple Processes With killall.....	118
More Process Related Commands You May Enjoy.....	118

Part 3 - Configuration And The Environment.....119

12 - The Environment.....120

What Is Stored In The Environment?.....	120
Examining The Environment.....	120
Some Interesting Variables.....	122
How Is The Environment Established?.....	123
What's In A Startup File?.....	123
Modifying The Environment.....	125
Which Files Should We Modify?.....	125
Text Editors.....	125
Using Your Text Editor.....	126
Why Comments Are Important.....	129
Activating Our Changes.....	130
Summing Up.....	130

13 - A Gentle Introduction To vi.....131

Why You Should Learn vi.....	131
A Little Background	132
Starting And Stopping vi.....	132
Compatibility Mode.....	133
Editing Modes.....	134
Entering Insert Mode.....	135
Saving Our Work.....	135
Moving The Cursor Around.....	136
Basic Editing.....	137
Appending Text.....	137
Opening A Line.....	138
Deleting Text.....	138
Cutting, Copying And Pasting Text.....	140
Joining Lines.....	141
Search And Replace.....	142
Searching Within A Line.....	142
Searching The Entire File.....	142
Global Search And Replace.....	143

Editing Multiple Files.....	144
Switching Between Files.....	145
Opening Additional Files For Editing.....	146
Copying Content From One File Into Another.....	147
Inserting An Entire File Into Another.....	148
Saving Our Work.....	149
Additional Reading.....	149
14 - Customizing Your Prompt.....	151
Anatomy Of A Prompt.....	151
Trying Some Alternate Prompt Designs.....	153
Adding Color.....	154
Terminal Confusion.....	154
Moving The Cursor.....	156
Saving The Prompt.....	158
Summing Up.....	158
Further Reading.....	158
Part 4 - Common Tasks And Essential Tools.....	159
15 – Package Management.....	160
Packaging Systems.....	160
How A Package System Works.....	161
Package Files.....	161
Repositories.....	161
Dependencies.....	162
High And Low-level Package Tools.....	162
Common Package Management Tasks.....	163
Finding A Package In A Repository.....	163
Installing A Package From A Repository.....	163
Installing A Package From A Package File.....	164
Removing A Package.....	164
Updating Packages From A Repository.....	165
Upgrading A Package From A Package File.....	165
Listing Installed Packages.....	165
Determining If A Package Is Installed.....	166
Displaying Info About An Installed Package.....	166
Finding Which Package Installed A File.....	166
Summing Up.....	167
The Linux Software Installation Myth.....	167
Additional Reading.....	168
16 - Storage Media.....	169

Mounting And Unmounting Storage Devices.....	169
Viewing A List Of Mounted File Systems.....	171
Why Unmounting Is Important.....	174
Determining Device Names.....	175
Creating New File Systems.....	178
Manipulating Partitions With fdisk.....	178
Creating A New File System With mkfs.....	180
Testing And Repairing File Systems.....	181
What The fsck?.....	182
Formatting Floppy Disks.....	182
Moving Data Directly To/From Devices.....	183
Creating CD-ROM Images.....	184
Creating An Image Copy Of A CD-ROM.....	184
Creating An Image From A Collection Of Files.....	184
A Program By Any Other Name.....	185
Writing CD-ROM Images.....	185
Mounting An ISO Image Directly.....	185
Blanking A Re-Writable CD-ROM.....	185
Writing An Image.....	186
Further Reading.....	186
Extra Credit.....	186
17 – Networking.....	188
Examining And Monitoring A Network.....	189
ping.....	189
traceroute.....	190
netstat.....	191
Transporting Files Over A Network.....	192
ftp.....	193
lftp – A Better ftp.....	195
wget.....	195
Secure Communication With Remote Hosts.....	195
ssh.....	196
Tunneling With SSH.....	199
scp And sftp.....	200
An SSH Client For Windows?.....	201
Further Reading.....	201
18 - Searching For Files.....	202
locate – Find Files The Easy Way.....	202
Where Does The locate Database Come From?.....	204
find – Find Files The Hard Way.....	204
Tests.....	205
Operators.....	207

Predefined Actions.....	209
User Defined Actions.....	212
Improving Efficiency.....	212
xargs.....	213
Dealing With Funny File Names.....	214
A Return To The Playground.....	214
Options.....	217
Further Reading.....	217
19 - Archiving And Backup.....	218
Compressing Files.....	218
gzip.....	219
bzip2.....	221
Don't Compress Again.....	222
Archiving Files.....	222
tar.....	222
zip.....	227
Synchronizing Files And Directories.....	230
Using rsync Over A Network.....	232
Further Reading.....	233
20 - Regular Expressions.....	234
What Are Regular Expressions?.....	234
grep.....	234
Meta Characters And Literals.....	236
The Any Character.....	237
Anchors.....	238
A Crossword Puzzle Helper.....	238
Bracket Expressions And Character Classes.....	239
Negation.....	239
Traditional Character Ranges.....	240
POSIX Character Classes.....	241
Reverting To Traditional Collation Order.....	244
POSIX Basic Vs. Extended Regular Expressions.....	245
POSIX.....	245
Alternation.....	246
Quantifiers.....	247
? - Match An Element Zero Or One Times.....	247
* - Match An Element Zero Or More Times.....	248
+ - Match An Element One Or More Times.....	249
{ } - Match An Element A Specific Number Of Times.....	249
Putting Regular Expressions To Work.....	250
Validating A Phone List With grep.....	250
Finding Ugly Filenames With find.....	251

Searching For Files With locate.....	252
Searching For Text In less And vim.....	252
Summing Up.....	254
Further Reading.....	254
21 - Text Processing.....	255
Applications Of Text.....	255
Documents.....	256
Web Pages.....	256
Email.....	256
Printer Output.....	256
Program Source Code.....	256
Revisiting Some Old Friends.....	256
cat.....	257
MS-DOS Text Vs. Unix Text.....	258
sort.....	258
uniq.....	266
Slicing And Dicing.....	267
cut.....	267
Expanding Tabs.....	270
paste.....	271
join.....	272
Comparing Text.....	274
comm.....	275
diff.....	275
patch.....	278
Editing On The Fly.....	279
tr.....	279
ROT13.....	280
sed.....	281
People Who Like sed Also Like.....	290
aspell.....	290
Summing Up.....	294
Further Reading.....	294
Extra Credit.....	294
22 - Formatting Output.....	295
Simple Formatting Tools.....	295
nl – Number Lines.....	295
fold – Wrap Each Line To A Specified Length.....	299
fmt – A Simple Text Formatter.....	299
pr – Format Text For Printing.....	302
printf – Format And Print Data.....	303
Document Formatting Systems.....	307

groff.....	308
Summing Up.....	314
Further Reading.....	314
23 – Printing.....	315
A Brief History Of Printing.....	315
Printing In The Dim Times.....	315
Character-based Printers.....	316
Graphical Printers.....	317
Printing With Linux.....	318
Preparing Files For Printing.....	318
pr – Convert Text Files For Printing.....	318
Sending A Print Job To A Printer.....	320
lpr – Print Files (Berkeley Style).....	320
lp – Print Files (System V Style).....	321
Another Option: a2ps.....	322
Monitoring And Controlling Print Jobs.....	325
lpstat – Display Print System Status.....	325
lpq – Display Printer Queue Status.....	326
lprm / cancel – Cancel Print Jobs.....	327
Summing Up.....	327
Further Reading.....	327
24 – Compiling Programs.....	328
What Is Compiling?.....	328
Are All Programs Compiled?.....	329
Compiling A C Program.....	330
Obtaining The Source Code.....	330
Examining The Source Tree.....	332
Building The Program.....	334
Installing The Program.....	338
Summing Up.....	338
Further Reading.....	338
Part 5 - Writing Shell Scripts.....	339
25 - Writing Your First Script.....	340
What Are Shell Scripts?.....	340
How To Write A Shell Script.....	340
Script File Format.....	341
Executable Permissions.....	341
Script File Location.....	342
Good Locations For Scripts.....	343

More Formatting Tricks.....	344
Long Option Names.....	344
Indentation And Line Continuation.....	344
Summing Up.....	345
Further Reading.....	345
26 – Starting A Project.....	346
First Stage: Minimal Document.....	346
Second Stage: Adding A Little Data.....	348
Variables And Constants.....	349
Assigning Values To Variables And Constants.....	352
Here Documents.....	353
Summing Up.....	356
Further Reading.....	356
27 – Top-Down Design.....	357
Shell Functions.....	358
Local Variables.....	361
Keep Scripts Running.....	362
Further Reading.....	365
28 – Flow Control: Branching With if.....	366
if.....	366
Exit Status.....	367
Test.....	369
File Expressions.....	369
String Expressions.....	372
Integer Expressions.....	373
A More Modern Version Of test.....	374
Combining Expressions.....	375
Portability Is The Hobgoblin Of Little Minds.....	377
Control Operators: Another Way To Branch.....	378
Summing Up.....	379
Further Reading.....	380
29 – Reading Keyboard Input.....	381
read – Read Values From Standard Input.....	382
Options.....	384
IFS.....	385
You Can’t Pipe read.....	387
Validating Input.....	388
Menus.....	389
Summing Up.....	391
Extra Credit.....	391

Further Reading.....	391
30 – Flow Control: Looping With while / until.....	392
Looping.....	392
while.....	392
Breaking Out Of A Loop.....	395
until.....	396
Reading Files With Loops.....	397
Summing Up.....	398
Further Reading.....	398
31 – Troubleshooting.....	399
Syntactic Errors.....	399
Missing Quotes.....	400
Missing Or Unexpected Tokens.....	400
Unanticipated Expansions.....	401
Logical Errors	403
Defensive Programming.....	403
Verifying Input.....	404
Design Is A Function Of Time.....	405
Testing.....	405
Test Cases.....	406
Debugging.....	407
Finding The Problem Area.....	407
Tracing.....	407
Examining Values During Execution.....	410
Summing Up.....	410
Further Reading.....	411
32 – Flow Control: Branching With case.....	412
case.....	412
Patterns.....	414
Summing Up.....	416
Further Reading.....	416
33 – Positional Parameters.....	417
Accessing The Command Line.....	417
Determining The Number of Arguments.....	418
shift – Getting Access To Many Arguments.....	419
Simple Applications.....	420
Using Positional Parameters With Shell Functions.....	421
Handling Positional Parameters En Masse.....	422
A More Complete Application.....	424
Summing Up.....	427

Further Reading.....	430
34 – Flow Control: Looping With for.....	431
for: Traditional Shell Form.....	431
Why i?.....	433
for: C Language Form.....	434
Summing Up.....	435
Further Reading.....	436
35 – Strings And Numbers.....	437
Parameter Expansion.....	437
Basic Parameters.....	437
Expansions To Manage Empty Variables.....	438
Expansions That Return Variable Names.....	440
String Operations.....	440
Arithmetic Evaluation And Expansion.....	443
Number Bases.....	443
Unary Operators.....	444
Simple Arithmetic.....	444
Assignment.....	446
Bit Operations.....	448
Logic.....	449
bc – An Arbitrary Precision Calculator Language.....	451
Using bc.....	452
An Example Script.....	453
Summing Up.....	454
Extra Credit.....	455
Further Reading.....	455
36 – Arrays.....	456
What Are Arrays?.....	456
Creating An Array.....	456
Assigning Values To An Array.....	457
Accessing Array Elements.....	458
Array Operations.....	460
Outputting The Entire Contents Of An Array.....	460
Determining The Number Of Array Elements.....	460
Finding The Subscripts Used By An Array.....	461
Adding Elements To The End Of An Array.....	461
Sorting An Array.....	462
Deleting An Array.....	463
Summing Up.....	464
Further Reading.....	464

37 – Exotica.....	465
Group Commands And Subshells.....	465
Process Substitution.....	466
Traps.....	468
Temporary Files.....	471
Asynchronous Execution.....	472
wait.....	472
Named Pipes.....	473
Setting Up A Named Pipe.....	474
Using Named Pipes.....	474
Summing Up.....	475
Further Reading.....	475
Index.....	476

Part 1 - Introduction

1 - Introduction

I want to tell you a story.

No, not the story of how, in 1991, Linus Torvalds wrote the first version of the Linux kernel. You can read that story in lots of Linux books. Nor am I going to tell you the story of how, some years earlier, Richard Stallman began the GNU project to create a free Unix-like operating system. That's an important story too, but most other Linux books have that one, as well.

No, I want to tell you the story of how you take back control of your computer.

When I began working with computers as a college student in the late 1970s, there was a revolution going on. The invention of the microprocessor had made it possible for ordinary people like you and me to actually own a computer. It's difficult for many people today to realize what the world was like when only big business and big government ran all the computers. Let's just say, you couldn't get much done.

Today, the world is very different. Computers are everywhere from tiny wristwatches to giant data centers to everything in between. In addition to ubiquitous computers, we also have a ubiquitous network connecting them together. This has created a wondrous new age of personal empowerment and creative freedom, but over the last couple of decades something else has been happening. A single giant corporation has been imposing its control over most of the world's computers and deciding what you can and cannot do with them. Fortunately, people from all over the world are doing something about it. They are fighting to maintain control of their computers by writing their own software. They are building Linux.

Many people speak of “freedom” in regards to Linux, but I don't think most people know what this freedom really means. Freedom is the power to decide what your computer does and the only way to have this freedom is to know what your computer is doing. Freedom is a computer that is without secrets, one where everything can be known if you care enough to find out.

Why Use The Command Line?

Have you ever noticed in the movies when the “super hacker,” -- you know, the guy who can break into the ultra-secure military computer in under thirty seconds -- sits down at

the computer, he never touches a mouse? It's because movie makers realize that we, as human beings, instinctively know the only way to really get anything done on a computer is by typing on a keyboard.

Most computer users today are only familiar with the *graphical user interface* (GUI) and have been taught by vendors and pundits that the *command line interface* (CLI) is a terrifying thing of the past. This is unfortunate because a good command line interface is a marvelously expressive way of communicating with a computer in much the same way as the written word is for human beings. It's been said that “graphical user interfaces make easy tasks easy, while command line interfaces make difficult tasks possible” and this is still very true today.

Since Linux is modeled after the Unix family of operating systems, it shares the same rich heritage of command line tools as Unix. Unix came into prominence during the early 1980s (although it was first developed a decade earlier) before the widespread adoption of the graphical user interface and, as a result, developed an extensive command line interface instead. In fact, one of the strongest reasons early adopters of Linux chose it over, say, Windows NT was the powerful command line interface which made the “difficult tasks possible.”

What This Book Is About

This book is a broad overview of “living” on the Linux command line. Unlike some books that concentrate on just a single program, such as the shell program, `bash`, I try to convey how to get along with the command line interface in a larger sense. How does it all work? What can it do? What's the best way to use it?

This is not a book about Linux system administration. While any serious discussion of the command line will invariably lead to system administration topics, this book only touches on a few administration issues. This book will, however, prepare the reader for additional study by providing a solid foundation in the use of the command line, an essential tool for any serious system administration task.

This book is very Linux-centric. Many other books try to broaden their appeal by including other platforms such as generic Unix and MacOS X. In doing so, they “water down” their content to feature only general topics. This book, on the other hand, only covers contemporary Linux distributions. Ninety-five percent of the content is useful for users of other Unix-like systems, but this book is highly targeted at the modern Linux command line user.

Who Should Read This Book

This book is for new Linux users who have migrated from other platforms. Most likely you are a “power user” of some version of Microsoft Windows. Perhaps your boss has told you to administer a Linux server or maybe you're just a desktop user who is tired of

all the security problems and want to give Linux a try. That's fine. All are welcome here.

That being said, there is no shortcut to Linux enlightenment. Learning the command line is challenging and takes real effort. It's not that it's so hard, but rather it's so *vast*. The average Linux system has literally *thousands* of programs you can employ on the command line. Consider yourself warned, learning the command line is not a casual endeavor.

But on the other hand, learning the Linux command line is extremely rewarding. If you think you're a “power user” now, just wait. You don't know what real power is – yet.

I also assume that you have no programming experience, but not to worry, we'll start you down that path as well.

What's In This Book

This material is presented in a carefully chosen sequence much like a tutor sitting next to you guiding you along. Many authors treat this material in what I call a “systematic” fashion, which makes sense if you already know the subject and need a reference manual but, in my experience, this confuses new users.

Another goal is to acquaint you with the Unix way of thinking which is different from the Windows way of thinking. Along the way, we'll go on a few side trips to help you understand why certain things work the way they do and how they got that way. Linux is not just a piece of software, it's also a small part of the larger Unix culture which has its own language and history. I might throw in a rant or two, as well.

This book is divided into five parts, each covering some aspect of the command line experience. Besides the first part, which you are reading now, this book contains:

- **Part 2 – Learning The Shell** starts our exploration of the basic language of the command line including such things as the structure of commands, file system navigation, command line editing, and finding help and documentation for commands.
- **Part 3 – Configuration And The Environment** covers editing configuration files that control the computer's operation from the command line.
- **Part 4 – Common Tasks And Essential Tools** explores many of the ordinary tasks that are commonly preformed from the command line. Unix-like operating systems, such as Linux, contain many “classic” command line programs that are used to perform powerful operations on data.
- **Part 5 – Writing Shell Scripts** introduces shell programming, an admittedly rudimentary, but easy to learn technique for automating many common computing tasks. By learning shell programming, you will become familiar with concepts

that can be applied to many other programming languages.

How To Read This Book

Start at the beginning of book and follow it to the end. I didn't write this as a reference work, it's really more like a story with a beginning, middle, and an end.

Prerequisites

To use this book, you will need is a working Linux installation. You can do this in one of two ways:

1. **Install Linux on a (not so new) computer.** It doesn't matter which distribution you choose, though most people today start out with either Ubuntu, Fedora, or OpenSUSE. If in doubt, try Ubuntu first. Installing a modern Linux distribution can be ridiculously easy or ridiculously difficult depending on your hardware. I suggest a desktop computer that is a couple of years old and has at least 256 megabytes of RAM and 6 gigabytes of free hard disk space. Avoid laptops and wireless networks if at all possible as these are often more difficult to get working.
2. **Use a “Live CD.”** One of the cool things you can do with many Linux distributions is run them directly from a CDROM without installing them at all. Just go into your BIOS setup and set your computer to “Boot from CDROM”, insert the live CD, and reboot. Using a live CD is a great way to test a computer for Linux compatibility prior to installation. The disadvantage to using a live CD is that it may be very slow compared to having Linux installed on your hard drive. Both Ubuntu and Fedora (among others) have live CD versions.

Regardless of how you install Linux, you will need to have occasional superuser (i.e. administrative) privileges to carry out the lessons in this book.

After you have a working installation, start reading and follow along with your own computer. Most of the material in this book is “hands on,” so sit down and get typing!

Why I Don't Call It “GNU/Linux”

In some quarters it's politically correct to call the Linux operating system the “GNU/Linux operating system.” The problem with “Linux” is that there is no completely correct way to name it because it was written by many different people in a vast, distributed development effort. Technically speaking, Linux is the name of the operating system's kernel, nothing more. The kernel is very important of course since it makes the operating system go, but it's not enough to form a

complete operating system.

Enter Richard Stallman, the genius-philosopher who founded the Free Software movement, started the Free Software Foundation, formed the GNU project, wrote the first version of the GNU C Compiler (gcc), created the GNU General Public License (the GPL), etc., etc., etc. He *insists* that you call it “GNU/Linux” to properly reflect the contributions of the GNU project. While the GNU project predates the Linux kernel and the project's contributions are extremely deserving of recognition, placing them in the name is unfair to everyone else who made significant contributions. Besides, I think “Linux/GNU” would be more technically accurate since the kernel boots first and everything else runs on top of it.

In popular usage, “Linux” refers to the kernel and all the other free and open source software found in the typical Linux distribution, that is, the entire Linux ecosystem, not just the GNU components. The operating system marketplace seems to prefer one-word names such as DOS, Windows, MacOS, Solaris, Irix, AIX. I have chosen to use the popular usage. If, however, you prefer to use “GNU/Linux” instead, please perform a mental search and replace while reading this book. I won't mind.

Acknowledgments

I want to thank the following people who helped make this book possible:

Jenny Watson, acquisitions editor at Wiley Publishing who originally suggested that I write a shell scripting book.

And lastly, the readers of LinuxCommand.org who have sent me so many kind emails. Their encouragement gave me the idea that I was really on to something!

Further Reading

Some Wikipedia articles on the famous people mentioned in this chapter:

http://en.wikipedia.org/wiki/Linux_Torvalds

http://en.wikipedia.org/wiki/Richard_stallman

The Free Software Foundation and the GNU Project:

http://en.wikipedia.org/wiki/Free_Software_Foundation

<http://www.fsf.org>

<http://www.gnu.org>

Part 2 - Learning The Shell

2 - What Is The Shell?

When we speak of the command line we are really referring to the *shell*. The shell is a program that takes keyboard commands and passes them to the operating system to carry out. Almost all Linux distributions supply a shell program from the GNU project called `bash`. The name “bash” is an acronym for “Bourne Again SHell”, a reference to the fact `bash` is an enhanced replacement for `sh`, the original Unix shell program written by Steve Bourne.

Terminal Emulators

When using a graphical user interface, we need another program called a *terminal emulator* to interact with the shell. If we look through our desktop menus, we will probably find one. KDE uses `konsole` and GNOME uses `gnome-terminal` though it's likely called simply “terminal” on our menu. There are a number of other terminal emulators available for Linux but they all basically do the same thing, give us access to the shell. You will probably develop a preference for one or another based on the number of bells and whistles it has.

Your First Keystrokes

So let's get started. Launch the terminal emulator! Once it comes up, we should see something like this:

```
[me@linuxbox ~]$
```

This is called a *shell prompt* and it will appear whenever the shell is ready to accept input. While it may vary in appearance somewhat depending the distribution, it will usually include your user name @ machine name followed by the current working directory (more about that in a little bit) and a dollar sign.

If the last character of the prompt is a pound sign (“#”) rather than a dollar sign, the terminal session has *superuser* privileges. This means either we are logged in as the root user or we selected a terminal emulator that provides superuser (administrative)

privileges.

Assuming that things are good so far, let's try some typing. Type some gibberish at the prompt like so:

```
[me@linuxbox ~]$ kaekfjaeifj
```

Since this command makes no sense, the shell will tell us so and give us another chance:

```
bash: kaekfjaeifj: command not found
[me@linuxbox ~]$
```

Command History

If we press the up-arrow key, we will see that the previous command “kaekfjaeifj” reappears after the prompt. This is called *command history*. Most Linux distributions remember the last five hundred commands by default. Press the down-arrow key and the previous command disappears.

Cursor Movement

Recall the previous command with the up-arrow key again. Now try the left and right-arrow keys. See how you can position the cursor anywhere on the command line? This makes editing commands easy.

A Few Words About Mice And Focus

While the shell is all about the keyboard, you can also use a mouse with your terminal emulator. There is a mechanism built into the X Window System, (the underlying engine that makes the GUI go) that supports a quick copy and paste technique. If you highlight some text by holding down the left mouse button and dragging the mouse over it (or double clicking on a word), it is copied into a buffer maintained by X. Pressing the middle mouse button will cause the text to be pasted at the cursor location. Try it.

Note: Don't be tempted to use `Ctrl-c` and `Ctrl-v` to perform copy and paste inside a terminal window. They don't work. Control codes have different meanings to the shell that predates Microsoft Windows.

2 - What Is The Shell?

Your graphical desktop environment (most likely KDE or GNOME) in an effort to behave in like Windows, probably has its *focus policy* set to “click to focus.” This means for a window get focus (become active) you need to click on it. This is contrary to traditional X behavior which is “focus follows mouse” which means that a window gets focus by just passing the mouse over it. The window will not come to the foreground until you click on it but it will be able to receive input. Setting the focus policy this way will make the copy and paste technique even more useful. Give it a try. I think if you give it a chance you will prefer it. You will find this setting in the configuration program for your window manager.

Try Some Simple Commands

Now that we have learned to type, let's try a few simple commands. The first one is `date`. This command displays the current time and date.

```
[me@linuxbox ~]$ date
Thu Oct 25 13:51:54 EDT 2007
```

A related command is `cal` which, by default, displays a calendar of the current month.

```
[me@linuxbox ~]$ cal
October 2007
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
  7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

To see the current amount of free space on your disk drives, type `df`:

```
[me@linuxbox ~]$ df
Filesystem      1K-blocks      Used Available Use% Mounted on
/dev/sda2        15115452    5012392   9949716  34% /
/dev/sda5        59631908   26545424  30008432  47% /home
/dev/sda1         147764       17370    122765  13% /boot
tmpfs            256856         0     256856   0% /dev/shm
```

Likewise, to display the amount of free memory, type the `free` command.

```
[me@linuxbox ~]$ free
              total        used         free       shared    buffers     cached
Mem:          513712      503976         9736           0         5312     122916
-/+ buffers/cache: 375748      137964
Swap:        1052248      104712      947536
```

Ending A Terminal Session

We can end a terminal session by either closing the terminal emulator window, or by entering the `exit` command at the shell prompt:

```
[me@linuxbox ~]$ exit
```

The Console Behind The Curtain

Even if we have no terminal emulator running, there remain several terminal sessions running behind the graphical desktop. Called *virtual terminals* or *virtual consoles*, these sessions can be accessed by pressing `Ctrl-Alt-F1` through `Ctrl-Alt-F6` on most systems. When a session is accessed, it presents a login prompt into which we can enter our user name and password. To return to the graphical desktop, type `Alt-F7`.

Further Reading

To learn more about Steve Bourne, father of the Bourne Shell, see this Wikipedia article:

http://en.wikipedia.org/wiki/Steve_Bourne

Here is an article about the concept of shells in computing:

[http://en.wikipedia.org/wiki/Shell_\(computing\)](http://en.wikipedia.org/wiki/Shell_(computing))

3 - Navigation

The first thing we need to learn to do (besides just typing) is how to navigate the file system on our Linux system. In this chapter we will introduce the following commands:

- `pwd` - print working directory
- `cd` - change directory
- `ls` - list directory contents

Understanding The File System Tree

Linux, like Windows, organizes its files in what is called a *hierarchical directory structure*. This means that they are organized in a tree-like pattern of directories (sometimes called folders in other systems), which may contain files and other directories. The first directory in the file system is called the *root directory*. The root directory contains files and subdirectories, which contain more files and subdirectories and so on and so on.

Note that unlike Windows, which has a separate file system tree for each drive, Unix-like systems such as Linux will have a single file system tree regardless of how many drives or storage devices are attached to the computer. Storage devices are attached (or more correctly, *mounted*) at various points on the tree according to the whims of the system administrator.

The Current Working Directory

Most of us are probably familiar with a graphical file manager which represents the file system tree as in Figure 1. Notice that the tree is usually shown upended, that is, with the root at the top and the various branches descending below.

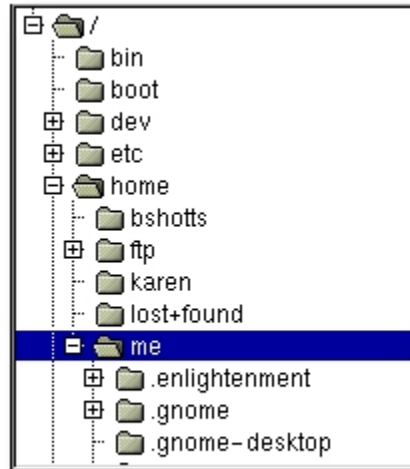


Figure 1: File system tree as shown by a graphical file manager

However, the command line has no pictures, so to navigate the file system tree we need to think of it a different way.

Imagine that the file system is a tree-shaped maze and we are able to stand inside of it. At any given time we are inside a single directory and we can see the files contained in the directory and the pathways to the directory above us (called the *parent directory*) and any subdirectories below us. The directory we are standing in is called the *current working directory*. To display the current working directory, we use the `pwd` (print working directory) command.

```
[me@linuxbox ~]$ pwd
/home/me
```

When we first log in to our system (or start a terminal emulator session) our current working directory is set to our *home directory*. Each user account is given one and when operating as a regular user, our home directory is the only place we are allowed to write files.

Listing The Contents Of A Directory

To list the files and directories in the current working directory, we use the `ls` command.

```
[me@linuxbox ~]$ ls
Desktop Documents Music Pictures Public Templates Videos
```

Actually we can use the `ls` command to list the contents of any directory, not just the current working directory and there are many other fun things it can do as well. We'll spend more time with `ls` in the next chapter.

Changing The Current Working Directory

To change your working directory (where we are standing in the maze) we use the `cd` command. To do this, type `cd` followed by the *pathname* of the desired working directory. A pathname is the route we take along the branches of the tree to get to the directory we want. Pathnames can be specified in one of two different ways; *absolute pathnames* or *relative pathnames*. Let's deal with absolute pathnames first.

Absolute Pathnames

An absolute pathname begins with the root directory and follows the tree branch by branch until the path to the desired directory or file is completed. For example, there is a directory on your system in which most of your system's programs are installed. The pathname of the directory is `/usr/bin`. This means from the root directory (represented by the leading slash in the pathname) there is a directory called "usr" which contains a directory called "bin".

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
[me@linuxbox bin]$ ls

...Listing of many, many files ...
```

Now we can see that we have changed the current working directory to `/usr/bin` and that it is full of files. Notice how your prompt has changed? As a convenience, it is usually set up to display the name of the working directory.

Relative Pathnames

Where an absolute pathname starts from the root directory and leads to its destination, a relative pathname starts from the working directory. To do this, it uses a couple of special symbols to represent relative positions in the file system tree. These special symbols are "." (dot) and ".." (dot dot).

The "." symbol refers to the working directory and the ".." symbol refers to the working directory's parent directory. Here is how it works. Let's change the working directory to `/usr/bin` again:

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

O.K., now let's say that we wanted to change the working directory to the parent of `/usr/bin` which is `/usr`. We could do that two different ways. Either with an absolute pathname:

```
[me@linuxbox bin]$ cd /usr
[me@linuxbox usr]$ pwd
/usr
```

Or, with a relative pathname:

```
[me@linuxbox bin]$ cd ..
[me@linuxbox usr]$ pwd
/usr
```

Two different methods with identical results. Which one should we use? The one that requires the least typing!

Likewise, we can change the working directory from `/usr` to `/usr/bin` in two different ways. Either using an absolute pathname:

```
[me@linuxbox usr]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

3 - Navigation

Or, with a relative pathname:

```
[me@linuxbox usr]$ cd ./bin
[me@linuxbox bin]$ pwd
/usr/bin
```

Now, there is something important that I must point out here. In almost all cases, you can omit the "./". It is implied. Typing:

```
[me@linuxbox usr]$ cd bin
```

would do the same thing. In general, if you do not specify a pathname to something, the working directory will be assumed.

Some Helpful Shortcuts

Shortcut	Result
cd	Changes the working directory to your home directory.
cd -	Changes the working directory to the previous working directory.
cd ~	Changes the working directory to your home directory.
cd ~ <i>user_name</i>	Changes the working directory to the home directory of <i>user_name</i> . For example, cd ~bob will change the directory to the home directory of user "bob."

Important facts about file names

1. File names that begin with a period character are hidden. This only means that `ls` will not list them unless you say `ls -a`. When your account was created, several hidden files were placed in your home directory to configure things for your account. Later on we will take a closer look at some of these files to see how you can customize your environment. In addition, some applications will place their configuration and settings files in your home directory as hidden files.

2. File names in Linux, like Unix, are case sensitive. The file names "File1" and "file1" refer to different files.
3. Linux has no concept of a "file extension" like some other operating systems. You may name files any way you like. The contents/purpose of a file is determined by other means.
4. While Linux supports long file names which may contain embedded spaces and punctuation characters, limit the punctuation characters to period, dash, and underscore. *Most importantly, do not embed spaces in file names.* If you want to represent spaces between words in a file name, use underscore characters. You will thank yourself later.

4 - Exploring The System

Now that we know how to move around the file system, it's time for a guided tour of your Linux system. Before we start however, I'm going to teach you some more commands that will be useful along the way:

- `ls` – list directory contents
- `file` – determine file type
- `less` – view file contents

More Fun With `ls`

You will probably find that the `ls` command is the most used command and for good reason. With it, you can see directory contents and determine a variety of important file and directory attributes. As we have seen, you can simply type `ls` to see a list of files and subdirectories contained in the current working directory:

```
[me@linuxbox ~]$ ls
Desktop Documents Music Pictures Public Templates Videos
```

Besides the current working directory, you can specify the directory to list, like so:

```
me@linuxbox ~]$ ls /usr
bin  games  kerberos  libexec  sbin  src
etc  include  lib      local  share  tmp
```

Or even specify multiple directories. In this example we will list both the user's home directory (symbolized by the “~” character) and the `/usr` directory:

```
[me@linuxbox ~]$ ls ~ /usr
/home/me:
```

```
Desktop Documents Music Pictures Public Templates Videos

/usr:
bin  games      kerberos  libexec  sbin    src
etc  include    lib       local    share   tmp
```

You can also change the format of the output to reveal more detail:

```
[me@linuxbox ~]$ ls -l
total 56
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Desktop
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Documents
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Music
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Pictures
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Public
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Templates
drwxrwxr-x 2 me me 4096 2007-10-26 17:20 Videos
```

By adding “-l” to the command we changed the output to the long format.

Options And Arguments

Now this brings us to a very important point about how most commands work. Commands are often followed by one or more *options* that modify their behavior and afterwards by one or more *arguments*, the items on which the command acts upon. So most commands look kind of like this:

```
command -options arguments
```

Most commands use options consisting of a single character preceded by a dash, for example “-l”, but many commands including those from the GNU Project also support *long options* consisting of a word preceded by two dashes. Also, many commands allow multiple short options to be strung together. In this example, the ls command is given two options, the “l” options to produce long format output, and the “t” option to sort the result by the file's modification time.

```
[me@linuxbox ~]$ ls -lt
```

4 - Exploring The System

We'll add the long option “--reverse” to reverse the order of the sort:

```
[me@linuxbox ~]$ ls -lt --reverse
```

The `ls` command has a large number of possible options. The most common ones are listed in the table below.

Option	Long Option	Description
-a	--all	List all files even those with names that begin with a period which are normally not listed (i.e. hidden.)
-d	--directory	Ordinarily, if you specify a directory, <code>ls</code> will list the contents of the directory, not the directory itself. Use this option in conjunction with the <code>-l</code> option to see details about the directory rather than its contents.
-F	--classify	This option will append an indicator character to the end of each listed name. For example a “/” if the name is a directory.
-h	--human-readable	In long format listings, display file sizes in human readable format rather than in bytes.
-l		Display results in long format.
-r	--reverse	Display the results in reverse order. Normally, <code>ls</code> displays its results in ascending alphabetical order.
-S		Sort results by file size.
-t		Sort by modification time.

A Longer Look At Long Format

As we saw before, the “-l” option causes `ls` to display its results in long format. This format contains a great deal of useful information. Here is the Examples directory from an Ubuntu system:

```
-rw-r--r-- 1 root root 3576296 2007-04-03 11:05 Experience ubuntu.ogg  
-rw-r--r-- 1 root root 1186219 2007-04-03 11:05 kubuntu-leaflet.png
```

```

-rw-r--r-- 1 root root 47584 2007-04-03 11:05 logo-Edubuntu.png
-rw-r--r-- 1 root root 44355 2007-04-03 11:05 logo-Kubuntu.png
-rw-r--r-- 1 root root 34391 2007-04-03 11:05 logo-Ubuntu.png
-rw-r--r-- 1 root root 32059 2007-04-03 11:05 oo-cd-cover.odf
-rw-r--r-- 1 root root 159744 2007-04-03 11:05 oo-derivatives.doc
-rw-r--r-- 1 root root 27837 2007-04-03 11:05 oo-maxwell.odt
-rw-r--r-- 1 root root 98816 2007-04-03 11:05 oo-trig.xls
-rw-r--r-- 1 root root 453764 2007-04-03 11:05 oo-welcome.odt
-rw-r--r-- 1 root root 358374 2007-04-03 11:05 ubuntu Sax.ogg

```

Let's look at the different fields from one of the files and examine their meanings:

Field	Meaning
<code>-rw-r--r--</code>	Access rights to the file. The leading dash means a regular file while a “d” indicates a directory. The next three characters are the access rights for the file's owner, the next three are for members of the file's group, and the final three are for everyone else. The full meaning of this is discussed in Chapter 10 – Permissions.
<code>1</code>	File's number of hard links. See the discussion of links later in this chapter.
<code>root</code>	The user name of the file's owner.
<code>root</code>	The name of the group which owns the file.
<code>32059</code>	Size of the files in bytes.
<code>2007-04-03 11:05</code>	Date and time of the file's last modification.
<code>oo-cd-cover.odf</code>	Name of the file.

Determining A File's Type With `file`

As we explore the system it will be useful to know what files contain. To do this we will use the `file` command determine a file's type. As we discussed earlier, in Linux, file names are not required to reflect a file's contents. While a file name like “picture.jpg” would normally be expected to contain a JPEG compressed image, it is not required to. You can invoke the `file` command this way:

```
file filename
```

When invoked, the `file` command will print a brief description of the file's contents. For example:

```
[me@linuxbox ~]$ file picture.jpg
picture.jpg: JPEG image data, JFIF standard 1.01
```

As you can imagine, there are many kinds of files. In fact, one of the common ideas in Unix-like operating systems such as Linux is that “everything is a file.” As we proceed with our lessons you will see just how true that statement is.

While many of the files on your system are familiar, for example MP3 and JPEG, there are many kinds that are a little less obvious and a few that are quite strange.

Viewing File Contents With `less`

The `less` command is a program to view text files. You will discover that throughout your Linux system there are many files that contain human-readable text and the `less` program provides a convenient way to examine them.

What Is “Text”?

There are many ways to represent information on a computer. All methods involve defining a relationship between the information and some numbers that will be used to represent it. Computers, after all, only understand numbers and all data is converted to numeric representation.

Some of these representation systems are very complex (such as compressed video files), while others are rather simple. One of the earliest and simplest is called *ASCII text*. ASCII (pronounced "As-Key") is short for American Standard Code for Information Interchange. This is a simple encoding scheme that was first used on Teletype machines to map keyboard characters to numbers.

Text is a simple one-to-one mapping of characters to numbers. It is very compact. Fifty characters of text translates to fifty bytes of data. It is important to understand that text only contains a simple mapping of characters to numbers. It is not the same as a word processor document such as one created by Microsoft Word or OpenOffice.org Writer. Those files, in contrast to simple ASCII text, contain many non-text elements that are used to describe it's structure and formatting. Plain ASCII text files contain only the characters themselves and a few rudimentary control codes like tabs, carriage returns and line feeds.

Throughout a Linux system, many files are stored in text format and there are many Linux tools that work with text files. Even Windows recognizes the importance of this format. The well-known NOTEPAD.EXE program is an editor for plain ASCII text files.

Why would we want to examine text files? Because many of the files that contain system settings (called *configuration files*) are stored in this format and being able to read them gives us insight as to how the system works. In addition, many of the actual programs that the system uses (called *scripts*) are stored in this format. In later chapters, we will learn how to edit text files in order to modify systems settings and write our own scripts, but for now we will just look at their contents.

The `less` command is used like this:

```
less filename
```

Once started, the `less` program allows you to scroll forward and backward through a text file. For example, to examine the file that defines all the system's user accounts, enter the following command:

```
[me@linuxbox ~]$ less /etc/passwd
```

Once the `less` program starts, we may view the contents of the file. If the file is longer than one page, we can scroll up and down. To exit `less`, press the “q” key.

The table below lists the most common keyboard commands used by `less`.

Command	Action
Page Up or b	Scroll back one page
Page Down or space	Scroll forward one page
Up Arrow	Scroll up one line
Down Arrow	Scroll down one line
G	Move to the end of the text file
1G	Move to the beginning of the text file
/characters	Search forward to the next occurrence of <i>characters</i>

n	Search for the next occurrence of the previous search
h	Display help screen
q	Quit less

Less Is More

The `less` program was designed as an improved replacement of an earlier Unix program called `more`. The name “less” is a play on the phrase “less is more” -- a motto of modernist architects and designers.

`less` falls into the class of programs called “pagers”, programs that allow the easy viewing of long text documents in a page by page manner. Whereas the `more` program could only page forward, the `less` program allows paging both forward and backward and has many other features as well.

A Guided Tour

The file system layout on your Linux system is much like that found on other Unix-like systems. The design is actually specified in a published standard called the *Linux Filesystem Hierarchy Standard*. Not all Linux distributions conform to the standard exactly but most come pretty close.

Next, we are going to wander around the file system ourselves to see what makes our Linux system tick. This will give you a chance to practice your navigation skills. One of the things we will discover is that many of the interesting files are in plain human-readable text. As we go about our tour, try the following:

1. `cd` into a given directory
2. List the directory contents with `ls -l`
3. If you see an interesting file, determine its contents with `file`
4. If it looks like it might be text, try viewing it with `less`

Remember the copy and paste trick! If you are using a mouse, you can double click on a file name to copy it and middle click to paste it into commands.

As we wander around, don't be afraid to look at stuff. As a regular user, you are largely prohibited from messing stuff up. That's the system administrators job! If a command complains about something, just move on to something else. Spend some time looking around. The system is yours to explore. Remember, in Linux, there are no secrets!

The table below lists of just a few of the directories you should explore. Feel free to try more!

Directory	Comments
/	The root directory. Where everything begins.
/bin	Contains binaries (programs) that must be present for the system to boot and run.
/boot	Contains the Linux kernel, initial RAM disk image (for drivers needed at boot time), and the boot loader. Interesting files: <ul style="list-style-type: none"> • /boot/grub/grub.conf or menu.lst which are used to configure the boot loader. • /boot/vmlinuz, the linux kernel
/dev	This is a special directory which contains <i>device nodes</i> . “Everything is a file” also applies to devices. Here is where the kernel maintains a list of all the devices it understands.
/etc	The /etc directory contains all of the system-wide configuration files. It also contains a collection of shell scripts that start each of the system services at boot time. Everything in this directory should be readable text. Interesting files: While everything in /etc is interesting, here are some of my all-time favorites: <ul style="list-style-type: none"> • /etc/crontab, a file that defines when automated jobs will run. • /etc/fstab, a table of storage devices and their associated mount points. • /etc/passwd, a list of the user accounts • /etc/X11/xorg.conf, the configuration file for the X Window system.
/home	In normal configurations, each user is given a directory in /home. Ordinary users can only write files in their home directories. This limitation protects the system from errant user activity.
/lib	Contains shared library files used by the core system programs. These are similar to DLLs in Windows.
/lost+found	Each formatted partition or device using a Linux file system

4 - Exploring The System

	such as ext3 will have this directory. It is used in the case of a partial recovery from a file system corruption event. Unless something really bad has happened to your system, this directory will remain empty.
/media	On modern Linux systems the /media directory will contain the mount points for removable media such as USB drives, CD-ROMs, etc. that are mounted automatically at insertion.
/mnt	On older Linux systems, the /mnt directory contains mount points for removable devices that have been mounted manually.
/opt	The /opt directory is used to install “optional” software. This is mainly used to hold commercial software products that may be installed on your system.
/proc	The /proc directory is special. It's not a real file system in the sense of files stored on your hard drive. Rather it is a virtual file system maintained by the Linux kernel. The “files” it contains are peepholes into the kernel itself. The files are readable and will give you a picture of how the kernel sees your computer.
/root	This is the home directory for the root account.
/sbin	This directory contains “system” binaries. These are programs that perform vital system tasks that are generally reserved for the superuser.
/tmp	The /tmp directory is intended for storage of temporary, transient files created by various programs. Some configurations cause this directory to be emptied each time the system is rebooted.
/usr	The /usr directory tree is likely the largest one on your Linux system. It contains all the programs and support files used by regular users.
/usr/bin	/usr/bin contains the executable programs installed by your Linux distribution. It is not uncommon for this directory to hold thousands of programs.
/usr/lib	The shared libraries for the programs in /usr/bin.
/usr/local	The /usr/local tree is where you install programs not

	included with your distribution that are intended for system-wide use. If you compile programs from source code, you normally install the executables in <code>/usr/local/bin</code> . Your distribution will create this tree for you but it will be empty unless you put something in it.
<code>/usr/sbin</code>	Contains more system administration programs.
<code>/usr/share</code>	<code>/usr/share</code> contains all the shared data used by programs in <code>/usr/bin</code> . This includes things like default configuration files, icons, screen backgrounds, sound files, etc.
<code>/usr/share/doc</code>	Most packages installed on your system will include some kind of documentation. In <code>/usr/share/doc</code> you will find documentation files organized by package.
<code>/var</code>	With the exception of <code>/tmp</code> and <code>/home</code> , the directories we have looked at so far remain relatively static, that is, their contents don't change. The <code>/var</code> directory tree is where data is stored that is likely to change. Various databases, spool files, user mail, etc. are located here.
<code>/var/log</code>	<code>/var/log</code> contains <i>log files</i> , records of various system activity. These are very important and should be monitored from time to time. The most useful one is <code>/var/log/messages</code> . Note that on some systems, you must be the superuser to view log files for security reasons.

Symbolic Links

As you were looking around, did you happen to see a directory listing with an entry like this?

```
lrwxrwxrwx 1 root root 11 2007-08-11 07:34 libc.so.6 -> libc-2.6.so
```

Notice how the first letter of the listing is “l” and the entry seems to have two file names? This is a special kind of a file called a *symbolic link* (also known as *soft links* or *symlinks*.) In most Unix-like systems it is possible to have a file referenced by multiple names. While the value of this may not be intuitively obvious, it is really a useful feature.

Picture this scenario: a program requires the use of a shared resource of some kind contained in a file named “foo”, but “foo” has frequent version changes and it would be

good to include the version number in the file name so the administrator or other interested party could see what version of “foo” is installed. This presents a problem. If we change the name of the shared resource, we have to track down every program that might use it and change them to look for a new resource name every time a new version of the resource is installed. That doesn't sound like fun at all.

Here is where symbolic links save the day. Let's say we install version 2.6 of “foo” which has the file name “foo-2.6” and then create a symbolic link simply called “foo” that points to “foo-2.6”. This means that when a program opens the file “foo”, it is actually opening the file “foo-2.6”. Now everybody is happy. The programs that rely on “foo” can find it and we can still see what actual version is installed. When it is time to upgrade to “foo-2.7”, we just add the file to our system, delete the symbolic link “foo” and create a new one that points to the new version. Not only does this solve the problem of the version upgrade, but it also allows to keep both versions on our machine. Imagine that “foo-2.7” has a bug (damn those developers!) and we need to revert to the old version. Again, we just delete the symbolic link pointing to the new version and create a new symbolic link pointing to the old version.

The directory listing above (from the `/lib` directory of a Fedora system) shows a symbolic link called “`libc.so.6`” that points to shared library file called “`libc-2.6.so`”. This means that programs looking for “`libc.so.6`” will actually get the file “`libc-2.6.so`”.

Hard Links

While we are on the subject of links, I want to mention that there is a second type of link called a *hard link*. Hard links also allow files to have multiple names but they do it in a different way. I'll talk more about the differences between symbolic and hard links in the next chapter.

5 - Manipulating Files And Directories

At this point, we are ready for some real work! This chapter will introduce you to the following commands:

- `cp` – Copy files and directories
- `mv` – move/rename files and directories
- `mkdir` – Create directories
- `rm` – Remove files and directories
- `ln` – Create hard and symbolic links

The first five commands are among the most frequently used Linux commands. They are the basic commands for manipulating both files and directories.

Now, to be frank, some of the tasks performed by these commands are more easily done with a graphical file manager. With a file manager, you can drag and drop a file from one directory to another, cut and paste files, delete files, etc. So why use these old command line programs?

The answer is power and flexibility. While it is easy to perform simple file manipulations with a graphical file manager, complicated tasks can be easier with the command line programs. For example, how would you copy all the HTML files from one directory to another, but only copy files that did not exist in the destination directory or were newer than the versions in the destination directory? Pretty hard with with a file manager. Pretty easy with the command line:

```
cp -u *.html destination
```

Wildcards

Before I begin with our commands, I want to talk about a shell feature that makes these commands so powerful. Since the shell uses filenames so much, it provides special characters to help you rapidly specify groups of filenames. These special characters are

5 - Manipulating Files And Directories

called *wildcards*. Wildcards allow you to select filenames based on patterns of characters. The table below lists the wildcards and what they select:

Wildcard	Meaning
*	Matches any characters
?	Matches any single character
[<i>characters</i>]	Matches any character that is a member of the set <i>characters</i> .
[! <i>characters</i>]	Matches any character that is not a member of the set <i>characters</i>
[[: <i>class</i> :]]	Matches any character that is a member of the specified <i>class</i> .

The table below lists the most commonly used character classes:

Character Class	Meaning
alnum	Matches any alphanumeric character
alpha	Matches any alphabetic character
digit	Matches any numeral
lower	Matches any lowercase letter
upper	Matches any uppercase letter

Using wildcards, it is possible to construct very sophisticated selection criteria for filenames. Here are some examples of patterns and what they match:

Pattern	Matches
*	All files
g*	Any file beginning with “g”
b*.txt	Any file beginning with “b” followed by any characters and ending with “.txt”
Data???	Any file beginning with “Data” followed by exactly three characters
[abc]*	Any file beginning with either an “a”, a “b”, or a “c”

<code>BACKUP.[0-9][0-9][0-9]</code>	Any file beginning with “BACKUP.” followed by exactly three numerals
<code>[:upper:]*</code>	Any file beginning with an uppercase letter
<code>![[:digit:]]*</code>	Any file not beginning with a numeral
<code>*[[:lower:]]123]</code>	Any file ending with a lowercase letter or the numerals “1”, “2”, or “3”

Wildcards can be used with any command that accepts file names as arguments, but more about that in the Chapter 8.

Character Ranges

If you are coming from another Unix-like environment or have been reading some other books on this subject, you may have encountered the `[A-Z]` or the `[a-z]` character range notations. These are traditional Unix and worked in older versions of Linux as well. They can still work but you have to be very careful with them because they will not produce the expected results unless properly configured. For now you should avoid using them and use character classes instead.

Wildcards Work In The GUI Too

Wildcards are especially valuable not only because they are used so frequently on the command line but are also supported by some graphical file managers as well.

- In **Nautilus** (the file manager for GNOME), you can select files using the Edit/Select Pattern menu item. Just enter a file selection pattern with wildcards and the files in the currently viewed directory will be highlighted for selection.
- In **Dolphin** and **Konqueror** (the file managers for KDE), you can enter wildcards directly on the location bar. For example, if you wanted to see all the files starting with a lowercase “u” in the `/usr/bin` directory, type “`/usr/bin/u*`” into the location bar and it will display the result.

As we will see, many ideas found originally in the command line interface find their way into the graphical interface too. It is one of the many things that makes the Linux desktop so powerful.

`mkdir` – Create Directories

The `mkdir` command is used to create directories. It works like this:

5 - Manipulating Files And Directories

```
mkdir directory...
```

A note on notation: When three periods follow an argument in the description of a command (like above), it means that the argument can be repeated, thus:

```
mkdir dir1
```

would create a single directory named “dir1”, while

```
mkdir dir1 dir2 dir3
```

would create three directories named “dir1”, “dir2”, and “dir3”.

cp – Copy Files And Directories

The cp command copies files or directories. It can be used two different ways:

```
cp item1 item2
```

to copy the single file or directory “item1” to file or directory “item2” and:

```
cp item... directory
```

to copy multiple items (either files or directories) into a directory.

Useful Options

Here are some of the commonly used options for cp:

Option	Meaning
-i, --interactive	Before overwriting an existing file, prompt the user for confirmation. If this option is not specified, cp will overwrite files silently.
-r, --recursive	Recursively copy directories. This means that if a directory being copied has subdirectories, copy them

	too.
<code>-u, --update</code>	When copying files from one directory to another, only copy files that either don't exist, or are newer than the existing corresponding files, in the destination directory.
<code>-v, --verbose</code>	Display informative messages as the copy is performed.

Examples

Command	Results
<code>cp file1 file2</code>	Copy <i>file1</i> to <i>file2</i> . If <i>file2</i> exists, it is overwritten by the contents of <i>file1</i> . If <i>file2</i> does not exist, it is created.
<code>cp -i file1 file2</code>	Same as above except that if <i>file2</i> exists, the user is prompted before it is overwritten.
<code>cp file1 file2 dir1</code>	Copy <i>file1</i> and <i>file2</i> into directory <i>dir1</i> . <i>dir1</i> must already exist.
<code>cp -r dir1 dir2</code>	Copy the contents of directory <i>dir1</i> to directory <i>dir2</i> . If directory <i>dir2</i> does not exist, it is created and after the copy will contain the same contents as directory <i>dir1</i> . If directory <i>dir2</i> does exist, then directory <i>dir1</i> (and its contents) will be copied into <i>dir2</i> . Since the <code>-r</code> option has been included, the copy will be recursive meaning that every subdirectory contained within <i>dir1</i> will be copied also.

mv – Move And Rename Files

The `mv` command performs either file moving or renaming depending on how it is used. In either case, the original file name no longer exists after the operation. `mv` is used in much same way as `cp`:

```
mv item1 item2
```

5 - Manipulating Files And Directories

to rename file or directory “item1” to “item2” or:

```
mv item... directory
```

move one or more items in one directory to another.

Useful Options

mv shares many of the same options as cp:

Option	Meaning
-i, --interactive	Before overwriting an existing file, prompt the user for confirmation. If this option is not specified, mv will overwrite files silently.
-r, --recursive	Recursively move directories. This means that if a directory being moved has subdirectories, move them too.
-u, --update	When moving files from one directory to another, only move files that either don't exist, or are newer than the existing corresponding files, in the destination directory.
-v, --verbose	Display informative messages as the move is performed.

Examples

Command	Results
mv <i>file1 file2</i>	Move <i>file1</i> to <i>file2</i> . If <i>file2</i> exists, it is overwritten by the contents of <i>file1</i> . If <i>file2</i> does not exist, it is created. In either case, <i>file1</i> ceases to exist.
mv -i <i>file1 file2</i>	Same as above except that if <i>file1</i> or <i>file2</i> exists, the user is prompted before it is overwritten.
mv <i>file1 file2 dir1</i>	Move <i>file1</i> and <i>file2</i> into directory <i>dir1</i> . <i>dir1</i> must already exist.
mv -r <i>dir1 dir2</i>	Move the contents of directory <i>dir1</i> to directory <i>dir2</i> . If directory <i>dir2</i> does not exist, it is created

and after the move will contain the same contents as directory *dir1*. If directory *dir2* does exist, then directory *dir1* (and it's contents) will be moved into *dir2*. Since the `-r` option has been included, the copy will be recursive meaning that every subdirectory contained within *dir1* will be moved also.

rm – Remove Files And Directories

The `rm` command is used to remove (delete) files and directories:

```
rm item...
```

where “item” is one or more files or directories.

Useful Options

Here are some of the common options for `rm`:

Option	Meaning
<code>-i, --interactive</code>	Before deleting an existing file, prompt the user for conformation. If this option is not specified, rm will delete files silently.
<code>-r, --recursive</code>	Recursively delete directories. This means that if a directory being deleted has subdirectories, delete them too. To delete a directory, this option must be specified.
<code>-f, --force</code>	Ignore nonexistent files and do not prompt. This overrides the <code>--interactive</code> option.
<code>-v, --verbose</code>	Display informative messages as the deletion is preformed.

Examples

Command	Results
---------	---------

5 - Manipulating Files And Directories

<code>rm file1</code>	Delete <i>file1</i> silently.
<code>rm -i file1</code>	Same as above except that the user is prompted for confirmation before the deletion is performed.
<code>rm -r file1 dir1</code>	Delete <i>file1</i> and <i>dir1</i> and its contents.
<code>rm -rf file1 dir1</code>	Same as above except that if either <i>file1</i> or <i>dir1</i> do not exist, <code>rm</code> will continue silently.

Be Careful With rm!

Unix-like operating systems such as Linux do not have an undelete command. Once you delete something with `rm`, it's gone. Linux assumes you're smart and you know what you're doing.

Be particularly careful with wildcards. Consider this classic example. Let's say you want to delete just the HTML files in a directory. To do this, you type:

```
rm *.html
```

which is correct, but what happens if you accidentally place a space between the "*" and the ".html" like so:

```
rm * .html
```

the `rm` command will delete all the files in the directory and then complain that there is no file called ".html".

Here is a useful tip. Whenever you use wildcards with `rm` (besides carefully checking your typing!), test the wildcard first with `ls`. This will let you see the files that will be deleted. Then press the up arrow key to recall the command and replace the `ls` with `rm`.

ln – Create Links

The `ln` command is used to create either hard or symbolic links. It is used in one of two ways:

```
ln file link
```

to create a hard link, and:

```
ln -s item link
```

to create a symbolic link where “item” is either a file or a directory.

Hard Links

Hard links are the original way of creating links, compared to symbolic links which are more modern. When you create a hard link, you are creating an additional directory entry for a file. By default, each file has a single hard link that gives it its name. Hard links have two important limitations:

1. A hard link cannot reference a file outside its own file system. This means a link may not reference a file that is not on the same disk partition as the link itself.
2. A hard link may not reference a directory.

A hard link is indistinguishable from the file itself. When you list a directory containing a hard link you will see no special indication of the presence of a link. In fact, all directory listings are the one default link that is created for each file. When deleting a hard link, the link is removed but the file itself continues to exist (that is, its space is not deallocated) until all links to the file are deleted.

We discuss hard links because you might encounter them from time to time and you need to be familiar, but modern practice prefers symbolic links which we will cover next.

Symbolic Links

Symbolic links were created to overcome some of the limitations of hard links. Symbolic links work by creating a special type of file that contains a text pointer to the referenced file or directory. In this regard, they operate in much the same way as a Windows shortcut though of course, they predate the Windows feature by many years ;-)

When operating on a file pointed to by a symbolic link, they are largely indistinguishable from one another. For example, if you write something to the symbolic link, the referenced file is written to. However when you delete a symbolic link, only the link is deleted, not the file itself. If the file is deleted before the symbolic link, the link will continue to exist, but will point to nothing. In this case the link is said to be *broken*. In many implementations, the `ls` command will display broken links in a distinguishing color, such as red, to reveal their presence.

I know the concept of links seems very confusing, but hang in there. We're going to try all this stuff and it will, hopefully, become clear.

Let's Build A Playground

Since we are going to do some real file manipulation, let's build a place to “play” with our file manipulation commands. First we need a directory to work in. We'll create one in our home directory and call it “playground”.

Creating Directories

To create a directory, the `mkdir` command is used. To create our playground directory we will first make sure we our current working directory is our home directory and will then create our new directory:

```
[me@linuxbox ~]$ cd  
[me@linuxbox ~]$ mkdir playground
```

To make our playground a little more interesting, let's create a couple of directories inside of it called “dir1” and “dir2”. To do this, we will change our current working directory to `playground` and execute another `mkdir`:

```
[me@linuxbox ~]$ cd playground  
[me@linuxbox playground]$ mkdir dir1 dir2
```

Notice that the `mkdir` command will accept multiple arguments allowing us to create both directories with a single command.

Copying Files

Next, let's get some data into our playground. We'll do this by copying a file. Using the `cp` command, we'll copy the `passwd` file from the `/etc` directory to the current working directory:

```
[me@linuxbox playground]$ cp /etc/passwd .
```

Notice how we used the shorthand for the current working directory, the single trailing period. So now if we perform an `ls`, we will see our file:

```
[me@linuxbox playground]$ ls -l
```

```
total 12
drwxrwxr-x 2 me  me 4096 2008-01-10 16:40 dir1
drwxrwxr-x 2 me  me 4096 2008-01-10 16:40 dir2
-rw-r--r-- 1 me  me 1650 2008-01-10 16:07 passwd
```

Now, just for fun, let's repeat the copy using the “-v” option (verbose) to see what it does:

```
[me@linuxbox playground]$ cp -v /etc/passwd .
`/etc/passwd' -> `./passwd'
```

The `cp` command performed the copy again, but this time displayed a concise message indicating what operation it was performing. Notice that `cp` overwrote the first copy without any warning. Again this is a case of `cp` assuming that you really know what you are doing. To get a warning, we'll include the “-i” (interactive) option:

```
[me@linuxbox playground]$ cp -i /etc/passwd .
cp: overwrite `./passwd'?
```

Responding to the prompt by entering a “y” will cause the file to be overwritten, any other character (for example, “n”) will cause `cp` to leave the file alone.

Moving And Renaming Files

Now, I don't know about you, but the name “passwd” doesn't seem very playful to me and this is playground after all, so let's change it to something else:

```
[me@linuxbox playground]$ mv passwd fun
```

Let's pass the fun around a little by moving our renamed file to each of the directories and back again:

```
[me@linuxbox playground]$ mv fun dir1
```

to move it first to directory `dir1`, then:

5 - Manipulating Files And Directories

```
[me@linuxbox playground]$ mv dir1/fun dir2
```

move it from `dir1` to `dir2`, then:

```
[me@linuxbox playground]$ mv dir2/fun .
```

finally bringing it back to the current working directory. Next, let's see the effect of `mv` on directories. First we will move our data file into `dir1` again:

```
[me@linuxbox playground]$ mv fun dir1
```

then move `dir1` into `dir2` and confirm it with `ls`:

```
[me@linuxbox playground]$ mv dir1 dir2
[me@linuxbox playground]$ ls -l dir2
total 4
drwxrwxr-x 2 me me 4096 2008-01-11 06:06 dir1
[me@linuxbox playground]$ ls -l dir2/dir1
total 4
-rw-r--r-- 1 me me 1650 2008-01-10 16:33 fun
```

Note that since `dir2` already existed, `mv` moved `dir1` into `dir2`. If `dir2` had not existed, `mv` would have renamed `dir1` to `dir2`. Lastly, let's put everything back:

```
[me@linuxbox playground]$ mv dir2/dir1 .
[me@linuxbox playground]$ mv dir1/fun .
```

Creating Hard Links

Now we'll try some links. First the hard links. Create some links to our data file like so:

```
[me@linuxbox playground]$ ln fun fun-hard
[me@linuxbox playground]$ ln fun dir1/fun-hard
[me@linuxbox playground]$ ln fun dir2/fun-hard
```


So now we have four instances of the file “fun”. Let's take a look our playground directory:

```
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me  me  4096 2008-01-14 16:17 dir1
drwxrwxr-x 2 me  me  4096 2008-01-14 16:17 dir2
-rw-r--r-- 4 me  me  1650 2008-01-10 16:33 fun
-rw-r--r-- 4 me  me  1650 2008-01-10 16:33 fun-hard
```

One thing you notice is that the second field in the listing for `fun` and `fun-hard` both contain a “4” which is the number of hard links that now exist for the file. You'll remember that a file will always have at least one because the file's name is created by a link. So, how do we know that `fun` and `fun-hard` are, in fact, the same file? In this case, `ls` is not very helpful. While we can see that `fun` and `fun-hard` are both the same size (field 5), our listing provides no way to be sure. To solve this problem, we're going to have to dig a little deeper.

When thinking about hard links, it is helpful to imagine that files are made up of two parts: the data part containing the file's contents and the name part which holds the file's name. When we create hard links, we are actually creating additional name parts that all refer to the same data part. The system assigns a chain of disk blocks to what is called an *inode* which is then associated with the name part. Each hard link therefore refers to a specific inode containing the file's contents.

The `ls` command has a way to reveal this information. It is invoked with the “-i” option:

```
[me@linuxbox playground]$ ls -li
total 16
12353539 drwxrwxr-x 2 me  me  4096 2008-01-14 16:17 dir1
12353540 drwxrwxr-x 2 me  me  4096 2008-01-14 16:17 dir2
12353538 -rw-r--r-- 4 me  me  1650 2008-01-10 16:33 fun
12353538 -rw-r--r-- 4 me  me  1650 2008-01-10 16:33 fun-hard
```

In this version of the listing the first field is the inode number and, as we can see, both `fun` and `fun-hard` share the same inode number thus confirming they are the same file.

Creating Symbolic Links

Symbolic links were created to overcome the two disadvantages of hard links: hard links cannot span physical devices and hard links cannot reference directories, only files.

5 - Manipulating Files And Directories

Symbolic links are a special type of file that contains a text pointer to the target file or directory.

Creating symbolic links is similar to creating hard links:

```
[me@linuxbox playground]$ ln -s fun fun-sym
[me@linuxbox playground]$ ln -s ../fun dir1/fun-sym
[me@linuxbox playground]$ ln -s ../fun dir2/fun-sym
```

The first example is pretty straight forward, we simply add the “-s” option to create a symbolic link rather than a hard link but what about the next two? Remember, when you create a symbolic link, you are creating a text description of where the target file is relative to the symbolic link. It's easier to see if you look at the ls output:

```
[me@linuxbox playground]$ ls -l dir1
total 4
-rw-r--r-- 4 me me 1650 2008-01-10 16:33 fun-hard
lrwxrwxrwx 1 me me 6 2008-01-15 15:17 fun-sym -> ../fun
```

The listing for fun-sym in dir1 shows that it is a symbolic link by the leading “l” in the first field and that it points to “../fun” which is correct. Relative to the location of fun-sym, fun is in the directory above it. Notice too, that the length of the symbolic link file is 6, the number of characters in the string “../fun” rather than the length of the file it is pointing to.

When creating symbolic links you can either use absolute pathnames:

```
ln -s /home/me/playground/fun dir1/fun-sym
```

or relative pathnames as we did in our earlier example. Using relative pathnames is more desirable because it allows a directory containing symbolic links to be renamed and/or moved without breaking the links.

In addition to regular files, symbolic links can also reference directories:

```
[me@linuxbox playground]$ ln -s dir1 dir1-sym
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me me 4096 2008-01-15 15:17 dir1
lrwxrwxrwx 1 me me 4 2008-01-16 14:45 dir1-sym -> dir1
```

```
drwxrwxr-x 2 me me 4096 2008-01-15 15:17 dir2
-rw-r--r-- 4 me me 1650 2008-01-10 16:33 fun
-rw-r--r-- 4 me me 1650 2008-01-10 16:33 fun-hard
lrwxrwxrwx 1 me me 3 2008-01-15 15:15 fun-sym -> fun
```

Removing Files And Directories

As we covered earlier, the `rm` command is used to delete files and directories. We are going to use it to clean up our playground a little bit. First, let's delete one of our hard links:

```
[me@linuxbox playground]$ rm fun-hard
[me@linuxbox playground]$ ls -l
total 12
drwxrwxr-x 2 me me 4096 2008-01-15 15:17 dir1
lrwxrwxrwx 1 me me 4 2008-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me me 4096 2008-01-15 15:17 dir2
-rw-r--r-- 3 me me 1650 2008-01-10 16:33 fun
lrwxrwxrwx 1 me me 3 2008-01-15 15:15 fun-sym -> fun
```

That worked as expected. The file `fun-hard` is gone and the link count shown for `fun` is reduced from four to three as indicated in the second field of the directory listing. Next, we'll delete the file `fun`, and just for enjoyment, we'll include the `-i` option to show what that does:

```
[me@linuxbox playground]$ rm -i fun
rm: remove regular file `fun'?
```

Enter `y` at the prompt and the file is deleted. But let's look at the output of `ls` now. Noticed what happened to `fun-sym`? Since it's a symbolic link pointing to a now-nonexistent file, the link is broken:

```
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 bshotts bshotts 4096 2008-01-15 15:17 dir1
lrwxrwxrwx 1 bshotts bshotts 4 2008-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 bshotts bshotts 4096 2008-01-15 15:17 dir2
lrwxrwxrwx 1 bshotts bshotts 3 2008-01-15 15:15 fun-sym -> fun
```

5 - Manipulating Files And Directories

Most Linux distributions configure `ls` to display broken links. On my Fedora box, broken links are displayed in blinking red text! The presence of a broken link is not in and of itself dangerous but it is rather messy. If we try to use a broke link it will do this:

```
[me@linuxbox playground]$ less fun-sym
fun-sym: No such file or directory
```

Let's clean up a little. We'll delete the symbolic links:

```
[me@linuxbox playground]$ rm fun-sym dir1-sym
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 bshotts bshotts 4096 2008-01-15 15:17 dir1
drwxrwxr-x 2 bshotts bshotts 4096 2008-01-15 15:17 dir2
```

One thing to remember about symbolic links is most file operations are carried out on the link's target not the link itself. `rm` is an exception. When you delete a link, it is the link that is deleted, not the target.

Finally, we will remove our playground. To do this we will return to our home directory and use `rm` with the recursive option (`-r`) to delete playground and all of its contents including its subdirectories:

```
[me@linuxbox playground]$ cd
[me@linuxbox ~]$ rm -r playground
```

Summing Up

We've covered a lot of ground here and I know it will take a while to fully sink in. You should go back and repeat the playground exercise over and over until it makes sense to you. It is important that you get a good understanding of basic file manipulation commands and wildcards. Feel free to expand on the playground exercise adding more files and directories, using wildcards to specify files for various operations. I realize that the concept of links is a little confusing at first. You may not create that many links in your lifetime but you will encounter them and when you need them, they can be a real lifesaver.

6 - Working With Commands

Up to this point, we have seen a series of mysterious commands each with its own mysterious options and arguments. In this chapter we will attempt to remove some of that mystery and even create some of our own commands. The commands introduced in this chapter are:

- `type` – Indicate how a command name is interpreted
- `which` – Display which executable program will be executed
- `man` – Display a command's manual page
- `apropos` – Display a list of appropriate commands
- `info` – Display a command's info entry
- `what is` – Display a very brief description of a command
- `alias` – Create an alias for a command

What Exactly Are Commands?

A command can be one of four different things:

1. **An executable program** like all those files we saw in `/usr/bin`. Within the category of executables, programs can be *compiled binaries* such as C and C++ programs or programs written in *scripting languages* such as the shell, perl, python, ruby, etc.
2. **A commands built into the shell itself.** `bash` supports a number of commands internally called *shell builtins*. The `cd` command, for example, is a shell builtin.
3. **A shell function.** These are miniature shell scripts incorporated into your *environment*. We will cover configuring your environment and writing shell functions in later chapters, but for now, just be aware that they exist.
4. **An aliases.** Commands that you can define yourself built from other commands.

Identifying Commands

It is often useful to know exactly which of the four kinds of commands is being used and Linux provides a couple of commands for this purpose.

type – Display A Command's Type

The `type` command is a shell builtin that displays the kind of command the shell will execute given a particular command name. It works like this:

```
type command
```

where “command” is the name of the command you want to display. Here are some examples:

```
[me@linuxbox ~]$ type type
type is a shell builtin
[me@linuxbox ~]$ type ls
ls is aliased to `ls --color=tty'
[me@linuxbox ~]$ type cp
cp is /bin/cp
```

Here we see the results for three different commands. Notice the one for `ls` (taken from a Fedora system) and how the `ls` command is actually an alias for the `ls` command with the “-- color=tty” option added. Now we know why the output from `ls` is displayed in color!

which – Display An Executable's Location

Sometimes there is more than one version of an executable program installed on your system. While this is not very common on desktop systems, it's not unusual on large servers. To determine the exact location of a given executable, the `which` command is used:

```
[me@linuxbox ~]$ which ls
/bin/ls
```

`which` only works for executable programs, not builtins nor aliases that are substitutes for actual executable programs. When we try to use `which` on a shell builtin, for

example `cd`, we get an error message:

```
[me@linuxbox ~]$ which cd
/usr/bin/which: no cd in
(/opt/jre1.6.0_03/bin:/usr/lib/qt-3.3/bin:/usr/kerberos/bin:/opt/jre1
.6.0_03/bin:/usr/lib/ccache:/usr/local/bin:/usr/bin:/bin:/home/me/bin
)
```

which is a fancy way of saying “command not found.”

Getting A Command's Documentation

With this knowledge of what a command is, we can now search for the documentation available for each kind of command.

help – Get Help For Shell Builtins

`bash` has a built-in help facility available for each of the shell builtins. To use it, type “help” followed by the name of the shell builtin. For example:

```
[me@linuxbox ~]$ help cd
cd: cd [-L|-P] [dir]
Change the current directory to DIR. The variable $HOME is the
default DIR. The variable CDPATH defines the search path for the
directory containing DIR. Alternative directory names in CDPATH are
separated by a colon (:). A null directory name is the same as the
current directory, i.e. `.'. If DIR begins with a slash (/), then
CDPATH is not used. If the directory is not found, and the shell
option `cdable_vars' is set, then try the word as a variable name.
If that variable has a value, then cd to the value of that variable.
The -P option says to use the physical directory structure instead of
following symbolic links; the -L option forces symbolic links to be
followed.
```

A note on notation: When square brackets appear in the description a command's syntax, they indicate optional items. A vertical bar character indicates mutually exclusive items. In the case of the `cd` command above:

```
cd [-L|-P] [dir]
```

This notation says that the command `cd` may be followed optionally by either a “-L” or a “-P” and further optionally followed by the argument “dir”.

As the output of `help` for the `cd` commands shows us, while it is concise and accurate, it

is by no means tutorial and as we can see, it also seems to mention a lot of things we haven't talked about yet! Don't worry. We'll get there.

--help – Display Usage Information

Many executable programs support a “--help” option that displays a description of the command's supported syntax and options. For example:

```
[me@linuxbox ~]$ mkdir --help
Usage: mkdir [OPTION] DIRECTORY...
Create the DIRECTORY(ies), if they do not already exist.

  -Z, --context=CONTEXT (SELinux) set security context to CONTEXT
Mandatory arguments to long options are mandatory for short options
too.
  -m, --mode=MODE      set file mode (as in chmod), not a=rwx - umask
  -p, --parents        no error if existing, make parent directories as
                      needed
  -v, --verbose        print a message for each created directory
  --help              display this help and exit
  --version            output version information and exit

Report bugs to <bug-coreutils@gnu.org>.
```

Some programs don't support the “--help” option, but try it anyway. Often times it results in an error message that will reveal the same usage information.

man – Display A Program's Manual Page

Most executable programs intended for command line use provide a formal piece of documentation called a *manual* or *man page*. A special paging program called `man` is used to view them. It is used like this:

```
man program
```

where “program” is the name of the command you are searching for.

Man pages vary somewhat in format but generally contain a title, a synopsis of the command's syntax, a description of the command's purpose, and a listing and description of each of the command's options. Man pages, however, do not usually include examples and are intended as a reference, not a tutorial. As an example, let's try viewing the man

page for the `ls` command:

```
[me@linuxbox ~]$ man ls
```

On most Linux systems, `man` uses `less` to display the manual page so all of the familiar `less` commands work while displaying the page.

The “manual” that `man` displays is broken into sections and not only covers user commands but also system administration commands, programming interfaces, file formats and more. The table below describes the layout of the manual:

Section	Contents
1	User commands
2	Programming interfaces kernel system calls
3	Programming interfaces to the C library
4	Special files such as device nodes and drivers
5	File formats
6	Games and amusements such as screen savers
7	Miscellaneous
8	System administration commands

Sometimes you need to look in a particular section of the manual to find what you are looking for. This is particularly true if are looking for a file format that is also the name of a command. Without specifying a section number you will always get the first instance of a match, probably in section 1. To specify a section number, use `man` like this:

```
man section search_term
```

For example:

```
[me@linuxbox ~]$ man 5 fstab
```

This will display the man page for the format of the `/etc/fstab` file.

apropos – Display Appropriate Commands

It is also possible to search the list of man pages for possible matches based on a search term. It's very crude but sometimes helpful. Here is an example of a search for man pages using the search term “floppy”:

```
[me@linuxbox ~]$ apropos floppy
create_floppy_devices (8) - udev callout to create all possible
                           floppy device based on the CMOS type
fdformat                (8) - Low-level formats a floppy disk
floppy                  (8) - format floppy disks
gfloppy                 (1) - a simple floppy formatter for the GNOME
mbadbblocks             (1) - tests a floppy disk, and marks the bad
                           blocks in the FAT
mformat                 (1) - add an MSDOS filesystem to a low-level
                           formatted floppy disk
```

The first field in each line of output is the name of the man page, the second field the section. Note that the man command with the “-k” option performs the exact same function as apropos.

what is – Display A Very Brief Description Of A Command

The `what is` program displays the name and one line description of a man page matching a specified keyword:

```
[me@linuxbox ~]$ what is ls
ls                    (1) - list directory contents
```

The Most Brutal Man Page Of Them All

As we have seen, the manual pages supplied with Linux and other Unix-like systems are intended as reference documentation and not as tutorials. Many man pages are hard to read but I think that the grand prize for difficulty has got to go to the man page for `bash`. As I was doing my research for this book, I gave it careful review to insure that I was covering most of its topics. When printed, it's over eighty pages long and extremely dense and its structure makes absolutely no sense to a new user.

But on the other hand, it is very accurate and concise, as well as, being extremely complete. So give it a look if you dare and look forward to the day when you can read it and it all makes sense.

info – Display A Program's Info Entry

The GNU Project provides an alternative to man pages for their programs called “info.” Info pages are displayed with a reader program named, appropriately enough, `info`. Info pages are *hyperlinked* much like web pages. Here is a sample:

```
File: coreutils.info, Node: ls invocation, Next: dir invocation,
Up: Directory listing

10.1 `ls': List directory contents
=====

The `ls' program lists information about files (of any type,
including directories). Options and file arguments can be intermixed
arbitrarily, as usual.

    For non-option command-line arguments that are directories, by
    default `ls' lists the contents of directories, not recursively, and
    omitting files with names beginning with `.'. For other non-option
    arguments, by default `ls' lists just the file name. If no non-
    option argument is specified, `ls' operates on the current directory,
    acting as if it had been invoked with a single argument of `.'.

    By default, the output is sorted alphabetically, according to the
--zz-Info: (coreutils.info.gz)ls invocation, 63 lines --Top-----
```

The `info` program reads *info files* which are tree structured into individual *nodes* each containing a single topic. Info files contain hyperlinks that can move you from node to node. A hyperlink can be identified by its leading asterisk and is activated by placing the cursor upon it and pressing the enter key.

To invoke `info`, type “info” followed optionally by the name of a program. Below is a table of commands used to control the reader while displaying an info page:

Command	Action
?	Display command help
PgUp	Display previous page
PgDn	Display next page
n	Next - Display the next node
p	Previous - Display the previous node
u	Up - Display the node above the currently displayed node, usually a menu.
Enter	Follow the hyperlink at the cursor location
q	Quit

All of the command line programs we have discussed so far are part of the GNU Project's "coreutils" package. So typing:

```
[me@linuxbox ~]$ info coreutils
```

will display a menu page with hyperlinks to each program contained in the coreutils package.

README And Other Program Documentation Files

Remember that many software packages installed on your system have documentation files residing in `/usr/share/doc`. Most of these are stored in plain text format and can be viewed with `less`. Some of the files are in HTML format and can be viewed with your web browser. You may encounter a few files ending with a ".gz" extension. This indicates that they have been compressed with the `gzip` compression program. The `gzip` package includes a special version of `less` called `zless` that will display the contents of `gzip`-compressed text files.

Creating Your Own Commands With `alias`

Now for our very first experience with programming! We will create a command of our own using the `alias` command. But before we start, I want to reveal a small command line trick. It's possible to put more than one command on a line by separating each command with a semicolon character. It works like this:

```
command1; command2; command3...
```

Here's the example we will use to demonstrate:

```
[me@linuxbox ~]$ cd /usr; ls; cd -  
bin  games  kerberos  lib64    local  share  tmp  
etc  include lib      libexec  sbin    src  
/home/me  
[me@linuxbox ~]$
```

As you can see, we have combined three commands on one line. First we change directory to /usr then list the directory and finally return to the original directory (by using 'cd -') so we end up where we started. Now let's turn this sequence into a new command using alias. The first thing we have to do is dream up a name for our new command. Let's try "test". But before we do that, it would be a good idea to find out if the name "test" is already being used. To find out, we can use the type command again:

```
[me@linuxbox ~]$ type test  
test is a shell builtin
```

Oops! The name "test" is already taken. Let's try "foo":

```
[me@linuxbox ~]$ type foo  
bash: type: foo: not found
```

Great! "foo" is not taken. So now let's create our alias:

```
[me@linuxbox ~]$ alias foo='cd /usr; ls; cd -'
```

Notice the structure of this command:

```
alias name='string'
```

After the command "alias" we give the command a name followed immediately (no whitespace allowed) by an equals sign followed immediately by a quoted string

6 - Working With Commands

containing the meaning to be assigned to the name. After you define your alias, it can be used anywhere the shell would expect a command. Let's try it:

```
[me@linuxbox ~]$ foo
bin  games  kerberos  lib64    local  share  tmp
etc  include lib      libexec  sbin    src
/home/me
[me@linuxbox ~]$
```

We can also use the `type` command again to see our alias:

```
[me@linuxbox ~]$ type foo
foo is aliased to `cd /usr; ls ; cd -`
```

To remove an alias, the `unalias` command is used, like so:

```
[me@linuxbox ~]$ unalias foo
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

While we purposefully avoided naming our alias with an existing command name, it is not uncommon to do so. This is often done to apply a commonly desired option to each invocation of a common command. For instance, we saw earlier how the `ls` command is often aliased to add color support:

```
[me@linuxbox ~]$ type ls
ls is aliased to `ls --color=tty`
```

To see all the aliases defined in your environment, use the `alias` command without arguments. Here are some of the aliases defined by default on a Fedora system. See if you can figure out what they all do:

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
```

There is one tiny problem with defining your own aliases on the command line. They vanish when your shell session ends. In a later chapter, we will see how you can add your own aliases to the files that establish your environment each time you log on, but for now, enjoy the fact that you have taken your first, albeit tiny, step into the world of shell programming!

Revisiting Old Friends

Now that we have learned how to find the documentation for commands, go and look up the documentation for all the commands we have encountered so far. Study what additional options are available and try them out!

7 – Redirection

In this lesson we are going to unleash what may be the coolest feature of the command line. It's called *I/O redirection*. The “I/O” stands for *input/output* and with this facility you can redirect the input and output of commands to and from files, as well as, connect multiple commands together into powerful command *pipelines*. To show off this facility, we will introduce the following commands:

- `cat` - Concatenate files
- `sort` - Sort lines of text
- `uniq` - Report or omit repeated lines
- `grep` - Print lines matching a pattern
- `wc` - Print newline, word, and byte counts for each file
- `head` - Output the first part of files
- `tail` - Output last part of file
- `tee` - Read from standard input and write to standard output and files

Standard Input, Output, And Error

Many of the programs that we have used so far produce output of some kind. This output often consists of two types. First, we have the program's results, that is, the data the program is designed to produce, and second, we have status and error messages that tell us how the program is getting along. If we look at a command like `ls`, we can see that it displays its results and its error messages on the screen.

Keeping with the Unix theme of “everything is a file,” programs such as `ls` actually send their results to a special file called *standard output* (often expressed as *stdout*) and their status messages to another file called *standard error* (*stderr*). By default, both standard output and standard error are linked to the screen.

In addition, many programs take input from a facility called *standard input* (*stdin*) which is, by default, attached to the keyboard.

I/O redirection allows us to change where output goes and where input comes from. Normally, output goes to the screen and input comes from the keyboard, but with I/O redirection, we can change that.

Redirecting Standard Output

I/O redirection allows us to redefine where standard output goes. To redirect standard output to another file besides the screen, we use the “>” redirection operator followed by the name of the file. Why would we want to do this? It's often useful to store the output of a command in a file. For example, we could tell the shell to send the output of the `ls` command to the file `ls-output.txt` instead of the screen:

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Here we created a long listing of the `/usr/bin` directory and sent the results to the file `ls-output.txt`. Let's examine the redirected output of the command:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me  me  167878 2008-02-01 15:07 ls-output.txt
```

Good, a nice, large, text file. If we look at the file with `less`, we will see that the file `ls-output.txt` does indeed contains the results from our previous `ls` command:

```
[me@linuxbox ~]$ less ls-output.txt
```

Now, let's repeat our redirection test, but this time with a twist. We'll change the name of the directory passed to `ls` to one that does not exist:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt
ls: cannot access /bin/usr: No such file or directory
```

We received an error message. It makes sense that an error was generated since we specified the non-existent directory `/bin/usr`, but why was the error message displayed on the screen rather than being redirected to the file `ls-output.txt`? The answer is that the `ls` program does not send its error messages to standard output. Instead, it, like most well-written Unix programs, sends its error messages to standard

error. Since we only redirected standard output and not standard error, the error message was still sent to the screen. We'll see how to redirect standard error in just a minute, but first, let's look at what happened to our output file:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me  me   0 2008-02-01 15:08 ls-output.txt
```

The file now has zero length! This is because when we redirect output with the “>” redirection operator, the destination file is always rewritten from the beginning. Since our `ls` command generated no results and only an error message, the redirection operation started to rewrite the file and then stopped because of the error resulting in its truncation. In fact, if we ever need to actually truncate a file (or create a new, empty file) we can use a trick like this:

```
[me@linuxbox ~]$ > ls-output.txt
```

Simply using the redirection operator with no command proceeding it will truncate an existing file or create a new, empty file.

So, how can we append redirected output to a file instead of overwriting the file from the beginning? For that, we use the “>>” redirection operator, like so:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
```

Using the “>>” operator will result in the output being appended to the file. If the file does not already exist, it is created just as though the “>” operator had been used. Let's put it to the test:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me  me  503634 2008-02-01 15:45 ls-output.txt
```

We repeated the command three times resulting in an output file three times as large.

Redirecting Standard Error

Redirecting standard error lacks the ease of a dedicated redirection operator. Rather, to redirect standard error we must refer to its *file descriptor*. A program can produce output on any of several numbered file streams. While we have referred to first three of these file streams as standard input, output and error, they are actually file descriptors zero, one and two respectively. Since standard error uses file descriptor number two, we can redirect standard error with this notation:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> ls-error.txt
```

The file descriptor “2” is placed immediately before the redirection operator to perform the redirection.

Redirecting Standard Output And Standard Error To One File

There are cases where we may wish to capture all of the output of a command to a single file. To do this, we must redirect both standard output and standard error at the same time. There are two ways to do this. First, the traditional way, which is more portable, as it works with old versions of the shell:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt 2>&1
```

And second, using a modern shorthand provided by recent versions of `bash`:

```
[me@linuxbox ~]$ ls -l /bin/usr &> ls-output.txt
```

Notice that in the first method, the order of the redirections is significant. The redirection of standard error must always occur *after* redirecting standard output or else it doesn't work.

Disposing Of Unwanted Output

Sometimes “silence is golden” and we don't want output from a command, we just want to throw it away. This applies particularly to error and status messages. The system provides a way to do this by redirecting output to a special file called “/dev/null”. This file is a system device called a *bit bucket* which accepts input and does nothing with it. To suppress error messages from a command we do this:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> /dev/null
```

/dev/null In Unix Culture

The bit bucket is an ancient Unix concept and due to its universality, has appeared in many parts of Unix culture. When someone says he/she is sending your comments to /dev/null, now you know what it means. For more examples, see the [Wikipedia article on “/dev/null”](#).

Redirecting Standard Input

Up to now, we haven't encountered any commands that make use of standard input (actually we have, but I'm saving that surprise for a little bit later), so we need to introduce one.

cat – Concatenate Files

The `cat` command reads one or more files and copies them standard output like so:

```
cat [file...]
```

In most cases, you can think of `cat` as being analogous to the `TYPE` command in DOS. You can use it to display files without paging, for example:

```
[me@linuxbox ~]$ cat ls-output.txt
```

will display the contents of the file `ls-output.txt`. `cat` is often used to display

short text files. Since `cat` can accept more than one file as an argument, it can also be used to join files together. Say we have downloaded a large file that has been split into multiple parts (I often see multimedia files split this way on USENET) and we want to join them back together. If the files were named:

```
movie.mpeg.001  movie.mpeg.002  ...  movie.mpeg.099
```

we could join them back together with this command:

```
cat movie.mpeg.0* > movie.mpeg
```

Since wildcards always expand in sorted order, the arguments will be arranged in the correct order.

This is all well and good, but what does this have to do with standard input? Nothing yet, but let's try something else. What happens if we type “`cat`” with no arguments:

```
[me@linuxbox ~]$ cat
```

Nothing happens, it just sits there like it's hung. It may seem that way but it's really doing exactly what it's supposed to.

If `cat` is not given any arguments, it reads from standard input and since standard input is by default attached to the keyboard, it's waiting for us to type something! Try this:

```
[me@linuxbox ~]$ cat
The quick brown fox jumped over the lazy dog.
```

Next, type a `Ctrl-d` (i.e. hold down the `Ctrl` key and press “`d`”) to tell `cat` that it has reached *end of file* (EOF) on standard input:

```
[me@linuxbox ~]$ cat
The quick brown fox jumped over the lazy dog.
The quick brown fox jumped over the lazy dog.
```

In the absence of file name arguments, `cat` copies standard input to standard output so we see our line of text repeated. We can use this behavior to create short text files. Let's say that we wanted to create a file called “`lazy_dog.txt`” containing the text in our example. We would do this:

```
[me@linuxbox ~]$ cat > lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Type the command followed by the text we want in to place in the file. Remember to type `Ctrl-d` at the end. Using the command line, we have implemented the world's dumbest word processor! To see our results, we can use `cat` to copy the file to stdout again:

```
[me@linuxbox ~]$ cat lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Now that we know how `cat` accepts standard input in addition to file name arguments, let's try redirecting standard input:

```
[me@linuxbox ~]$ cat < lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Using the “<” operator, we redirect the source of standard input from the keyboard to the file `lazy_dog.txt`.

Before we move on, check out the man page for `cat`, as it has several interesting options.

Pipelines

The ability of commands to read data from standard input and send to standard output is utilized by a shell feature called pipelines. Using the pipe operator “|” (vertical bar), the output of one command can be *piped* into the input of another:

```
command1 | command2
```

To fully demonstrate this we are going to need some commands. Remember how I said there was one we already knew that accepts standard input? It's `less`. We can use `less` to page the output of any command that sends its results to standard output:

```
[me@linuxbox ~]$ ls -l /usr/bin | less
```

We will use this technique constantly.

Filters

Pipelines are often used to perform complex operations on data. It is possible to put several commands together into a pipeline. Frequently, the commands used this way are referred to as *filters*. Filters take input, change it somehow and then output it. The first one we will try is `sort`. Imagine we wanted to make a list of all of the executable programs in `/bin` and `/usr/bin`, put them in sorted order and view it:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | less
```

Since we specified two directories (`/bin` and `/usr/bin`), the output of `ls` would have consisted of two sorted lists, one for each directory. By including `sort` in our pipeline, we changed the data to produce a single, sorted list.

uniq - Report Or Omit Repeated Lines

The `uniq` command is often used in conjunction with `sort`. `uniq` inputs a sorted list of data and, by default, removes any duplicates from the list. So, to make sure our list has no duplicates (that is, any programs of the same name that appear in both the `/bin` and `/usr/bin` directories) we will add `uniq` to our pipeline:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | less
```

If we want to see the list of duplicates, we add the “-d” option to `uniq` like so:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq -d | less
```

wc – Print Line, Word, And Byte Counts

The `wc` (word count) command is used to display the number of lines, words, and bytes contained in files. For example:

```
[me@linuxbox ~]$ wc ls-output.txt
7902  64566 503634 ls-output.txt
```

In this case it prints out three numbers: lines, words, and bytes contained in `ls-output.txt`. Like our previous commands, if executed without command line arguments, `wc` accepts standard input. The “-l” option limits its output to only report lines. Adding it to a pipeline is a handy way to count things. To see the number of programs we have in our sorted list, we can do this:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | wc -l
2728
```

grep – Print Lines Matching A Pattern

`grep` is a powerful program used to find text patterns within files. It's used like this:

```
grep pattern [file...]
```

When `grep` encounters “pattern” in the file, it prints out the line containing it. The pattern that `grep` can match can be very complex, but for now we will concentrate on simple text matches. We'll cover the advanced patterns, called *regular expressions* in a later chapter.

Let's say we want to find all the files in our list of programs that had the word “zip” embedded in the name. Such a search might give us an idea of some of the programs on our system that had something to do with file compression. We would do this:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

There are a couple of handy options for `grep`: “-i” which causes `grep` to ignore case when performing the search (normally searches are case sensitive) and “-v” which tells `grep` to only print lines that do not match the pattern.

head / tail – Print First/Last Part Of Files

Sometimes you don't want all of the output from a command. You may only want either the first or last few lines. The `head` and `tail` commands print the beginning section or the ending section of a file. By default they print the first/last ten lines but this can be adjusted with the “-n” option:

```
[me@linuxbox ~]$ head -n 5 ls-output.txt
total 343496
-rwxr-xr-x 1 root root      31316 2007-12-05 08:58 [
-rwxr-xr-x 1 root root      8240 2007-12-09 13:39 411toppm
-rwxr-xr-x 1 root root    111276 2007-11-26 14:27 a2p
-rwxr-xr-x 1 root root    25368 2006-10-06 20:16 a52dec
[me@linuxbox ~]$ tail -n 5 ls-output.txt
-rwxr-xr-x 1 root root      5234 2007-06-27 10:56 znew
-rwxr-xr-x 1 root root       691 2005-09-10 04:21 zonetab2pot.py
-rw-r--r-- 1 root root       930 2007-11-01 12:23 zonetab2pot.pyc
-rw-r--r-- 1 root root       930 2007-11-01 12:23 zonetab2pot.pyo
lrwxrwxrwx 1 root root         6 2008-01-31 05:22 zsoelim -> soelim
```

These can be used in pipelines as well:

```
[me@linuxbox ~]$ ls /usr/bin | tail -n 5
znew
zonetab2pot.py
zonetab2pot.pyc
zonetab2pot.pyo
zsoelim
```

`tail` has an option which allows you to view files in real-time. This is useful for watching the progress of log files as they are being written. In the following example, we will look at the `messages` file in `/var/log`. You will require superuser privileges to do this on some Linux distributions since the `/var/log/messages` file may contain security information:

```
[me@linuxbox ~]$ tail -f /var/log/messages
Feb  8 13:40:05 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 13:40:05 twin4 dhclient: bound to 192.168.1.4 -- renewal in
1652 seconds.
Feb  8 13:55:32 twin4 mountd[3953]: /var/NFSv4/musicbox exported to
both 192.168.1.0/24 and twin7.localdomain in
192.168.1.0/24,twin7.localdomain
```

```
Feb  8 14:07:37 twin4 dhclient: DHCPREQUEST on eth0 to 192.168.1.1
port 67
Feb  8 14:07:37 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 14:07:37 twin4 dhclient: bound to 192.168.1.4 -- renewal in
1771 seconds.
Feb  8 14:09:56 twin4 smartd[3468]: Device: /dev/hda, SMART
Prefailure Attribute: 8 Seek_Time_Performance changed from 237 to 236
Feb  8 14:10:37 twin4 mountd[3953]: /var/NFSv4/musicbox exported to
both 192.168.1.0/24 and twin7.localdomain in
192.168.1.0/24,twin7.localdomain
Feb  8 14:25:07 twin4 sshd(pam_unix)[29234]: session opened for user
me by (uid=0)
Feb  8 14:25:36 twin4 su(pam_unix)[29279]: session opened for user
root by me(uid=500)
```

Using the “-f” option, `tail` continues to monitor the file and when new lines are appended, they immediately appear on the display. This continues until you type `Ctrl-C`.

tee – Read From Stdin And Output To Stdout And Files

In keeping with our plumbing metaphor, Linux provides a command called `tee` which creates a “tee” fitting on our pipe. The `tee` program reads standard input and copies it to both standard output (allowing the data to continue down the pipeline) and to one or more files. This is useful for capturing a pipeline's contents at an intermediate stage of processing. Here we repeat one of our earlier examples, this time including `tee` to capture the entire directory listing to the file `ls.txt` before `grep` filters the pipeline's contents:

```
[me@linuxbox ~]$ ls /usr/bin | tee ls.txt | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

Summing Up

As always, you should check out the documentation of each of the commands we have covered in this chapter. I have only described their most basic usage. They all have a number of interesting options. As you gain Linux experience, you will see that the redirection feature of the command line is extremely useful for solving specialized problems. There are many commands that make use of standard input and output, and almost all command line programs use standard error to display their informative messages.

Linux Is About Imagination

When I am asked to explain the difference between Windows and Linux, I often use a toy analogy.

Windows is like a Game Boy. You go to the store and buy one all shiny new in the box. You take it home, turn it on and play with it. Pretty graphics, cute sounds. After a while though, you get tired of the game that came with it so you go back to the store and buy another one. This cycle repeats over and over. Finally, you go back to the store and say to the person behind the counter, "I want a game that does this!" only to be told that no such a game exists because there is no "market demand" for it. Then you say, "But I only need to change this one thing!" The person behind the counter says you can't change it. The games are all sealed up in their cartridges. You discover that your toy is limited to the games that others have decided that you need and no more.

Linux, on the other hand, is like the world's largest Erector Set. You open it up and it's just a huge collection of parts. A lot of steel struts, screws, nuts, gears, pulleys, motors, and a few suggestions on what to build. So you start to play with it. You build one of the suggestions and then another. After a while you discover that you have your own ideas of what to make. You don't ever have to go back to the store, as you already have everything you need. The Erector Set takes on the shape of your imagination. It does what you want.

Your choice of toys is, of course, a personal thing, so which toy would you find more satisfying?

8 - Seeing The World As The Shell Sees It

In this chapter we are going to look at some of the “magic” that occurs on the command line when you press the enter key. While we will examine several interesting and complex features of the shell, we will do it with just one new command:

- `echo` – Display A Line Of Text

Expansion

Each time you type a command line and press the enter key, `bash` performs several processes upon the text before it carries out your command. We have seen a couple of cases of how a simple character sequence, for example “*”, can have a rather large meaning to the shell. The process that makes this happen is called *expansion*. With expansion, you type something and it is expanded into something else before the shell acts upon it. To demonstrate what we mean by this, let's take a look at the `echo` command. `echo` is a shell builtin that performs a very simple task. It prints out its text arguments on standard output:

```
[me@linuxbox ~]$ echo this is a test
this is a test
```

That's pretty straight forward. Whatever is passed to it get displayed. Let's try another example:

```
[me@linuxbox ~]$ echo *
Desktop Documents ls-output.txt Music Pictures Public Templates
Videos
```

So what just happened? Why didn't `echo` print “*”? As you recall from our work with wildcards, the “*” character means match any characters in a file name, but what we didn't see in our original discussion was how the shell does that. The simple answer is that the shell expands the “*” into something else (in this instance, the names of the files

in the current working directory) before the `echo` command is executed. When the enter key is pressed, the shell automatically expands any qualifying characters on the command line before the command is carried out, so the `echo` command never saw the “*”, only its expanded result. Knowing this, we can see that `echo` behaved as expected.

Pathname Expansion

The mechanism by which wildcards work is called *pathname expansion*. If we try some of the techniques that we employed in our earlier chapters, we will see that they are really expansions. Given a home directory that looks like this:

```
[me@linuxbox ~]$ ls
Desktop  ls-output.txt  Pictures  Templates
Documents Music          Public    Videos
```

we could carry out the following expansions:

```
[me@linuxbox ~]$ echo D*
Desktop Documents
```

and:

```
[me@linuxbox ~]$ echo *s
Documents Pictures Templates Videos
```

or even:

```
[me@linuxbox ~]$ echo [[:upper:]]*
Desktop Documents Music Pictures Public Templates Videos
```

and looking beyond our home directory:

```
[me@linuxbox ~]$ echo /usr/*/share
/usr/kerberos/share /usr/local/share
```

Tilde Expansion

As you may recall from our introduction to the `cd` command, the tilde character (“~”) has a special meaning. When used at the beginning of a word, it expands into the name of the home directory of the named user, or if no user is named, your home directory:

```
[me@linuxbox ~]$ echo ~  
/home/me
```

If user “foo” has an account, then:

```
[me@linuxbox ~]$ echo ~foo  
/home/foo
```

Arithmetic Expansion

The shell allows arithmetic to be performed by expansion. This allows us to use the shell prompt as a calculator:

```
[me@linuxbox ~]$ echo $((2 + 2))  
4
```

Arithmetic expansion only supports integers (whole numbers, no decimals) but can perform quite a number of different operations. In the table below, we see some of the supported operators:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division, but remember, since expansion only supports integer arithmetic, results are the integers.
%	Modulo, which is fancy-speak for remainder.
**	Exponentiation

Spaces are not significant in arithmetic expressions and they may be nested. For

example, to multiply five squared by three:

```
[me@linuxbox ~]$ echo $((5**2) * 3)
75
```

Here is an example with division and remainder:

```
[me@linuxbox ~]$ echo Five divided by two equals $(5/2)
Five divided by two equals 2
[me@linuxbox ~]$ echo with $(5%2) left over.
with 1 left over.
```

Brace Expansion

Perhaps the strangest expansion is called *brace expansion*. With it, you can create multiple text strings from a pattern containing braces. Here's an example:

```
[me@linuxbox ~]$ echo Front-{A,B,C}-Back
Front-A-Back Front-B-Back Front-C-Back
```

Patterns to be brace expanded may contain a leading portion called a *preamble* and a trailing portion called a *postscript*. The brace expression itself may contain either a comma-separated list of strings, or a range of integers or single characters. The pattern may not contain embedded whitespace. Here is an example using a range of integers:

```
[me@linuxbox ~]$ echo Number_{1..5}
Number_1 Number_2 Number_3 Number_4 Number_5
```

A range of letters in reverse order:

```
[me@linuxbox ~]$ echo {Z..A}
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

Brace expansions may be nested:

8 - Seeing The World As The Shell Sees It

```
[me@linuxbox ~]$ echo a{A{1,2},B{3,4}}b  
aA1b aA2b aB3b aB4b
```

So what is this good for? The most common application is to make lists of files or directories to be created. For example, if you were a photographer (like me!) and had a large collection of images that you wanted to organize into years and months, the first thing you might do is create a series of directories named in numeric “Year-Month” format. This way, the directory names will sort in chronological order. You could type out a complete list of directories but that's a lot of work and it's error-prone too. Instead, you could do this:

```
[me@linuxbox ~]$ cd Pics  
[me@linuxbox Pics]$ mkdir {2007..2009}-0{1..9} {2007..2009}-{10..12}  
[me@linuxbox Pics]$ ls  
2007-01 2007-07 2008-01 2008-07 2009-01 2009-07  
2007-02 2007-08 2008-02 2008-08 2009-02 2009-08  
2007-03 2007-09 2008-03 2008-09 2009-03 2009-09  
2007-04 2007-10 2008-04 2008-10 2009-04 2009-10  
2007-05 2007-11 2008-05 2008-11 2009-05 2009-11  
2007-06 2007-12 2008-06 2008-12 2009-06 2009-12
```

Pretty slick!

Parameter Expansion

We're only going to touch briefly on parameter expansion in this chapter but we'll be extensively covering it later. It's a feature that is more useful in shell scripts than directly on the command line. Many of its capabilities have to do with the system's ability to store small chunks of data and to give each chunk a name. Many such chunks, more properly called *variables*, are available for your examination. For example, the variable named “USER” contains your user name. To invoke parameter expansion and reveal the contents of USER you would do this:

```
[me@linuxbox ~]$ echo $USER  
me
```

To see a list of available variables try this:

```
[me@linuxbox ~]$ printenv | less
```


You may have noticed that with other types of expansion, if you mistype a pattern, the results will be that the expansion will not take place and that the `echo` command will simply display the mistyped pattern. With parameter expansion, if you misspell the name of a variable the expansion will still take place but will result in an empty string:

```
[me@linuxbox ~]$ echo $SUER
[me@linuxbox ~]$
```

Command Substitution

Command substitution allows us to use the output of a command as an expansion:

```
[me@linuxbox ~]$ echo $(ls)
Desktop Documents ls-output.txt Music Pictures Public Templates
Videos
```

One of my favorites goes something like this:

```
[me@linuxbox ~]$ ls -l $(which cp)
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

Here we passed the results of `which cp` as an argument to the `ls` command thereby getting the listing of of the `cp` program without having to know its full pathname. We are not limited to just simple commands. Entire pipelines can be used (only partial output shown):

```
[me@linuxbox ~]$ file $(ls /usr/bin/* | grep zip)
/usr/bin/bunzip2:      symbolic link to `bzip2'
/usr/bin/bzip2:       ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), dynamically linked (uses shared libs), for
GNU/Linux 2.6.9, stripped
/usr/bin/bzip2recover: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), dynamically linked (uses shared libs), for
GNU/Linux 2.6.9, stripped
/usr/bin/funzip:      ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), dynamically linked (uses shared libs), for
GNU/Linux 2.6.9, stripped
/usr/bin/gpg-zip:     Bourne shell script text executable
```

```
/usr/bin/gunzip:      symbolic link to `../../bin/gunzip`  
/usr/bin/gzip:       symbolic link to `../../bin/gzip`  
/usr/bin/mzip:       symbolic link to `mtools`
```

In this example, the results of the pipeline became the argument list of the `file` command.

There is an alternate syntax for command substitution which you will often see in older shell programs which is also supported in `bash`. It uses *back-quotes* instead of the dollar sign and parentheses:

```
[me@linuxbox ~]$ ls `which cp`  
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

Quoting

Now that we've seen how many ways the shell can perform expansions, it's time to learn how we can control it. Take for example:

```
[me@linuxbox ~]$ echo this is a      test  
this is a test
```

or:

```
[me@linuxbox ~]$ echo The total is $100.00  
The total is 00.00
```

In the first example, *word-splitting* by the shell removed extra whitespace from the `echo` command's list of arguments and in the second example, parameter expansion substituted an empty string for the value of “\$1” because it was an undefined variable. The shell provides a mechanism called *quoting* to selectively suppress unwanted expansions.

Double Quotes

The first type of quoting we will look at is *double quotes*. If you place text inside double quotes, all the special characters used by the shell lose their special meaning and treated like ordinary characters. The exceptions are “\$”, “\” (backslash), and “`” (back-quote). This means that word-splitting, pathname expansion, tilde expansion, and brace

expansion are suppressed but that parameter expansion and command substitution are still carried out. Using double quotes is how we cope with filenames containing embedded spaces. Say you were the unfortunate victim of a file called `two words.txt`. If you tried to use this on the command line, word-splitting would cause this to be treated as two separate arguments rather than the desired single argument:

```
[me@linuxbox ~]$ ls -l two words.txt
ls: cannot access two: No such file or directory
ls: cannot access words.txt: No such file or directory
```

By using double quotes, we stop the word-splitting and get the desired result, further we can even repair the damage!:

```
[me@linuxbox ~]$ ls -l "two words.txt"
-rw-rw-r-- 1 me me 18 2008-02-20 13:03 two words.txt
[me@linuxbox ~]$ mv "two words.txt" two_words.txt
```

There! Now we don't have to keep typing those pesky double quotes.

Remember, parameter expansion, arithmetic expansion, and command substitution still take place within double quotes:

```
[me@linuxbox ~]$ echo "$USER $((2+2)) $(cal)"
me 4 February 2008
Su Mo Tu We Th Fr Sa
      1 2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29
```

We should take a moment to look at the effect of double quotes on command substitution. First let's look a little deeper at how word splitting works. In our earlier example we saw how word-splitting appears to remove extra spaces in our text:

```
[me@linuxbox ~]$ echo this is a test
this is a test
```

By default, word-splitting looks for the presence of spaces, tabs, and newlines (linefeed

8 - Seeing The World As The Shell Sees It

characters) and treats them as *delimiters* between words. This means that unquoted spaces, tabs, and newlines are not considered part of the text. They only serve as separators. Since they separate the words into different arguments, our example command line contains a command followed by four distinct arguments. If we add double quotes:

```
[me@linuxbox ~]$ echo "this is a    test"
this is a    test
```

word-splitting is suppressed and the embedded spaces are not treated as delimiters, rather they become part of the argument. Once the double quotes are added, our command line contains a command followed by a single argument.

The fact that newlines are considered delimiters by the word-splitting mechanism causes an interesting, albeit subtle, effect on command substitution. Consider the following:

```
[me@linuxbox ~]$ echo $(cal)
February 2008 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
[me@linuxbox ~]$ echo "$(cal)"
February 2008
Su Mo Tu We Th Fr Sa
          1 2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29
```

In the first instance, the unquoted command substitution resulted in a command line containing thirty-eight arguments. In the second, a command line with one argument that includes the embedded spaces and newlines.

Single Quotes

If you need to suppress *all* expansions, use *single quotes*. Here is a comparison of unquoted, double quotes, and single quotes:

```
[me@linuxbox ~]$ echo text ~/.txt {a,b} $(echo foo) $((2+2)) $USER
text /home/me/ls-output.txt a b foo 4 me
[me@linuxbox ~]$ echo "text ~/.txt {a,b} $(echo foo) $((2+2)) $USER"
text ~/.txt {a,b} foo 4 me
```

```
[me@linuxbox ~]$ echo 'text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER'
text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
```

As we can see, with each succeeding level of quoting, more and more of the expansions are suppressed.

Escaping Characters

Sometimes we only want to quote a single character. To do this, we can precede a character with a backslash which in this context is called the *escape character*. Often this is done inside double quotes to selectively prevent an expansion:

```
[me@linuxbox ~]$ echo "The balance for user $USER is: \$5.00"
The balance for user me is: $5.00
```

It is also common to use escaping to eliminate the special meaning of a character in a file name. For example, it is possible to use many characters in file names that normally have special meaning to the shell. These would include “\$”, “!”, “&”, and others. To include a special character in a file name you can do this:

```
[me@linuxbox ~]$ mv bad\&filename good_filename
```

To allow a backslash character to appear, escape it by typing “\\”. Note that within single quotes, the backslash loses its special meaning and is treated as an ordinary character.

Backslash Escape Sequences

In addition to its role as the escape character, the backslash is also used as part of a notation to represent certain special characters called *control codes*. In ASCII, the first thirty two characters in its coding scheme were used to transmit commands to teletype-like devices. Some of these codes are still familiar like tab, backspace, linefeed, and carriage return, while others are not, such as, null, end-of-transmission, and acknowledge.

The table below lists some of the common escape sequences you are likely to encounter. The idea behind this representation using the backslash originated in the C programming language and has been adopted by many others, including the

shell.

Escape Sequence	Meaning
<code>\a</code>	Bell (“Alert” - causes the computer to beep)
<code>\b</code>	Backspace
<code>\n</code>	Newline. On Unix-like systems this produces a linefeed.
<code>\r</code>	Carriage return
<code>\t</code>	Tab

Adding the “-e” option to `echo` will enable interpretation of escape sequences. You may also place them inside of `$' '`. Here, using the `sleep` command which is simple program that just waits for the specified number of seconds and then exits, we can create a primitive countdown timer:

```
sleep 10; echo -e "Time's up.\a"
```

Summing Up

As we move forward with using the shell, you will find that expansions and quoting will be used with increasing frequency, so it makes sense to get a good understanding of the way this works. Try looking at the at the “EXPANSION” and “QUOTING” sections of the bash man page for a more formal discussion of this topic.

9 - Advanced Keyboard Tricks

I often kiddingly describe Unix as, “the operating system for people who like to type.” Of course, the fact that it even has a command line is testament to that. But command line users don't like to type *that* much. Why else would so many commands have such short names like `cp`, `ls`, `mv`, and `rm`? In fact, one of the most cherished goals of the command line is laziness; doing the most work with the fewest number of keystrokes. Another goal is never having to lift your fingers from the keyboard, never reaching for the mouse. In this chapter, we will look at `bash` features that make keyboard use faster and more efficient.

In this chapter, the following commands will make an appearance:

- `clear` – clear the screen
- `history` – display the contents of the history list

Command Line Editing

`bash` uses a library (a shared collection of routines that different programs can use) called *Readline* to implement command line editing. We have already seen some of this. For example, we know that the arrow keys move the cursor but there are many more features.

Cursor Movement

The following table lists the keys used to move the cursor:

Key	Action
<code>Ctrl-a</code>	Move cursor to the beginning of the line.
<code>Ctrl-e</code>	Move cursor to the end of the line.
<code>Ctrl-f</code>	Move cursor forward one character. Same as the right arrow key.
<code>Ctrl-b</code>	Move cursor backward one character. Same as the left arrow key.
<code>Alt-f</code>	Move cursor forward one word.

9 - Advanced Keyboard Tricks

Alt-b	Move cursor backward one word.
Ctrl-l	Clear the screen and move the cursor to the top left corner. Note that there is a command named <code>clear</code> that does the same thing.

Modifying Text

Key	Action
Ctrl-d	Delete the character at the cursor location
Ctrl-t	Transpose (exchange) the character at the cursor location with the one preceding it.
Alt-t	Transpose the word at the cursor location with the one preceding it.
Alt-l	Convert the characters from the cursor location to the end of the word to lowercase.
Alt-u	Convert the characters from the cursor location to the end of the word to uppercase.

Cutting And Pasting (Killing And Yanking) Text

The Readline documentation uses the terms *killing* and *yanking* to refer to what we would commonly call cutting and pasting. Items that are cut are stored in a buffer called the *kill-ring*.

Key	Action
Ctrl-k	Kill text from the cursor location to the end of line.
Ctrl-u	Kill text from the cursor location to the beginning of the line.
Alt-d	Kill text from the cursor location to the end of the current word.
Alt-Backspace	Kill text from the cursor location to the beginning of the current word. If the cursor is at the beginning of a word, kill the previous word.
Ctrl-y	Yank text from the kill-ring and insert it at the cursor location.

The Meta Key

If you venture into the documentation for Readline, which can be found in the READLINE section of the bash man page, you will encounter the term “meta key”. On modern keyboards this maps to the Alt key but it wasn't always so.

Back in the dim times (before PCs but after Unix) not everybody had their own computer. What they might have had was a device called a *terminal*. A terminal was a communication device that featured a text display screen and a keyboard and just enough electronics inside to display text characters and move the cursor around. It was attached (usually by serial cable) to a larger computer or the communication network of a larger computer. There were many different brands of terminals and they all had different keyboards and display feature sets. Since they all tended to at least understand ASCII, software developers wanting to have portable applications wrote to the lowest common denominator. Unix systems have a very elaborate way to dealing with terminals and their different display features. Since the developers of Readline could not be sure of the presence of a dedicated extra control key, they invented one and called it “meta.” While the Alt key serves as the meta key on modern keyboards, you can also press and release the escape key to get the same effect as holding down the Alt key in case you're still using a terminal (which you can still do in Linux!)

For more information, see the Wikipedia article, “Computer terminal.”

Completion

Another way that the shell can help you is through a mechanism called *completion*. Completion occurs when you press the tab key while typing a command line. Let's see how this works. Given a home directory that looks like this:

```
[me@linuxbox ~]$ ls
Desktop  ls-output.txt  Pictures  Templates  Videos
Documents Music           Public
```

Try typing the following but **don't press the enter key**:

```
[me@linuxbox ~]$ ls l
```

Now press the tab key:

9 - Advanced Keyboard Tricks

```
[me@linuxbox ~]$ ls ls-output.txt
```

See how the shell completed the line for you? Let's try another one. Again, don't press enter:

```
[me@linuxbox ~]$ ls D
```

Press tab:

```
[me@linuxbox ~]$ ls D
```

No completion, just a beep. This happened because “D” matches more than one entry in the directory. For completion to be successful, the “clue” you give it has to be unambiguous. If we go further:

```
[me@linuxbox ~]$ ls Do
```

Then press tab:

```
[me@linuxbox ~]$ ls Documents
```

The completion is successful.

While this example shows completion of pathnames, which is its most common use, completion will also work on variables (if the beginning of the word is a “\$”), user names (if the word begins with “~”), host names (if the beginning of the word is “@”. Host name completion only works for host names listed in `/etc/hosts`), and commands (if the word is the first word on the line.)

There are a number of control and meta key sequences that are associated with completion:

Key	Action
Alt-?	Display list of possible completions. <i>On most systems you can also do this by pressing the tab key a second time which is much easier.</i>
Alt-*	Insert all possible completions. This is useful when you want to use

```
more than one possible match.
```

There quite a few more that I find rather obscure. You can a list in the `bash` man page under “`READLINE`”.

Programmable Completion

Recent versions of `bash` have a facility called *programmable completion*. Programmable completion allows you (or more likely, your distribution provider) to add additional completion rules. Usually this is done to add support for particular applications. For example it is possible to add completions for the option list of a command or match particular file types that an application supports. Ubuntu has a fairly large set defined by default. Programmable completion is implemented by shell functions, a kind of mini shell script that we will cover in later chapters. If you are curious, try:

```
set | less
```

and see if you can find them. Not all distributions include them by default.

Using History

As we discovered in Chapter 2, `bash` maintains a history of commands that have been entered. This list of commands is kept in your home directory in a file called `.bash_history`. The history facility is a useful resource for reducing the amount of typing you have to do, especially when combined with command line editing.

Searching History

At any time, we can view the contents of the history list by:

```
[me@linuxbox ~]$ history | less
```

By default, `bash` stores the last five hundred commands you have entered. We will see how to adjust this value in later chapters. Let's say we want to find the commands we used to list `/usr/bin`. One way we could do this:

```
[me@linuxbox ~]$ history | grep /usr/bin
```

And let's say that among our results we got a line containing an interesting command like this:

```
88  ls -l /usr/bin > ls-output.txt
```

The number “88” is the line number of the command in the history list. We could use this immediately using another type of expansion called *history expansion*. To use our discovered line we could do this:

```
[me@linuxbox ~]$ !88
```

`bash` will expand “!88” into the contents of the eighty-eighth line in the history list. There are other forms of history expansion that we will cover a little later.

`bash` also provides the ability to search the history list incrementally. This means that we can tell `bash` to search the history list as we enter characters, with each additional character further refining our search. To start incremental search type `Ctrl-r` followed by the text you are looking for. When you find it, you can either type `Enter` to execute the command or type `Ctrl-j` to copy the line from the history list to the current command line. To find the next occurrence of the text (moving “up” the history list), type `Ctrl-r` again. To quit searching, type either `Ctrl-g` or `Ctrl-c`. Here we see it in action:

```
[me@linuxbox ~]$
```

First type `Ctrl-r`:

```
(reverse-i-search)`':
```

The prompt changes to indicate that we are performing a reverse incremental search. It is “reverse” because we are searching from “now” to some time in the past. Next, we start typing our search text. In this example “/usr/bin”:

```
(reverse-i-search)`/usr/bin': ls -l /usr/bin > ls-output.txt
```

Immediately, the search returns our result. With our result, we can execute the command by pressing `enter`, or we can copy the command to our current command line for further editing by typing `Ctrl-j`. Let's copy it. Type `Ctrl-j`:

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Our shell prompt returns and our command line is loaded and ready for action!

The table below lists some of the keystrokes used to manipulate the history list:

Key	Action
Ctrl-p	Move to the previous history entry. Same action as the up arrow.
Ctrl-n	Move to the next history entry. Same action as the down arrow.
Alt-<	Move to the beginning (top) of the history list.
Alt->	Move to the end (bottom) of the history list, i.e. the current command line.
Ctrl-r	Reverse incremental search. Searches incrementally from the current command line up the history list.
Ctrl-s	Forward incremental search. Searches incrementally from the current position in the history list downward.
Alt-p	Reverse search, non-incremental. With this key you type in the search string and press enter before the search is performed.
Alt-n	Forward search, non-incremental.
Ctrl-o	Execute the current item in the history list and advance to the next one. This is handy if you are trying to re-execute a sequence of commands in the history list.

History Expansion

As we have seen, the shell offers a specialized type of expansion for items in the history list by using the “!” character. We have already seen how the exclamation point can be followed by a number to insert an entry from the history list. There are a number of other expansion features:

Sequence	Action
!!	Repeat the last command. It is probably easier to press up arrow and enter.
! <i>number</i>	Repeat history list item <i>number</i> .
! <i>string</i>	Repeat last history list item starting with <i>string</i> .
! <i>?string</i>	Repeat last history list item containing <i>string</i> .

I would caution against using the “!string” and “!?string” forms unless you are absolutely sure of the contents of the history list items.

There are many more elements available in the history expansion mechanism but I fear that this subject is already too arcane and that your head will explode if I continue. The HISTORY EXPANSION section of the `bash` man page goes into all the gory details. Feel free to explore!

Summing Up

In this chapter we have covered *some* of the keyboard tricks that the shell provides to help hardcore typists reduce their workloads. I suspect that as time goes by and you become more involved with the command line, you will refer back to this chapter to pick up more of these tricks. For now, consider them optional and potentially helpful.

10 – Permissions

Operating systems in the Unix tradition differ from those in the MS-DOS tradition in that they are not only a *multitasking* systems but also *multi-user* systems as well.

What exactly does this mean? It means that more than one person can be using the computer at the same time. While your computer will likely have only one keyboard and monitor, it can still be used by more than one user. For example, if your computer is attached to a network, or the Internet, remote users can log in via SSH (secure shell) and operate the computer. In fact, remote users can execute graphical applications and have the graphical output displayed on a remote computer. The X Window System supports this as part of its basic design.

The multi-user capability of Linux is not a recent "innovation," but rather a feature that is deeply ingrained into the design of the operating system. If you remember the environment in which Unix was created, this makes perfect sense. Years ago before computers were "personal," they were large, expensive, and centralized. A typical university computer system, for example, consisted of a large central computer located in some building on campus and terminals were located throughout the campus, each connected to the large central computer. The computer would support many users at the same time.

In order to make this practical, a method had to be devised to protect the users from each other. After all, you could not allow the actions of one user to crash the computer, nor could you allow one user to interfere with the files belonging to another user.

In this chapter we are going to look at this essential part of system security and we will introduce the following commands:

- `id` – Display user identity
- `chmod` – Change a file's mode
- `umask` – Set the default file permissions
- `su` – Run a shell as another user
- `sudo` – Execute a command as another user
- `chown` – Change a file's owner

- `chgrp` – Change a file's group ownership
- `passwd` – Change a user's password

Owners, Group Members, And Everybody Else

When we were exploring the system back in Chapter 4, did you encounter a situation like this? For example, when trying to examine the `/etc/shadow` file:

```
[me@linuxbox ~]$ file /etc/shadow
/etc/shadow: regular file, no read permission
[me@linuxbox ~]$ less /etc/shadow
/etc/shadow: Permission denied
```

The reason for this error message is that you, as a regular user, do not have permission to read this file.

In the Unix security model, a user may *own* files and directories. When you own a file or directory, you have control over its access. Users can, in turn, belong to a *group* consisting of one or more users who are given access to files and directories by their owners. In addition to granting access to a group, an owner may also grant some set of access rights to everybody which in Unix terms is referred to as the *world*. To find out information about our identity, we use the `id` command:

```
[me@linuxbox ~]$ id
uid=500(me) gid=500(me) groups=500(me)
```

Let's look at the output. When user accounts are created, users are assigned a number called a *user ID* or *uid* which is then, for the sake of the humans, mapped to a user name. The user is assigned a *primary group ID* or *gid* and may belong to additional groups. The above example is from a Fedora system. On other systems, such as Ubuntu, the output may look a little different:

```
[me@linuxbox ~]$ id
uid=1000(me) gid=1000(me)
groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(v
ideo),46(plugdev),108(lpadmin),114(admin),1000(me)
```

As we can see the uid and gid numbers are different. This is simply because Fedora starts its numbering of regular user accounts at 500 while Ubuntu starts at 1000. We can also

see that the Ubuntu user belongs to a lot more groups. This has to do with the way Ubuntu manages privileges for system devices and services.

So where does this information come from? Like so many things in Linux, from a couple of text files. User accounts are defined in the `/etc/passwd` file and groups are defined in the `/etc/group` file. When user accounts and groups are created, these files are modified along with `/etc/shadow` which holds information about the user's password. For each user account, the `/etc/passwd` file defines the user (login) name, uid, gid, the account's real name, home directory, and login shell. If you examine the contents of `/etc/passwd` and `/etc/group` you will notice that besides the regular user accounts, there are accounts for the superuser (uid 0) and various other system users.

In the next chapter, when we cover processes, we will see that some of these other “users” are, in fact, quite busy.

While many Unix-like systems assign regular users to a common group such as “users”, modern Linux practice is to create a unique, single-member group with the same name as the user. This makes certain types of permission assignment easier.

Reading, Writing, And Executing

Access rights to files and directories are defined in terms of read access, write access, and execution access. If we look at the output of the `ls` command we can get some clue as to how this is implemented:

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me    0 2008-03-06 14:52 foo.txt
```

The first ten characters of the listing are the *file attributes*. The first of these characters is the *file type*. Here are the file types you are most likely to see:

Attribute	File Type
-	A regular file.
d	A directory.
l	A symbolic link. Notice that with symbolic links, the remaining file attributes are always “rwxrwxrwx” and are dummy values. The real file attributes are those of the file the symbolic link points to.
c	A <i>character special file</i> . This file type refers to a device that handles data as a stream of bytes such as a terminal or modem.
b	A <i>block special file</i> . This file type refers to a device that handles

	data in blocks such as a hard drive or CD-ROM drive
--	---

The remaining nine characters of the file attributes, called the *file mode*, represent the read, write, and execute permissions for the file's owner, the file's group owner, and everybody else:

Owner	Group	World
rwx	rwx	rwx

When set, the *r*, *w*, and *x* mode attributes have the following effect on files and directories:

Attribute	Files	Directories
r	Allows a file to be opened and read.	Allows a directory's contents to be listed if the execute attribute is also set.
w	Allows a file to be written to or truncated. This attribute does not allow files to be renamed nor deleted.	Allows files within a directory to be created, deleted, and renamed if the execute attribute is also set.
x	Allows a file to be treated as a program and executed. Program files written in scripting languages must also be set readable to be executed.	Allows a directory to be entered, i.e. <i>cd directory</i> .

Here are some examples of file attribute settings:

File Attributes	Meaning
-rwx-----	A regular file that is readable, writable, and executable by the file's owner. No one else has any access.
-rw-----	A regular file that is readable and writable by the file's owner. No one else has any access.
-rw-r--r--	A regular file that is readable and writable by the file's owner. Members of the owner group may read the file. The file is world-readable.
-rwxr-xr-x	A regular file that is readable, writable, and executable by the

	file's owner. The file may be read and executed by everybody else.
<code>-rw-rw----</code>	A regular file that is readable and writable by the file's owner and members of the file's group owner only.
<code>lrwxrwxrwx</code>	A symbolic link. All symbolic links have “dummy” permissions. The real permissions are kept with the actual file pointed to by the symbolic link.
<code>drwxrwx---</code>	A directory. The owner and the members of the owner group may enter the directory, create, rename and remove files within the directory.
<code>drwxr-x---</code>	A directory. The owner may enter the directory and create, rename and delete files within the directory. Members of the owner group may enter directory and but cannot create, delete or rename files.

chmod – Change file mode

To change the mode (permissions) of a file or directory, the `chmod` command is used. Be aware that in order to change the mode of a file or directory, you must be either the file or directory's owner or the superuser. `chmod` supports two distinct ways of specifying what changes to make, octal number representation, or symbolic representation. We will cover octal number representation first.

What The Heck Is Octal?

Octal (base 8), and its cousin, *hexadecimal* (base 16) are number systems often used to express numbers on computers. We humans, owing to the fact that we (or at least most of us) were born with ten fingers, count using a base 10 number system. Computers, on the other the other hand, were born with only one finger and thus do all all their counting in *binary* (base 2). Their number system only has two numerals, zero and one. So in binary, counting looks like this:

0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011...

In octal, counting is done with the numerals zero through seven, like so:

0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21...

Hexadecimal counting uses the numerals zero through nine plus the letters “A” through “F”:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13...

While we can see the sense in binary (since computers only have one finger), what are octal and hexadecimal good for? The answer has to do with human convenience. Many times, small portions of data are represented on computers as *bit patterns*. Take for example an RGB color. On most computer displays, each pixel is composed of three color components: eight bits of red, eight bits of green, and eight bits of blue. A lovely medium blue would be a twenty-four digit number:

```
010000110110111111001101
```

How would you like to read and write those kinds of numbers all day? I didn't think so. Here's where another number system would help. Each digit in a hexadecimal number represents four digits in binary. In octal, each digit represents three binary digits. So our twenty-four digit medium blue could be condensed down to a six digit hexadecimal number:

```
436FCD
```

Since the digits in the hexadecimal number “line up” with the bits in the binary number we can see that the red component of our color is “43”, the green “6F”, and the blue “CD”.

These days, hexadecimal notation (often spoken as “hex”) is more common than octal but as we shall soon see, octal's ability to express three bits of binary will be very useful...

With octal notation we use octal numbers to set the pattern of desired permissions. Since each digit in an octal number represents three binary digits, this maps nicely to the scheme used to store the file mode. This table shows what we mean:

Octal	Binary	File Mode
0	000	- - -
1	001	- - X
2	010	- W -
3	011	- W X
4	100	r - -
5	101	r - X
6	110	r W -
7	111	r W X

By using three octal digits, we can set the file mode for the owner, group owner, and world:

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me    0 2008-03-06 14:52 foo.txt
[me@linuxbox ~]$ chmod 600 foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw----- 1 me    me    0 2008-03-06 14:52 foo.txt
```

By passing the argument “600”, we were able to set the permissions of the owner to read and write while removing all permissions from the group owner and world. Though remembering the octal to binary mapping may seem inconvenient, you will usually only have to use a few common ones: 7 (rwx), 6 (rw-), 5 (r-x), 4 (r--), and 0 (---).

chmod also supports a symbolic notation for specifying file modes. Symbolic notation is divided into three parts, who the change is to affect, the operation to be performed, and what permission is set. To specify who is affected, a combination of the characters “u”, “g”, “o”, and “a” is used as follows:

Symbol	Meaning
u	Short for “user” but means the file or directory owner.
g	Group owner.
o	Short for “others” but means world.
a	Short for “all.” The combination of “u”, “g”, and “o”.

The operation may be a “+” indicating that a permission is to be added, a “-” indicating that a permission is to be taken away, or a “=” indicating that only the specified permissions are to be applied and that all others are to be removed.

Permissions are specified with the “r”, “w”, and “x” characters. Here are some examples of symbolic notation:

Notation	Meaning
u+x	Add execute permission for the owner.
u-x	Remove execute permission for the owner.
o-rw	Remove the read and write permission for anyone besides the owner and group owner.
go=rw	Set the group owner and anyone besides the owner to have read and

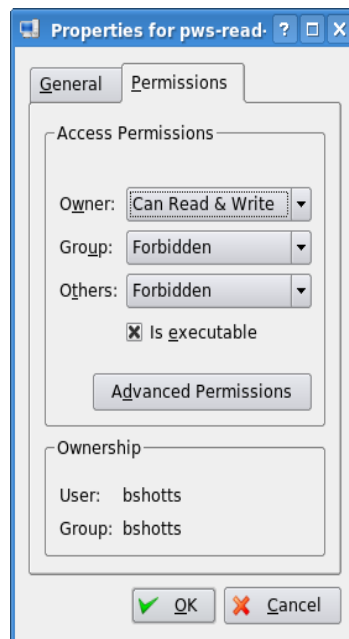
	write permission. If either the group owner or world had execute permissions, they are removed.
<code>u+x, go=rX</code>	Add execute permission for the owner and set the permissions for the group and others read and execute. Multiple specifications may be separated by commas.

While I prefer to use octal notation, some folks really like the symbolic. Symbolic notation does offer the advantage of allowing you to set a single attribute without disturbing any of the others.

Take a look at the `chmod` man page for more details and a list of options. A word of caution regarding the “`--recursive`” option: it acts on both files and directories, so it's not as useful as one would hope since you would rarely want files and directories to have the same permissions.

Setting File Mode With The GUI

Now that we have seen how the permissions on files and directories are set, we can better understand the permission dialogs in the GUI. In both Nautilus (GNOME) and Konqueror (KDE), right-clicking a file or directory icon will expose a properties dialog. Here is an example from KDE 3.5:



Here we can see the settings for the owner, group, and world. In KDE, clicking on the “Advanced Permissions” button brings up another dialog that allows you to set each of the mode attributes individually. Another victory for understanding brought to us by the command line!

umask – Set Default Permissions

The `umask` command controls the default permissions given to a file when it is created. It uses octal notation to express a *mask* of bits to be removed from a file's mode attributes. Let's take a look:

```
[me@linuxbox ~]$ rm -f foo.txt
[me@linuxbox ~]$ umask
0002
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me    0 2008-03-06 14:53 foo.txt
```

We first removed any old copy of `foo.txt` to make sure we were starting fresh. Next, we ran the `umask` command without an argument to see the current value. It responded with the value `0002` (the value `0022` is another common value) which is the octal representation of our mask. We next create a new instance of the file `foo.txt` and observe its permissions.

We can see that the owner and group both get read and write permission while everyone else only gets read permission. The reason that world does not have write permission is because of the value of the mask. Let's repeat our example, this time setting the mask ourselves:

```
[me@linuxbox ~]$ rm foo.txt
[me@linuxbox ~]$ umask 0000
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-rw- 1 me    me    0 2008-03-06 14:58 foo.txt
```

When we set the mask to `0000` (effectively turning it off), we see that the file is now world writable. To understand how this works, we have to look at octal numbers again. If we take the mask and expand it into binary and compare it to the attributes we can see what happens:

Original file mode	---rw-rw-rw-
Mask	000000000010
Result	---rw-rw-r--

Ignore for the moment the leading zeros (we'll get to those in a minute) and observe that where the one appears in our mask, an attribute was removed, in this case, the removal of the world write permission. That's what the mask does. Everywhere a one appears in the binary value of the mask, an attribute is unset. If we look at a mask value of 0022, we can see what it does:

Original file mode	---rw-rw-rw-
Mask	000000010010
Result	---rw-r--r--

Again, where a one appears in the binary value, the corresponding attribute is unset. Play with some values (try some sevens) to get used to how this works. When you're done, remember to clean up:

```
[me@linuxbox ~]$ rm foo.txt; umask 0002
```

Most of the time you won't have to change the mask, the default provided by your distribution will be fine. In some high-security situations, however, you will want to control it.

Some Special Permissions

Though we will usually see an octal permission mask expressed as a three digit number, it is more technically correct to express it in four digits. Why? Because, in addition to read, write, and execute permission, there are some other, less used, permission settings.

The first of these is the *setuid bit* (octal 4000). When applied to an executable file, it sets the *effective user ID* from that of real user (the user actually running the program) to that of the program's owner. Most often this is given to a few programs owned by the superuser. When an ordinary user runs a program that is "*setuid root*", the program runs with the effective privileges of the superuser. This

allows the program to access files and directories that the ordinary user would normally be prohibited from accessing. Clearly, this raises security concerns and the number of setuid programs must be held to an absolute minimum.

The second is the *setgid bit* (octal 2000) which, like the setuid bit, changes the *effective group ID* from the *real group ID* of the user to that of the file owner. If the setgid bit is set on a directory, newly created files in the directory will be given the group ownership of the directory rather than the group ownership of the file's creator. This is useful in a shared directory when it is desirable for members of a common group who need access to all the files in the directory regardless of the file owner's primary group.

The third is called the *sticky bit* (octal 1000). This is a holdover from ancient Unix where it was possible to mark an executable file as "not swappable." On files, Linux ignores the sticky bit, but if applied to a directory, it prevents users from deleting or renaming files unless the user is either the owner of the directory, the owner of the file, or the superuser. This is often used to control access to a shared directory, such as, /tmp.

Here are some examples of using chmod with symbolic notation to set these special permissions. First assigning setuid to a program:

```
chmod u+s program
```

Next, assigning setgid to a directory:

```
chmod g+s dir
```

Finally, assigning the sticky bit to a directory:

```
chmod +t dir
```

When viewing the output from ls, you can determine the special permissions. Here are some examples. First, a program that is setuid:

```
-rwsr-xr-x
```

A directory that has the setgid attribute:

```
drwxrwsr-x
```

A directory with the sticky bit set:

```
drwxrwxrwt
```

Changing Who You Are

At various times, we may find it necessary to take on the identity of another user. Often

we want to gain superuser privileges to carry out some administrative task, but it is also possible to “become” another regular user for such things as testing an account. There are three ways to take on an alternate identity:

1. You could log out and log back in as the desired user.
2. You could use the `SU` command.
3. You could use the `SUDO` command.

We will skip the first technique since we know how to do that and it lacks the convenience of the other two. Using either `SU` or `SUDO` will permit you to temporarily assume the identity of another user from within your own shell session. The `SU` command allows you to assume the identity of another user and either start a new shell session with that user's user and group IDs, or issue a single command as that user. The `SUDO` command allows an administrator to set up a configuration file called `/etc/sudoers` and define specific commands that particular users are permitted to execute under an assumed identity. The choice of which command to use is largely determined by which Linux distribution you use. Your distribution probably includes both commands but your distribution's configuration will favor either one or the other. We'll start with `SU`.

su – Run A Shell With Substitute User And Group IDs

The `SU` command is used to start a shell as another user. The command syntax looks like this:

```
su [-[1]] [user]
```

If the “-l” option is included, then the resulting shell session is a *login shell* for the specified user. This means that the user's environment is loaded and the working directory is changed to the user's home directory. This is usually what you want. If the user is not specified, the superuser is assumed. Notice that (strangely) the “-l” may be abbreviated “-”, which is how it is most often used. To start a shell for the superuser, we would do this:

```
[me@linuxbox ~]$ su -  
Password:  
[root@linuxbox ~]#
```

After entering the command, we are prompted for the superuser's password. If it is

successfully entered, a new shell prompt appears indicating that this shell has superuser privileges (the trailing “#” rather than a “\$”) and the current working directory is now the home directory for the superuser (normally /root.) Once in the new shell, we can carry out commands as the superuser. When finished, enter the exit command to return to the previous shell:

```
[root@linuxbox ~]# exit
[me@linuxbox ~]$
```

It is also possible to execute a single command rather than starting a new interactive command by using `SU` this way:

```
su -c 'command'
```

Using this form, a single command line is passed to the new shell for execution. It is important to enclose the command in quotes, as we do not want expansion to occur in our shell, but rather in the new shell:

```
[me@linuxbox ~]$ su -c 'ls -l /root/*'
Password:
-rw----- 1 root root      754 2007-08-11 03:19 /root/anaconda-ks.cfg

/root/Mail:
total 0
[me@linuxbox ~]$
```

sudo – Execute A Command As Another User

The `sudo` command is in many ways like `su`, but has some important additional capabilities. The administrator can configure `sudo` to allow ordinary users to execute commands as a different user (usually the superuser) in a very controlled way. In particular, a user may be restricted to one or more specific commands and no others. Another important difference is that the use of `sudo` does not require access to the superuser's password. To authenticate using `sudo`, the user uses his/her own password. Let's say, for example, that `sudo` has been configured to allow us to run a fictitious backup program called “`backup_script`” which requires superuser privileges. With `sudo` it would be done like this:

```
[me@linuxbox ~]$ sudo backup_script
Password:
System Backup Starting...
```

After entering the command, we are prompted for our password (not the superuser's) and once the authentication is complete, the specified command is carried out. One important difference between `SU` and `SUDO` is that `SUDO` does not start a new shell nor does it load another user's environment. This means that commands do not need to be quoted any differently than they would be without using `SUDO`. Note that this behavior can be overridden by specifying various options. See the `SUDO` man page for details.

To see what privileges are granted by `SUDO`, use the “-l” option to list them:

```
[me@linuxbox ~]$ sudo -l
User me may run the following commands on this host:
    (ALL) ALL
```

Ubuntu And sudo

One of the recurrent problems for regular users is how to perform certain tasks that require superuser privileges. These tasks include installing and updating software, editing system configuration files, and accessing devices. In the Windows world, this is often done by giving users administrative privileges. This allows users to perform these tasks. However it also enables programs executed by the user to have the same abilities. This is desirable in most cases but it also permits malware such as viruses to have free reign of the computer.

In the Unix world there has always been a larger division between regular users and administrators owing to the multi-user heritage of Unix. The approach taken in Unix is to grant superuser privileges only when needed. To do this, the `SU` and `SUDO` commands are commonly used.

Up until a couple of years ago, most Linux distributions relied on `SU` for this purpose. `SU` didn't require the configuration that `SUDO` required and having a root account is traditional in Unix. This introduces a problem. Users will be tempted to operate as root unnecessarily. In fact, some users will operate their systems as the root user exclusively since it does away with all those annoying “permission denied” messages. This is how you reduce the security of a Linux system to that of a Windows system. Not a good idea.

When Ubuntu was introduced, its creators took a different tact. By default, Ubuntu disables logins to the root account (by failing to set a password for the account) and instead uses `SUDO` to grant superuser privileges. The initial user account is granted full access to superuser privileges via `SUDO` and may grant similar powers to subsequent user account accounts.

chown – Change File Owner And Group

The `chown` command is used to change the owner and group owner of a file or directory. Superuser privileges are required to use this command. The syntax of `chown` looks like this:

```
chown [owner][:group] file...
```

`chown` can change the file owner and/or the file group owner depending on the first argument of the command. Here are some examples:

Argument	Results
<code>bob</code>	Changes the ownership of the file from its current owner to user <code>bob</code> .
<code>bob:users</code>	Changes the ownership of the file from its current owner to user <code>bob</code> and changes the file group owner to group <code>users</code> .
<code>:admins</code>	Changes the group owner to the group <code>admins</code> . The file owner is unchanged.
<code>bob:</code>	Change the file owner from the current owner to user <code>bob</code> and changes the group owner to the login group of user <code>bob</code> .

Let's say that we have two users. The first, `bob`, who has access to superuser privileges and `tony` who does not. User `bob` wants to copy a file from his home directory to the home directory of user `tony`. Since user `bob` wants `tony` to be able to edit the file, `bob` changes the ownership of the copied file from `bob` to `tony`:

```
[bob@linuxbox ~]$ sudo cp myfile.txt ~tony
Password:
[bob@linuxbox ~]$ sudo ls ~tony/myfile.txt
```

```
-rw-r--r-- 1 root root 8031 2008-03-20 14:30 /home/tony/myfile.txt
[ bob@linuxbox ~ ]$ sudo chown tony: ~tony/myfile.txt
[ bob@linuxbox ~ ]$ sudo ls -l ~tony/myfile.txt
-rw-r--r-- 1 tony tony 8031 2008-03-20 14:30 /home/tony/myfile.txt
```

Here we see `bob`, copy the file from his directory to the home directory of `tony`. Next, `bob` changes the ownership of the file from `root` (a result of using `sudo`) to `tony`. Using the trailing colon in the first argument, `bob` also changed the group ownership of the file to the login group of `tony` which happens to be group `tony`.

Notice how that after the first use of `sudo`, `bob` was not prompted for his password? This is because `sudo`, in most configurations, “trusts” you for several minutes until its timer runs out.

chgrp – Change Group Ownership

In older versions of Unix, the `chown` command only changed file ownership, not group ownership. For that purpose, a separate command, `chgrp` was used. It works much the same as `chown` except for being more limited.

Exercising Our Privileges

Now that we have learned how this permissions thing works, it's time to show it off. We going to demonstrate the solution to a common problem – setting up a shared directory. Let's imagine that we have two users named “`bill`” and “`karen`.” They both have music CD collections and wish to set up a shared directory where they will each store their music files as Ogg Vorbis or MP3. User `bill` has access to superuser privileges via `sudo`.

The first thing that needs to happen is to create a group that will have both `bill` and `karen` as members. Using the graphical user management tool (we'll look at doing this with command line programs in a later chapter), `bill` creates a group called `music` and adds users `bill` and `karen` to it:



Next, `bill` creates the directory for the music files:

```
[bill@linuxbox ~]$ sudo mkdir /usr/local/share/Music
Password:
```

Since `bill` is manipulating files outside his home directory, superuser privileges are required. After the directory is created, it has the following ownerships and permissions:

```
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxr-xr-x 2 root root 4096 2008-03-21 18:05 /usr/local/share/Music
```

As we can see, the directory is owned by `root` and has 755 permissions. To make this directory sharable, `bill` needs to change the group ownership and the group permissions to allow writing:

```
[bill@linuxbox ~]$ sudo chown :music /usr/local/share/Music
[bill@linuxbox ~]$ sudo chmod 775 /usr/local/share/Music
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxrwxr-x 2 root music 4096 2008-03-21 18:05 /usr/local/share/Music
```

O.K. So what does this all mean? It means that we now have a directory, `/usr/local/share/Music` that is owned by `root` and allows read and write

access to group `music`. Group `music` has members `bill` and `karen`, thus `bill` and `karen` can create files in directory `/usr/local/share/Music`. Other users can list the contents of the directory but cannot create files there.

But we still have a problem. With the current permissions, files and directories created within the `Music` directory will have the normal permissions of the users `bill` and `karen`:

```
[bill@linuxbox ~]$ > /usr/local/share/Music/test_file
[bill@linuxbox ~]$ ls -l /usr/local/share/Music
-rw-r--r-- 1 bill  bill  0 2008-03-24 20:03 test_file
```

Actually there are two problems. First, the default `umask` on this system is `0022` which prevents group members from writing files belonging to other members of the group. This would not ordinarily be a problem if the shared directory only contained files, but since this directory will store music and music is usually organized in a hierarchy of artists and albums, members of the group will need the ability to create files and directories inside directories created by other members. We need to change the `umask` used by `bill` and `karen` to `0002` instead.

Second, each file and directory created by one member will be set to the primary group of the user rather than the group `music`. This can be fixed by setting the `setgid` bit on the directory:

```
[bill@linuxbox ~]$ sudo chmod g+s /usr/local/share/Music
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxrwsr-x 2 root music 4096 2008-03-24 20:03 /usr/local/share/Music
```

Now we test to see if the new permissions fix the problem. `bill` sets his `umask` to `0002`, removes the previous test file, creates a new test file and directory:

```
[bill@linuxbox ~]$ umask 0002
[bill@linuxbox ~]$ rm /usr/local/share/Music/test_file
[bill@linuxbox ~]$ > /usr/local/share/Music/test_file
[bill@linuxbox ~]$ mkdir /usr/local/share/Music/test_dir
[bill@linuxbox ~]$ ls -l /usr/local/share/Music
drwxrwsr-x 2 bill  music 4096 2008-03-24 20:24 test_dir
-rw-rw-r-- 1 bill  music 0 2008-03-24 20:22 test_file
[bill@linuxbox ~]$
```


Both files and directories are now created with the correct permissions to allow all members of the group `MUSIC` to create files and directories inside the `MUSIC` directory.

The one remaining issue is `umask`. The necessary setting only lasts until the end of session and must be reset. In the next part of the book, we'll look at making the change to `umask` permanent.

Changing Your Password

The last topic we'll cover in this chapter is setting passwords for yourself (and for other users if you have access to superuser privileges.) To set or change a password, the `passwd` command is used. The command syntax looks like this:

```
passwd [user]
```

To change your password just enter the `passwd` command. You will be prompted for your old password and your new password:

```
[me@linuxbox ~]$ passwd  
(current) UNIX password:  
New UNIX password:
```

The password command will try to enforce use of “strong” passwords. This means the it will refuse to accept passwords that are too short, too similar to previous passwords, are dictionary words, or too easily guessed:

```
[me@linuxbox ~]$ passwd  
(current) UNIX password:  
New UNIX password:  
BAD PASSWORD: is too similar to the old one  
New UNIX password:  
BAD PASSWORD: it is WAY too short  
New UNIX password:  
BAD PASSWORD: it is based on a dictionary word
```

If you have superuser privileges, you can specify a user name as an argument to the `passwd` command to set the password for another user. There are other options available to the superuser to allow account locking, password expiration, etc. See the `passwd` man page for details.

11 - Processes

Modern operating systems are usually *multitasking*, meaning that they create the illusion of doing more than one thing at once by rapidly switching from one executing program to another. The Linux kernel manages this through the use of *processes*. Processes are how Linux organizes the different programs waiting for their turn at the CPU.

You have probably noticed that sometimes your computer will be sluggish or that an application stops responding. In this chapter we will look at some of the tools available at the command line that let us examine what programs are doing and how to terminate processes that are misbehaving.

This chapter will introduce the following commands:

- `ps` – Report A Snapshot Of Current Processes
- `top` – Display Tasks
- `jobs` – List Active Jobs
- `bg` – Place A Job In The Background
- `fg` – Place A Job In The Foreground
- `kill` – Send A Signal To A Process
- `killall` – Kill Processes By Name
- `shutdown` – Shutdown Or Reboot The System

How A Process Works

When your system starts up, the kernel initiates a few of its own activities as processes and it launches a program called `init`. `init`, in turn, runs a series of shell scripts (located in `/etc`) called *init scripts* which start all the system services. Many of these services are implemented as *daemon programs*, programs that just sit in the background and do their thing without having any user interface. So even if we are not logged in, the system is at least a little busy performing routine stuff.

The fact that a program can launch other programs is expressed in the process scheme as

a *parent process* producing a *child process*.

The kernel maintains information about each process to help keep things organized. For example, each process is assigned a number called a *process ID* or *PID*. PIDs are assigned in ascending order with `init` always getting PID 1. The kernel also keeps track of the memory assigned to each process, as well as, the processes' readiness to resume execution. Like files, process also have owners and user IDs, effective user IDs, etc.

Viewing Processes

The most commonly used command to view processes (there are several) is `ps`. The `ps` program has a lot of options but in its simplest form, it is used like this:

```
[me@linuxbox ~]$ ps
  PID TTY          TIME CMD
 5198 pts/1        00:00:00 bash
10129 pts/1        00:00:00 ps
```

The result in this example lists two processes, process 5198 and process 10129 which are `bash` and `ps` respectively. As we can see, by default, `ps` doesn't show us very much, just the processes associated with the current terminal session. To see more, we need to add some options but before we do that, let's look at the other fields produced by `ps`. TTY is short for "Teletype" and refers to the *controlling terminal* for the process. Unix is showing its age here. The `TIME` field is the amount of CPU time consumed by the process. As we can see, neither process makes the computer work very hard.

If we add an option, we can get a bigger picture of what the system is doing:

```
[me@linuxbox ~]$ ps x
  PID TTY          STAT TIME COMMAND
 2799 ?           Ssl   0:00 /usr/libexec/bonobo-activation-server -ac
 2820 ?           Sl    0:01 /usr/libexec/evolution-data-server-1.10 --
15647 ?           Ss    0:00 /bin/sh /usr/bin/startkde
15751 ?           Ss    0:00 /usr/bin/ssh-agent /usr/bin/dbus-launch --
15754 ?           S     0:00 /usr/bin/dbus-launch --exit-with-session
15755 ?           Ss    0:01 /bin/dbus-daemon --fork --print-pid 4 -pr
15774 ?           Ss    0:02 /usr/bin/gpg-agent -s -daemon
15793 ?           S     0:00 start_kdeinit --new-startup +kcmnit_start
15794 ?           Ss    0:00 kdeinit Running...
15797 ?           S     0:00 dcopserver -nosid
```

and many more...

Adding the “x” option (note that there is no leading dash) tells `ps` to show all of our process regardless of what terminal (if any) they are controlled by. The presence of a “?” in the TTY column indicates no controlling terminal. Using this option, we see a list of every process that we own.

Since the system is running a lot of processes, `ps` produces a long list. It is often helpful to pipe the output from `ps` into `less` for easier viewing. Some option combinations also produce long lines of output, so maximizing the terminal emulator window may be a good idea too.

A new column titled STAT has been added to the output. STAT is short for “state” and reveals the current status of the process:

State	Meaning
R	Running. This means that the process is running or ready to run.
S	Sleeping. A process is not running, rather it is waiting for an event, such as a keystroke or network packet.
D	Uninterruptible Sleep. Process is waiting for I/O such as a disk drive.
T	Stopped. Process has been instructed to stop. More on this later.
Z	A defunct or “zombie” process. This is a child process that has terminated but has not been cleaned up by its parent.
<	A high priority process. It's possible to grant more importance to a process and it will, as a result, receive more time on the CPU. This property of a process is called <i>niceness</i> . A process with high priority is said to be less <i>nice</i> because it's taking more of the CPU's time which leaves less for everybody else.
N	A low priority process. A process with low priority (a “nice” process) will only get processor time after other processes with higher priority have been serviced.

Another popular set of options is “aux” without a leading dash. This gives us even more information:

```
[me@linuxbox ~]$ ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0  2136   644 ?        Ss   Mar05   0:31  init
root         2  0.0  0.0     0     0 ?        S<   Mar05   0:00  [kt]
root         3  0.0  0.0     0     0 ?        S<   Mar05   0:00  [mi]
root         4  0.0  0.0     0     0 ?        S<   Mar05   0:00  [ks]
root         5  0.0  0.0     0     0 ?        S<   Mar05   0:06  [wa]
root         6  0.0  0.0     0     0 ?        S<   Mar05   0:36  [ev]
root         7  0.0  0.0     0     0 ?        S<   Mar05   0:00  [kh]

and many more...
```

Using the options without the leading dash invokes the command with “BSD style” behavior. The Linux version of `ps` can emulate the behavior of the `ps` program found in several different Unix implementations. With these options, we get these additional columns:

Header	Meaning
USER	User ID. This is the owner of the process.
%CPU	CPU usage in percent.
%MEM	Memory usage in percent.
VSZ	Virtual memory size.
RSS	Resident Set Size. The amount of physical memory (RAM) the process is using in kilobytes.
START	Time when the process started. For values over twenty four hours, a date is used.

Viewing Processes Dynamically With `top`

While the `ps` command can reveal a lot about what the machine is doing, it provides only a snapshot of the machine's state at the moment the `ps` command is executed. To see a more dynamic view of the machine's activity we use the `top` command:

```
[me@linuxbox ~]$ top
```

The `top` program displays a continuously updating (by default, every 3 seconds) display of the system processes listed in order of process activity. The name “top” comes from

the fact that the `top` program is used to see the “top” processes on the system. The top display consists of two parts. A system summary at the top of the display, followed by a table of processes sorted by CPU activity:

```

top - 14:59:20 up 6:30, 2 users, load average: 0.07, 0.02, 0.00
Tasks: 109 total, 1 running, 106 sleeping, 0 stopped, 2 zombie
Cpu(s): 0.7%us, 1.0%sy, 0.0%ni, 98.3%id, 0.0%wa, 0.0%hi, 0.0%si
Mem: 319496k total, 314860k used, 4636k free, 19392k buff
Swap: 875500k total, 149128k used, 726372k free, 114676k cach

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
 6244 bshotts   39   19 31752 3124 2188 S   6.3   1.0   16:24.42 trackerd
11071 bshotts   20    0  2304 1092  840 R   1.3   0.3    0:00.14 top
 6180 bshotts   20    0  2700 1100  772 S   0.7   0.3    0:03.66 dbus-dae
 6321 bshotts   20    0 20944 7248 6560 S   0.7   2.3   2:51.38 multiloa
 4955 root      20    0  104m 9668 5776 S   0.3   3.0   2:19.39 Xorg
    1 root      20    0  2976  528  476 S   0.0   0.2    0:03.14 init
    2 root      15   -5     0     0     0 S   0.0   0.0    0:00.00 kthreadd
    3 root      RT   -5     0     0     0 S   0.0   0.0    0:00.00 migratio
    4 root      15   -5     0     0     0 S   0.0   0.0    0:00.72 ksoftirq
    5 root      RT   -5     0     0     0 S   0.0   0.0    0:00.04 watchdog
    6 root      15   -5     0     0     0 S   0.0   0.0    0:00.42 events/0
    7 root      15   -5     0     0     0 S   0.0   0.0    0:00.06 khelper
   41 root      15   -5     0     0     0 S   0.0   0.0    0:01.08 kblockd/
   67 root      15   -5     0     0     0 S   0.0   0.0    0:00.00 kseriod
  114 root      20    0     0     0     0 S   0.0   0.0    0:01.62 pdflush
  116 root      15   -5     0     0     0 S   0.0   0.0    0:02.44 kswapd0
    
```

The system summary contains a lot of good stuff. Here's a rundown:

Row	Field	Meaning
1	top	Name of the program.
	14:59:20	Current time of day.
	up 6:30	This is called <i>uptime</i> . It is the amount of time since the machine was last booted. In this example, the system has been up for six and a half hours.
	2 users	There are two users logged in.
	load average:	<i>Load average</i> refers to the number of processes that are waiting to run, that is, the number of processes that are in a runnable state and are sharing the CPU. Three values are shown, each for a different period of time. The first is

		the average for the last 60 seconds, the next the previous 5 minutes, and finally the previous 15 minutes. Values under 1.0 indicates that the machine is not busy.
2	Tasks:	This summarizes the number of processes and their various process states.
3	Cpu(s):	This row describes the character of the activities that the CPU is performing.
	0.7%us	0.7% of the CPU is being used for <i>user processes</i> . This means processes outside of the kernel itself.
	1.0%sy	1.0% of the CPU is being used for <i>system (kernel) processes</i> .
	0.0%ni	0.0% of the CPU is being used by “nice” (low priority) processes.
	98.3%id	98.3% of the CPU is idle.
	0.0%wa	0.0% of the CPU is waiting for I/O.
4	Mem:	Shows how physical RAM is being used.
5	Swap:	Shows how swap space is being used.

The top program accepts a number of keyboard commands, the two most interesting are `h` which displays the program's help screen, and `q` which quits `top`.

Both major desktop environments provide graphical applications that display information similar to `top` (in much the same way that Task Manager in Windows works), but I find that `top` is better than the graphical versions because it is faster and it consumes far fewer system resources. After all, your system monitor program shouldn't be the source of the system slowdown that you are trying to track.

Controlling Processes

Now that we can see and monitor processes, let's gain some control over them. For our experiments, we're going to use a little program called `xlogo` as our guinea pig. The `xlogo` program is a sample program supplied with the X Window System (the underlying engine that makes the graphics on your display go) that simply displays a resizable window containing the X logo. First, we'll get to know our test subject:

```
[me@linuxbox ~]$ xlogo
```

After entering the command, a small window containing the logo should appear somewhere on the screen. On some systems, `xlogo` may print a warning message but it may be safely ignored.

We can verify that `xlogo` is running by resizing its window. If the logo is redrawn in the new size, then the program is running.

Notice how our shell prompt has not returned? This is because the shell is waiting for the program to finish, just like all the other programs we have used so far. If we close the `xlogo` window, the prompt returns.

Interrupting A Process

Let's observe what happens when we run `xlogo` again. First, enter the `xlogo` command and verify that the program is running. Next, return to the terminal window and type `Ctrl-c`.

```
[me@linuxbox ~]$ xlogo
[me@linuxbox ~]$
```

In a terminal, typing `Ctrl-c`, *interrupts* a program. This means that we politely asked the program to terminate. After typing `Ctrl-c`, the `xlogo` window closed and the shell prompt returned.

Many (but not all) command line programs can be interrupted by using this technique.

Putting A Process In The Background

But let's say we wanted to get the shell prompt back without terminating the `xlogo` program. This is done by placing the program in the *background*. You can think of the terminal as having a *foreground* (with stuff visible on the surface like the shell prompt) and a background (with hidden stuff behind the surface.) To launch a program so that it is immediately placed in the background, we follow the command with a “&” character:

```
[me@linuxbox ~]$ xlogo &
[1] 28236
[me@linuxbox ~]$
```


After entering the command, the `xlogo` window appeared and the shell prompt returned but some funny numbers were printed too. This message is part of a shell feature called *job control*. With this message, the shell is telling us that we have started job number 1 (“[1]”) and that it has PID 28236. If we run `ps`, we can see our process:

```
[me@linuxbox ~]$ ps
  PID TTY          TIME CMD
 10603 pts/1        00:00:00 bash
 28236 pts/1        00:00:00 xlogo
 28239 pts/1        00:00:00 ps
```

The shell's job control facility also gives us a way to list the jobs that are have been launched from our terminal. Using the `jobs` command, we can see this list:

```
[me@linuxbox ~]$ jobs
[1]+  Running                  xlogo &
```

The results show that we have one job, numbered “1”, that it is running, and that the command was `xlogo &`.

Returning A Process To The Foreground

A process in the background is immune from keyboard input including any attempt interrupt it with a `Ctrl-c`. It is possible to return a process to the foreground. Using the `fg` command, it is done this way:

```
[me@linuxbox ~]$ jobs
[1]+  Running                  xlogo &
[me@linuxbox ~]$ fg %1
xlogo
```

The command `fg` followed by a percent sign and the job number (called a *jobspec*) does the trick. If you have only one background job, the *jobspec* is optional.

Stopping (Pausing) A Process

Sometimes we'll want to stop a process without terminating it. Often this is done to allow a foreground process to be moved to the background. To stop a foreground process, type `Ctrl-z`. Let's try it. At the command prompt, type `xlogo`, the Enter key, then `Ctrl-z`:

```
[me@linuxbox ~]$ xlogo
[1]+  Stopped                  xlogo
[me@linuxbox ~]$
```

After stopping `xlogo`, we can verify that the program has stopped by attempting to resize the `xlogo` window. We will see that it appears quite dead. We can either restore the program to the foreground using the `fg` command or move the program to the background with the `bg` command:

```
[me@linuxbox ~]$ bg %1
[1]+  xlogo &
[me@linuxbox ~]$
```

As with the `fg` command, the jobspec is optional if there is only one job.

Moving a process from the foreground to the background is handy if we launch a graphical program from the command but forget to place it in the background by appending the trailing “&”.

But why would you want to launch a graphical program from the command line? There are two reasons. First, the program you wish to run might not be listed on the windows manager's menus (such as `xlogo`.) Secondly, by launching a program from the command line, you might be able to see error messages that would otherwise be invisible to you if the program were launched graphically. Sometimes, a program will fail to start up when launched from the graphical menu. By launching it from the command line instead, you may see an error message that will reveal the problem. We will also see in later chapters, some graphical programs have many interesting and useful command line options.

Signals

The `kill` command is used to “kill” programs. This allows us to terminate programs that need killing. Here's an example:

```
[me@linuxbox ~]$ xlogo &
[1] 28401
[me@linuxbox ~]$ kill 28401
[1]+  Terminated              xlogo
```

We first launch `xlogo` in the background. The shell prints the jobspec and the PID of

the background process. Next, we use the `kill` command and specify the PID of the process we want to terminate. We could have also specified the process using a jobspec (for example, “%1”) instead of a PID.

While this is all very straightforward, there is more to it than that. The `kill` command doesn't exactly “kill” programs, rather it sends them *signals*. Signals are one of several ways that the operating system communicates with programs. We have already seen signals in action with the use of Ctrl-c and Ctrl-z. When the terminal receives one of these keystrokes, it sends a signal to the program in the foreground. In the case of Ctrl-c, a signal called INT (Interrupt) is sent; with Ctrl-z, a signal called TSTP (Terminal Stop.) Programs, in turn, “listen” for signals and may act upon them as they are received. The fact that a program can listen and act upon signals allows a program to do things like save work in progress when it is sent a termination signal.

Sending Signals To Processes With `kill`

The `kill` command is used to send signals to programs. Its most common syntax looks like this:

```
kill [-signal] PID...
```

If no signal is specified on the command line, then the TERM (Terminate) signal is sent by default. The `kill` command is most often used to send the following signals:

Number	Name	Meaning
1	HUP	<p>Hangup. This is a vestige of the good old days when terminals were attached to remote computers with phone lines and modems. The signal is used to indicate to programs that the controlling terminal has “hung up.” The effect of this signal can be demonstrated by closing a terminal session. The foreground program running on the terminal will be sent the signal and will terminate.</p> <p>This signal is also used by many daemon programs to cause a reinitialization. This means that when a daemon is sent this signal, it will restart and re-read its configuration file. The Apache web server is an example of a daemon that uses the HUP signal in this way.</p>

2	INT	Interrupt. Performs the same function as the Ctrl-c key sent from the terminal. It will usually terminate a program.
9	KILL	Kill. This signal is special. Whereas programs may choose to handle signals sent to them in different ways including ignoring them all together, the KILL signal is never actually sent to the target program. Rather, the kernel immediately terminates the process. When a process is terminated in this manner, it is given no opportunity to “clean up” after itself including saving its work. For this reason, the KILL signal should only be used as a last resort when other termination signals fail.
15	TERM	Terminate. This is the default signal sent by the <code>kill</code> command. If a program is still “alive” enough to receive signals, it will terminate.
18	CONT	Continue. This will restore a process after a STOP signal.
19	STOP	Stop. This signal causes a process to pause without terminating. Like the KILL signal, it is not sent to the target process and thus it cannot be ignored.

Let's try out the `kill` command:

```
[me@linuxbox ~]$ xlogo &
[1] 13546
[me@linuxbox ~]$ kill -1 13546
[1]+  Hangup                xlogo
```

In this example, we start the `xlogo` program in the background and then send it a HUP signal with `kill`. The `xlogo` program terminates and the shell indicates that the background process has received a hangup signal. You may need to press the enter key a couple of times before you see the message. Note that signals may be specified either by number or by name including a name prefixed with the letters “SIG”:

```
[me@linuxbox ~]$ xlogo &
[1] 13601
[me@linuxbox ~]$ kill -INT 13601
[1]+  Interrupt                xlogo
[me@linuxbox ~]$ xlogo &
[1] 13608
[me@linuxbox ~]$ kill -SIGINT 13608
[1]+  Interrupt                xlogo
```

Repeat the example above and try out the other signals. Remember you can also use jobspecs in place of PIDs.

Processes, like files, have owners and you must be the owner of a process (or the superuser) in order to send it signals with `kill`.

In addition to the list of signals above, which are most often used with `kill`, there are other signals frequently used by the system. Here is a list of other common signals you are likely to see:

Number	Name	Meaning
3	QUIT	Quit.
11	SEGV	Segmentation Violation. This signal is sent if a program makes illegal use of memory, that is, it tried to write somewhere it was not allowed to.
20	TSTP	Terminal Stop. This is the signal sent by the Ctrl-z key by the terminal. Unlike the STOP signal, the TSTP signal is received by the process and may be ignored.
28	WINCH	Window Change. This is a signal sent by the system when a window changes size. Some programs, like <code>top</code> and <code>less</code> will respond to this signal by redrawing themselves to fit the new window dimensions.

For the curious, you can get the complete list of signals with the following command:

```
[me@linuxbox ~]$ kill -l
```

Sending Signals To Multiple Processes With `killall`

It's also possible to send signals to multiple processes matching a specified program or user name by using the `killall` command. Here is the syntax:

```
killall [-u user] [-signal] name...
```

To demonstrate, we will start a couple of instances of the `xlogo` program and then terminate them:

```
[me@linuxbox ~]$ xlogo &
[1] 18801
[me@linuxbox ~]$ xlogo &
[2] 18802
[me@linuxbox ~]$ killall xlogo
[1]-  Terminated          xlogo
[2]+  Terminated          xlogo
```

Remember, as with `kill`, you must have superuser privileges to send signals to processes that do not belong to you.

More Process Related Commands You May Enjoy

Since monitoring processes is an important system administration task, there are a lot of commands for it. Here are some you can play with:

Command	Description
<code>pstree</code>	Outputs a process list arranged in a tree-like pattern showing the parent/child relationships between processes.
<code>vmstat</code>	Outputs a snapshot of system resource usage including, memory, swap and disk I/O. To see a continuous display, follow the command with a time delay (in seconds) for updates. For example: <code>vmstat 5</code> . Terminate the output with Ctrl-c.
<code>xload</code>	A graphical program that draws a graph showing system load over time.
<code>tload</code>	Similar to the <code>xload</code> program, but draws the graph in the terminal. Terminate the output with Ctrl-c.

Part 3 - Configuration And The Environment

12 - The Environment

As we discussed earlier, the shell maintains a body of information during your shell session called the *environment*. Data stored in the environment is used by programs to determine facts about your configuration. While most programs use *configuration files* to store program settings, some programs will also look for values stored in the environment to adjust their behavior. By knowing this, we can use the environment to customize our shell experience.

In this chapter we will work with the following commands:

- `printenv` – Print part or all of the environment
- `set` – Set shell options
- `export` – Export environment to subsequently executed programs
- `alias` – Create an alias for a command

What Is Stored In The Environment?

The shell stores two basic types of data in the environment though with `bash` the types are largely indistinguishable. They are *environment variables* and *shell variables*. Shell variables are bits of data placed there by `bash`, and environment variables are basically everything else. In addition to variables, the shell also stores some programmatic data, namely *aliases* and *shell functions*. We covered aliases in Chapter 6 and shell functions (which are related to shell scripting) will be covered in Part 5.

Examining The Environment

We can use both the `set` builtin in `bash` or the `printenv` program to see what is stored in the environment. The `set` command will show both the shell and environment variables while `printenv` will only display the latter. Since the list of environment contents will be fairly long, it is best to pipe the output of either command into `less`:

```
[me@linuxbox ~]$ printenv | less
```


Doing so, we should get something that looks like this:

```
KDE_MULTIHEAD=false
SSH_AGENT_PID=6666
HOSTNAME=linuxbox
GPG_AGENT_INFO=/tmp/gpg-Pd0t7g/S.gpg-agent:6689:1
SHELL=/bin/bash
TERM=xterm
XDG_MENU_PREFIX=kde-
HISTSIZE=1000
XDG_SESSION_COOKIE=6d7b05c65846c3eaf3101b0046bd2b00-1208521990.996705-1177056199
GTK2_RC_FILES=/etc/gtk-2.0/gtkrc:/home/me/.gtkrc-2.0:/home/bshotts/.kde/share/config/gtkrc-2.0
GTK_RC_FILES=/etc/gtk/gtkrc:/home/me/.gtkrc:/home/bshotts/.kde/share/config/gtkrc
GS_LIB=/home/me/.fonts
WINDOWID=29360136
QTDIR=/usr/lib/qt-3.3
QTINC=/usr/lib/qt-3.3/include
KDE_FULL_SESSION=true
USER=me
LS_COLORS=no=00:fi=00:di=00;34:ln=00;36:pi=40;33:so=00;35:bd=40;33;01:cd=40;33;01:or=01;05;37;41:mi=01;05;37;41:ex=00;32:*.cmd=00;32:*.exe:
```

What we see is a list of environment variables and their values. For example we see a variable called `USER` which contains the value “me”. The `printenv` command can also list the value of a specific variable:

```
[me@linuxbox ~]$ printenv USER
me
```

The `set` command, when used without options or arguments, will display both the shell and environment variables as well as any defined shell functions. Unlike `printenv`, its output is courteously sorted in alphabetical order:

```
[me@linuxbox ~]$ set | less
```

It is also possible to view the contents of a variable using the `echo` command, like this:

```
[me@linuxbox ~]$ echo $HOME
/home/me
```

One element of the environment that neither `set` nor `printenv` displays is aliases. To see them, enter the `alias` command without arguments:

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
```

Some Interesting Variables

The environment contains quite a few variables and your environment may differ from the one presented here but you will likely see the following variable in your environment:

Variable	Contents
PATH	A colon-separated list of directories that are searched when you enter the name of a executable program.
SHELL	The name of your shell program.
USER	Your user name.
DISPLAY	The name of your display if you are running a graphical environment. Usually this is “:0” meaning the first display generated by the X server.
TERM	The name of your terminal type. Unix-like systems support many terminal protocols and this variable sets the protocol to be used with your terminal emulator.
PS1	Prompt String 1. This defines the contents of your shell prompt. As we will later see, this can be extensively customized.
LANG	Defines the character set and collation order of your language.
TZ	Specifies your timezone. Most Unix-like systems maintain the computers internal clock in <i>Coordinated Universal Time</i> (UTC) and then display the local time by applying an offset specified by this variable.

HOME	The pathname of your home directory.
PWD	The current working directory.
OLD_PWD	The previous current working directory.
PAGER	The name of the program to be used for paging output. Often this is set to <code>/usr/bin/less</code> .
EDITOR	Than name of the program to be used for text editing.

Don't worry if some of these values are not set on your system.

How Is The Environment Established?

When you log on to your system, the `bash` program starts and reads a series of configuration scripts called *startup files* that define the default environment shared by all users. This is followed by more startup files in your home directory that define your personal environment. Here is a list of the files (not all are required) in the order in which they are read by `bash`:

File	Contents
<code>/etc/profile</code>	A global configuration script that applies to all users.
<code>~/.bash_profile</code>	A user's personal startup file. Can be used to extend or override settings in the global configuration script.
<code>~/.bash_login</code>	If <code>~/.bash_profile</code> is not found, <code>bash</code> attempts to read this file.
<code>~/.profile</code>	If neither <code>~/.bash_profile</code> nor <code>~/.bash_login</code> are found, <code>bash</code> attempts to read this file. This is the default in some version of Ubuntu.
<code>~/.bashrc</code>	This file contains additional configuration settings, typically aliases and shell functions. It is read whenever a user logs in or when new terminal session is started.

Take a look at your system and see which of these file you have. Remember -- since most of the file names listed above start with a period (meaning that they are hidden), you will need to use the “-a” option when using `ls`.

What's In A Startup File?

If we take a look inside a typical `.bash_profile`, it looks something like this:

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin
export PATH
```

Lines that begin with a “#” are *comments* and not read by the shell. These are for human readability. The first interesting thing occurs on the fourth line with the following code:

```
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
```

This is called an *if statement*, and we will cover it fully when we get to shell scripting in Part 5, but for now I will translate:

```
If the file “~/.bashrc” exists, then
    read the “~/.bashrc” file.
```

So we can see that this bit of code is how the shell gets the contents of `.bashrc`. The next thing in our startup file has to do with the `PATH` variable.

Have you ever wondered how the shell knows where to find commands when you enter them on the command line? For example, when you enter `ls`, the shell does not search the entire computer to find `/bin/ls` (the full pathname of the `ls` command), rather it searches a list of directories that are contained in the `PATH` variable.

The `PATH` variable is initially set by the `/etc/profile` startup file and with this code:

```
PATH=$PATH:$HOME/bin
```

is modified to add the directory `$HOME/bin` to the end of the list. This is an example of parameter expansion that we covered in Chapter 8. To demonstrate how this works, try the following:

```
[me@linuxbox ~]$ foo="This is some "  
[me@linuxbox ~]$ echo $foo  
This is some  
[me@linuxbox ~]$ foo=$foo"text."  
[me@linuxbox ~]$ echo $foo  
This is some text.
```

Using this technique, we can append text to the end of a variable's contents.

By adding the string `$HOME/bin` to the end of the `PATH` variable's contents, the directory `$HOME/bin` is added to the list of directories searched when a command is entered. What this means is if you want to create a directory within your home directory for storing your own private programs, the shell is ready to accommodate you. All you have to do is call it `bin` and you are ready to go.

Lastly, we have:

```
export PATH
```

The `export` command tells the shell to make the contents of `PATH` available to child process of this shell.

Modifying The Environment

Since we know where the startup files are and what they contain, we can modify them to customize our environment.

Which Files Should We Modify?

As a general rule, if you need to add directories to your `PATH`, or define additional environment variables, place those changes in `.bash_profile` (or equivalent according to your distribution. For example, Ubuntu uses `.profile`.) For everything else, place the changes in `.bashrc`. Unless you are the system administrator and need to change the defaults for all users of the system, restrict your modifications to the files in your home directory. It is certainly possible to change the files in `/etc` such as `profile`, and in many cases it would be sensible to do so, but for now let's play it safe.

Text Editors

To edit (i.e. modify) the shell's startup files, as well as, most of the other configuration files on the system, we use a program called a *text editor*. A text editor is a program that

is in some ways like a word processor in that it allows you to edit the words on the screen with a moving cursor. It differs from a word processor by only supporting pure text and often contain features designed for writing programs. Text editors are the central tool used by software developers to write code and by system administrators to manage the configuration files that control the system.

There are a lot of different text editors available for Linux and your system probably has several installed. Why so many different ones? Probably because programmers like writing them and since programmers use them extensively, they write editors to express their own desires as to how they should work.

Text editors fall into two basic categories, graphical and text based. GNOME and KDE both include some popular graphical editors. GNOME ships with an editor called `gedit` which is usually called “Text Editor” in the GNOME menu. KDE usually ships with three which are (in order of increasing complexity) `kedit`, `kwrite`, and `kate`.

There are many text based editors. The popular ones you will encounter are `nano`, `vi`, and `emacs`. The `nano` editor is a simple, easy-to-use editor designed as a replacement for the `pico` editor supplied with the PINE email suite. The `vi` editor (actually, on most Linux systems it is replaced by a program named `vim` which is short for “Vi IMproved”) is the traditional editor for Unix-like systems. It will be the subject of our next chapter. The `emacs` editor was originally written by Richard Stallman. It is a gigantic, all-purpose, does-everything programming environment. While always available, it is seldom installed on most Linux systems by default.

Using Your Text Editor

All text editors can be invoked from command line by typing the name of the editor followed by the name of the file you want to edit. If the file does not already exist, the editor will assume that you want to create a new file. Here is an example using `gedit`:

```
[me@linuxbox ~]$ gedit some_file
```

This command will start the `gedit` text editor and load the file named “some_file” if it exists.

All of the graphical text editors are pretty self-explanatory, so we won't cover them here. Instead, we will concentrate on our first text based text editor, `nano`. Let's fire up `nano` and edit the `.bashrc` file. But before we do that, let's practice some “safe computing.” Whenever you edit an important configuration file, it is always a good idea to create a backup copy of the file first. This protects us in case we mess the file up while editing. To create a backup of the `.bashrc` file, do this:

```
[me@linuxbox ~]$ cp .bashrc .bashrc.bak
```

It doesn't matter what you call the backup file, just be sure you can understand it. The extensions “.bak”, “.sav”, “.old”, and “.orig” are all popular ways of indicating a backup file.

Now that we have a backup file, we'll start the editor:

```
[me@linuxbox ~]$ nano .bashrc
```

Once nano starts, you'll get a screen like this:

```
GNU nano 2.0.3          File: .bashrc
# .bashrc
# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi
# User specific aliases and functions

[ Read 8 lines ]
^G Get Help ^O WriteOut ^R Read Fil ^Y Prev Pag ^K Cut Text ^C Cur Pos
^X Exit     ^J Justify  ^W Where Is ^V Next Pag ^U UnCut Te ^T To Spell
```

Note: If your system does not have nano installed you may use a graphical editor instead.

The screen consists of a header at the top, the text of the file being edited in the middle and a menu of commands at the bottom. Since nano was designed to replace the text editor supplied with an email client, it is rather short on editing features.

The first command you should learn is any text editor is how to exit the program. In the

case of `nano`, you type `Ctrl-x` to exit. This is indicated in the menu at the bottom of the screen. The notation “`^X`” means `Ctrl-x`. A number of other programs express control characters this way as well.

The second command we need to know is how to save our work. With `nano` it's `Ctrl-o`. With this knowledge under our belts, we're ready to do some editing. Using the down arrow key and/or the page down key, move the cursor to the end of the file then add the following lines to the `.bashrc` file:

```
umask 0002
export HISTCONTROL=ignoredups
export HISTSIZE=1000
alias l='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

Note: your distribution may already include some of these but duplicates won't hurt anything.

Here is the meaning of our additions:

Line	Meaning
<code>umask 0002</code>	Sets the <code>umask</code> to solve the problem with shared directories we discussed in Chapter 10.
<code>export HISTCONTROL=ignoredups</code>	Causes the shell's history recording feature to ignore a command if the same command was just recorded.
<code>export HISTSIZE=1000</code>	Increase the size of the command history from the default of 500 lines to 1000 lines.
<code>alias l='ls -d .* --color=auto'</code>	Create a new command called “ <code>l.</code> ” which displays all directory entries that begin with a dot.
<code>alias ll='ls -l --color=auto'</code>	Create a new command called “ <code>ll</code> ” which displays a long format directory listing.

As we can see, many of our additions are not intuitively obvious so it would be a good

idea to add some comments to our `.bashrc` file to help explain things to the humans. Using the editor, change our additions to look like this:

```
# Change umask to make directory sharing easier
umask 0002

# Ignore duplicates in command history and increase
# history size to 1000 lines
export HISTCONTROL=ignoredups
export HISTSIZE=1000

# Add some helpful aliases
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

Ah, much better! With our changes complete, type `Ctrl-o` to save our modified `.bashrc` file and `Ctrl-x` to exit `nano`.

Why Comments Are Important

Whenever you modify configuration files it's a good idea to add some comments to document your changes. Sure, you will remember what you changed tomorrow but what about six months from now? Do yourself a favor and add some comments. While you're at it, it's not a bad idea to keep a log of what changes you make.

Shell scripts and the `bash` startup files use a “#” symbol to begin a comment. Other configuration files may use other symbols. Most configuration files will have comments. Use them as a guide.

You will often see lines in configuration files that are *commented out* to prevent them from being interpreted by the affected program. This is done to give the reader suggestions for possible configuration choices or examples of correct configuration syntax. For example, the `.bashrc` file of Ubuntu 8.04 contains these lines:

```
# some more ls aliases
#alias ll='ls -l'
#alias la='ls -A'
#alias l='ls -CF'
```

The last three lines are valid alias definitions that have been commented out. If you remove the leading “#” symbols from these three lines, a technique called *uncommenting*, you will activate the aliases. Conversely, if you add a “#” symbol

to the beginning of a line you can deactivate a configuration line while preserving the information it contains.

Activating Our Changes

The changes we have made to our `.bashrc` will not take affect until we close our terminal session and start a new one since the `.bashrc` file is only read at the beginning of a session. However, we can force `bash` to re-read the modified `.bashrc` file with the following command:

```
[me@linuxbox ~]$ source .bashrc
```

After doing this we should be able to see the effect of our changes. Try out one of the new aliases:

```
[me@linuxbox ~]$ ll
```

Summing Up

In this chapter we learned an essential skill-- editing configuration files with a text editor. Moving forward, as you read man pages for commands, take note of the environment variables that commands support. You may find a gem or two. In later chapters we will learn about shell functions, a powerful feature that you can also include in the `bash` startup files to add to your arsenal of custom commands.

13 - A Gentle Introduction To vi

There is an old joke about a visitor to New York City asking a passerby for directions to the city's famous classical music venue:

Visitor: Excuse me, how do I get to Carnegie Hall?

Passerby: Practice, practice, practice!

Learning the Linux command line, like becoming an accomplished pianist, is not something that you pick up in an afternoon. It takes years of practice. In this chapter we will introduce the `vi` text editor, one of the core programs in the Unix tradition. `vi` is somewhat notorious for its difficult user interface, but when you see a master sit down at the keyboard and begin to “play” you will indeed be witness to some great art. We won't become masters in this chapter, but when you are done you will know how to play “chopsticks” in `vi`.

Why You Should Learn vi

In this modern age of graphical editors and easy-to-use text-based editors such as `nano`, why should you learn `vi`? I can think of three good reasons:

- `vi` is always available. This can be a lifesaver if you have a system with no graphical interface such as a remote server or a local system with a broken X configuration. `nano`, while increasingly popular is still not universal. POSIX, a standard for program compatibility on Unix systems, requires that `vi` be present.
- `vi` is lightweight and fast. Honestly, for many tasks it's easier to bring up `vi` than it is to find your graphical text editor in the menus and wait for its multiple megabytes to load. In addition, `vi` is designed for typing speed. As we shall see, a skilled `vi` user never has to lift his or her fingers from the keyboard while editing.
- You don't want other Linux and Unix users to think you are a sissy.

O.K. maybe two good reasons.

A Little Background

`vi` was first written in 1976 by University of California at Berkley student Bill Joy who later went on to co-found Sun Microsystems. `vi` derives its name from the word “visual” since it was intended to allow editing on a video terminal with a moving cursor. Previous to *visual editors*, there were *line editors* which operated on a single line of text at a time. To specify a change, you tell a line editor to go to a particular line and describe what change to make such as adding or deleting text. With the advent of video terminals (rather than printer-based terminals such as teletypes) you could perform visual editing. `vi` actually incorporates a powerful line editor called `ex` and you can use line editing commands in while using `vi`.

Most Linux distributions don't include real `vi`, rather they ship with an enhanced replacement called `vim` (which is short for “vi improved”) written by Bram Moolenaar. `vim` is a substantial improvement over traditional Unix `vi` and it is usually is symbolically linked (or aliased) to the name “vi” on your system. In the discussions that follow we will assume that you have a program called “vi” that is really `vim`.

Starting And Stopping vi

To start `vi`, we simply type the following:

```
[me@linuxbox ~]$ vi
```

And a screen like this should appear:

```
~
~
~
~                          VIM - Vi Improved
~
~                          version 7.1.138
~                          by Bram Moolenaar et al.
~                          Vim is open source and freely distributable
~
~                          Sponsor Vim development!
~ type :help sponsor<Enter>   for information
~
~ type :q<Enter>              to exit
~ type :help<Enter>  or <F1>  for on-line help
~ type :help version7<Enter>  for version info
~
~                          Running in Vi compatible mode
~ type :set nocp<Enter>      for Vim defaults
```

```
~      type  :help cp-default<Enter> for info on this
~
~
~
```

Just as we did with nano earlier, the first thing to learn is how to exit. To exit you enter the following command:

```
:q
```

and the shell prompt should return. If, for some reason, vi will not let you quit (it's usually because you made a change to file that has not yet been saved), you can tell vi that you really mean it by adding an exclamation point to the command:

```
:q!
```

Tip: If you get “lost” in vi, try pressing the ESC key twice to return it to a sensible state.

Compatibility Mode

In the example startup screen above (taken from Ubuntu 8.04), we see the text “Running in Vi compatible mode.” This means that vim will run in a mode that is closer to the normal behavior of vi rather than enhanced behavior of vim. For purposes of this chapter, we will want to run vim with its enhanced behavior. To do this you have a few options:

Try running vim instead of vi.

If that works, consider adding alias 'vi=vim' to your .bashrc file.

Alternately, use this command to add a line to your vim configuration file:

```
echo "set nocp" >> ~/.vimrc
```


Entering Insert Mode

In order to add some text to our file, we must first enter *insert mode*. To do this, press the “i” key. Afterwards you should see the following at the bottom of the screen if vim is running in its usual enhanced mode (in vi compatible mode this will not appear):

```
-- INSERT --
```

Now we can add enter some text. Try this:

```
The quick brown fox jumped over the lazy dog.
```

To exit insert mode and return to command mode, press the Esc key.

Saving Our Work

To save the change we just made to our file, we must enter an *ex command* while in command mode. This is easily done by pressing the “:” key. After doing this, a colon character should appear at the bottom of the screen:

```
:
```

To write our modified file, we follow the colon with a “w” then Enter:

```
:w
```

The file will be written to the hard drive and you should get a confirmation message at the bottom of the screen like this:

```
"foo.txt" [New] 1L, 46C written
```

Tip: If you read the vim documentation, you will notice that they (confusingly) call command mode *normal mode* and ex commands *command mode*. Beware.

Moving The Cursor Around

While in command mode, `vi` offers a large number of movement commands, some of which it shares with `less`. Here is a subset:

Key	Moves The Cursor
<code>l</code> or Right Arrow	Right one character,
<code>h</code> or Left Arrow	Left one character.
<code>j</code> or Down Arrow	Down one line.
<code>k</code> or Up Arrow	Up one line.
<code>0</code> (zero)	To the beginning of the current line.
<code>^</code>	To the first non-whitespace character on the current line.
<code>\$</code>	To the end of the current line.
<code>w</code>	To the beginning of the next word or punctuation character.
<code>W</code>	To the beginning of the next word ignoring punctuation characters.
<code>b</code>	To the beginning of the previous word or punctuation character.
<code>B</code>	To the beginning of the previous word ignoring punctuation characters.
<code>Ctrl-f</code> or Page Down	Down one page.
<code>Ctrl-b</code> or Page Up	Up one page
<code>numberG</code>	To line <i>number</i> . For example, <code>1G</code> moves to the first line of the file.
<code>G</code>	To the last line of the file.

Why are the `h`, `j`, `k`, and `l` keys used for cursor movement? It's because when `vi` was originally written, not all video terminals had arrow keys, and by using regular keyboard keys for cursor movement, skilled typists would never have to lift their fingers from the typing area of the keyboard.

Many commands in `vi` can be prefixed with a number as with the “`G`” command listed above. By prefixing a command with a number, we may specify the number of times a command is to be carried out. For example, the command “`5j`” causes `vi` to move the

cursor down five lines.

Basic Editing

Most editing consists of a few basic operations such as inserting text, deleting text and moving text around by cutting and pasting. `vi` of course supports all of these operations in its own unique way. `vi` also provides a limited form of undo. If we press the “u” key while in command mode, `vi` will undo the last change that you made. This will come in handy as we try out some of the basic editing commands.

Appending Text

`vi` has several different ways of entering insert mode. We have already used the `i` command to insert text.

Let's go back to our `foo.txt` file for a moment:

```
The quick brown fox jumped over the lazy dog.
```

If we wanted to add some text to the end of this sentence, we would discover that the `i` command will not do it, since we can't move the cursor beyond the end of the line. `vi` provides a command to append text, the sensibly named “a” command. If we move the cursor to the end of the line and type “a”, the cursor will move past the end of the line and `vi` will enter insert mode. This will allow us to add some more text:

```
The quick brown fox jumped over the lazy dog. It was cool.
```

Remember to press the `ESC` key to exit insert mode.

Since you will almost always want to append text to the end of a line, `vi` offers a shortcut to move to end of the current line and start appending. It's the “A” command. Let's try it and add some more lines to our file.

First, we'll move the cursor to the beginning of the line using the “0” (zero) command. Now type “A” and add the following lines of text:

```
The quick brown fox jumped over the lazy dog. It was cool.  
Line 2  
Line 3  
Line 4
```

Line 5

Again, press the ESC key to exit insert mode.

Opening A Line

Another way we can insert text is by “opening” a line. This inserts a blank line between two existing lines and enters insert mode. This has two variants:

Command	Opens
<code>o</code>	The line below the current line.
<code>O</code>	The line above the current line.

We can demonstrate this as follows: place the cursor on “Line 3” then press the `o` key.

```
The quick brown fox jumped over the lazy dog.  It was cool.  
Line 2  
Line 3  
  
Line 4  
Line 5
```

As we can see, a new line was opened below the third line and we entered insert mode. Exit insert mode by pressing the ESC key. Press the `u` key to undo our change.

Press the `O` key to open the line above the cursor:

```
The quick brown fox jumped over the lazy dog.  It was cool.  
Line 2  
  
Line 3  
Line 4  
Line 5
```

Exit insert mode by pressing the ESC key and undo our change by pressing `u`.

Deleting Text

As we might expect, `vi` offers a variety of ways to delete text. There are two main ways.

First, the `x` key will delete a character at the cursor location. `x` may be preceded by a number specifying how many characters are to be deleted. The `d` key is more general purpose. Like `x`, it may be preceded by a number specifying the number of times the deletion is to be performed. In addition, `d` is always followed by a movement command that controls the size of the deletion. Here are some examples:

Command	Deletes
<code>x</code>	The current character.
<code>3x</code>	The current character and the next two characters.
<code>dd</code>	The current line.
<code>5dd</code>	The current line and the next four lines.
<code>dW</code>	From the current cursor position to the beginning of the next word.
<code>d\$</code>	From the current cursor location to the end of the current line.
<code>d0</code>	From the current cursor location to the beginning of the line.
<code>d^</code>	From the current cursor location to the first non-whitespace character in the line.
<code>dG</code>	From the current line to the end of the file.
<code>d20G</code>	From the current line to the twentieth line of the file.

Place the cursor on the word “It” on the first line of our text. Press the `x` key repeatedly until the rest of the sentence is deleted. Next, press the `u` key repeatedly until the deletion is undone.

Note: Real `vi` only support a single level of undo. `vim` supports multiple levels.

Let's try the deletion again, this time using the `d` command. Again, move the cursor to the word “It” and press `dW` to delete the word:

```
The quick brown fox jumped over the lazy dog. was cool.
Line 2
Line 3
Line 4
Line 5
```

Press `d$` to delete from the cursor position to the end of the line:

```
The quick brown fox jumped over the lazy dog.  
Line 2  
Line 3  
Line 4  
Line 5
```

Press `dG` to delete from the current line to the end of the file:

```
~  
~  
~  
~  
~
```

Press `u` three times to undo the deletion.

Cutting, Copying And Pasting Text

The `d` command not only deletes text, it also “cuts” text. Each time we use the `d` command the deletion is copied into a paste buffer (think clipboard) that we can later recall with the `p` command to paste the contents of the buffer behind the cursor or the `P` command to paste the contents in front of the cursor.

The `y` command is used to “yank” (copy) text in much the same way the `d` command is used to cut. Here are some examples combining the `y` command with various movement commands:

Command	Copies
<code>yy</code>	The current line.
<code>5yy</code>	The current line and the next four lines.
<code>yw</code>	From the current cursor position to the beginning of the next word.
<code>y\$</code>	From the current cursor location to the end of the current line.
<code>y0</code>	From the current cursor location to the beginning of the line.
<code>y^</code>	From the current cursor location to the first non-

	whitespace character in the line.
yG	From the current line to the end of the file.
y20G	From the current line to the twentieth line of the file.

Let's try some copy and paste. Place the cursor on the first line of the text and type yy to copy the current line. Next, move the cursor to the last line (G) and type p to paste the line below the current line:

```
The quick brown fox jumped over the lazy dog.  It was cool.
Line 2
Line 3
Line 4
Line 5
The quick brown fox jumped over the lazy dog.  It was cool.
```

Just as before, the u command will undo our change. With the cursor still positioned on the last line of the file, type P to paste the text above the current line:

```
The quick brown fox jumped over the lazy dog.  It was cool.
Line 2
Line 3
Line 4
The quick brown fox jumped over the lazy dog.  It was cool.
Line 5
```

Try out some of the other y commands in the table above and get to know the behavior of both the p and P commands. When you are done, return the file to its original state.

Joining Lines

vi is rather strict about its idea of a line. Normally, it is not possible to move the cursor to the end of a line and delete the end-of-line character to join one line with the one below it. Because of this, vi provides a specific command (J, not to be confused with j which is for cursor movement) to join lines together.

If we place the cursor on line 3 and type the J command. Here's what happens:

```
The quick brown fox jumped over the lazy dog.  It was cool.
```

```
Line 2  
Line 3Line 4  
Line 5
```

Search And Replace

vi has the ability to move the cursor to locations based on searches. It can do this on both a single line or over an entire file. It can also perform text replacements with or without confirmation from the user.

Searching Within A Line

The `f` command searches a line and moves the cursor to the next instance of a specified character. For example, the command `fa` would move the cursor to the next occurrence of the character “a” within the current line. After performing a character search within a line, the search may be repeated by typing a semicolon.

Searching The Entire File

To move the cursor to the next occurrence of a word or phrase, the `/` command is used. This works the same way as we learned earlier in the `less` program. When you type the `/` command a “/” will appear at the bottom of the screen. Next, you type the word or phrase to be searched for followed by the `Enter` key. The cursor will move to the next location containing the search string. A search may be repeated using the previous search string with the `n` command. Here's an example:

```
The quick brown fox jumped over the lazy dog. It was cool.  
Line 2  
Line 3  
Line 4  
Line 5
```

Place the cursor on the first line of the file. Type:

```
/Line
```

followed by the `Enter` key. The cursor will move to line 2 and the word “Line” will be highlighted. Next, type `n` and the cursor will move to line 3. Repeating the `n` command will move the cursor down the file until it runs out of matches. While we have so far

only used words and phrases for our search patterns, `vi` allows the use of *regular expressions*, a powerful method of expressing complex text patterns. We will cover regular expressions in some detail in a later chapter.

Global Search And Replace

`vi` uses an `ex` command to perform search and replace operations (called “substitution” in `vi`) over a range of lines or the entire file. To change the word “Line” to “line” for the entire file we would enter the following command:

```
:%s/Line/line/g
```

Let's break this command down into separate items and see what each one does:

Item	Meaning
:	The colon character starts an <code>ex</code> command.
%	Specifies the range of lines for the operation. % is a shortcut meaning from the first line to the last line. Alternately, the range could have been specified <code>1, 5</code> (since our file is five lines long), or <code>1, \$</code> which means “from line 1 to the last line in the file.” If the range of lines is omitted, the operation is only performed on the current line.
s	Specifies the operation. In this case, substitution (search and replace.)
/Line/line/	The search pattern and the replacement text.
g	This means “global” in the sense that the search and replace is performed on every instance of the search string in the line. If omitted, only the first instance of the search string on each line is replaced.

After executing our search and replace command our file looks like this:

```
The quick brown fox jumped over the lazy dog. It was cool.
line 2
line 3
line 4
line 5
```

We can also specify a substitution command with user confirmation. This is done by adding a “c” to the end of the command. For example:

```
:%s/line/Line/gc
```

This command will change our file back to its previous form, however before each substitution, vi stops and asks us to confirm the substitution with this message:

```
replace with Line (y/n/a/q/l/^E/^Y)?
```

Each of the characters within the parentheses are possible choices as follows:

Key	Action
y	Perform the substitution.
n or Enter	Skip this instance of the pattern.
a	Perform the substitution on this and all subsequent instances of the pattern.
q or ESC	Quit substituting.
l	Perform this substitution and then quit. Short for “last.”
Ctrl-e, Ctrl-y	Scroll down and scroll up respectively. Useful for viewing the context of the proposed substitution.

If you type y the substitution will be performed, n or Enter will cause vi to skip this instance move on to the next one.

Editing Multiple Files

It's often useful to edit more than one file at a time. You might need to make changes to multiple files or you may need to copy content from one file into another. With vi we can open multiple files for editing by specifying them on the command line:

```
vi file1 file2 file3...
```

Let's exit our existing vi session and create a new file for editing. Type :q! to exit vi without saving your modified text. Next, we'll create an additional file in our home

directory that we can play with. We'll create by capturing some output from the `ls` command:

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Let's edit our old file and our new one with `vi`:

```
[me@linuxbox ~]$ vi foo.txt ls-output.txt
```

`vi` will start up and we will see the first file on the screen:

```
The quick brown fox jumped over the lazy dog. It was cool.  
Line 2  
Line 3  
Line 4  
Line 5
```

Switching Between Files

To switch from one file to the next, use this `ex` command:

```
:n
```

To move back to the previous file use:

```
:N
```

While we can move from one file to another, `vi` enforces a policy that prevents you from switching files if the current file has unsaved changes. To force `vi` to switch files and abandon your changes, append an exclamation point (!) to the command.

In addition to the switching method described above, `vim` (and some versions of `vi`) also provide some `ex` commands that make multiple files easier to manage. We can view a list of files being edited with the `:buffers` command. Doing so will display a list of the files at the bottom of the display:

```
:buffers
 1 %a  "foo.txt"           line 1
 2     "ls-output.txt"     line 0
Press ENTER or type command to continue
```

To switch to another buffer (file), type `:buffer` followed by the number of the buffer you wish to edit. For example, to switch from buffer 1 which contains the file `foo.txt` to buffer two containing the file `ls-output.txt` we would type this:

```
:buffer 2
```

and our screen now displays the second file.

Opening Additional Files For Editing

It's also possible to add files to our current editing session. The ex command `:e` (short for “edit”) followed by a file name will open an additional file. Let's end our current editing session and return to the command line.

Start `vi` again with just one file:

```
[me@linuxbox ~]$ vi foo.txt
```

To add our second file, enter:

```
:e ls-output.txt
```

And it should appear on the screen. The first file is still present as we can verify:

```
:buffers
 1 #   "foo.txt"           line 1
 2 %a  "ls-output.txt"     line 0
Press ENTER or type command to continue
```

Note: You cannot switch to files loaded with the `:e` command using either the `:n` or `:N` commands. To switch files, use the `:buffer` command followed by the

```
buffer number.
```

Copying Content From One File Into Another

Often while editing multiple files, we will want to copy a portion of one file into another file that we are editing. This is easily done using the usual yank and paste commands we used earlier. We can demonstrate as follows. First, using our two files, switch to buffer 1 (foo.txt) by entering:

```
:buffer 1
```

which should give us this:

```
The quick brown fox jumped over the lazy dog.  It was cool.  
Line 2  
Line 3  
Line 4  
Line 5
```

Next, move the cursor to the first line, and type yy to yank (copy) the line.

Switch to the second buffer by entering:

```
:buffer 2
```

The screen will now contain some file listings like this (only a portion is shown here):

```
total 343700  
-rwxr-xr-x 1 root root      31316 2007-12-05 08:58 [  
-rwxr-xr-x 1 root root       8240 2007-12-09 13:39 411toppm  
-rwxr-xr-x 1 root root    111276 2008-01-31 13:36 a2p  
-rwxr-xr-x 1 root root     25368 2006-10-06 20:16 a52dec  
-rwxr-xr-x 1 root root     11532 2007-05-04 17:43 aafire  
-rwxr-xr-x 1 root root       7292 2007-05-04 17:43 aainfo
```

Move the cursor to the first line and paste the line we copied from the preceding file by typing the p command:

```
total 343700
the quick brown fox jumped over the lazy dog.  It was cool.
-rwxr-xr-x 1 root root      31316 2007-12-05 08:58 [
-rwxr-xr-x 1 root root      8240 2007-12-09 13:39 411toppm
-rwxr-xr-x 1 root root     111276 2008-01-31 13:36 a2p
-rwxr-xr-x 1 root root     25368 2006-10-06 20:16 a52dec
-rwxr-xr-x 1 root root     11532 2007-05-04 17:43 aafire
-rwxr-xr-x 1 root root      7292 2007-05-04 17:43 aainfo
```

Inserting An Entire File Into Another

It's also possible to insert an entire file into one that we are editing. To see this in action let's end our vi session and start a new one with just a single file:

```
[me@linuxbox ~]$ vi ls-output.txt
```

We will see our file listing again:

```
total 343700
-rwxr-xr-x 1 root root      31316 2007-12-05 08:58 [
-rwxr-xr-x 1 root root      8240 2007-12-09 13:39 411toppm
-rwxr-xr-x 1 root root     111276 2008-01-31 13:36 a2p
-rwxr-xr-x 1 root root     25368 2006-10-06 20:16 a52dec
-rwxr-xr-x 1 root root     11532 2007-05-04 17:43 aafire
-rwxr-xr-x 1 root root      7292 2007-05-04 17:43 aainfo
```

Move the cursor to the third line then enter the following ex command:

```
:r foo.txt
```

The `:r` command (short for "read") inserts the specified file before the cursor position. Our screen should now look like this:

```
total 343700
-rwxr-xr-x 1 root root      31316 2007-12-05 08:58 [
-rwxr-xr-x 1 root root      8240 2007-12-09 13:39 411toppm
the quick brown fox jumped over the lazy dog.  It was cool.
Line 2
```

```
Line 3
Line 4
Line 5
-rwxr-xr-x 1 root root      111276 2008-01-31 13:36 a2p
-rwxr-xr-x 1 root root      25368 2006-10-06 20:16 a52dec
-rwxr-xr-x 1 root root      11532 2007-05-04 17:43 aafire
-rwxr-xr-x 1 root root       7292 2007-05-04 17:43 aainfo
```

Saving Our Work

Like everything else in `vi`, there are several different ways to save our edited files. We have already covered the `ex` command `:w` but there are some others you may also find helpful.

In command mode, typing `ZZ` will save the current file and exit `vi`. Likewise, the `ex` command `:wq` will combine the `:w` and `:q` commands into one that will both save the file and exit.

The `:w` command may also specify an optional file name. This acts like a “Save As...” in programs you may already know. For example, if we were editing `foo.txt` and wanted to save an alternate version called `foo1.txt`, we would enter the following:

```
:w foo1.txt
```

Note: While the command above saves the file under a new name, it does not change the name of the file you are editing. As you continue to edit, you will still be editing `foo.txt`, not `foo1.txt` as you might expect.

Additional Reading

Even with all that we have covered in this chapter, we have barely scratched the surface of what `vi` and `vim` can do. Here are a couple of on-line resources that you can use to continue your journey towards `vi` mastery:

[Learning The vi Editor](#) – A Wikibook from Wikipedia offers a concise guide to `vi` and several of its work-a-likes including `vim`. It's available at:

<http://en.wikibooks.org/wiki/Vi>

[The Vim Book](#) - The `vim` project has a 570 page book that covers (almost) all of the features in `vim`. You can find it at:

<ftp://ftp.vim.org/pub/vim/doc/book/vimbook-OPL.pdf>.

14 - Customizing Your Prompt

In this chapter we will look at a seemingly trivial detail, our shell prompt. This examination will reveal some of the inner workings of the shell and our terminal emulator program itself.

Like so many things in Linux, the shell prompt is highly configurable and while we have pretty much taken it for granted, the prompt is a really useful device once we learn how to control it.

Anatomy Of A Prompt

Our default prompt looks something like this:

```
[me@linuxbox ~]$
```

Notice that it contains our user name, our host name and our current working directory, but how did it get that way? Very simply, it turns out. The prompt is defined by an environment variable named `PS1` (short for “prompt string one”.) We can view the contents of `PS1` with the `echo` command:

```
[me@linuxbox ~]$ echo $PS1
[\u@\h \w]\$
```

Note: Don't worry if your results are not exactly the same the example above. Every Linux distribution defines the prompt string a little differently, some quite exotically.

From the results, we can see that `PS1` contains a few of the characters we see in our prompt such as the brackets, the at-sign, and the dollar sign, but the rest are a mystery. The astute among you will recognize these as *backslash-escaped special characters* like those we saw in Chapter 8. Here is a partial list of the characters that the shell treats

specially in the prompt string:

Sequence	Value Displayed
\a	ASCII bell. This makes the computer beep when it is encountered.
\d	The current date in day, month, date format. For example, “Mon May 26”
\h	Host name of the local machine minus the trailing domain name.
\H	Full host name.
\j	Number of jobs running in the current shell session.
\l	The name of the current terminal device.
\n	A newline character.
\r	A carriage return.
\s	Name of the shell program.
\t	The current time in 24 hour hours:minutes:seconds format.
\T	The current time in 12 hour format.
\@	The current time in 12 hour AM/PM format.
\A	The current time in 24 hour hours:minutes format.
\u	The user name of the current user.
\v	The version number of the shell.
\V	The version and release numbers of the shell.
\w	The name current working directory.
\W	The last part of the current working directory name.
\!	The history number of the current command.
\#	The number of commands entered into this shell session.
\\$	This displays a “\$” character unless you have superuser privileges. In that case it displays a “#” instead.
\[Signals the start of a series of one or more non-printing characters. This is used to embed non-printing control characters used to manipulate the terminal emulator in some way, such as moving the cursor or changing text colors.
\]	Signals the end of a non-printing character sequence.

Trying Some Alternate Prompt Designs

With this list of special characters we can change the prompt to see the effect. First, we'll back up the existing string so we can restore it later. To do this, we will copy the existing string into another shell variable that we create ourselves. We can do it like this:

```
[me@linuxbox ~]$ ps1_old="$PS1"
```

We create a new variable called `ps1_old` and assign the value of `PS1` to it. Later, when we restore the original value, we will reverse the process. We can verify that the value has been copied with the `echo` command:

```
[me@linuxbox ~]$ echo $ps1_old
[\u@\h \w]\$
```

We can restore the original prompt at any time during our terminal session by simply reversing the process:

```
[me@linuxbox ~]$ PS1="$ps1_old"
```

Now that we are ready to proceed, let's see what happens if we have an empty prompt string:

```
[me@linuxbox ~]$ PS1=
```

If we assign nothing to the prompt string, we get nothing. No prompt string at all! The prompt is still there, but displays nothing just as we asked it to. Since this is kind of disconcerting to look at, we'll replace it with a minimal prompt:

```
PS1="\$ "
```

That's better. At least now we can see what we are doing. Notice the trailing space within the double quotes. This provides the space between the dollar sign and the cursor when the prompt is displayed.

Let's add a bell to our prompt:

```
$ PS1="\a\$ "
```

Now we should hear a beep each time the prompt is displayed. This probably gets annoying but it might be useful if we needed notification when an especially long-running command were executed.

Next, let's try to make an informative prompt with some host name and time-of-day information:

```
$ PS1="\A \h \$ "  
17:33 linuxbox $
```

Adding time-of-day to our prompt would be useful if we need to keep track of when we performed certain tasks. Finally, we'll make a new prompt that is similar to our original:

```
17:37 linuxbox $ PS1="<\u@\h \w>\$ "  
<me@linuxbox ~>$
```

Try out the other sequences listed in the table above and see if you can come up with a brilliant new prompt.

Adding Color

Most terminal emulator programs respond to certain non-printing character sequences to control such things as character attributes (like color, bold text and the dreaded blinking text) and cursor position. We'll cover cursor position in a little bit, but first we'll look at color.

Terminal Confusion

Back in ancient times, when terminals were hooked to remote computers, there were many competing brands of terminals and they all worked differently. They had different keyboards and they all had different ways of interpreting control information. Unix and Unix-like systems have a two rather complex subsystems to deal with the babel of terminal control (called `termcap` and `terminfo`) and if you look in the deepest recesses of your terminal emulator settings you may find a setting for the type of terminal emulation.

In an effort to make terminals speak some sort of common language, the American National Standards Institute (ANSI) developed a standard set of character sequences to control video terminals. Old time DOS users will remember the ANSI .SYS file that was used to enable interpretation of these codes.

Character color is controlled by sending the terminal emulator an *ANSI escape code* embedded in the stream of characters to be displayed. The control code does not “print out” on the display, rather it is interpreted by the terminal as an instruction. As we saw in the table above, the `\[` and `\]` sequences are used to encapsulate non-printing characters. An ANSI escape code begins with an octal 033 (the code generated by the escape key) followed by an optional character attribute followed by an instruction. For example, the code to set the text color to normal (attribute = 0), black text is:

```
\033[0;30m
```

Here is a table of the available text colors. Notice that the colors are divided into two groups differentiated by the application of the bold character attribute (1) which creates the appearance of “light” colors:

Sequence	Text Color	Sequence	Text Color
<code>\033[0;30m</code>	Black	<code>\033[1;30m</code>	Dark Gray
<code>\033[0;31m</code>	Red	<code>\033[1;31m</code>	Light Red
<code>\033[0;32m</code>	Green	<code>\033[1;32m</code>	Light Green
<code>\033[0;33m</code>	Brown	<code>\033[1;33m</code>	Yellow
<code>\033[0;34m</code>	Blue	<code>\033[1;34m</code>	Light Blue
<code>\033[0;35m</code>	Purple	<code>\033[1;35m</code>	Light Purple
<code>\033[0;36m</code>	Cyan	<code>\033[1;36m</code>	Light Cyan
<code>\033[0;37m</code>	Light Grey	<code>\033[1;37m</code>	White

Let's try to make a red prompt. We'll insert the escape code at the beginning:

```
<me@linuxbox ~->$ PS1="\[\033[0;31m\]<\u@\h \w>\$ "
```

```
<me@linuxbox ~->$
```

That works, but notice that all the text that we type after the prompt is also red. To fix this, we will add another escape code to the end of the prompt that tells the terminal emulator to return to the previous color:

```
<me@linuxbox ~->$ PS1="\[\033[0;31m\]<\u@\h \w>\$\[\033[0m\] "
<me@linuxbox ~->$
```

That's better!

It's also possible to set the text background color using the codes listed below. The background colors do not support the bold attribute.

Sequence	Background Color	Sequence	Background Color
\033[0;40m	Black	\033[0;44m	Blue
\033[0;41m	Red	\033[0;45m	Purple
\033[0;42m	Green	\033[0;46m	Cyan
\033[0;43m	Brown	\033[0;47m	Light Grey

We can create a prompt with a red background by applying a simple change to the first escape code:

```
<me@linuxbox ~->$ PS1="\[\033[0;41m\]<\u@\h \w>\$\[\033[0m\] "
<me@linuxbox ~->$
```

Try out the color codes and see what you can create!

Note: Besides the normal (0) and bold (1) character attributes, text may also be given underscore (4), blinking (5), and inverse (7) attributes, as well. In the interests of good taste, many terminal emulators refuse to honor the blinking attribute however.

Moving The Cursor

Escape codes can be used to position the cursor. A common use of this is to provide a clock or some other kind of information at a different location on the screen such as an upper corner each time the prompt is drawn. Here is a list of the escape codes that position the cursor:

Escape Code	Action
\033[<i>l</i> ; <i>c</i> H	Move the cursor to line <i>l</i> and column <i>c</i> .
\033[<i>n</i> A	Move the cursor up <i>n</i> lines.

<code>\033[nB</code>	Move the cursor down n lines.
<code>\033[nC</code>	Move the cursor forward n characters.
<code>\033[nD</code>	Move the cursor backward n characters.
<code>\033[2J</code>	Clear the screen and move the cursor to the upper left corner (line 0, column 0.)
<code>\033[K</code>	Clear from the cursor position to the end of the current line.
<code>\033[s</code>	Store the current cursor position.
<code>\033[u</code>	Recall the stored cursor position.

Using the codes above, we'll construct a prompt that draws a red bar at the top of the screen containing a clock (rendered in yellow text) each time the prompt is displayed. The code for the prompt is this formidable looking string:

```
PS1="\[\033[s\033[0;0H\033[0;41m\033[K\033[1;33m\t\033[0m\033[u\  
<\u@\h \w>\$ "
```

Let's take a look at each part of the string to see what it does:

Sequence	Action
<code>\[</code>	Begins a non-printing character sequence. The real purpose of this is to allow <code>bash</code> to correctly calculate the size of the visible prompt. Without this, command line editing features will improperly position the cursor.
<code>\033[s</code>	Store the cursor position. This is needed to return to the prompt location after the bar and clock have been drawn at the top of the screen. <i>Be aware that some terminal emulators do not honor this code.</i>
<code>\033[0;0H</code>	Move the cursor to the upper left corner, which is line zero, column zero.
<code>\033[0;41m</code>	Set the background color to red.
<code>\033[K</code>	Clear from the current cursor location (the top left corner) to the end of the line. Since the background color is now red, the line is cleared to that color creating our bar. Note that clearing to the end of the line does not change the cursor position which remains at the upper left corner.

<code>\033[1;33m</code>	Set the text color to yellow.
<code>\t</code>	Display the current time. While this is a “printing” element, we still include it in the non-printing portion of the prompt since we don't want <code>bash</code> to include the clock when calculating the true size of the displayed prompt.
<code>\033[0m</code>	Turn off color. This affects both the text and background.
<code>\033[u</code>	Restore the cursor position saved earlier.
<code>\]</code>	End non-printing characters sequence.
<code><\u@\h \w>\\$</code>	Prompt string.

Saving The Prompt

Obviously, we don't want to be typing that monster all the time, so we'll want to store our prompt someplace. We can make the prompt permanent by adding it to our `.bashrc` file. To do so, add these two lines to the file:

```
PS1="\[\033[s\033[0;0H\033[0;41m\033[K\033[1;33m\t\033[0m\033[u\  
<\u@\h \w>\$ "  
  
export PS1
```

Summing Up

Believe it or not, there is much more that can be done with prompts involving shell functions and scripts that we haven't covered here, but this is a good start. Not everyone will care enough to change their prompt since the default prompt is usually satisfactory, but for those who like to tinker, the shell provides the opportunity for many hours of trivial fun.

Further Reading

The *Bash Prompt HOWTO* from the [Linux Documentation Project](http://www.linuxdocumentationproject.org/) provides a pretty complete discussion of what the shell prompt can be made to do. It is available at:

<http://tldp.org/HOWTO/Bash-Prompt-HOWTO/>

Wikipedia has a good article on the ANSI Escape Codes:

http://en.wikipedia.org/wiki/ANSI_escape_code

Part 4 - Common Tasks And Essential Tools

15 – Package Management

If you have spent any time in the Linux community, you have no doubt heard your share of opinions as to which of the many Linux distributions, is “best.” These discussions often get really silly focusing on such things the prettiness of the desktop background (some people won't use Ubuntu because its default color scheme is brown!) and other trivial matters.

For me, the most important determinant of distribution quality is the *packaging system* and the vitality of the distribution's support community. As you spend more time with Linux, you will see that its software landscape is extremely dynamic. Things are constantly changing. Most of the top-tier Linux distributions release new versions every six months and many individual program updates every day. To keep up with this blizzard of software, we need good tools for *package management*.

Package management is a method of installing and maintaining software on the system. Today, most people can satisfy all of their software needs by installing *packages* from their Linux distributor. This contrasts with the early days of Linux when one had to download and compile *source code* in order to install software. Not that there is anything wrong with compiling source code, in fact, having access to source code is the great wonder of Linux. It give you (and everybody else) the ability to examine and improve the system. It's just that having a pre-compiled package is faster and easier to deal with.

In this chapter, we will look at some of the command line tools used for package management. While all of the major distributions provide powerful and sophisticated graphical programs for maintaining the system, it is important to learn about the command line programs too. They can perform many tasks that are difficult (or impossible) to do with their graphical counterparts.

Packaging Systems

Different distributions use different packaging systems and as a general rule, a package intended for one distribution is not compatible with another distribution. Most distributions fall into one of two “camps” of packaging technologies. There is the Debian “.deb” camp and the Red Hat “.rpm” camp. There are some important exceptions such as Gentoo, Slackware, and Foresight, but most others use one of these two basic systems.

Packaging System	Distributions (Partial Listing)
Debian Style (.deb)	Debian, Ubuntu, Xandros, Linspire
Red Hat Style (.rpm)	Fedora, CentOS, Red Hat Enterprise Linux, OpenSUSE, Mandriva, PCLinuxOS

How A Package System Works

You may be familiar with the methods of software distribution found in the proprietary software industry. This usually entails buying a piece of installation media such as an “install disk” and then running an “installation wizard” to install a new application on the system.

Linux doesn't work that way. Virtually all software for a Linux system will be found on the Internet. Most of it will be provided by the distribution vendor in the form of *package files* and the rest will be available in source code form that can be installed manually. We'll talk a little about how to install software by compiling source code in a later chapter.

Package Files

The basic unit of software in a packaging system is, of course, the *package file*. A package file is a compressed collection of files that comprise the software package. A package may consist of numerous programs and data files that support the programs. In addition to the files that will be installed, the package file will also include meta data about the package, such as a text description of the package and its contents. Many packages also contain pre- and post-installation scripts that perform configuration tasks before and after the package installation.

Package files are created by a person known as a *package maintainer*, often (but not always) an employee of the distribution vendor. The package maintainer obtains the software in source code form from the *upstream provider* (the author of the program), compiles it, creates the package meta data and any necessary installation scripts. Often, the package maintainer will apply modifications to the original source code to improve the program's integration with the other parts of the Linux distribution.

Repositories

While some software projects choose to perform their own packaging and distribution, most packages today are created by the distribution vendors and interested third parties. Packages are made available to the users of a distribution in central repositories that may contain many thousands of packages each specially built and maintained for the distribution.

The distribution may maintain several different repositories for different stages of the software development life cycle. For example, there will usually be a “testing” repository that contains packages that have just been built and are intended for use by brave souls who are looking for bugs before they are released for general distribution. A distribution will often have a “development” repository where work-in-progress packages are kept that are destined for inclusion in the distribution's next major release.

A distribution may also have related third-party repositories. Often these are needed to supply software that, for legal reasons such as patents or DRM anti-circumvention issues, cannot be included with the distribution. Perhaps the best known case is that of encrypted DVD support, which is not legal in the United States. The third-party repositories operate in countries where software patents and anti-circumvention laws do not apply. These repositories are usually wholly independent of the distribution they support and to use them, one must know about them and manually include them in the configuration files for the package management system.

Dependencies

Programs seldom “stand-alone”, rather they rely on the presence of other software components to get their work done. Common activities, such as input/output for example, are handled by routines shared by many programs. These routines are stored in what are called *shared libraries* which provide essential services to more than one program. If a package requires a shared resource like a shared library, it is said to have a *dependency*. Modern package management systems all provide some method of *dependency resolution* to ensure that when a package is installed, all of its dependencies are installed too.

High And Low-level Package Tools

Package management systems usually consist of two types of tools, low-level tools which handle tasks such as installing and removing package files, and high-level tools that perform meta-data searching and dependency resolution. In this chapter we will look at the tools supplied with Debian-style systems (such as Ubuntu and many others) and those used by recent Red Hat products. While all Red Hat-style distributions use the same low-level program (`rpm`), they rely on different high-level tools. For our discussion, we will cover the high-level program `yum`, used by Fedora, Red Hat Enterprise Linux, and CentOS. Other Red Hat-style distributions provide high-level tools with comparable features.

Distributions	Low-Level Tools	High-Level Tools
Debian-Style	<code>dpkg</code>	<code>apt-get</code> , <code>aptitude</code>
Fedora, Red Hat	<code>rpm</code>	<code>yum</code>

Enterprise Linux, CentOS		
--------------------------	--	--

Common Package Management Tasks

There are many operations that can be performed with the command line package management tools. We will look at the most common. Be aware that the low-level tools also support creation of package files, an activity outside the scope of this book.

In the discussion below, the term “*package_name*” refers to the actual name of a package rather than the name of a “*package_file*,” that is, the name of the file that contains the package.

Finding A Package In A Repository

Using the high-level tools to search repository meta-data, a package can be located based on its name or description.

Style	Command(s)
Debian	apt-get update apt-cache search <i>search_string</i>
Red Hat	yum search <i>search_string</i>

Example: To search a yum repository for the emacs text editor, this command could be used:

```
yum search emacs
```

Installing A Package From A Repository

High-level tools permit a package to be downloaded from a repository and installed with full dependency resolution.

Style	Command(s)
Debian	apt-get update apt-get install <i>package_name</i>
Red Hat	yum install <i>package_name</i>

Example: To install the emacs text editor from an apt repository:

```
apt-get update; apt-get install emacs
```

Installing A Package From A Package File

If a package file has been downloaded from a source other than a repository, it can be installed directly (though without dependency resolution) using a low-level tool.

Style	Command(s)
Debian	<code>dpkg --install <i>package_file</i></code>
Red Hat	<code>rpm -i <i>package_file</i></code>

Example: If the `emacs-22.1-7.fc7-i386.rpm` package file had been downloaded from a non-repository site, it would be installed this way:

```
rpm -i emacs-22.1-7.fc7-i386.rpm
```

Note: Since this technique uses the low-level `rpm` program to perform the installation, no dependency resolution is performed. If `rpm` discovers a missing dependency, `rpm` will exit with an error.

Removing A Package

Packages can be un-installed using either the high-level or low-tools. The high-level tools are shown below.

Style	Command(s)
Debian	<code>apt-get remove <i>package_name</i></code>
Red Hat	<code>yum erase <i>package_name</i></code>

Example: To un-install the `emacs` package from a Debian-style system:

```
apt-get remove emacs
```

Updating Packages From A Repository

The most common package management task is keeping the system up-to-date with the latest packages. The high-level tools can perform this vital task in one single step.

Style	Command(s)
Debian	apt-get update apt-get upgrade
Red Hat	yum update

Example: To apply any available updates to the installed packages on a Debian-style system:

```
apt-get update; apt-get upgrade
```

Upgrading A Package From A Package File

If an updated version of a package has been downloaded from a non-repository source, it can be installed replacing the previous version:

Style	Command(s)
Debian	dpkg --install <i>package_file</i>
Red Hat	rpm -U <i>package_file</i>

Example: Updating an existing installation of emacs to the version contained in the package file `emacs-22.1-7.fc7-i386.rpm` on a Red Hat system:

```
rpm -U emacs-22.1-7.fc7-i386.rpm
```

Note: dpkg does not have a specific option for upgrading a package versus installing one as rpm does.

Listing Installed Packages

These commands can be used to display list of all the packages installed on the system:

Style	Command(s)
Debian	<code>dpkg --list</code>
Red Hat	<code>rpm -qa</code>

Determining If A Package Is Installed

These low-level tools can be used to display if a specified package is installed:

Style	Command(s)
Debian	<code>dpkg --status <i>package_name</i></code>
Red Hat	<code>rpm -q <i>package_name</i></code>

Example: To determine if the emacs package is installed on a Debian style system:

```
dpkg --status emacs
```

Displaying Info About An Installed Package

if the name of an installed package is known, the following commands can be used to display a description of the package:

Style	Command(s)
Debian	<code>apt-cache show <i>package_name</i></code>
Red Hat	<code>yum info <i>package_name</i></code>

Example: To see a description of the emacs package on a Debian-style system:

```
apt-cache show emacs
```

Finding Which Package Installed A File

To determine what package is responsible for the installation of a particular file, the following commands can be used:

Style	Command(s)
-------	------------

Debian	<code>dpkg --search <i>file_name</i></code>
Red Hat	<code>rpm -qf <i>file_name</i></code>

Example: To see what package installed the `/usr/bin/vim` file on a Red Hat system:

```
rpm -qf /usr/bin/vim
```

Summing Up

In the chapters that follow, we may need to install additional packages if necessary programs are not already installed on your system. With our new found knowledge (and appreciation) of package management, we should have no problem installing and managing the programs we need.

The Linux Software Installation Myth

People migrating from other platforms sometimes fall victim to the myth that software is somehow difficult to install under Linux and that the variety of packaging schemes used by different distributions is a hindrance. Well, it is a hindrance, but only to proprietary software vendors who wish to distribute binary-only versions of their secret software.

The Linux software ecosystem is based on the idea of open source code. If a program developer releases source code for a product, it is likely that a person associated with a distribution will package the product and include it in their repository. This method ensures that the product is well integrated into the distribution and the user is given the convenience of “one-stop shopping” for software rather than having the need to search for each product's web site.

Device drivers are handled in much the same way except that instead of being separate items in a distribution's repository, they become part of the Linux kernel itself. Generally speaking, there is no such thing as a “driver disk” in Linux. Either the kernel supports a device or it doesn't and the Linux kernel supports a lot of devices. Many more in fact than Windows does. Of course, this fact is of no consolation if the particular device you need is not supported but when that happens you need to look at the cause. A lack of driver support is usually caused by one of three things:

1. **The device is too new.** Since many hardware vendors don't actively support Linux development, it falls upon a member of the Linux community to write the kernel driver code. This takes time.
2. **The device is too exotic.** Not all distributions include every possible device driver. Each distribution builds their own kernels and since kernels are very configurable (after all it's what makes it possible to run Linux on everything from wristwatches to mainframes) they may have overlooked your device. By locating and downloading the source code for the driver, it is possible for you (yes, you) to compile and install the driver yourself. This process is not unduly difficult, but it is rather involved. We'll talk about compiling software in a later chapter.
3. **The hardware vendor is evil.** They have neither released source code for a Linux driver nor have they released the technical documentation for somebody to create one for them. This means that the hardware vendor is trying to keep the programming interfaces to the device a secret. Since we don't want secret devices in our computers, I suggest that you remove the offending hardware and pitch it into the trash because that's what it is.

Additional Reading

Spend some time getting to know the package management system for your distribution. Each distribution provides documentation for its package management tools. In addition, here are some more generic sources:

The [Debian GNU/Linux FAQ](#) chapter on package management provides an overview of package management on Debian systems :

<http://www.debian.org/doc/FAQ/ch-pkgtools.en.html>

The home page for the RPM project:

<http://www.rpm.org>

The home page for the YUM project at Duke University:

<http://linux.duke.edu/projects/yum/>

16 - Storage Media

In previous chapters we looked at manipulating data at the file level. In this chapter we will consider data at the device level. Linux has amazing capabilities for handling storage devices which can be physical storage, such as hard disks, or network storage, or various virtual storage devices like RAID (Redundant Array of Independent Disks) and LVM (Logical Volume Manager.)

However, since this is not a book about system administration, we will not try to cover this entire topic in depth. What we will try to do is introduce some of the concepts and key commands that are used to manage storage devices.

To carry out the exercises in this chapter we will use a USB flash drive, a CD-RW disk (if your systems is equipped with a CD-ROM burner) and a floppy disk (again, if your system is so equipped.)

We will look at the following commands:

- `mount` – Mount a file system
- `umount` – Unmount a file system
- `fsck` – Check and repair a file system
- `fdisk` – Partition table manipulator
- `mkfs` – Create a file system
- `fdformat` – Format a floppy disk
- `dd` – Write block oriented data directly to a device
- `genisoimage (mkisofs)` – Create an ISO 9660 image file
- `wodim (cdrrecord)` – Write data to optical storage media
- `md5sum` – Calculate an MD5 checksum

Mounting And Unmounting Storage Devices

Recent advances in the Linux desktop have made storage device management extremely

easy for desktop users. For the most part, you attach a device to your system and it “just works.” Back in the old days (say, 2004), this stuff had to be done manually. On non-desktop systems (i.e. servers) this is still a largely manual affair since servers often have extreme storage needs and complex configuration requirements.

The first step in managing a storage device is attaching the device to the file system tree. This process, called *mounting*, allows the device to participate with the operating system. As we recall from Chapter 3, Unix-like operating systems, like Linux, maintain a single file system tree with devices attached at various points. This contrasts other operating systems such as MS-DOS and Windows that maintain separate trees for each device (for example C:\, D:\, etc.)

There is a file named `/etc/fstab` that lists the devices (typically hard disk partitions) that are to be mounted at boot time. Here is an example `/etc/fstab` file from a Fedora 7 system:

LABEL=/12	/	ext3	defaults	1 1
LABEL=/home	/home	ext3	defaults	1 2
LABEL=/boot	/boot	ext3	defaults	1 2
tmpfs	/dev/shm	tmpfs	defaults	0 0
devpts	/dev/pts	devpts	gid=5,mode=620	0 0
sysfs	/sys	sysfs	defaults	0 0
proc	/proc	proc	defaults	0 0
LABEL=SWAP-sda3	swap	swap	defaults	0 0

Most of the file systems listed in this example file are virtual and are not applicable to our discussion. The interesting ones for our purposes are the first three:

LABEL=/12	/	ext3	defaults	1 1
LABEL=/home	/home	ext3	defaults	1 2
LABEL=/boot	/boot	ext3	defaults	1 2

These are the hard disk partitions. Each line of the file consists of six fields as follows:

Field	Contents	Description
1	Device	Traditionally, this field contains the actual name of a device file associated with the physical device such as <code>/dev/hda1</code> (the first partition of the master device on the first IDE channel) but with the today's computers which have lots of devices that are hot pluggable (like USB drives), many modern Linux distributions associate a device with a text label

		instead. This label (which is added to the storage media when it is formatted) is read by the operating system when the device is attached to the system. That way, no matter which device file is assigned to the actual physical device it can still be correctly identified.
2	Mount Point	The directory where the device is attached to the file system tree.
3	File System Type	Linux allows many file system types to be mounted. Most native Linux file systems are ext3 but many others are supported such as FAT16 (msdos), FAT32 (vfat), NTFS (ntfs), CD-ROM (iso9660), etc.
4	Options	File systems can be mounted with various options. It is possible, for example, to mount a files system as read-only, or prevent any programs from being executed from them (a useful security feature for removable media.)
5	Frequency	A single number that specifies if and when a file system is to be backed up with the dump command.
6	Order	A single number that specifies in what order file systems should be checked with the fsck command.

Viewing A List Of Mounted File Systems

The `mount` command is used to mount file systems. Entering the command without arguments will display a list of the file systems currently mounted:

```
[me@linuxbox ~]$ mount
/dev/sda2 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/sda5 on /home type ext3 (rw)
/dev/sda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
fusectl on /sys/fs/fuse/connections type fusectl (rw)
/dev/sdd1 on /media/disk type vfat (rw,nosuid,nodev,noatime,
```

```
uhelper=hal,uid=500,utf8,shortname=lower)
twin4:/musicbox on /misc/musicbox type nfs4 (rw,addr=192.168.1.4)
```

The format of the listing is: *device on mount point type file system type (options)*. For example, the first line shows that device `/dev/sda2` is mounted as the root file system and it is of type `ext3` and it is both readable and writable (the option “`rw`”.) This listing also has two interesting entries at the bottom of the list. The next to last entry shows a 2 gigabyte SD memory card in a card reader mounted at `/media/disk` and the last entry is a network drive mounted at `/misc/musicbox`.

For our first experiment, we will work with a CD-ROM. First, let's look at a system before a CD-ROM is inserted:

```
[me@linuxbox ~]$ mount
/dev/mapper/VolGroup00-LogVol100 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/hda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
```

This listing is from a CentOS 5 system which is using LVM (Logical Volume Manager) to create its root file system. Like many modern Linux distributions, this system will attempt to automatically mount the CD-ROM after insertion. After we insert the disk, we see the following:

```
[me@linuxbox ~]$ mount
/dev/mapper/VolGroup00-LogVol100 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/hda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
/dev/hdc on /media/live-1.0.10-8 type iso9660 (ro,noexec,nosuid,
nodev,uid=500)
```

At the end of the listing we see that the CD-ROM (which is device `/dev/hdc` on this system) has been mounted on `/media/live-1.0.10-8` and is type `iso9660` (a CD-

ROM.) For purposes of our experiment, we're interested in the name of the device. When you conduct this experiment yourself, the device name will most likely be different.

Warning: In the examples that follow, it is vitally important that you pay close attention to the actual device names in use on your system and **do not use the names used in the text!**

Now that we have the device name of the CD-ROM drive, let's unmount the disk and remount it another location in the file system tree. To do this, we become the superuser and unmount the disk with the `umount` (notice the spelling) command:

```
[me@linuxbox ~]$ su -  
Password:  
[root@linuxbox ~]# umount /dev/hdc
```

The next step is to create a new *mount point* for the disk. A mount point is simply a directory somewhere on the file system tree. Nothing special about it. It doesn't even have to be an empty directory, though if you mount a device on a non-empty directory, you will not be able to see the directory's previous contents until you unmount the device. For our purposes, we will create a new directory:

```
[root@linuxbox ~]# mkdir /mnt/cdrom
```

Finally, we mount the CD-ROM at the new mount point:

```
[root@linuxbox ~]# mount -t iso9660 /dev/hdc /mnt/cdrom
```

Afterwards, we can examine the contents of the CD-ROM via the new mount point:

```
[root@linuxbox ~]# cd /mnt/cdrom  
[root@linuxbox cdrom]# ls
```

Notice however, what happens when we try to unmount the CD-ROM:

```
[root@linuxbox cdrom]# umount /dev/hdc
```

```
umount: /mnt/cdrom: device is busy
```

Why is this? The reason is we cannot unmount a device if the device is being used by someone or some process. In this case, we changed our working directory to the mount point for the CD-ROM and that causes the device to be busy. We can easily remedy the issue by changing the working directory to something other than the mount point:

```
[root@linuxbox cdrom]# cd  
[root@linuxbox ~]# umount /dev/hdc
```

Now the device unmounts successfully.

Why Unmounting Is Important

If you look at the output of the `free` command which displays statistics about memory usage, you will see a statistic called “buffers.” Computer systems are designed to go as fast as possible. One of the impediments to system speed are slow devices. Printers are a good example. Even the fastest printer is extremely slow by computer standards. A computer would be very slow indeed if it had to stop and wait for a printer to finish printing a page. In the early days of PCs (before multi-tasking), this was a real problem. If you were working on a spreadsheet or text document, the computer would stop and become unavailable every time you printed. The computer would send the data to the printer as fast as the printer could accept it, but it was very slow since printers don't print very fast. This problem was solved by the advent of the *printer buffer*, a device containing some RAM memory that would sit between the computer and the printer. With the printer buffer in place, the computer would send the printer output to the buffer and it would be quickly stored in the fast RAM and the computer could go back to work without waiting. Meanwhile, the printer buffer would slowly *spool* the data to the printer from the buffer's memory at the speed that the printer could accept it.

This idea of buffering is used extensively in computers to make them faster. Don't let the need to occasionally read or write data to/from slow devices impede the speed of the system. Operating systems store data read from and to be written to storage devices in memory for as long as possible before actually having to interact with the slower device. On a Linux system for example, you will notice that the system seems to fill up memory as it is used. This does not mean the Linux is “using“ all the memory, it means that Linux is taking advantage of all the available memory to do as much buffering as it can.

This buffering causes writing to storage devices to be done very quickly, because the writing to the physical device is being deferred to a future time. In the meantime the data destined for the device is piling up in memory. From time to time the operating system will write this data to the physical device.

Unmounting a device entails writing all the remaining data to the device so that it can be safely removed. If the device is removed without unmounting it first, there exists the possibility that not all the data destined for the device has been transferred. In some cases this data may include vital directory updates and this will lead to *file system corruption*, one of the worst things that can happen on a computer.

Determining Device Names

It's sometimes difficult to determine the name of a device. Back in the old days, it wasn't very hard. A device was always in the same place and it didn't change. Unix-like systems like it that way. Back when Unix was developed, “changing a disk drive” involved using a forklift to remove a washing machine-sized device from the computer room. In recent years the typical desktop hardware configuration has become quite dynamic and Linux has evolved to become more flexible than its ancestors.

In the examples above we took advantage of the modern Linux desktop's ability to “automagically” mount the device and then determine the name after the fact. But what if you are managing a server or some other environment where this does not occur? How can we figure it out?

First, let's look at how the system names devices. If we list the contents of the `/dev` directory (where all devices live) we can see that there are lots and lots of devices:

```
[me@linuxbox ~]$ ls /dev
```

The contents of this listing reveal some patterns of device naming. Here are a few:

Pattern	Device
<code>/dev/fd*</code>	Floppy disk drives.
<code>/dev/hd*</code>	IDE (PATA) disks on older systems. Typical motherboards contain two IDE connectors, or <i>channels</i> each with a cable with two attachment points for drives. The first drive on the cable is called the <i>master</i> device and the second is called the <i>slave</i> device. The devices names are ordered such that <code>/dev/hda</code>

	referred to the master device on the first channel, <code>/dev/hdb</code> , the slave device on the first channel, <code>/dev/hdc</code> , the master device on the second channel, and so on. A trailing digit indicates the partition number on the device. For example, <code>/dev/hda1</code> refers to the first partition on the first hard drive on the system while <code>/dev/hda</code> refers to the entire drive.
<code>/dev/lp*</code>	Printers.
<code>/dev/sd*</code>	SCSI disks. On recent Linux systems, the kernel treats all disk-like devices (this includes PATA/SATA hard disks, flash drives, USB mass storage devices such as portable music players and digital cameras) as SCSI disks. The rest of the naming system is similar to the older <code>/dev/hd*</code> naming scheme described above.
<code>/dev/sr*</code>	Optical drives (CD/DVD readers and burners)

In addition, we often see symbolic links such as `/dev/cdrom`, `/dev/dvd` and `/dev/floppy` which point to the actual device files provided as a convenience.

If you are working on a system that does not automatically mount removable devices, you can use the following technique to determine how the removable device is named when it is attached. First, we start a real-time view of the `/var/log/messages` file (you may require superuser privileges for this):

```
[me@linuxbox ~]$ sudo tail -f /var/log/messages
```

The last few lines of the file will be displayed and then pause. Next, plug in the removable device. In this example we will use a 16 MB flash drive. Almost immediately, the kernel will notice the device and probe it:

```
Jul 23 10:07:53 linuxbox kernel: usb 3-2: new full speed USB device
using uhci_hcd and address 2
Jul 23 10:07:53 linuxbox kernel: usb 3-2: configuration #1 chosen
from 1 choice
Jul 23 10:07:53 linuxbox kernel: scsi3 : SCSI emulation for USB Mass
Storage devices
Jul 23 10:07:58 linuxbox kernel: scsi scan: INQUIRY result too short
(5), using 36
Jul 23 10:07:58 linuxbox kernel: scsi 3:0:0:0: Direct-Access      Easy
Disk                1.00 PQ: 0 ANSI: 2
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] 31263 512-byte
```



```
hardware sectors (16 MB)
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Write Protect is
off
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Assuming drive
cache: write through
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] 31263 512-byte
hardware sectors (16 MB)
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Write Protect is
off
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Assuming drive
cache: write through
Jul 23 10:07:59 linuxbox kernel: sdb: sdb1
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Attached SCSI
removable disk
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: Attached scsi generic
sg3 type 0
```

After the display pauses again, type Ctrl-c to get the prompt back. The interesting parts of the output are the repeated references to “[sdb]” which matches our expectation of a SCSI disk device name. Knowing this, two lines become particularly illuminating:

```
Jul 23 10:07:59 linuxbox kernel: sdb: sdb1
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Attached SCSI
removable disk
```

This tells us the device name is `/dev/sdb` for the entire device and `/dev/sdb1` for the first partition on the device. As you have no doubt noticed, working with Linux is full of interesting detective work!

Tip: Using the `tail -f /var/log/messages` technique is a great way to watch what the system is doing in near real-time.

With our device name in hand, we can now mount the flash drive:

```
[me@linuxbox ~]$ sudo mkdir /mnt/flash
[me@linuxbox ~]$ sudo mount /dev/sdb1 /mnt/flash
[me@linuxbox ~]$ df
Filesystem            1K-blocks      Used Available Use% Mounted on
/dev/sda2             15115452    5186944   9775164  35% /
/dev/sda5             59631908   31777376  24776480  57% /home
/dev/sda1             147764      17277    122858   13% /boot
tmpfs                 776808      0        776808   0% /dev/shm
/dev/sdb1             15560      0        15560   0% /mnt/flash
```

The device name will remain the same as long as it remains physically attached to the computer and the computer is not rebooted.

Creating New File Systems

Let's say that we wanted to reformat the flash drive with a Linux native file system rather than the FAT32 system it has now. This involves two steps: 1. (optional) create a new partition layout if the existing one is not to our liking, and 2. create a new, empty file system on the drive.

Warning! In the following exercise, we are going to format a flash drive. Use a drive that contains nothing you care about because it will be erased! Again, **make absolutely sure you are specifying the correct device name for your system, not the one shown in the text. Failure to heed this warning could result in you formatting the wrong drive!**

Manipulating Partitions With `fdisk`

The `fdisk` program allows us to directly interact with disk-like devices (such as hard disk drives and flash drives) at a very low level. With this tool we can edit, delete, and create partitions on the device. To work with our flash drive, we must first unmount it (if needed) and then invoke the `fdisk` program as follows:

```
[me@linuxbox ~]$ sudo umount /dev/sdb1
[me@linuxbox ~]$ sudo fdisk /dev/sdb
```

Notice that we must specify the device in terms of the entire device, not by partition number. After the program starts up, we will see the following prompt:

```
Command (m for help):
```

Entering an “m” will display the program menu:

```
Command action
  a  toggle a bootable flag
  b  edit bsd disklabel
  c  toggle the dos compatibility flag
  d  delete a partition
```

```
l  list known partition types
m  print this menu
n  add a new partition
o  create a new empty DOS partition table
p  print the partition table
q  quit without saving changes
s  create a new empty Sun disklabel
t  change a partition's system id
u  change display/entry units
v  verify the partition table
w  write table to disk and exit
x  extra functionality (experts only)
```

Command (m for help):

The first thing we will want to do is examine the existing partition layout. We do this by entering “p” to print the partition table for the device:

Command (m for help): **p**

```
Disk /dev/sdb: 16 MB, 16006656 bytes
1 heads, 31 sectors/track, 1008 cylinders
Units = cylinders of 31 * 512 = 15872 bytes
```

Device	Boot	Start	End	Blocks	Id	System
/dev/sdb1		2	1008	15608+	b	W95 FAT32

In this example, we see a 16 MB device with a single partition (1) that uses 1006 of the available 1008 cylinders on the device. The partition is identified as Windows 95 FAT32 partition. Some programs will use this identifier to limit the kinds of operation that can be done to the disk, but most of the time it is not critical to change it. However in the interests of demonstration, we will change it to indicate a Linux partition. To do this, we must first find out what ID is used to identify a Linux partition. In the listing above we see that the ID “b” is used to specify the existing partition. To see a list of the available partition types, we refer back to the program menu. There we can see the following choice:

```
l  list known partition types
```

We enter “l” at the prompt a large list of possible types is displayed. Among them we see “b” for our existing partition type and “83” for Linux.

Going back to the menu, we see this choice to change a partition ID:

```
t  change a partition's system id
```

We enter “t” at the prompt enter the new ID:

```
Command (m for help): t
Selected partition 1
Hex code (type L to list codes): 83
Changed system type of partition 1 to 83 (Linux)
```

This completes all the changes that we need to make. Up to this point the device has been untouched (all the changes have been stored in memory, not on the physical device) so we will write the modified partition table to the device and exit. To do this, we enter “w” at the prompt:

```
Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.

WARNING: If you have created or modified any DOS 6.x
partitions, please see the fdisk manual page for additional
information.
Syncing disks.
[me@linuxbox ~]$
```

If we had decided to leave the device unaltered, we could have entered “q” at the prompt which would have exited the program without writing the changes. We can safely ignore the ominous sounding warning message.

Creating A New File System With mkfs

With our partition editing done (lightweight though it might have been) it’s time to create a new file system on our flash drive. To do this, we will use `mkfs` (short for “make file system”) which can create file systems in a variety of formats. To create an `ext3` file system on the device, we use the “-t” option to specify the “ext3” system type followed by the name of device containing the partition we wish to format:

```
[me@linuxbox ~]$ sudo mkfs -t ext3 /dev/sdb1
mke2fs 1.40.2 (12-Jul-2007)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
3904 inodes, 15608 blocks
780 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=15990784
2 block groups
8192 blocks per group, 8192 fragments per group
1952 inodes per group
Superblock backups stored on blocks:
    8193

Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 34 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
[me@linuxbox ~]$
```

The program will display a lot of information when ext3 is the chosen file system type. To re-format the device to its original FAT32 file system, specify “vfat” as the file system type:

```
[me@linuxbox ~]$ sudo mkfs -t vfat /dev/sdb1
```

This process of partitioning and formatting can be used anytime additional storage devices are added to the system. While we worked with a tiny flash drive, the same process can be applied to internal hard disks and other removable storage devices such as USB hard drives.

Testing And Repairing File Systems

In our earlier discussion of the `/etc/fstab` file, we saw some mysterious digits at the end of each line. Each time the system boots, it routinely checks the integrity of the file systems before mounting them. This is done by the `fscck` program (short for “file system check”) The last number in each `fstab` entry specifies the order the devices are to be checked. In our example above, we see that the root file system is checked first followed by the `home` and `boot` file systems. Devices with a zero as the last digit are not routinely checked.

In addition to checking the integrity of file systems, `fsck` can also repair corrupt file systems with varying degrees of success depending on the amount of damage. On Unix-like file systems, recovered portions of files are placed in the `lost+found` directory located in the root of each file system.

To check our flash drive (which should be unmounted first), we could do the following:

```
[me@linuxbox ~]$ sudo fsck /dev/sdb1
fsck 1.40.8 (13-Mar-2008)
e2fsck 1.40.8 (13-Mar-2008)
/dev/sdb1: clean, 11/3904 files, 1661/15608 blocks
```

In my experience, file system corruption is quite rare unless there is a hardware problem, such as a failing disk drive. On most systems, file system corruption detected at boot time will cause the system to stop and direct you to run `fsck` before continuing.

What The fsck?

In Unix culture, the word “fsck” is often used in place of a popular word with which it shares three letters. This is especially appropriate given that you will probably be uttering the aforementioned word if you find yourself in a situation where you are forced to run `fsck`.

Formatting Floppy Disks

For those of us still using computers old enough to be equipped with floppy diskette drives, we can manage those devices too. Preparing a blank floppy for use is a two step process. First we perform a low-format on the diskette, then create a file system. To accomplish the formatting, we use the `fdformat` program specifying the name of the floppy device (usually `/dev/fd0`):

```
[me@linuxbox ~]$ sudo fdformat /dev/fd0
Double-sided, 80 tracks, 18 sec/track. Total capacity 1440 kB.
Formatting ... done
Verifying ... done
```

Next, we apply a FAT file system to the diskette with `mkfs`:

```
[me@linuxbox ~]$ sudo mkfs -t msdos /dev/fd0
```

Notice that we use the “msdos” file system type to get the older (and smaller) style file allocation tables. After a diskette is prepared, it may be mounted like other devices.

Moving Data Directly To/From Devices

While we usually think of data on our computers always being organized into files, it is also possible to think of the data in “raw” form. If we look at a disk drive for example, we will see that it consists of a large number of “blocks” of data that the operating system sees as directories and files. However, if we could treat a disk drive as simply a large collection of data blocks, we could perform useful tasks such as cloning devices.

The `dd` program performs this task. It copies blocks of data from one place to another. It uses a unique syntax (for historical reasons) and is usually used this way:

```
dd if=input_file of=output_file [bs=block_size [count=blocks]]
```

Let’s say we had two USB flash drives of the same size and we wanted to exactly copy the first drive to the second. If we attached both drives to the computer and they are assigned to devices `/dev/sdb` and `/dev/sdc` respectively, we could copy everything on the first drive to the second drive with the following:

```
dd if=/dev/sdb of=/dev/sdc
```

Alternately, if only the first device were attached to the computer, we could copy its contents to an ordinary file for later restoration or copying:

```
dd if=/dev/sdb of=flash_drive.img
```

Warning! The `dd` command is very powerful. Though its name derives from “data definition,” it is sometimes called “destroy disk” because users often mistype either the `if` or `of` specifications. **Always double check your input and output specifications before pressing enter!**

Creating CD-ROM Images

Writing a recordable CD-ROM (either a CD-R or CD-RW) consists of two steps, first constructing an *iso image file* that is the exact file system image of the CD-ROM and second, writing the image file onto the CD-ROM media.

Creating An Image Copy Of A CD-ROM

If we want to make an iso image of an existing CD-ROM, we can use `dd` to read all the data blocks off the CD-ROM and copy them to a local file. Say we had an Ubuntu CD and we wanted to make an iso file that we could later use to make more copies. After inserting the CD and determining its device name (we'll assume `/dev/cdrom`) we can make the iso file like so:

```
dd if=/dev/cdrom of=ubuntu.iso
```

This technique works for data DVDs as well, but will not work for music CDs as they do not use a file system for storage. For music CDs, look at the `cdrecord` command.

Creating An Image From A Collection Of Files

To create an iso image file containing the contents of a directory, we use the `genisoimage` program. To do this, we first create a directory containing all the files we wish to include in the image and then execute the `genisoimage` command to create the image file. For example, if we had created a directory called `~/cd-rom-files` and filled it with files for our CD-ROM, we could create an image file named `cd-rom.iso` with the following command:

```
genisoimage -o cd-rom.iso -R -J ~/cd-rom-files
```

The “-R” option adds meta-data for the *Rock Ridge extensions*, which allows the use of long file names and POSIX style file permissions. Likewise, the “-J” option enables the *Joliet extensions* which permit long file names for Windows.

A Program By Any Other Name...

If you look at on-line tutorials for creating and burning optical media such as CD-ROMs and DVDs you will frequently encounter two programs called `mkisofs` and `cdrecord`. These programs were part of a popular package called “`cdrtools`” authored by Jörg Schilling. In the summer of 2006, Mr. Schilling made a license change to a portion of the `cdrtools` package which, in the opinion of many in the Linux community, created a license incompatibility with the GNU GPL. As a result, a *fork* of the `cdrtools` project was started that now includes replacement programs for `cdrecord` and `mkisofs` named `wodim` and `genisoimage` respectively.

Writing CD-ROM Images

After we have an image file, we can burn it onto our optical media. Most of the commands we will discuss below can be applied to both recordable CD-ROM and DVD media.

Mounting An ISO Image Directly

There is a trick that we can use to mount an iso image while it is still on our hard disk and treat it as though it was already on optical media. By adding the “`-o loop`” option to `mount` (along with the required “`-t iso9660`” file system type) we can mount the image file as though it were a device and attach it to the file system tree:

```
mkdir /mnt/iso_image
mount -t iso9660 -o loop image.iso /mnt/iso_image
```

In the example above, we created a mount point named `/mnt/iso_image` and then mounted the image file `image.iso` at that mount point. After the image is mounted, it can be treated just as though it was a real CD-ROM or DVD. *Remember to unmount the image when it is no longer needed.*

Blanking A Re-Writable CD-ROM

Rewritable CD-RW media needs to be erased or *blanked* before it can be reused. To do this, we can use `wodim` specifying the device name for the CD writer and the type of blanking to be performed. The `wodim` program offers several types. The most minimal (and fastest) is the “`fast`” type:

```
wodim dev=/dev/cdrw blank=fast
```

Writing An Image

To write an image, we again use `wodim` specifying the name of the optical media writer device and the name of the image file:

```
wodim dev=/dev/cdrw image.iso
```

In addition to the device name and image file, `wodim` supports a very large set of options. Two common ones are “-v” for verbose output, and “-dao” which writes the disk in *disk-at-once* mode. This option should be used if you are preparing a disk for commercial reproduction. The default mode for `wodim` is *track-at-once* which is useful for recording music tracks.

Further Reading

We have just touched on the many ways that the command line can be used to manage storage media. Take a look at the man pages of the commands we have covered. Some of them support huge numbers of options and operations. Also, look on-line for tutorials for adding hard drives to your Linux system (there are many) as well as working with optical media.

Extra Credit

It’s often useful to verify the integrity of an iso image that we have downloaded. In most cases a distributor of an iso image will also supply a *checksum file*. A checksum is the result of an exotic mathematical calculation that will result in a number that represents the content of the target file. If the contents of the file changes by even one bit, the resulting checksum will be much different. The most common method of checksum generation uses the `md5sum` program. When you use `md5sum` it produces a unique hexadecimal number:

```
md5sum image.iso  
34e354760f9bb7fbf85c96f6a3f94ece image.iso
```

After you download an image you should run `md5sum` against it and compare the results with the `md5sum` value supplied by the publisher.

In addition to checking the integrity of a downloaded file, we can use `md5sum` to verify newly written optical media. To do this we first calculate the checksum of the image file and then calculate a checksum for the media. The trick to verifying the media is to limit the calculation only the portion of the optical media that contains the image. We do this by determining the number of 2048 byte blocks the image contains (optical media is always written in 2048 byte blocks) and reading that many blocks from the media. On some types of media, this is not required. A CD-R written in disk-at-once mode can be checked this way:

```
md5sum /dev/cdrom
34e354760f9bb7fbf85c96f6a3f94ece /dev/cdrom
```

However many types of media, such as DVDs require a precise calculation of the number of blocks. In the example below, we check the integrity of the image file `dvd-image.iso` and the disk in the DVD reader `/dev/dvd`. Can you figure out how this works?

```
md5sum dvd-image.iso; dd if=/dev/dvd bs=2048 count=$(( $(stat -c "%s"
dvd-image.iso) / 2048 )) | md5sum
```

17 – Networking

When it comes to networking, there is probably nothing that cannot be done with Linux. Linux is used to build all sorts of networking systems and appliances, including firewalls, routers, name servers, NAS (Network Attached Storage) boxes and the list goes on and on.

Just as the subject of networking is vast, so are the number of commands that can be used to configure and control it. We will focus our attention on just a few of the most frequently used ones. The commands chosen for examination include those used to monitor networks and those used to transfer files. In addition, we are going to explore the SSH program that is used to perform remote logins. This chapter will cover:

- `ping` - Send an ICMP ECHO_REQUEST to network hosts
- `tracert` - Print the route packets trace to a network host
- `netstat` - Print network connections, routing tables, interface statistics, masquerade connections, and multicast memberships
- `whois` - Client for the whois directory service
- `arp` - Manipulate the system ARP cache
- `ftp` - Internet file transfer program
- `wget` - Non-interactive network downloader
- `ssh` - OpenSSH SSH client (remote login program)

We're going to assume a little background in networking. In this, the Internet age, everyone using a computer needs a basic understanding of networking concepts. To make full use of this chapter you should familiar with the following terms:

- IP (Internet Protocol) Address
- Host and Domain Name
- URI (Uniform Resource Identifier)

Please see the “Additional Reading” section below for some useful articles regarding these terms.

Note: Some of the commands we will cover may (depending on your distribution) require installation from your distribution’s package repositories and some may require superuser privileges to execute.

Examining And Monitoring A Network

Even if we’re not the system administrator, it’s often helpful to examine the performance and operation of the network.

ping

The most basic network command is `ping`. The `ping` command sends a special network packet called an ICMP ECHO_REQUEST to a specified host. Most network devices receiving this packet will reply to it allowing the network connection to be verified.

Note: It is possible to configure most network devices (including Linux hosts) to ignore these packets. This is usually done for security reasons to partially obscure a host from a potential attacker. It is also common for firewalls to be configured to block ICMP traffic.

For example, to see if we can reach `linuxcommand.org` (one of my favorite sites ;-), we can use `ping` like this:

```
[me@linuxbox ~]$ ping linuxcommand.org
```

Once started, `ping` continues to send packets at a specified interval (default is one second) until it is interrupted:

```
[me@linuxbox ~]$ ping linuxcommand.org
PING linuxcommand.org (66.35.250.210) 56(84) bytes of data.
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=1
ttl=43 time=107 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=2
ttl=43 time=108 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=3
ttl=43 time=106 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=4
ttl=43 time=106 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=5
ttl=43 time=105 ms
```

```
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=6
ttl=43 time=107 ms

--- linuxcommand.org ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 6010ms
rtt min/avg/max/mdev = 105.647/107.052/108.118/0.824 ms
```

After it is interrupted (in this case after the sixth packet by pressing Ctrl-c), `ping` prints performance statistics. A properly performing network will exhibit zero percent packet loss. A successful “ping” will indicate that the elements of the network, that is, its interface cards, cabling, routing and gateways, are in generally good working order.

traceroute

The `traceroute` program displays a listing of all the “hops” network traffic takes to get from the local system to a specified host. For example, to see the route taken to reach `slashdot.org`, we would do this:

```
[me@linuxbox ~]$ traceroute slashdot.org
```

The output looks like this:

```
traceroute to slashdot.org (216.34.181.45), 30 hops max, 40 byte
packets
 1 ipcop.localdomain (192.168.1.1)  1.066 ms  1.366 ms  1.720 ms
 2 * * *
 3 ge-4-13-ur01.rockville.md.bad.comcast.net (68.87.130.9)  14.622
ms 14.885 ms 15.169 ms
 4 po-30-ur02.rockville.md.bad.comcast.net (68.87.129.154) 17.634
ms 17.626 ms 17.899 ms
 5 po-60-ur03.rockville.md.bad.comcast.net (68.87.129.158) 15.992
ms 15.983 ms 16.256 ms
 6 po-30-ar01.howardcounty.md.bad.comcast.net (68.87.136.5) 22.835
ms 14.233 ms 14.405 ms
 7 po-10-ar02.whitemarsh.md.bad.comcast.net (68.87.129.34) 16.154
ms 13.600 ms 18.867 ms
 8 te-0-3-0-1-cr01.philadelphia.pa.ibone.comcast.net (68.86.90.77)
21.951 ms 21.073 ms 21.557 ms
 9 pos-0-8-0-0-cr01.newyork.ny.ibone.comcast.net (68.86.85.10)
22.917 ms 21.884 ms 22.126 ms
10 204.70.144.1 (204.70.144.1) 43.110 ms 21.248 ms 21.264 ms
11 cr1-pos-0-7-3-1.newyork.savvis.net (204.70.195.93) 21.857 ms
cr2-pos-0-0-3-1.newyork.savvis.net (204.70.204.238) 19.556 ms cr1-
```

```

pos-0-7-3-1.newyork.savvis.net (204.70.195.93) 19.634 ms
12 cr2-pos-0-7-3-0.chicago.savvis.net (204.70.192.109) 41.586 ms
42.843 ms cr2-tengig-0-0-2-0.chicago.savvis.net (204.70.196.242)
43.115 ms
13 hr2-tengigabitethernet-12-1.elkgrovech3.savvis.net
(204.70.195.122) 44.215 ms 41.833 ms 45.658 ms
14 csr1-ve241.elkgrovech3.savvis.net (216.64.194.42) 46.840 ms
43.372 ms 47.041 ms
15 64.27.160.194 (64.27.160.194) 56.137 ms 55.887 ms 52.810 ms
16 slashdot.org (216.34.181.45) 42.727 ms 42.016 ms 41.437 ms

```

In the output we can see that connecting from our test system to `slashdot.org` requires traversing sixteen routers. For routers that provided identifying information we see their host names, IP addresses and performance data which includes three samples of round-trip time from the local system to the router. For routers that do not (because of router configuration, network congestion, firewalls, etc.), we see asterisks as in the line for hop number two.

netstat

The `netstat` program is used to examine various network settings and statistics. Through the use of its many options, we can look at a variety of features of our network setup. Using the “-ie” option we can examine the network interfaces in our system:

```

[me@linuxbox ~]$ netstat -ie
eth0    Link encap:Ethernet  HWaddr 00:1d:09:9b:99:67
        inet addr:192.168.1.2 Bcast:192.168.1.255 Mask:255.255.255.0
        inet6 addr: fe80::21d:9ff:fe9b:9967/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:238488 errors:0 dropped:0 overruns:0 frame:0
        TX packets:403217 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:100
        RX bytes:153098921 (146.0 MB)  TX bytes:261035246 (248.9 MB)
        Memory:fdfc0000-fdfe0000

lo      Link encap:Local Loopback
        inet addr:127.0.0.1  Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING  MTU:16436  Metric:1
        RX packets:2208 errors:0 dropped:0 overruns:0 frame:0
        TX packets:2208 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:111490 (108.8 KB)  TX bytes:111490 (108.8 KB)

```

In the example above, we see that our test system has two network interfaces. The first,

called `eth0` is the Ethernet interface and the second, called `lo`, is the *loopback* interface, a virtual interface that the system uses to “talk to itself.”

When performing causal network diagnostics, the important things to look for here are the presence of the word “UP” at the beginning of the fourth line for each interface indicating that the network interface is enabled, and the presence of a valid IP address in the `inet addr` field on the second line. For systems using DHCP (Dynamic Host Configuration Protocol), a valid IP address in this field will verify that the DHCP is working.

Using the “-r” option will display the kernel’s network routing table. This shows how the network is configured to send packets from network to network:

```
[me@linuxbox ~]$ netstat -r
Kernel IP routing table
Destination Gateway Genmask Flags MSS Window irtt Iface
192.168.1.0 * 255.255.255.0 U 0 0 0 eth0
default 192.168.1.1 0.0.0.0 UG 0 0 0 eth0
```

In this simple example, we see a typical routing table for a client machine on a LAN (Local Area Network) behind a firewall/router. The first line of the listing shows the destination `192.168.1.0`. IP addresses that end in zero refer to networks rather than individual hosts, so this destination means any host on the LAN. The next field, `Gateway` is the name or IP address of the gateway (router) used to go from the current host to the destination network. An asterisk in this field indicates that no gateway is needed.

The last line contains the destination `default`. This means that any traffic destined for a network not otherwise listed in the table. In our example, we see that the gateway is defined as a router with the address of `192.168.1.1` which presumably knows what to do with the destination traffic.

The `netstat` program has many options and we have only looked at a couple. Check out the `netstat` man page for a complete list.

Transporting Files Over A Network

What good is a network unless we know how to move files across it? There are many programs that move data over networks. We will cover two of them now and several more in later sections.

ftp

One of the true “classic” programs is `ftp`. `ftp` gets its name from the protocol it uses, the *File Transfer Protocol*. FTP is used widely on the Internet for file downloads. Most, if not all, web browsers support it and you often see URIs starting with the protocol `ftp://`.

But before there were web browsers, there was the `ftp` program. `ftp` is used to communicate with *FTP servers*, machines that contain files that can be uploaded and downloaded over a network.

The FTP (in its original form) is not secure because it sends account names and passwords in *cleartext*. This means that they are not encrypted and anyone *sniffing* the network can see them. Because of this, almost all FTP done over the Internet is done by *anonymous FTP servers*. An anonymous server allows anyone to login using the login name “anonymous” and a meaningless password.

In the example below, we show a typical session with the `ftp` program downloading an iso image of Ubuntu located in the `/pub/cd_images/Ubuntu-8.04` directory of the anonymous FTP server `fileserver`:

```
[me@linuxbox ~]$ ftp fileserver
Connected to fileserver.localdomain.
220 (vsFTPd 2.0.1)
Name (fileserver:me): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd pub/cd_images/Ubuntu-8.04
250 Directory successfully changed.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
-rw-rw-r-- 1 500 500 733079552 Apr 25 03:53 ubuntu-8.04-
desktop-i386.iso
226 Directory send OK.
ftp> lcd Desktop
Local directory now /home/me/Desktop
ftp> get ubuntu-8.04-desktop-i386.iso
local: ubuntu-8.04-desktop-i386.iso remote: ubuntu-8.04-desktop-
i386.iso
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for ubuntu-8.04-desktop-
i386.iso (733079552 bytes).
226 File send OK.
```

```
733079552 bytes received in 68.56 secs (10441.5 kB/s)
ftp> bye
```

Here is an explanation of the commands entered during this session:

Command	Meaning
<code>ftp fileserver</code>	Invoke the <code>ftp</code> program and have it connect to the FTP server <code>fileserver</code> .
<code>anonymous</code>	Login name. After the login prompt, a password prompt will appear. Some servers will accept a blank password, other will require a password in the form of a email address. In that case, try “ user@example.com ” or some such.
<code>cd pub/cd_imges/Ubuntu-8.04</code>	Change to the directory on the remote system containing the desired file. Note that on most anonymous FTP servers the files for public downloading are found somewhere under the <code>pub</code> directory.
<code>ls</code>	List the directory on the remote system.
<code>lcd Desktop</code>	Change the directory on the local system to <code>~/Desktop</code> . In the example, the <code>ftp</code> program was invoked when the working directory was <code>~</code> . This command changes the working directory to <code>~/Desktop</code> .
<code>get ubuntu-8.04-desktop-i386.iso</code>	Tell the remote system to transfer the file <code>ubuntu-8.04-desktop-i386.iso</code> to the local system. Since the working directory on the local system was changed to <code>~/Desktop</code> , the file will be downloaded there.
<code>bye</code>	Log off the remote server and end the <code>ftp</code> program session. The commands <code>quit</code> and <code>exit</code> may also be used.

Typing “`help`” at the “`ftp>`” prompt will display a list of the supported commands. Using

ftp on a server where sufficient permissions have been granted, it is possible to perform many ordinary file management tasks. It's clumsy but it does work.

lftp – A Better ftp

ftp is not the only command line FTP client. In fact, there are many. One of better (and more popular) ones is lftp by Alexander Lukyanov. It works much like the traditional ftp program but has many additional convenience features including multiple protocol support (including HTTP), automatic re-try on failed downloads, background processes, tab completion of path names, and many more.

wget

Another popular command line program for file downloading is wget. It is useful for downloading content from both web and FTP sites. Single files, multiple files, and even entire sites can be downloaded. To download the first page of linuxcommand.org we could do this:

```
[me@linuxbox ~]$ wget http://linuxcommand.org/index.php
--11:02:51-- http://linuxcommand.org/index.php
           => `index.php'
Resolving linuxcommand.org... 66.35.250.210
Connecting to linuxcommand.org|66.35.250.210|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]

      [ <=>                               ] 3,120      ---K/s

11:02:51 (161.75 MB/s) - `index.php' saved [3120]
```

The program's many options allow wget to recursively download, download files in the background (allowing you to log off but have the downloads continue), and continue downloading a partially downloaded file. These many features are well documented in its better-than-average man page.

Secure Communication With Remote Hosts

For many years, Unix-like operating systems have had the ability to be remotely administered via a network. In the early days, before the general adoption of the Internet, there were a couple of popular programs that were used to log in to remote hosts. These were the rlogin and telnet programs. These programs however suffer from the same fatal flaw that the ftp program does, they transmit all their communications

(including log in names and passwords) in cleartext. This makes them wholly inappropriate for use in the Internet age.

ssh

To address this problem, a new protocol called SSH (Secure Shell) was developed. SSH solves the two basic problems of securely communicating with a remote host. First, it authenticates that the remote host is who it says it is (thus preventing so-called “man in the middle” attacks), and second, it encrypts all of the communications between the local and remote hosts.

SSH consists of two parts. An SSH server runs on the remote host listening for incoming connections on port twenty-two while an SSH client is used on the local system to communicate with the remote server.

Most Linux distributions ship an implementation of SSH called OpenSSH from the BSD project. Some distributions include both the client and the server packages by default (for example, Red Hat family distributions) while others (such as Ubuntu) only supply the client. To enable a system to receive remote connections, it must have the `openssh-server` package installed, configured and running, and (if the system is either running or is behind a firewall) it must allow incoming network connections on TCP port 22.

Tip: If you don’t have a remote system to connect to but want to try these examples, make sure the `OpenSSH-server` package is installed on your system and use `localhost` as the name of the remote host. That way, your machine will create network connections with itself.

The SSH client program used to connect to remote SSH servers is called, appropriately enough, `ssh`. To connect to a remote host named `remote-sys`, we would use the `ssh` client program like so:

```
[me@linuxbox ~]$ ssh remote-sys
The authenticity of host 'remote-sys (192.168.1.4)' can't be
established.
RSA key fingerprint is
41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.
Are you sure you want to continue connecting (yes/no)?
```

The first time the connection is attempted, a message is displayed indicating that the authenticity of the remote host cannot be established. This is because the client program has never seen this remote host before. To accept the credentials of the remote host, enter “yes” when prompted. Once the connection is established the user is prompted for

his/her password:

```
Warning: Permanently added 'remote-sys,192.168.1.4' (RSA) to the list
of known hosts.
me@remote-sys's password:
```

After the password is successfully entered, we receive the shell prompt from the remote system:

```
Last login: Sat Aug 30 13:00:48 2008
[me@remote-sys ~]$
```

The remote shell session continues until the user enters the `exit` command at the remote shell prompt thereby closing the remote connection. At this point, the local shell session resumes and the local shell prompt reappears.

It is also possible to connect to remote systems using a different user name. For example, if the local user “me” had an account named “bob” on a remote system, user me could log in to the account bob on the remote system as follows:

```
[me@linuxbox ~]$ ssh bob@remote-sys
bob@remote-sys's password:
Last login: Sat Aug 30 13:03:21 2008
[bob@remote-sys ~]$
```

As stated before, SSH verifies the authenticity of the remote host. If remote host does not successfully authenticate, the following message appears:

```
[me@linuxbox ~]$ ssh remote-sys
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!    @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle
attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.
Please contact your system administrator.
Add correct host key in /home/me/.ssh/known_hosts to get rid of this
```

```
message.  
Offending key in /home/me/.ssh/known_hosts:1  
RSA host key for remote-sys has changed and you have requested strict  
checking.  
Host key verification failed.
```

This message is caused by one of two possible situations. First, an attacker may be attempting a “man-in-the-middle” attack. This is rare since, everybody knows that SSH alerts the user to this. The more likely culprit is that the remote system has been changed somehow, for example, it has had its operating system or SSH server reinstalled. In the interests of security and safety however, the first possibility should not be dismissed out of hand. Always check with the administrator of the remote system when this message occurs.

After it has been determined that the message is due to a benign cause, it is safe to correct the problem on the client side. This is done by using a text editor (vim perhaps?) to remove the obsolete key from the `~/.ssh/known_hosts` file. In the example message above we see this:

```
Offending key in /home/me/.ssh/known_hosts:1
```

This means that line one of the `known_hosts` file contains the offending key. Delete this line from the file and then the SSH program will be able to accept new authentication credentials from the remote system.

Besides opening a shell session on a remote system, `ssh` also allows one to execute a single command on a remote system. For example to execute the `free` command on a remote host named `remote-sys` and having the results displayed on the local system:

```
[me@linuxbox ~]$ ssh remote-sys free  
me@twin4's password:  
      total        used        free      shared  buffers    cached  
Mem:   775536     507184     268352          0     110068     154596  
-/+ buffers/cache: 242520 533016  
Swap:  1572856          0     1572856  
[me@linuxbox ~]$
```

It's possible to use this technique in more interesting ways such as this example where we perform an `ls` on the remote system and redirect the output to a file on the local system:

```
[me@linuxbox ~]$ ssh remote-sys 'ls *' > dirlist.txt
me@twin4's password:
[me@linuxbox ~]$
```

Notice the use of the single quotes in the command above. This is done because we do not want the pathname expansion performed on the local machine rather we want it to be performed on the remote system. Likewise, if we had wanted the output redirected to a file on the remote machine, we could have placed the redirection operator and the file name within the single quotes:

```
[me@linuxbox ~]$ ssh remote-sys 'ls * > dirlist.txt'
```

Tunneling With SSH

Part of what happens when you establish a connection with a remote host via SSH, is that an *encrypted tunnel* is created between the local and remote systems. Normally, this tunnel is used to allow commands typed at the local system to be transmitted safely to the remote system and the results to be transmitted safely back. In addition to this basic function, the SSH protocol allows most types of network traffic to be sent through the encrypted tunnel creating a sort of VPN (Virtual Private Network) between the local and remote systems.

Perhaps the most common use of this feature is to allow X Window system traffic to be transmitted. On a system running an X server (that is, a machine that is displaying a GUI), it is possible to launch and run a X client program (a graphical application) on a remote system and have its display appear on the local system. It's easy to do, here's an example: let's say we were sitting at a Linux system called `linuxbox` which is running an X server, and we wanted to run the `xload` program on a remote system named `remote-sys` and see the program's graphical output on our local system. We could do this:

```
[me@linuxbox ~]$ ssh -X remote-sys
me@remote-sys's password:
Last login: Mon Sep 08 13:23:11 2008
[me@remote-sys ~]$ xload
```

After the `xload` command is executed on the remote system, its window appears on the local system. On some systems you may need to use the “-Y” option rather than the “-X” option to do this.

scp And sftp

The OpenSSH package also includes two programs that can make use of an SSH encrypted tunnel to copy files across the network. The first, `scp` (secure copy) is used much like the familiar `cp` program to copy files. The most notable difference is that the source or destination pathnames may be preceded with the name of a remote host followed by a colon character. For example, if we wanted to copy a document named `document.txt` from our home directory on the remote system `remote-sys` to the current working directory on our local system, we could do this:

```
[me@linuxbox ~]$ scp remote-sys:document.txt .
me@remote-sys's password:
document.txt                100% 5581      5.5KB/s   00:00
[me@linuxbox ~]$
```

As with SSH, you may apply a user name to the beginning of the remote host's name if the desired remote host account name does not match that of the local system:

```
[me@linuxbox ~]$ scp bob@remote-sys:document.txt .
```

The second SSH file copying program is `sftp` which, as its name implies, is a secure replacement for the `ftp` program. `sftp` works much like the original `ftp` program that we used earlier, however instead of transmitting everything in cleartext, it uses an SSH encrypted tunnel. `sftp` has an important advantage over conventional `ftp` in that it does not require an FTP server to be running on the remote host. It only requires the SSH server. This means that any remote machine that can connect with the SSH client can also be used as a FTP-like server. Here is a sample session:

```
[me@linuxbox ~]$ sftp remote-sys
Connecting to remote-sys...
me@remote-sys's password:
sftp> ls
ubuntu-8.04-desktop-i386.iso
sftp> lcd Desktop
sftp> get ubuntu-8.04-desktop-i386.iso
Fetching /home/me/ubuntu-8.04-desktop-i386.iso to ubuntu-8.04-
desktop-i386.iso
/home/me/ubuntu-8.04-desktop-i386.iso 100% 699MB 7.4MB/s 01:35
sftp> bye
```


The SFTP protocol is supported by many of the graphical file managers found in Linux distributions. Using either Nautilus (GNOME) or Konqueror (KDE), we can enter a URI that begins with `sftp://` into the location bar and operate on files stored on a remote system running an SSH server.

An SSH Client For Windows?

Let's say you are sitting at a Windows machine but you need to log in to your Linux server and get some real work done, what do you do? Get an SSH client program for your Windows box of course! There are a number of these. The most popular one is probably PuTTY by Simon Tatham and his team. The PuTTY program displays a terminal window and allow a Windows user to open an SSH (or telnet) session on a remote host. The program also provides analogs for the scp and sftp programs.

PuTTY is available at <http://www.chiark.greenend.org.uk/~sgtatham/putty/>

Further Reading

We've only touched on a few of the most common command line networking tools available for Linux. For a broader (albeit dated) look at network administration, the Linux Documentation Project provides the Linux Network Administrators Guide available at:

<http://tldp.org/LDP/nag2/index.html>

The Wikipedia contains many good networking articles. Here are some of the basics:

http://en.wikipedia.org/wiki/Internet_protocol_address

http://en.wikipedia.org/wiki/Host_name

http://en.wikipedia.org/wiki/Uniform_Resource_Identifier

18 - Searching For Files

As we have wandered around our Linux system, one thing has become abundantly clear: a typical Linux system has a lot of files! This begs the question, “how do you find things?” We already know that the Linux file system is well organized according to conventions that have been passed down from one generation of Unix-like system to the next, but still, the sheer number of files presents a sometimes daunting problem.

In this chapter we will look at two tools that are used to find files on a system. These tools are:

- `locate` – Find files by name
- `find` – Search for files in a directory hierarchy

We will also look at a command that is often used with the file search commands to process the resulting list of files:

- `xargs` – Build and execute command lines from standard input

In addition, we will introduce a couple of commands to assist us in or exploration:

- `touch` – Change file times
- `stat` – Display file or file system status

locate – Find Files The Easy Way

The `locate` program performs a rapid search of a database of pathnames and outputs every name that matches a given substring. Say, for example, we want to find all the programs with names that begin with the “zip.” Since we are looking for programs, we can assume that the directory containing the programs would end with “bin/”. Therefore we could try to use `locate` this way to find our files:

```
[me@linuxbox ~]$ locate bin/zip
```

`locate` will search its database of pathnames and output any that contain the string

“bin/zip”:

```
/usr/bin/zip  
/usr/bin/zipcloak  
/usr/bin/zipgrep  
/usr/bin/zipinfo  
/usr/bin/zipnote  
/usr/bin/zipsplit
```

If the search requirement is not so simple, `locate` can be combined with other tools such as `grep` to design more interesting searches:

```
[me@linuxbox ~]$ locate zip | grep bin  
/bin/bunzip2  
/bin/bzip2  
/bin/bzip2recover  
/bin/gunzip  
/bin/gzip  
/usr/bin/funzip  
/usr/bin/gpg-zip  
/usr/bin/preunzip  
/usr/bin/prezip  
/usr/bin/prezip-bin  
/usr/bin/unzip  
/usr/bin/unzipsfx  
/usr/bin/zip  
/usr/bin/zipcloak  
/usr/bin/zipgrep  
/usr/bin/zipinfo  
/usr/bin/zipnote  
/usr/bin/zipsplit
```

The `locate` program has been around for a number of years and there are several different variants in common use. The two most common ones used in modern Linux distributions are `slocate` and `mlocate` though they are usually symbolically linked to a generic program named `locate`. The different versions of `locate` have overlapping options sets and some versions include regular expression matching (which we’ll cover in an upcoming chapter) and wild card support. Check the man page for `locate` to determine which version of `locate` is used on your system.

Where Does The locate Database Come From?

You may notice that, on some distributions, `locate` fails to work just after the system is installed, but if you try again the next day, it works fine. What gives? The `locate` database is created by another program named `updatedb`. Usually it is run periodically as a *cron job*, that is, a task performed at regular intervals by the cron daemon. Most systems equipped with `locate` run `updatedb` once a day. Since the database is not updated continuously, you will notice that very recent files do not show up when using `locate`. To overcome this, it's possible to run the `updatedb` program manually. To do this, become the superuser and run `updatedb` at the prompt.

find – Find Files The Hard Way

While the `locate` program can find a file based solely on its name, the `find` program searches a given directory (and its subdirectories) for files based on variety of attributes. We're going to spend a lot of time with `find` because it has a lot of interesting features that we will see again and again when we start to cover programming concepts in later chapters.

In its simplest use, `find` is given one or more names of directories to search. For example, to produce a list of ones home directory:

```
[me@linuxbox ~]$ find ~
```

On most active user accounts, this will produce a large list. Since the list is sent to standard output, we can pipe the list into other programs. Let's use `wc` to count the number of files:

```
[me@linuxbox ~]$ find ~ | wc -l  
47068
```

Wow, we've been busy! The beauty of `find` is how it can be used to identify files that meet certain criteria. It does this through the (slightly strange) application of *options*, *tests*, and *actions*. We'll look at the tests first.

Tests

Let's say that we wanted a list of directories from our search. To do this, we could add the following test:

```
[me@linuxbox ~]$ find ~ -type d | wc -l  
1695
```

Adding the test `-type d` limited the search to directories. Conversely, we could have limited the search to regular files with this test:

```
[me@linuxbox ~]$ find ~ -type f | wc -l  
38737
```

Here are the common file type tests supported by find:

File Type	Description
b	Block special device file
c	Character special device file
d	Directory
f	Regular file
l	Symbolic link

We can also search by file size and by file name by adding some additional tests: Let's look for all the regular files that match the wild card pattern `*.JPG` and are larger than one megabyte:

```
[me@linuxbox ~]$ find ~ -type f -name "*.JPG" -size +1M | wc -l  
840
```

In this example we add the `-name` test followed by the wild card pattern. Notice how we enclose it in quotes to prevent pathname expansion by the shell. Next we add the `-size` test followed by the string `+1M`. The leading plus sign indicates that we are looking for files larger than the specified number. A leading minus sign would change the meaning of the string to be less than the specified number. No sign means match the value exactly. The trailing letter `"M"` indicates that the unit of measurement is

megabytes. The following characters may be used to specify units:

Character	Unit
b	512 byte blocks. This is the default if no unit is specified.
c	Bytes
w	Two byte words
k	Kilobytes (Units of 1024 bytes)
M	Megabytes (Units of 1048576 bytes)
G	Gigabytes (Units of 1073741824 bytes)

`find` supports a large number of different tests. Here is a rundown of the common ones. Note that in cases where a numeric argument is required, the same “+” and “-” notation discussed above can be applied:

Test	Description
<code>-cmin <i>n</i></code>	Matches files or directories whose content or attributes were modified <i>n</i> minutes ago.
<code>-cnewer <i>file</i></code>	Matches files or directories whose contents or attributes were modified more recently than those of <i>file</i> .
<code>-ctime <i>n</i></code>	Matches files or directories whose contents or attributes were modified more recently than <i>n</i> *24 hours.
<code>-empty</code>	Matches empty files and directories.
<code>-group <i>name</i></code>	Matches file or directories belonging to <i>group</i> . <i>group</i> may be expressed as either a group name or as a numeric group ID.
<code>-iname <i>pattern</i></code>	Like the <code>-name</code> test but case insensitive.
<code>-inum <i>n</i></code>	Match files with inode number <i>n</i> . This is helpful for finding all the hard links to a particular inode.
<code>-mmin <i>n</i></code>	Matches files or directories whose contents were modified <i>n</i> minutes ago.
<code>-mtime <i>n</i></code>	Matches files or directories whose contents were modified <i>n</i> *24 hours ago.
<code>-name <i>pattern</i></code>	Match file and directories with the specified wild card <i>pattern</i> .
<code>-newer <i>file</i></code>	Matches files and directories whose contents were modified

	more recently than the specified <i>file</i> . This is very useful when writing shell scripts that perform file backups. Each time you backup, update a file (such as a log), then use <code>find</code> to determine the files that have changed since the last backup.
<code>-nouser</code>	Matches file and directories that do not belong to a valid user. This can be used to find files belonging to deleted accounts or to detect activity by attackers.
<code>-nogroup</code>	Matches files and directories that do not belong to a valid group.
<code>-perm mode</code>	Matches files or directories that have permissions set to the specified <i>mode</i> . <i>mode</i> may be expressed by either octal or symbolic notation.
<code>-samefile name</code>	Similar to the <code>-inum</code> test. Matches files that share the same inode number as file <i>name</i> .
<code>-size n</code>	Matches files of size <i>n</i> .
<code>-type c</code>	Matches files of type <i>c</i> .
<code>-user name</code>	Matches files or directories belonging to user <i>name</i> . The user may be expressed by a user name or by a numeric user ID.

This is not a complete list. The `find` man page has all the details.

Operators

Even with all the tests that `find` provides, we may still need a better way of describe the *logical relationships* between the tests. For example, what if we needed to determine if all the files and subdirectories in a directory had secure permissions? We would look for all the files with permissions that are not 0600 and the directories with permissions that are not 0711. Fortunately, `find` provides a way to combine tests using *logical operators* to create more complex logical relationships. To express the aforementioned test, we could do this:

```
[me@linuxbox ~]$ find ~ \( -type f -not -perm 0600 \) -or \( -type d
-not -perm 0711 \)
```

Yikes! That sure looks weird. What is all this stuff? Actually, the operators are not that

complicated once you get to know them. Here is the list:

Operator	Description
-and	Matches if the tests on both sides of the operator are true. May be shortened to -a. Note that when no operator is present, -and is implied by default.
-or	Matches if a test on either side of the operator is true. May be shortened to -o.
-not	Matches if the test following the operator is false. May be abbreviated with an exclamation point (!).
()	Groups tests and operators together to form larger expressions. This is used to control the precedence of the logical evaluations. By default, <code>find</code> evaluates from left to right. It is often necessary to override the default evaluation order to obtain the desired result. Even if not needed, it is helpful sometimes to include the grouping characters to improve readability of the command. Note that since the parentheses characters have special meaning to the shell, they must be quoted when using them on the command line to allow them to be passed as arguments to <code>find</code> . Usually the backslash character is used to escape them.

With this list of operators in hand, let's deconstruct our `find` command. When viewed from the uppermost level, we see that our tests are arranged as two groupings separated by an `-or` operator:

```
( expression 1 ) -or ( expression 2 )
```

This makes sense since we are searching for files with a certain set of permissions and for directories with a different set. If we are looking for both files and directories, why do we use `-or` instead of `-and`? It's because as `find` scans through the files and directories, each one is evaluated to see if it matches the specified tests. We want to know if it is *either* a file with bad permissions *or* a directory with bad permissions. It can't be both at the same time. So if we expand the grouped expressions, we can see it this way:

```
( file with bad perms ) -or ( directory with bad perms )
```

So our next challenge is how to test for "bad permissions." How do we do that? Actually we don't. What we will test for is "not good permissions" since we know what "good permissions" are. In the case of files, we define good as 0600 and for directories, as

0711. The expression that will test files for “not good” permissions is:

```
-type f -and -not -perms 0600
```

and for directories, it would be:

```
-type d -and -not -perms 0711
```

As is noted in the table of operators above, the `-and` operator can be safely removed since it is implied by default. So if we put this all back together, we get our final command:

```
find ~ ( -type f -not -perms 0600 ) -or ( -type d -not -perms 0711 )
```

However, since the parentheses have special meaning to the shell, we must escape them to prevent the shell from trying to interpret them. Preceding each one with a backslash character does the trick.

There is another feature of the logical operators that is important to understand. Let’s say that we have two expressions separated by a logical operator:

```
expr1 -operator expr2
```

In all cases, *expr1* will always be performed, however the operator will determine if *expr2* is performed. Here’s how it works:

Results of <i>expr1</i>	Operator	<i>expr2</i> is...
True	-and	Always performed
False	-and	Never performed
True	-or	Never performed
False	-or	Always performed

Why does this happen? It’s done for performance. Take `-and` for example. We know that the expression *expr1* -and *expr2* cannot be true if the result of *expr1* is false, so there is no point in performing *expr2*. Likewise, if we have the expression *expr1* -or *expr2* and the result of *expr1* is true, there is no point in performing *expr2* as we already know that the expression *expr1* -or *expr2* is true.

OK, so it helps it go faster. Why is this important? It’s important because we can rely on this behavior to control how actions are performed as we shall soon see...

Predefined Actions

All this searching and testing is fine, but we want to get some work done! Having a list

of results from our `find` command is useful, but what we really want to do is act on the items on the list. Fortunately, `find` provides for actions to be performed based on the search results. There are a set of predefined actions and several ways to apply user-defined actions. First let's look at a few of the predefined actions:

Action	Description
<code>-delete</code>	Delete the currently matching file.
<code>-ls</code>	Performs the equivalent of <code>ls -dils</code> on the matching file. Output is sent to standard output.
<code>-print</code>	Outputs the full pathname of the matching file to standard output. This is the default action if no other action is specified.
<code>-quit</code>	Quit once a match has been made.

As with the tests, there are many more actions. See the `find` man page for full details.

In our very first example we did this:

```
find ~
```

which produced a list of every file and subdirectory contained within our home directory. It produced a list because the `-print` action is implied if no other action is specified. Thus our command could also be expressed as:

```
find ~ -print
```

We can use `find` to delete files that meet certain criteria. For example to delete files that have the file extension `".BAK"` (which is often used to designate backup files), we could use this command:

```
find ~ -type f -name '*.BAK' -delete
```

In this example, every file in the user's home directory (and its subdirectories) is searched for file names ending in `.BAK`. When they are found, they are deleted.

Warning: It should go without saying that you should *use extreme caution* when

using the `-delete` action. Always test the command by first substituting the `-print` action for `-delete` to confirm the search results.

Before we go on, let's take another look at how the logical operators affect actions. Consider the following command:

```
find ~ -type f -name '*.BAK' -print
```

As we have seen, this command will look for every regular file (`-type f`) whose name ends with `.BAK` (`-name '*.BAK'`) and will output the relative pathname of the file to standard output (`-print`). However, the reason the command performs the way it does is determined by the logical relationships between each of the tests and actions. Remembering that there is, by default, an implied `-and` relationship between each test and action, we can also express the command this way:

```
find ~ -type f -and -name '*.BAK' -and -print
```

With our command fully expressed, let's look at how the logical operators affect its execution:

Test/Action	Is Performed Only If...
<code>-print</code>	<code>-type f</code> and <code>-name '*.BAK'</code> are true
<code>-name '*.BAK'</code>	<code>-type f</code> is true
<code>-type f</code>	Is always performed since it is the first test/action in an <code>-and</code> relationship.

Since the logical relationship between the tests and actions determines which of them are performed, we can see that the order of the tests and actions is important. For instance, if we were to reorder the tests and actions so that the `-print` action was the first one, the command would behave much differently:

```
find ~ -print -and -type f -and -name '*.BAK'
```

This version of the command will print each file and then (the `-print` action always evaluates to true) test for file type and the specified file extension.

User Defined Actions

In addition to the predefined actions, we can also invoke arbitrary commands. The traditional way of doing this is with the `-exec` action. This action works like this:

```
-exec command {} ;
```

where *command* is the name of a command, {} is a symbolic representation of the current pathname and the semicolon is a required delimiter indicating the end of the command. Here's an example of using `-exec` to act like the `-delete` action discussed earlier:

```
-exec rm '{}' ';' 
```

Again, since the brace and semicolon characters have special meaning to the shell, they must be quoted.

It's also possible to execute a user defined action interactively. By using the `-ok` action in place of `-exec`, the user is prompted before each execution of the specified command:

```
find ~ -type f -name 'foo*' -ok ls -l '{}' ';'
< ls ... /home/me/bin/foo > ? y
-rwxr-xr-x 1 me  me 224 2007-10-29 18:44 /home/me/bin/foo
< ls ... /home/me/foo.txt > ? y
-rw-r--r-- 1 me  me   0 2008-09-19 12:53 /home/me/foo.txt
```

In this example, we search for files with names starting with the string "foo" and execute the command `ls -l` each time one is found. Using the `-ok` action causes the user to be prompted before the `ls` command is executed.

Improving Efficiency

When the `-exec` action is used it, launches a new instance of the specified command each time a matching file is found. There are times when we might prefer to combine all of the search results and launch a single instance of the command. For example, rather than executing the commands like this:

```
ls -l file1
```

```
ls -l file2
```

we may prefer to execute it this way:

```
ls -l file1 file2
```

thus causing the command to be executed only one time rather than multiple times. There are two ways we can do this. The traditional way, using the external command `xargs` and the alternate way using a new feature in `find` itself. We'll talk about the alternate way first.

By changing the trailing semicolon character to a plus sign, we activate the ability of `find` to combine the results of the search into an argument list for a single execution of the desired command. Going back to our example:

```
find ~ -type f -name 'foo*' -exec ls -l '{}' ';'
-rwxr-xr-x 1 me  me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me  me  0 2008-09-19 12:53 /home/me/foo.txt
```

will execute `ls` each time a matching file is found. By changing the command to:

```
find ~ -type f -name 'foo*' -exec ls -l '{}' '+'
-rwxr-xr-x 1 me  me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me  me  0 2008-09-19 12:53 /home/me/foo.txt
```

we get the same results but the system only has to execute the `ls` command once.

xargs

The `xargs` command performs an interesting function. It accepts input from standard input and converts it into an argument list for a specified command. With our example, we would use it like this:

```
find ~ -type f -name 'foo*' -print | xargs ls -l
-rwxr-xr-x 1 me  me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me  me  0 2008-09-19 12:53 /home/me/foo.txt
```

Here we see the output of the `find` command piped into `xargs` which in turn, constructs an argument list for `ls` command and then executes it.

Note: While the number of arguments that can be placed into a command line is quite large, it's not unlimited. It is possible to create commands that are too long for the shell to accept. When a command line exceeds the maximum length supported by the system, `xargs` executes the specified command with the maximum number of arguments possible and then repeats this process until

standard input is exhausted. To see the maximum size of the command line, execute `xargs` with the `--show-limits` option.

Dealing With Funny File Names

Unix-like systems allow embedded spaces (and even newlines!) in file names. This causes problems for programs like `xargs` that construct argument lists for other programs. An embedded space will be treated as a delimiter and the resulting command will interpret each space separated word as a separate argument. To overcome this, `find` and `xarg` allow the optional use of a *null character* as argument separator. A null character is defined in ASCII as the character represented by the number zero (as opposed to, for example,, the space character which is defined in ASCII as the character represented by the number 32.) The `find` command provides the action `-print0` which produces null separated output and the `xargs` command has the `--null` option which accepts null separated input. Here's an example:

```
find ~ -iname '*.jpg' -print0 | xargs --null ls -l
```

Using this technique, we can insure that all files, even those containing embedded spaces in their names, are handled correctly.

A Return To The Playground

It's time to put `find` to some (almost) practical use. We'll create a playground and try out some of what we have learned about `find`.

First, let's create a playground with lots of subdirectories and files:

```
[me@linuxbox ~]$ mkdir -p playground/dir-{1..100}
[me@linuxbox ~]$ touch playground/dir-{1..100}/file-{A..Z}
```

Marvel in the power of the command line! With these two lines we created a playground directory containing one hundred subdirectories each containing twenty-six empty files. Try that with the GUI!

The method we employed to accomplish this magic involved a familiar command (`mkdir`), an exotic shell expansion (braces) and a new command, `touch`. By combining `mkdir` with the `-p` option (which causes `mkdir` to create the parent directories of the specified paths) with brace expansion, we were able to create one

hundred directories.

The `touch` command is usually used to set or update the access, change, and modify times of files. However, if a filename argument is that of a nonexistent file, an empty file is created.

In our playground, we created one hundred instances of a file named `file-A`. Let's find them:

```
[me@linuxbox ~]$ find playground -type f -name 'file-A'
```

Note that unlike `ls`, `find` does not produce results in sorted order. Its order is determined by the layout of the storage device. To confirm that we actually have one hundred instances of the file we can confirm it this way:

```
[me@linuxbox ~]$ find playground -type f -name 'file-A' | wc -l
100
```

Next, let's look at finding files based on their modification times. We will find that this is helpful when creating backups or organizing files in chronological order. To do this, we will first create a reference file whose modification time will be used to compare against:

```
[me@linuxbox ~]$ touch playground/timestamp
```

This will create an empty file named `timestamp` and set its modify its modification time to the current time. We can verify this by using another handy command, `stat`, which is kind of a souped-up version of `ls`. The `stat` command reveals all that the system understands about a file and its attributes:

```
[me@linuxbox ~]$ stat playground/timestamp
  File: `playground/timestamp'
  Size: 0          Blocks: 0          IO Block: 4096 regular empty file
Device: 803h/2051d Inode: 14265061 Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1001/ me)   Gid: ( 1001/ me)
Access: 2008-10-08 15:15:39.000000000 -0400
Modify: 2008-10-08 15:15:39.000000000 -0400
Change: 2008-10-08 15:15:39.000000000 -0400
```

If we touch the file again and then examine it with `stat`, we will see that the file's times

have been updated.:

```
[me@linuxbox ~]$ touch playground/timestamp
[me@linuxbox ~]$ stat playground/timestamp
  File: `playground/timestamp'
  Size: 0          Blocks: 0          IO Block: 4096 regular empty file
Device: 803h/2051d Inode: 14265061 Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1001/ me)   Gid: ( 1001/ me)
Access: 2008-10-08 15:23:33.000000000 -0400
Modify: 2008-10-08 15:23:33.000000000 -0400
Change: 2008-10-08 15:23:33.000000000 -0400
```

Next, let's use `find` to update some of our playground files:

```
[me@linuxbox ~]$ find playground -type f -name 'file-B' -exec touch
'{}' ';' 
```

This will update all files in the playground named `file-B`. Next we'll use `find` to identify the updated files by comparing the all the files to the reference file `timestamp`:

```
[me@linuxbox ~]$ find playground -type f -newer playground/timestamp
```

The results will contain all one hundred instances of `file-B`. Since we performed a `touch` on all the files in the playground named `file-B` after we updated `timestamp`, they are now “newer” than it is and thus can be identified with the `-newer` test.

Finally, let's go back to the bad permissions test we performed earlier and apply it to playground:

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 \) -or \(
-type d -not -perm 0711 \)
```

This command will list all one hundred directories and twenty-six hundred files in playground (as well as, `timestamp` and `playground` itself for a total of 2702) because none of them meet our definition of “good permissions.” With our knowledge of operators and actions, we can add actions to this command to apply new permissions to the files and directories in our playground:


```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec
chmod 0600 '{}' ';' \) -or \( -type d -not -perm 0711 -exec chmod
0711 '{}' ';' \)
```

On a day-to-day basis, we might find it easier to issue two commands, one for the directories and one for the files, rather than this one large compound command, but it's nice to know that we can do it this way if we want. The important point here is to understand how the operators and actions can be used together to perform useful tasks.

Options

Finally, we have the options. The options are used to control the scope of a `find` search. They may be included with other tests and actions when constructing `find` expressions. Here is a list of the most commonly used ones:

Option	Description
<code>-depth</code>	Direct <code>find</code> to process a directory's files before the directory itself. This option is automatically applied when the <code>-delete</code> action is specified.
<code>-maxdepth <i>levels</i></code>	Set the maximum number of levels that <code>find</code> will descend into a directory tree when performing tests and actions.
<code>-mindepth <i>levels</i></code>	Set the minimum number of levels that <code>find</code> will descend into a directory tree before applying tests and actions.
<code>-mount</code>	Direct <code>find</code> not to traverse directories that are mounted on other file systems.
<code>-noleaf</code>	Direct <code>find</code> not to optimize its search based on the assumption that it is searching a Unix-like file system. This is needed when scanning DOS/Windows file systems and CD-ROMs.

Further Reading

The `locate`, `updatedb`, `find`, and `xargs` programs are all part the GNU Project's `findutils` package. The GNU Project provides a website with extensive on-line documentation which is quite good and should be read if you are using these programs in high security environments:

<http://www.gnu.org/software/findutils/>

19 - Archiving And Backup

One of the primary tasks of a computer system's administrator is that of keeping the system's data secure. One way this is done is by performing timely backups of the system's files. But even if we're not system administrators, it is often useful to make copies of things and to move large collections of files from place to place and from device to device.

In this chapter we will look at several common programs that are used to manage collections of files. There are the file compression programs:

- `gzip` – Compress or expand files
- `bzip2` – A block sorting file compressor

The archiving programs:

- `tar` – Tape archiving utility
- `zip` – Package and compress files

And the file synchronization program:

- `rsync` – Remote file and directory synchronization

Compressing Files

Throughout the history of computing, there has been a struggle to get the most data into the smallest available space, whether that space be memory, storage devices or network bandwidth. Many of the data services that we take for granted today, such as portable music players, high definition television, or broadband Internet, owe their existence to effective *data compression* techniques.

Data compression is the process of removing *redundancy* from data. Let's consider an imaginary example. Say we had a picture file with the dimensions of one hundred pixels by one hundred pixels that was entirely black. In terms of data storage (assuming twenty-four bits, or three bytes per pixel), the image will occupy thirty thousand bytes of storage:

$$100 * 100 * 3 = 30,000$$

An image that is all one color contains entirely redundant data. If we were clever, we

could encode the data in such a way that we simply describe the fact that we have a block of thirty thousand black pixels. So instead of storing a block of data containing thirty thousand zeros (black is usually represented in image files as zero), we could compress the data into the number 30,000 followed by a zero to represent our data. Such a data compression scheme is called *run-length encoding* and is one of the most rudimentary compression techniques. Today's techniques are much more advanced and complex but the basic goal remains the same – get rid of redundant data.

Compression algorithms (the mathematical techniques used to carry out the compression) fall into two general categories, *lossless* and *lossy*. Lossless compression preserves all the data contained in the original. This means that when a file is restored from a compressed version the restored file is exactly the same as the original uncompressed version. Lossy compression, on the other hand, removes data as the compression is performed to allow more compression to be applied. When a lossy file is restored it does not match the original version, rather it is a close approximation. Examples of lossy compression are JPEG (for images) and MP3 (for music.) In our discussion, we will look exclusively at lossless compression since most data on computers cannot tolerate any data loss.

gzip

The `gzip` program is used to compress one or more files. When executed, it replaces the original file with a compressed version of the original. The corresponding `gunzip` program is used to restore compressed files to their original uncompressed form. Here is an example:

```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me  me  15738 2008-10-14 07:15 foo.txt
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me  me  3230 2008-10-14 07:15 foo.txt.gz
[me@linuxbox ~]$ gunzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me  me  15738 2008-10-14 07:15 foo.txt
```

In this example we create a text file named `foo.txt` from a directory listing. Next we run `gzip` which replaces the original file with a compressed version named `foo.txt.gz`. In the directory listing of `foo.*` we see that the original file has been replaced with the compressed version and that the compressed version about one fifth the size of the original. We can also see that the compressed file has the same permissions and time stamp as the original.

Next, we run the `gzip` program to uncompress the file. Afterward we can see that the compressed version of the file has been replaced with the original, again with the permissions and time stamp preserved.

`gzip` has many options. Here are a few:

Option	Description
<code>-c</code>	Write output to standard output and keep original files. May also be specified with <code>--stdout</code> and <code>--to-stdout</code> .
<code>-d</code>	Decompress. This causes <code>gzip</code> to act like <code>gunzip</code> . May also be specified with <code>--decompress</code> or <code>--uncompress</code> .
<code>-f</code>	Force compression even if compressed version of the original file already exists. May also be specified with <code>--force</code> .
<code>-h</code>	Display usage information. May also be specified with <code>--help</code> .
<code>-l</code>	List compression statistics for each file compressed. May also be specified with <code>--list</code> .
<code>-r</code>	If one or more arguments on the command line are directories, recursively compress files contained within them. May also be specified with <code>--recursive</code> .
<code>-t</code>	Test the integrity of a compressed file. May also be specified with <code>--test</code> .
<code>-v</code>	Display verbose messages while compressing. May also be specified with <code>--verbose</code> .
<code>-number</code>	Set amount of compression. <i>number</i> is an integer in the range of 1 (fastest, least compression) to 9 (slowest, most compression). The values 1 and 9 may also be expressed as <code>--fast</code> and <code>--best</code> , respectively. The default value is 6.

Going back to our earlier example:

```
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ gzip -tv foo.txt.gz
foo.txt.gz: OK
[me@linuxbox ~]$ gzip -d foo.txt.gz
```

Here we replaced the file `foo.txt` with the compressed version named `foo.txt.gz`. Next, we tested the integrity of the compressed version using the `-t` and `-v` options.

Finally, we decompressed the file back to its original form.

`gzip` can also be used in interesting ways using standard input and output:

```
[me@linuxbox ~]$ ls -l /etc | gzip > foo.txt.gz
```

This command creates a compressed version of the results of a directory listing.

`gunzip` assumes that file names end in the extension `.gz`, so it's not necessary to specify it so long the specified name is not in conflict with an existing uncompressed file:

```
[me@linuxbox ~]$ gunzip foo.txt
```

If our goal was only to view the contents of a compressed text file, we can do this:

```
[me@linuxbox ~]$ gunzip -c foo.txt | less
```

Alternately, there is a program supplied with `gzip` called `zcat` that is equivalent to `gunzip` with the `-c` option. It can be used like the `cat` command on `gzip` compressed files:

```
[me@linuxbox ~]$ zcat foo.txt.gz | less
```

bzip2

The `bzip2` program by Julian Seward is similar to `gzip` but uses a different compression algorithm that achieves higher levels of compression at the cost of compression speed. In most regards it works in the same fashion as `gzip`. A file compressed with `bzip2` is denoted with the extension `.bz2`:

```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-r--r-- 1 me me 15738 2008-10-17 13:51 foo.txt
[me@linuxbox ~]$ bzip2 foo.txt
[me@linuxbox ~]$ ls -l foo.txt.bz2
-rw-r--r-- 1 me me 2792 2008-10-17 13:51 foo.txt.bz2
[me@linuxbox ~]$ bunzip2 foo.txt.bz2
```

As we can see, `bzip2` can be used the same way as `gzip`. All the options (except for `-r`) that we discussed for `gzip` are also supported in `bzip2`. Note however that the compression level option (`-number`) has a somewhat different meaning to `bzip2`. `bzip2` comes with `bunzip2` and `bzcat` for decompressing files.

`bzip2` also comes with the `bzip2recover` program which will try to recover damaged `.bz2` files.

Don't Compress Again

I sometimes see people do something like this:

```
$ gzip picture.jpg
```

where they attempt to compress a file that is already compressed with an effective compression algorithm.

Don't do it. You're probably just wasting your time, and space! If you apply compression to a file that is already compressed, you will actually end up making the resulting file larger. This is because all compression techniques involve some overhead that is added to the file to describe the compression. If you try to compress a file that already contains no redundant information, the compression will not result in any savings to offset the additional overhead.

Archiving Files

A common file management task often used in conjunction with compression is *archiving*. Archiving is the process of gathering up many files and bundling them together into a single large file. Archiving is often done as a part of performing system back ups and when old data is moved from a system to some other type of long-term storage.

tar

In the Unix-like world of software, the `tar` program is the classic tool for archiving files. Its name, short for *tape archive*, reveals its roots as a tool for making backup tapes. While it is still used for that traditional task, it is equally adept at other types of storage devices as well. You will often see file names that end with the extension `.tar` or `.tgz` which indicates a "plain" tar archive and a gzipped archive respectively. A tar archive can consist of a group of separate files or of one or more directory hierarchies or a mixture of both. The command syntax works like this:

```
tar mode [options] pathname...
```

where *mode* is one of the following operating modes (only a partial list is shown here, see the `tar` man page for a complete list):

Mode	Description
c	Create an archive from a list of files and/or directories
x	Extract an archive.
r	Append specified pathnames to the end of an archive.
t	List the contents of an archive.

`tar` uses a slightly non-standard way of expressing options, so some examples may be the best way to show how this works. First let's re-create our playground from the previous chapter:

```
[me@linuxbox ~]$ mkdir -p playground/dir-{1..100}
[me@linuxbox ~]$ touch playground/dir-{1..100}/file-{A-Z}
```

Next, let's create a tar archive of the entire playground:

```
[me@linuxbox ~]$ tar cf playground.tar playground
```

This command creates a tar archive named `playground.tar` that contains the entire playground directory hierarchy. As we can see, the mode and the option `f` which is used to specify the name of the tar archive may be joined together and no leading dash is required. Note however that the mode must always be specified first before any other option.

To list the contents of the archive, we can do this:

```
[me@linuxbox ~]$ tar tf playground.tar
```

For a more detailed listing, we can add the `v` (verbose) option:

```
[me@linuxbox ~]$ tar tvf playground.tar
```

Now, let's extract the playground in a new location. We will do this by creating a new

directory named `foo` and change directory and extract the tar archive:

```
[me@linuxbox ~]$ mkdir foo
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground.tar
[me@linuxbox foo]$ ls
playground
```

If we examine the contents of `~/foo/playground`, we will see that the archive was successfully installed, creating a precise reproduction of the original files. There is one caveat however, unless you are operating as the superuser, files and directories restored from archives take on the ownership of the user performing the restoration rather than the original owner.

Another interesting behavior of `tar` is the way it handles pathnames in archives. The default is for pathnames to always be relative rather than absolute. `tar` does this by simply removing any leading slash from the pathname when creating the archive. Here is a demonstration. We will recreate our archive this time specifying an absolute pathname:

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ tar cf playground2.tar ~/playground
```

Remember, `~/playground` will expand into `/home/me/playground` when we press the enter key, so we will get an absolute pathname for our demonstration. Next, we will restore the archive as before and watch what happens:

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground2.tar
[me@linuxbox foo]$ ls
home  playground
[me@linuxbox foo]$ ls home
me
[me@linuxbox foo]$ ls home/me
playground
```

Here we can see that when we restored our second archive, it recreated the directory `home/me/playground` relative to our current working directory, `~/foo`, not relative to the root directory as would have been the case with an absolute pathname. This may seem like an odd way for it to work, but it's actually more useful this way as it allows us to restore archives to any location rather than being forced to restore them to only their

original locations. Repeating the exercise with the inclusion of the verbose option (`v`) will give a clearer picture of what's going on.

Let's consider a hypothetical, yet practical example of `tar` in action. Imagine we wanted to copy the home directory and its contents from one system to another and we had a large USB hard drive that we could use for the transfer. On our modern Linux system, the drive is "automagically" mounted in the `/media` directory and let's imagine that the disk has a volume name of `BigDisk` when we attach it. To make the `tar` archive we could do the following:

```
[me@linuxbox ~]$ sudo tar cf /media/BigDisk/home.tar /home
```

After the tar file is written, we unmount the drive and attach it to the second computer. Again it is mounted at `/media/BigDisk`. To restore the archive we do this:

```
[me@linuxbox2 ~]$ cd /  
[me@linuxbox2 /]$ sudo tar xf /media/BigDisk/home.tar
```

What's important to see here is that we must first change directory to `/` so that the restoration is relative to the root directory since all pathnames within the archive are relative.

When restoring an archive, it's possible to limit what in the archive is restored. For example, if we wanted to restore a single file from an archive, it could be done like this:

```
tar xf archive.tar pathname
```

By adding a trailing *pathname* to the command, `tar` will only restore the specified file. Multiple pathnames may be specified. Note that *pathname* must be the full, exact relative pathname as stored in the archive. When specifying pathnames, wildcards are not normally supported, however the GNU version `tar` (which is the version most often found in Linux distributions) supports them with the `--wildcards` option. Here is an example using our previous `playground.tar` file:

```
[me@linuxbox ~]$ cd foo  
[me@linuxbox foo]$ tar xf ../playground.tar --wildcards 'home/me/dir-  
*/file-A'
```

This command will extract (restore) only files matching the specified pathname including the wildcard `dir-*`.

`tar` is often used in conjunction with `find` to produce archives. In this example we will use `find` to produce a set of files to include in an archive:

```
[me@linuxbox ~]$ find playground -name 'file-A' -exec tar rf
playground.tar '{}' '+'
```

Here we use `find` to match all the files in `playground` named `file-A` and then using the `-exec` action, invoke `tar` in the append mode (`r`) to add the matching files to the archive `playground.tar`.

Using `tar` with `find` is a good way of creating *incremental backups* of a directory tree or an entire system. By using `find` to match files newer than a timestamp file, we could create an archive that only contained files newer than the last archive, assuming that the timestamp file is updated right after each archive is created.

`tar` can also make use of both standard input and output. Here is a comprehensive example:

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name 'file-A' | tar cf - --files-
from=- | gzip > playground.tgz
```

In this example, we used the `find` program to produce a list of matching files and piped them into `tar`. If the filename “-” is specified, it is taken to mean standard input or output as needed (by the way, this convention of using “-” to represent standard input/output is used by a number of other programs too.) The `--files-from` option (which may be also be specified as `-T`) causes `tar` to read its list of pathnames from a file rather than the command line. Lastly, the archive produced by `tar` is piped into `gzip` to create the compressed archive `playground.tgz`. The `.tgz` extension is the conventional extension given to `gzip`-compressed tar files. The extension `.tar.gz` is also sometimes used.

While we used the `gzip` program externally to produced our compressed archive, modern versions of GNU `tar` support both `gzip` and `bzip2` compression directly with the use of the `z` and `j` options respectively. Using our previous example as a base, we can simplify it this way:

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar czf
playground.tgz -T -
```

If we had wanted to create a bzip2 compressed archive instead, we could have done this:

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar cjf
playground.tbz -T -
```

By simply changing the compression option from `z` to `j` (and changing the output file's extension to `.tbz` to indicate a bzip2 compressed file) we enabled bzip2 compression.

Another interesting way standard input and output can be employed is for transferring files between systems over a network. Imagine that we had two machines running a Unix-like system equipped with `tar` and `ssh`. In such a scenario, we could transfer a directory from a remote system (named `remote-sys` for this example) to our local system:

```
[me@linuxbox ~]$ mkdir remote-stuff
[me@linuxbox ~]$ cd remote-stuff
[me@linuxbox remote-stuff]$ ssh remote-sys tar cf - Documents | tar
xf -
me@remote-sys's password:
[me@linuxbox remote-stuff]$ ls
Documents
```

Here we were able to copy a directory named `Documents` from the remote system `remote-sys` and to a directory within the directory named `remote-stuff` on the local system. How did we do this? First we launched the `tar` program on the remote system using `ssh`. As you recall, `ssh` allows us to execute a program remotely on networked computer and “see” the results on the local system. By “see” we mean that the standard output produced on the remote system is sent to the local system for viewing. We can take advantage of this to have `tar` create an archive (the `C` mode) and send it to standard output rather than a file (the `f` option with the dash argument) thereby transporting the archive over the encrypted tunnel provided by `ssh` to the local system. On the local system, we execute `tar` and have it expand an archive (the `x` mode) supplied from standard input (again, the `f` option with the dash argument.)

zip

The `zip` program is both a compression tool and an archiver. The file format utilized by

the program is familiar to almost every Windows user as it reads and writes `.zip` files which are extremely common in the Windows world but less so in Linux where `gzip` is the predominant compression program with `bzip2` being a close second.

In its most basic usage, `zip` is invoked like this:

```
zip options zipfile file...
```

For example, to make an zip archive of our playground we would do this:

```
[me@linuxbox ~]$ zip -r playground.zip playground
```

We include the `-r` option for recursion or else only the `playground` directory (and none of its contents) is stored. The addition of the extension `.zip` is automatic and thus the following command is exactly equivalent:

```
[me@linuxbox ~]$ zip -r playground playground
```

However, we will include the file extension for clarity.

During the creation of the zip archive, `zip` will normally display a series of messages like this:

```
adding: playground/dir-20/file-Z (stored 0%)
adding: playground/dir-20/file-Y (stored 0%)
adding: playground/dir-20/file-X (stored 0%)
adding: playground/dir-87/ (stored 0%)
adding: playground/dir-87/file-S (stored 0%)
```

These messages show the status of each file added to the archive. `zip` will add files to the archive using one of two storage methods, either it will “store” a file, as shown here, without compression, or it will “deflate” the file which performs compression. The numeric value displayed after the storage method indicates the amount of compression achieved. Since our playground only contains empty files, no compression is performed on its contents.

Extracting the contents of a zip file is straightforward using the `unzip` program:

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ unzip ../playground.zip
```

One thing to note about `zip` as opposed to `tar` is that if an existing archive is specified, it is updated rather than replaced. What this means is that the existing archive is preserved but any new files are added and any matching files are replaced.

Files may be listed and extracted selectively from a zip archive by specifying them to `unzip`:

```
[me@linuxbox ~]$ unzip -l playground.zip playground/dir-87/file-Z
Archive:  ../playground.zip
 Length   Date   Time    Name
-----
          0  10-05-08 09:25  playground/dir-87/file-Z
-----
          0                                  1 file
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ unzip ./playground.zip playground/dir-87/file-Z
Archive:  ../playground.zip
replace playground/dir-87/file-Z? [y]es, [n]o, [A]ll, [N]one,
[r]ename: y
extracting: playground/dir-87/file-Z
```

Using the `-l` option causes `unzip` to merely list the contents of the archive without extracting the file. If no file(s) are specified, `unzip` will list all files in the archive as we would expect. The `-v` option can be added to increase the verbosity of the listing. Note that when the archive extraction conflicts with an existing file, the user is prompted before the file is replaced.

Like `tar`, `zip` can make use of standard input and output though its implementation is somewhat less useful. It is possible to pipe a list of file names to `zip` via the `-@` option:

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name "file-A" | zip -@ file-A.zip
```

Here we use `find` to generate a list of files matching the test `-name "file-A"` and pipe the list into `zip` which creates the archive `file-A.zip` containing the selected files.

`zip` also supports writing its output to standard output but its use is limited because very few programs can make use of its output. Unfortunately, the `unzip` program, strangely, does not accept standard input. This prevents `zip` and `unzip` from being used together to perform network file copying like `tar`.

`zip` can however accept standard input, so it can be used to compress the output of other

programs:

```
[me@linuxbox ~]$ ls -l /etc/ | zip ls-etc.zip -  
adding: - (deflated 80%)
```

In this example we pipe the output of `ls` into `zip`. Like `tar`, `zip` interprets the trailing dash as meaning use standard input for the input file.

The `unzip` program allows its output to be sent to standard output when the `-p` (for pipe) option is specified:

```
[me@linuxbox ~]$ unzip -p ls-etc.zip | less
```

We touched on some of the basic things that `zip/unzip` can do. They both have a lot of options (though some are platform specific to other systems) that add to their flexibility. The man pages for both `zip` and `unzip` are pretty good and contain useful examples. However the main use these programs tends to be exchanging files with Windows systems rather than an everyday tool for performing compression and archiving on Linux where `tar` and `gzip` are greatly preferred.

Synchronizing Files And Directories

A common strategy for maintaining a backup copy of a system involves keeping one or more directories synchronized with another located on either the local system (usually a removable storage device of some kind) or with a remote system. We might for example have a local copy of a web site under development and from time to time synchronize it with the “live” copy on a remote web server.

In the Unix-like world the preferred tool for this task is `rsync`. This program can synchronize both local and remote directories using the *rsync remote-update protocol* which allows `rsync` to quickly detect the differences between two directories and perform the minimum amount of copying required to bring them into sync. This makes `rsync` very fast and economical to use compared to other kinds of copy programs.

`rsync` is invoked like this:

```
rsync options source destination
```

where *source* and *destination* are one of the following:

- A local file or directory
- A remote file or directory in the form of *[user@]host:path*

- A remote rsync server specified with a URI of `rsync://[user@]host[:port]/path`

Note that either the source or destination must be a local file. Remote to remote copying is not supported.

Let's try `rsync` out on some local files. First let's clean out our `foo` directory:

```
[me@linuxbox ~]$ rm -rf foo/*
```

Next we'll synchronize the `playground` directory with a corresponding copy in `foo`:

```
[me@linuxbox ~]$ rsync -av playground foo
```

We've included both the `-a` option (for archiving – causes recursion and preservation of file attributes) and the `-v` option (verbose output) to make a *mirror* of the `playground` directory within `foo`. While the command runs, we will see a list of the files and directories being copied. At the end we will see a summary message like this:

```
sent 135759 bytes  received 57870 bytes  387258.00 bytes/sec
total size is 3230  speedup is 0.02
```

indicating the amount of copying performed. If we run the command again, we will see a different result:

```
[me@linuxbox ~]$ rsync -av playground foo
building file list ... done

sent 22635 bytes  received 20 bytes  45310.00 bytes/sec
total size is 3230  speedup is 0.14
```

Notice how there was no listing of files. This is because `rsync` detected that there were no differences between `~/playground` and `~/foo/playground` and therefore it didn't need to copy anything. If we modify a file in `playground` and run `rsync` again:

```
[me@linuxbox ~]$ touch playground/dir-99/file-Z
[me@linuxbox ~]$ rsync -av playground foo
```

```
building file list ... done
playground/dir-99/file-Z
sent 22685 bytes  received 42 bytes  45454.00 bytes/sec
total size is 3230  speedup is 0.14
```

we see that `rsync` detected the change and copied only the updated file.

As a practical example, let's consider the imaginary external hard drive that we used earlier with `tar`. If we attach the drive to our system and it's once again mounted at `/media/BigDisk`, we can perform a useful system back up by first creating a directory on the external drive named `/backup` and then use `rsync` to copy the most important stuff from our system to the external drive:

```
[me@linuxbox ~]$ mkdir /media/BigDisk/backup
[me@linuxbox ~]$ sudo rsync -av --delete /etc /home /usr/local
/media/BigDisk/backup
```

In this example, we copied the `/etc`, `/home`, and `/usr/local` directories from our system to our imaginary storage device. We included the `--delete` option to remove files that may have existed on the backup device that no longer existed on the source device (this would be irrelevant the first time we made a backup but would be useful on subsequent copies.) Repeating the procedure of attaching the external drive and running this `rsync` command would be a useful (though possibly not ideal) way of keeping a small system backed up. Of course, an alias would be helpful here too. We could create an alias and add it to our `.bashrc` file to provide this feature:

```
alias backup='sudo rsync -av --delete /etc /home /usr/local
/media/BigDisk/backup'
```

Now, all we have to do is attach our external drive and run the `backup` command to do the job.

Using `rsync` Over A Network

One of the real beauties of `rsync` is that it can be used to copy files over a network. The “r” in `rsync` stands for “remote” after all. Remote copying can be done in one of two ways. First, it can communicate with another system that has `rsync` installed along with a remote shell program such as `ssh`. Let's say we had another system on our local network with a lot of available hard drive space and we wanted to perform our backup

operation using the remote system instead of an external drive. Assuming that it already had a directory named `/backup` where we could deliver our files, we could do this:

```
[me@linuxbox ~]$ sudo rsync -av --delete --rsh=ssh /etc /home
/usr/local remote-sys:/backup
```

We made two changes to our command to facilitate the network copy. First, we added the `--rsh=ssh` option which instructs `rsync` to use the `ssh` program as its remote shell. In this way we were able to use an `ssh` encrypted tunnel to securely transfer the data from the local system to the remote host. Second, we specified the remote host by prefixing its name (in this case the remote host is named `remote-sys`) to the destination path name.

The second way that `rsync` can be used to synchronize files over a network is by means of a *rsync server*. `rsync` can be configured to run as a daemon and listen to incoming requests for synchronization. This is often done to allow mirroring of a remote system. For example, Red Hat Software maintains a large repository of software packages under development for its Fedora distribution. It is useful for software testers to mirror this collection during the testing phase of the distribution release cycle. Since files in the repository change frequently (often more than once a day) it is desirable to maintain a local mirror by periodic synchronization rather than by bulk copying of the repository. One of these repositories is kept at Georgia Tech and we could mirror it using our local copy of `rsync` and their `rsync` server like this:

```
[me@linuxbox ~]$ mkdir fedora-devel
[me@linuxbox ~]$ rsync -av -delete rsync://rsync.gtlib.gatech.edu/fed
ora-linux-core/development/i386/os fedora-devel
```

In this example, we use the URI of the remote `rsync` server which consists of a protocol (`rsync://`) followed by the remote host name (`rsync.gtlib.gatech.edu`) followed by the pathname of the repository.

Further Reading

The man pages for all of the commands discussed here are pretty clear and contain useful examples. In addition, the GNU project has a good online manual for their version of `tar`. It can be found here:

<http://www.gnu.org/software/tar/manual/index.html>

20 - Regular Expressions

In the next few chapters, we are going to look at various tools that are used to manipulate text. As we have seen, text data plays an important role on all Unix-like systems such as Linux. But before we can fully appreciate all of the features offered by these tools, we have to first examine a technology that is frequently associated with the most sophisticated uses of these tools – regular expressions.

As we have navigated the many features and facilities offered by the command line, we have encountered some truly arcane shell features and commands, such as shell expansion and quoting, keyboard shortcuts, and command history, not to mention the vi editor. Regular expressions continue this “tradition” and may be (arguably) the most arcane feature of them all. This is not to suggest that the time it takes to learn about them is not worth the effort. Quite the contrary. A good understanding, will enable us to perform amazing feats, though their full value may not be immediately apparent.

What Are Regular Expressions?

Simply put, regular expressions are a symbolic notation that is used to identify patterns in text. In some ways, they resemble the shell’s wildcard method of matching file and pathnames but on a much grander scale. Regular expressions are supported by many command line tools and by most programming languages to facilitate the solution of text manipulation problems. However, to further confuse things, not all regular expressions are the same, they vary slightly from tool to tool and from programming language to language. For our discussion, we will limit ourselves to regular expressions as described in the POSIX standard which will cover most of the command line tools as opposed to many programming languages (most notably *perl*) which use slightly larger and richer sets of notations.

grep

The main program we will use to work with regular expressions is our old pal `grep`. The name “grep” is actually derived from the phrase “global regular expression print” so we can see that `grep` has something to do with regular expressions. In essence, `grep` searches text files for the occurrence of a specified regular expression and outputs any

line containing a match to standard output.

So far we have used `grep` with fixed strings, like so:

```
[me@linuxbox ~]$ ls /usr/bin | grep zip
```

which will list all the files in the `/usr/bin` directory whose names contain the substring “zip”.

The `grep` program accepts options and arguments this way:

```
grep [options] regex [file...]
```

where *regex* is a regular expression.

Here is a list of the commonly used `grep` options:

Option	Description
-i	Ignore case. Do not distinguish between upper and lower case characters. May also be specified <code>--ignore-case</code> .
-v	Invert match. Normally, <code>grep</code> prints lines that contain a match. This option causes <code>grep</code> to print every line that does not contain a match. May also be specified <code>--invert-match</code> .
-c	Print the number of matches (or non-matches if the <code>-v</code> option is also specified) instead of the lines themselves. May also be specified <code>--count</code> .
-l	Print the name of each file that contains a match instead of the lines themselves. May also be specified <code>--files-with-matches</code> .
-L	Like the <code>-l</code> option but prints only the names of files that do not contain matches. May also be specified <code>--files-without-match</code> .
-n	Prefix each matching line with the number of the line within the file. May also be specified <code>--line-number</code> .
-h	For multi-file searches, suppress the output of file names. May also be specified <code>--no-filename</code> .

In order to more fully explore `grep`, let’s create some text files to search:

```
[me@linuxbox ~]$ ls /bin > dirlist-bin.txt
[me@linuxbox ~]$ ls /usr/bin > dirlist-usr-bin.txt
[me@linuxbox ~]$ ls /sbin > dirlist-sbin.txt
[me@linuxbox ~]$ ls /usr/sbin > dirlist-usr-sbin.txt
[me@linuxbox ~]$ ls dirlist*.txt
dirlist-bin.txt  dirlist-sbin.txt  dirlist-usr-sbin.txt
dirlist-usr-bin.txt
```

We can perform a simple search of our list of files like so:

```
[me@linuxbox ~]$ grep bzip dirlist*.txt
dirlist-bin.txt:bzip2
dirlist-bin.txt:bzip2recover
```

In this example, `grep` searches all of the listed files for the string `bzip` and finds two matches, both in the file `dirlist-bin.txt`. If we were only interested in the list of files that contained matches rather than the matches themselves, we could specify the `-l` option:

```
[me@linuxbox ~]$ grep -l bzip dirlist*.txt
dirlist-bin.txt
```

Conversely, if we wanted only to see a list of the files that did not contain a match, we could do this:

```
[me@linuxbox ~]$ grep -L bzip dirlist*.txt
dirlist-home.txt
dirlist-sbin.txt
dirlist-usr-bin.txt
dirlist-usr-sbin.txt
```

Meta Characters And Literals

While it may not seem apparent, our `grep` searches have been using regular expressions all along, albeit very simple ones. The regular expression “`bzip`” is taken to mean that a match will occur only if the line in the file contains at least four characters and that somewhere in the line the characters “`b`”, “`z`”, “`i`”, and “`p`” are found in that order with no other characters in between. The characters in the string “`bzip`” are all *literal characters*, in that they match themselves. In addition to literals, regular expressions may also

include *meta characters* that are used to specify more complex matches. Regular expression meta characters consist of the following:

`^ $. [] { } - ? * + () | \`

All other characters are considered literals though the backslash character is used in a few cases to create *meta sequences*, as well as, allowing the meta characters to be escaped and treated as literals instead of being interpreted as meta characters.

Note: As we can see, many of the regular expression meta characters are also characters that have meaning to the shell when expansion is performed. When we pass regular expressions containing meta characters on the command line, it is vital that they be enclosed in quotes to prevent the shell from attempting to expand them.

The Any Character

The first meta character we will look at is the dot or period character which is used to match any character. If we include it in a regular expression it will match any character in that character position. Here's an example:

```
[me@linuxbox ~]$ grep -h '.zip' dirlist*.txt
bunzip2
bzip2
bzip2recover
gunzip
gzip
funzip
pgp-zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
```

Here we searched for the any line in our files that match the regular expression “.zip”. There are a couple of interesting things to note about the results. Notice how the zip program was not found. This is because the inclusion of the dot in our regular expression increased the length of the required match to four characters in length and the name “zip” only contains three, so it does not match. Second, if there had been any files in our lists that contained the file extension .zip they would have also been matched because the period character in the file extension is treated as “any character” too.

Anchors

The caret (^) and dollar sign (\$) characters are treated as *anchors* in regular expressions. This means that they cause the match to occur only if the regular expression is found at the beginning of the line (^) or at the end of the line (\$):

```
[me@linuxbox ~]$ grep -h '^zip' dirlist*.txt
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
[me@linuxbox ~]$ grep -h 'zip$' dirlist*.txt
gunzip
gzip
funzip
gpg-zip
preunzip
prezip
unzip
zip
[me@linuxbox ~]$ grep -h '^zip$' dirlist*.txt
zip
```

Here we search the list of files for the string “zip” located at the beginning of the line, the end of the line, and on a line where it is at both the beginning of the line and the end of the line (i.e. by itself on the line.) Note that the regular expression ‘^\$’ (a beginning and an end with nothing in between) will match blank lines.

A Crossword Puzzle Helper

Even with our limited knowledge of regular expressions at this point we can do something useful.

My wife loves crossword puzzles and sometimes she will ask me for help with a particular question. Something like, “what’s a five letter word whose third letter is ‘j’ and last letter is ‘r’ that means...?” This kind of question got me thinking.

Did you know that your Linux system contains a dictionary? It does. Take a look in the `/usr/share/dict` directory and you might find one, or several. The dictionary files located there are just long lists of words, one per line, arranged in alphabetical order. On my system, the `words` file contains just over 98,500 words.

To find possible answers to the crossword puzzle question above, we could do this:

```
[me@linuxbox ~]$ grep -i '^..j.r$' /usr/share/dict/words
Major
major
```

Using this regular expression, we can find all the words in our dictionary file that are five letters long and have a “j” in the third position and an “r” in the last position.

Bracket Expressions And Character Classes

In addition to matching any character at a given position in our regular expression, we can also match a single character from a specified set of characters using *bracket expressions*. With bracket expressions we can specify a set of characters (including characters that would otherwise be interpreted as meta characters) that will be matched. Here is an example using a two character set:

```
[me@linuxbox ~]$ grep -h '[bg]zip' dirlist*.txt
bzip2
bzip2recover
gzip
```

Here we match any line that contains the string “bzip” or “gzip”

A set may contain any number of characters and meta characters lose their special meaning when placed within brackets. However, there are two cases where meta characters are used within bracket expressions, but they have different meanings. The first is the caret (^) which is used to indicate negation, and the dash (-) which is used to indicate a character range.

Negation

If the first character in a bracket expression is a caret (^), the remaining characters are taken to be a set of characters that must not be present at the given character position. Modifying our previous example:

```
[me@linuxbox ~]$ grep -h '[^bg]zip' dirlist*.txt
bunzip2
gunzip
funzip
```

```
gpg-zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
```

With negation activated, we get a list of files that contain the string “zip” preceded by any character except “b” or “g”. Notice that the file `zip` was not found. A negated character set still requires a character at the given position, but the character must not be a member of the negated set.

The caret character only invokes negation if it is the first character within a bracket expression, otherwise it loses its special meaning and becomes an ordinary character in the set.

Traditional Character Ranges

If we wanted to construct a regular expression that would find every file in our lists that began with an upper case letter we could do this:

```
[me@linuxbox ~]$ grep -h '^[ABCDEFGHJKLMNOPQRSTUVWXYZ]' dirlist*.txt
```

Just a matter of putting all twenty-six upper case letters in a bracket expression. But the idea of all that typing is deeply troubling, so there is another way:

```
[me@linuxbox ~]$ grep -h '^[A-Z]' dirlist*.txt
MAKEDEV
ControlPanel
GET
HEAD
POST
X
X11
Xorg
MAKEFLOPPIES
NetworkManager
NetworkManagerDispatcher
```

By using a character range, we can abbreviate the twenty-six letters down to three. Any range of characters can be expressed this way including multiple ranges, such as this expression that matches all file names starting with letters and numbers:


```
[me@linuxbox ~]$ grep -h '^[A-Za-z0-9]' dirlist*.txt
```

In character ranges, we see that the dash character is treated specially, so how do we actually include a dash character in a bracket expression? Make it the first character in the expression. Consider these two examples:

```
[me@linuxbox ~]$ grep -h '[A-Z]' dirlist*.txt
```

This will match every file name containing an upper case letter. While:

```
[me@linuxbox ~]$ grep -h '[-AZ]' dirlist*.txt
```

will match every file name containing a dash, or a upper case “A” or an uppercase “Z”.

POSIX Character Classes

The traditional character ranges are an easily understood and effective way to handle the problem of quickly specifying sets of characters. Too bad they don't always work. While we have not encountered any problems with our use of `grep` so far, we might run into problems using other programs.

Back in Chapter 5, we looked at how wildcards are used to perform pathname expansion. In that discussion we talked about how character ranges could be used in a manner almost identical to the way they are used in regular expressions, but here's the problem:

```
[me@linuxbox ~]$ ls /usr/sbin/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]*  
/usr/sbin/MAKEFLOPPIES  
/usr/sbin/NetworkManagerDispatcher  
/usr/sbin/NetworkManager
```

(Depending on the Linux distribution, we will get a different list of files, possibly an empty list. This example is from Ubuntu) This command produces the expected result, just files whose names begin with an uppercase letter, but:

```
[me@linuxbox ~]$ ls /usr/sbin/[A-Z]*  
/usr/sbin/biosdecode  
/usr/sbin/chat
```

```
/usr/sbin/chgpasswd
/usr/sbin/chpasswd
/usr/sbin/chroot
/usr/sbin/cleanup-info
/usr/sbin/complain
/usr/sbin/console-kit-daemon
```

(Only a partial listing of the results is shown.) With this command we get an entirely different result. Why is that? It's a long story, but we'll try to keep it short.

Back when Unix was first developed, it only knew about ASCII characters and this feature reflects that fact. In ASCII, the first thirty-two characters (numbers 0-31) are control codes (things like tabs, backspaces, and carriage returns.) The next thirty-two (32-63) contain printable characters including most punctuation characters and the numerals zero through nine. The next thirty-two (numbers 64-95) contain the uppercase letters and a few more punctuation symbols. The final thirty-one (numbers 96-127) contain the lowercase letters and a few more punctuation symbols. Based on this arrangement, systems using ASCII used a *collation order* that looked like this:

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

This differs from proper dictionary order which is like this:

AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz

As the popularity of Unix spread beyond the United States, there grew a need to support characters that were not found in U.S. English. The ASCII table was expanded to use a full eight bits adding characters numbers 128-255 which accommodated many more languages. To support this ability, the POSIX standards introduced a concept called a *locale* which could be adjusted to set the character set needed for a particular location. We can see the language setting of our system using this command:

```
[me@linuxbox ~]$ echo $LANG
en_US.UTF-8
```

With this setting, POSIX compliant applications will use a dictionary collation order rather than ASCII order. This explains the behavior of the commands above. A character range of [A-Z] when interpreted in dictionary order includes all of the alphabetic characters except the lowercase "z" and hence our results.

To partially work around this problem, the POSIX standard includes a number of character classes which provide useful ranges of characters. They are described in the table below:

Character Class	Description
<code>[:alnum:]</code>	The alphanumeric characters. In ASCII, equivalent to: <code>[A-Za-z0-9]</code>
<code>[:word:]</code>	The same as <code>[:alnum:]</code> with the addition of the underscore (<code>_</code>) character.
<code>[:alpha:]</code>	The alphabetic characters. In ASCII, equivalent to: <code>[A-Za-z]</code>
<code>[:blank:]</code>	Includes the space and tab characters.
<code>[:cntrl:]</code>	The ASCII control codes. Includes the ASCII characters zero through thirty-one and 127.
<code>[:digit:]</code>	The numerals zero through nine.
<code>[:graph:]</code>	The visible characters. In ASCII it includes characters 33 through 126.
<code>[:lower:]</code>	The lowercase letters.
<code>[:punct:]</code>	The punctuation characters. In ASCII, equivalent to: <code>[- !"#\$%&'()*+,-./:;<=>?@[_`{ }~]</code>
<code>[:print:]</code>	The printable characters. All the characters in <code>[:graph:]</code> plus the space character.
<code>[:space:]</code>	The whitespace characters including space, tab, carriage return, newline, vertical tab, and form feed. In ASCII, equivalent to: <code>[\t\r\n\v\f]</code>
<code>[:upper:]</code>	The upper case characters.
<code>[:xdigit:]</code>	Characters used to express hexadecimal numbers. In ASCII, equivalent to: <code>[0-9A-Fa-f]</code>

Even with the character classes, there is still no convenient way to express partial ranges such as `[A-M]`.

Using character classes we can repeat our directory listing and see an improved result:

```
[me@linuxbox ~]$ ls /usr/sbin/[[:upper:]]*
/usr/sbin/MAKEFLOPPIES
/usr/sbin/NetworkManagerDispatcher
/usr/sbin/NetworkManager
```

Remember however that this is not an example of a regular expression, rather it is the shell performing pathname expansion. We show it here because POSIX character classes can be used for both.

Reverting To Traditional Collation Order

You can have your system use the traditional (ASCII) collation order by changing the value of the `LANG` environment variable. As we saw above, the `LANG` variable contains the name of the language and character set used in your locale. This value was originally determined as your Linux was installed when you selected an installation language.

To see the locale settings, use the `locale` command:

```
[me@linuxbox ~]$ locale
LANG=en_US.UTF-8
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_PAPER="en_US.UTF-8"
LC_NAME="en_US.UTF-8"
LC_ADDRESS="en_US.UTF-8"
LC_TELEPHONE="en_US.UTF-8"
LC_MEASUREMENT="en_US.UTF-8"
LC_IDENTIFICATION="en_US.UTF-8"
LC_ALL=
```

To change the locale to use the traditional Unix behaviors, set the `LANG` variable to `POSIX`:

```
[me@linuxbox ~]$ export LANG=POSIX
```

Note that this change converts the system to use U.S. English (more specifically ASCII) for its character set, so carefully consider if this is really what you want.

You can make this change permanent by adding this line to your `.bashrc` file:

```
export LANG=POSIX
```

POSIX Basic Vs. Extended Regular Expressions

Just when we thought this couldn't get any more confused, we discover that POSIX also splits regular expression implementations into two kinds, *basic regular expressions (BRE)* and *extended regular expressions (ERE)*. The features we have covered so far are supported by any application that is POSIX-compliant and implements BRE. Our `grep` program is one such program.

What's the difference between BRE and ERE? It's a matter of meta characters. With BRE the following meta characters are recognized:

`^ $. [] *`

All other characters are considered literals. With ERE, the following meta characters (and their associated functions) are added:

`() { } ? + |`

However (and this is the fun part) the “(“, “)“, “{“, and “}” characters are treated in as meta characters in BRE *if* they are escaped with a backslash, whereas with ERE, preceding any meta character with a backslash causes it to be treated as a literal. We'll try to point out any weirdness that comes along in the discussions that follow.

Since the features we are going to discuss next are part of ERE, we are going to need to use a different `grep`. Traditionally, this has been performed by the `egrep` program but the GNU version of `grep` also supports extended regular expressions when the `-E` option is used.

POSIX

During the 1980's Unix became a very popular commercial operating system, but by 1988, the Unix world was in turmoil. Many computer manufacturers had licensed the Unix source code from its creators, AT&T and were supplying versions of the operating system with their systems. However in their efforts to create product differentiation, each manufacturer was adding proprietary changes and extensions. This was starting to limit the compatibility of software. As always with proprietary vendors, each was trying to play a winning game of “lock-in” with their customers. This dark time in the history of Unix is known today as “*the Balkanization.*”

Enter the IEEE (Institute of Electrical and Electronics Engineers.) In the mid 1980s they began developing a set of standards that would define how Unix (and Unix-like) systems would perform. These standards, formally known as IEEE 1003, define the *application programming interfaces (APIs)* and shell and utilities

that are to be found on a standard Unix-like system. The name “POSIX” which stands for *Portable Operating System Interface* (with the “X” added to the end for extra snappiness) was suggested by Richard Stallman (yes, *that* Richard Stallman) and was adopted by the IEEE.

Alternation

The first of the extended regular expression features we will discuss is called *alternation*, which is the facility that allows a match to occur from among a set of expressions. Just as a bracket expression allows a single character to match from a set of specified characters, alternation allows matches from a set of strings or other regular expressions.

To demonstrate, we’ll use `grep` in conjunction with `echo`. First, let’s try a plain old string match:

```
[me@linuxbox ~]$ echo "AAA" | grep AAA
AAA
[me@linuxbox ~]$ echo "BBB" | grep AAA
[me@linuxbox ~]$
```

A pretty straightforward example in which we pipe the output of `echo` into `grep` and see the results. When a match occurs, we see it printed out, when no match occurs, we see no results.

Now we’ll add alternation. The vertical bar meta character is used to signify it:

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB'
AAA
[me@linuxbox ~]$ echo "BBB" | grep -E 'AAA|BBB'
BBB
[me@linuxbox ~]$ echo "CCC" | grep -E 'AAA|BBB'
[me@linuxbox ~]$
```

Here we see the regular expression `'AAA|BBB'` which means “match either the string AAA or the string BBB.” Notice that since this is an extended feature we added the `-E` option to `grep` (though we could have just used the `egrep` program instead) and we enclosed the regular expression in single quotes to prevent the shell from interpreting the vertical bar meta character as a redirection operator. Alternation is not limited to two choices:

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB|CCC'
AAA
```

To combine alternation with other regular expression elements, we can use () to separate the alternation:

```
[me@linuxbox ~]$ grep -Eh '^(bz|gz|zip)' dirlist*.txt
```

This expression will match file name in our lists that starts with either “bz”, “gz”, or “zip”. Had we left off the parentheses:

```
[me@linuxbox ~]$ grep -Eh '^bz|gz|zip' dirlist*.txt
```

the meaning of this regular expression changes to match any file name that begins with “bz” or contains “gz” or contains “zip”.

Quantifiers

Extended regular expressions support several ways to specify the number of times an element is matched.

? - Match An Element Zero Or One Times

This quantifier means, in effect, “make the preceding element optional.” Let’s say we wanted to check a phone number for validity and we considered a phone number to be valid if it matched either of these two forms:

```
(nnn) nnn-nnnn
nnn nnn-nnnn
```

where “n” is a numeral. We could construct a regular expression like this:

```
^\(?:[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

In this expression, we follow the “(“ and “)” characters with question marks to indicate that they are to be matched zero or one times. Again, since the parentheses are normally meta characters (in ERE) , we precede them with backslashes to cause them to be treated as literals instead.

Let’s try it:

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)\)? [0-9][0-9][0-9]$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'
555 123-4567
[me@linuxbox ~]$ echo "AAA 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'
[me@linuxbox ~]$
```

Here we see that the expression matches both forms of the phone number, but does not match one containing non-numeric characters.

* - Match An Element Zero Or More Times

Like the `?` meta character, the `*` is used to denote an optional item, however, unlike the `?`, the item may occur any number of times, not just once. Let's say we wanted to see if a string was a sentence, that is, it starts with an uppercase letter, then contains any number of upper and lowercase letters and spaces and ends with a period. To match this (very crude) definition of a sentence, we could use a regular expression like this:

```
[:upper:][:upper:][:lower: ]*\.
```

The expression consists of three items, a bracket expression containing the `[:upper:]` character class, a bracket expression containing both the `[:upper:]` and `[:lower:]` character classes and a space, and a period escaped with a backslash. The second element is trailed with an `*` meta character, so that after the leading uppercase letter in our sentence, any number of upper and lowercase letters and spaces may follow it and still match:

```
[me@linuxbox ~]$ echo "This works." | grep -E '[:upper:][:upper:][:lower: ]*\.'
```

```
This works.
```

```
[me@linuxbox ~]$ echo "This Works." | grep -E '[:upper:][:upper:][:lower: ]*\.'
```

```
This Works.
```

```
[me@linuxbox ~]$ echo "this does not" | grep -E '[:upper:][:upper:][:lower: ]*\.'
```

```
[me@linuxbox ~]$
```

The expression matches the first two tests but not the third since it lacks the required leading uppercase character and trailing period.

+ - Match An Element One Or More Times

The + meta character works much like the * except it requires that there be at least one instance of the preceding element to cause a match. Here is a regular expression that will only match lines that consist of groups of one or more alphabetic characters separated by single spaces:

```
^([[[:alpha:]]+ ?)+$
```

```
[me@linuxbox ~]$ echo "This that" | grep -E '^([[[:alpha:]]+ ?)+$'
This that
[me@linuxbox ~]$ echo "a b c" | grep -E '^([[[:alpha:]]+ ?)+$'
a b c
[me@linuxbox ~]$ echo "a b 9" | grep -E '^([[[:alpha:]]+ ?)+$'
[me@linuxbox ~]$ echo "abc d" | grep -E '^([[[:alpha:]]+ ?)+$'
[me@linuxbox ~]$
```

We see that this expression does not match the line “a b 9” because it contains a non-alphabetic character, nor does it match “abc d” because more than one space character separates the characters “c” and “d”.

{ } - Match An Element A Specific Number Of Times

The { and } meta characters are used to express minimum and maximum numbers of required matches. They may be specified in four possible ways:

Specifier	Meaning
{ <i>n</i> }	Match the preceding element if it occurs exactly <i>n</i> times.
{ <i>n</i> , <i>m</i> }	Match the preceding element if it occurs at least <i>n</i> times but no more than <i>m</i> times.
{ <i>n</i> ,}	Match the preceding element if it occurs <i>n</i> or more times.
{, <i>m</i> }	Match the preceding element if it occurs no more than <i>m</i> times.

Going back to our earlier example with the phone numbers, we can use this method of specifying repetitions to simplify our original regular expression from:

```
^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

to:

```
^\(?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$
```

Let’s try it:

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^(\?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^(\?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$'
555 123-4567
[me@linuxbox ~]$ echo "5555 123-4567" | grep -E '^(\?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$'
[me@linuxbox ~]$
```

As we can see, our revised expression can successfully validate numbers both with and without the parentheses while rejecting those numbers that are not properly formatted.

Putting Regular Expressions To Work

Let's look at some of the commands we already know and see how they can be used with regular expressions.

Validating A Phone List With grep

In our earlier example, we looked at single phone numbers and checked them for proper formatting. A more realistic scenario, would be checking a list of numbers instead, so let's make a list. We'll do this by reciting a magical incantation to the command line. It will be magic because we have not covered most of the commands involved, but worry not. We will get there in future chapters. Here is the incantation:

```
[me@linuxbox ~]$ for i in $(seq 1 10); do echo "(${RANDOM:0:3}) ${RANDOM:0:3}-${RANDOM:0:4}" >> phonenumber.txt; done
```

This command will produce a file named `phonenumber.txt` containing ten phone numbers. Each time the command is repeated, another ten numbers are added to the list. We can also change the value `10` near the beginning of the command to produce more or fewer numbers. If we examine the contents of the file however, we see we have a problem:

```
[me@linuxbox ~]$ cat phonenumber.txt
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
```

```
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
```

Some of the numbers are malformed, which is perfect our purposes since we will use `grep` to validate them.

One useful method of validation would be to scan the file for invalid numbers and display the resulting list on the display:

```
[me@linuxbox ~]$ grep -Ev '^([0-9]{3}\) [0-9]{3}-[0-9]{4}$'
phonelist.txt
(292) 108-518
(129) 44-1379
[me@linuxbox ~]$
```

Here we use the `-v` option to produce an inverse match so that we will only output the lines in the list that do not match the specified expression. The expression itself includes the anchor meta characters at each end to insure that the number has no extra characters at either end. This expression also requires that the parentheses be present in a valid number unlike our earlier phone number example.

Finding Ugly Filenames With `find`

The `find` command supports a test based on a regular expression. There is an important consideration to keep in mind when using regular expressions in `find` versus `grep`. Whereas `grep` will print a line when the line *contains* a string that matches an expression, `find` requires that the pathname *exactly match* the regular expression. In the following example, we will use `find` with a regular expression to find every pathname that contains any character that is not a member of the following set:

```
[ -_./0-9a-zA-Z ]
```

Such a scan would reveal pathnames that contain embedded spaces and other potentially offensive characters:

```
[me@linuxbox ~]$ find . -regex '.*[^-_./0-9a-zA-Z].*'
```

Due to the requirement for an exact match of the entire pathname, we use `.*` at both ends

of the expression to match zero or more instances of any character. In the middle of the expression we use a negated bracket expression containing our set of acceptable pathname characters.

Searching For Files With `locate`

The `locate` program supports both basic (the `--regex` option) and extended (the `--regex` option) regular expressions. With it, we can perform many of the same operations that we performed earlier with our `dirlist` files:

```
[me@linuxbox ~]$ locate --regex 'bin/(bz|gz|zip)'  
/bin/bzcat  
/bin/bzcmp  
/bin/bzdiff  
/bin/bzegrep  
/bin/bzexe  
/bin/bzfgrep  
/bin/bzgrep  
/bin/bzip2  
/bin/bzip2recover  
/bin/bzless  
/bin/bzmore  
/bin/gzexe  
/bin/gzip  
/usr/bin/zip  
/usr/bin/zipcloak  
/usr/bin/zipgrep  
/usr/bin/zipinfo  
/usr/bin/zipnote  
/usr/bin/zipsplit
```

Using alternation, we perform a search for pathnames that contain either `bin/bz`, `bin/gz`, or `bin/zip`.

Searching For Text In `less` And `vim`

`less` and `vim` both share the same method of searching for text. Pressing the `/` key followed by a regular expression will perform a search. If we use `less` to view our `phonelist.txt` file:

```
[me@linuxbox ~]$ less phonelist.txt
```

Then search for our validation expression:

```
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
~
~
~
/^([0-9]{3}\) [0-9]{3}-[0-9]{4}$
```

less will highlight the strings that match leaving the invalid ones easy to spot:

```
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
~
~
~
(END)
```

vim, on the other hand, supports basic regular expressions, so our search expression would look like this:

```
/([0-9]\{3\}) [0-9]\{3\}-[0-9]\{4\}
```

As we can see, the expression is mostly the same, however, many of the characters that are considered meta characters in extended expressions are considered literals in basic expressions. They are only treated as meta characters when escaped with a backslash. Depending on the particular configuration of vim on our system, the matching will be highlighted. If not, try this command mode command:

```
:hlsearch
```

to activate search highlighting.

Note: Depending on your distribution, `vim` may or may not support text search highlighting. Ubuntu, in particular, supplies a very stripped-down version of `vim` by default. On such systems, you may want to use your package manager to install a more complete version of `vim`.

Summing Up

In this chapter, we've seen a few of the many uses of regular expressions. We can find even more if we use regular expressions to search for additional applications that use them. We can do that by searching the man pages:

```
[me@linuxbox ~]$ cd /usr/share/man/man1
[me@linuxbox man1]$ zgrep -El 'regex|regular expression' *.gz
```

The `zgrep` program provides a front end for `grep` allowing it to read compressed files. In our example, we search the compressed section one man page files located in their usual location. The result of this command is a list of files containing either the string “regex” or “regular expression”. As we can see, regular expressions show up in a lot of programs.

There was one feature found in basic regular expressions that we did not cover. Called *back references*, these will be discussed in the next chapter.

Further Reading

There are many online resources for learning regular expressions including various tutorials and cheat sheets.

In addition, the Wikipedia has good articles on the following background topics:

POSIX: <http://en.wikipedia.org/wiki/Posix>

ASCII: <http://en.wikipedia.org/wiki/Ascii>

21 - Text Processing

All Unix-like operating systems rely heavily on text files for many types of data storage. So it makes sense that there would be many tools for manipulating text and indeed there are. In this chapter, we will look at a group of programs that are used to “slice and dice” text. In the next chapter we will again look at text processing, but we will look at those programs that are used to format text for printing and other kinds of human consumption.

This chapter will revisit some old friends and introduce us to some new ones:

- `cat` - Concatenate files and print on the standard output
- `sort` - Sort lines of text files
- `uniq` - Report or omit repeated lines
- `cut` - Remove sections from each line of files
- `paste` - Merge lines of files
- `join` - Join lines of two files on a common field
- `comm` - Compare two sorted files line by line
- `diff` - Compare files line by line
- `patch` - Apply a diff file to an original
- `tr` - Translate or delete characters
- `sed` - Stream editor for filtering and transforming text
- `aspell` - Interactive spell checker

Applications Of Text

So far, we have learned a couple of text editors (`nano` and `vim`), looked a bunch of configuration files, and have witnessed the output of dozens of commands, all in text. But what else is text used for? For many things, it turns out.

Documents

Many people write documents using plain text formats. While it is easy to see how a small text file could be useful for keeping simple notes, it is also possible to write large documents in text format as well. One popular approach is to write a large document in a text format and then use a *markup language* to describe the formatting of the finished document. Many scientific papers are written using this method as Unix-based text processing systems were among the first systems that supported advanced typographical layout needed by writers in technical disciplines.

Web Pages

The world's most popular type of electronic document is probably the web page. Web pages are text documents that use either *HTML (Hypertext Markup Language)* or *XML (Extensible Markup Language)* as markup languages to describe the document's visual format.

Email

Email is an intrinsically text medium. Even non-text attachments are converted into a text representation for transmission. We can see this for ourselves by downloading an email message and then viewing it in `less`. We will see that the message begins with a *header* that describes the source of the message and the various processing it received during its journey followed by the *body* of the message with its content.

Printer Output

On Unix-like systems, output destined for a printer is sent as plain text or (if the page contains graphics) is converted into a text format *page description language* known as *PostScript* which is then sent to a program that generates the graphic dots to be printed.

Program Source Code

Many of the command line programs found on Unix-like systems were created to support system administration and software development and the text processing programs are no exception. Many of them are designed to solve software development problems. The reason why text processing is important to software developers is that all software starts out as text. *Source code*, the part of the program the programmer actually writes, is always in text format.

Revisiting Some Old Friends

Back in Chapter 7 (Redirection), we learned about some commands that are able to

accept standard input in addition to command line arguments. We only touched on them briefly then but now we will take a closer look at how they can be used to perform text processing.

cat

The `cat` program has a number of interesting options. Many of them are used to help better visualize text content. One example is the `-A` option which is used to display non-printing characters in the text. There are cases when we want to know if control characters are embedded in our otherwise visible text. The most common of these are tab characters (as opposed to spaces) and carriage returns often present as end-of-line characters in MS-DOS style text files. Another common situation is a file containing lines of text with trailing spaces.

Let's create a test file using `cat` as primitive word processor. We just enter the command `cat` (along with specifying a file for redirected output) and begin typing our text. When we are finished, we type `Ctrl-d` to indicate to `cat` that we have reached end-of-file. In this example, we enter a leading tab character and follow the line with some trailing spaces:

```
[me@linuxbox ~]$ cat > foo.txt
    The quick brown fox jumped over the lazy dog.
[me@linuxbox ~]$
```

Next, we will use `cat` with the `-A` option to display the text:

```
[me@linuxbox ~]$ cat -A foo.txt
^IThe quick brown fox jumped over the lazy dog.  $
[me@linuxbox ~]$
```

As we can see in the results, the tab character in our text is represented by `^I`. This is a common notation that means "Control-I" which, as it turns out, is the same as a tab character. We also see that a `$` appears at the true end of the line indicating that our text contains trailing spaces.

MS-DOS Text Vs. Unix Text

One of the reasons you may want to use `cat` to look for non-printing characters in text is to spot hidden carriage returns. Where do hidden carriage returns come from? DOS and Windows! Unix and DOS don't define the end of a line the same way in text files. Unix ends a line with a linefeed character (ASCII 10) while MS-DOS and its derivatives use the sequence carriage return (ASCII 13) and linefeed to terminate each line of text.

There are several ways to convert files from DOS to Unix format. On many Linux systems a program called `dos2unix` is provided which will do the job. However if you don't have `dos2unix` on your system, don't worry. The process of converting from DOS to Unix format is very simple. It only involves the removal of the offending carriage returns and that is easily accomplished by a couple of the programs discussed later in this chapter.

`cat` also has options that are used to modify text. The two most prominent are the `-n` which numbers lines, and `-s` which suppresses the output of multiple blank lines. We can demonstrate thusly:

```
[me@linuxbox ~]$ cat > foo.txt
The quick brown fox

jumped over the lazy dog.
[me@linuxbox ~]$ cat -ns foo.txt
 1      The quick brown fox
 2
 3      jumped over the lazy dog.
[me@linuxbox ~]$
```

In this example, we create a new version of our `foo.txt` test file containing two lines of text separated by two blank lines. After processing by `cat` with the `-ns` options, the extra blank line is removed and the remaining lines are numbered. While this is not much of a process to perform on text, it is a process.

sort

The `sort` program sorts the contents of standard input or one or more files specified on the command line and sends the results to standard output. Using the same technique that we used with `cat`, we can demonstrate processing of standard input directly from the

keyboard:

```
[me@linuxbox ~]$ sort > foo.txt
c
b
a
[me@linuxbox ~]$ cat foo.txt
a
b
c
```

After entering the command, we type the letters “c”, “b”, and “a” followed once again with Ctrl-d to indicate end-of-file. We then view the resulting file and see that the lines now appear in sorted order.

Since `sort` can accept multiple files on the command line as arguments, it possible to *merge* multiple files into a single sorted whole. For example, if we had three text files and wanted to combine them into a single sorted file, we could do something like this:

```
sort file1.txt file2.txt file3.txt > final_sorted_list.txt
```

`sort` has several interesting options. Here is a partial list:

Option	Long Option	Description
-b	--ignore-leading-blanks	By default, sorting is performed based on the entire line starting with the first character in the line. This option causes sort to ignore leading spaces in lines and calculates sorting based on the first non-whitespace character on the line.
-f	--ignore-case	Makes sorting case insensitive.
-n	--numeric-sort	Performs sorting based on the numeric evaluation of a string. Using this option allows sorting to be performed on numeric values rather than alphabetic values.
-r	--reverse	Sort in reverse order. Results are in descending rather than

		ascending order.
-k	--key= <i>field1</i> [, <i>field2</i>]	Sort based on a key field located from <i>field1</i> to <i>field2</i> rather than the entire line. See discussion below.
-m	--merge	Treat each each argument as the name of a presorted file. Merge multiple files into a single sorted result without performing any additional sorting.
-o	--output= <i>file</i>	Send sorted output to <i>file</i> rather than standard output.
-t	--field-separator= <i>char</i>	Define the field separator character. By default fields are separated by spaces or tabs.

Most of the options above are pretty self-explanatory, however some are not. First let's look at the `-n` option used for numeric sorting. Using this option it is possible to sort values based on numeric values. We can demonstrate this by sorting the results of the `du` command to determine the largest users of disk space. Normally, the `du` command lists the results of a summary in pathname order:

```
[me@linuxbox ~]$ du -s /usr/share/* | head
252      /usr/share/aclocal
96       /usr/share/acpi-support
8        /usr/share/adduser
196     /usr/share/albumart
344     /usr/share/alsa
8       /usr/share/alsa-base
12488   /usr/share/anthony
8       /usr/share/apmd
21440   /usr/share/app-install
48      /usr/share/application-registry
```

In this example, we pipe the results into `tail` to limit the results to the first ten lines. We can produce a numerically sorted list to show the ten largest consumers of space this way:

```
[me@linuxbox ~]$ du -s /usr/share/* | sort -nr | head
509940   /usr/share/locale-langpack
```

```

242660    /usr/share/doc
197560    /usr/share/fonts
179144    /usr/share/gnome
146764    /usr/share/myspell
144304    /usr/share/gimp
135880    /usr/share/dict
76508     /usr/share/icons
68072     /usr/share/apps
62844     /usr/share/foomatic

```

By using the `-nr` options, we produce a reverse numerical sort with the largest values appearing first in the results. This sort works because the numerical values occur at the beginning of each line. But what if we want to sort a list based on some value found within the line? For example, the results of an `ls -l`:

```

[me@linuxbox ~]$ ls -l /usr/bin | head
total 152948
-rwxr-xr-x 1 root  root    34824 2008-04-04 02:42 [
-rwxr-xr-x 1 root  root   101556 2007-11-27 06:08 a2p
-rwxr-xr-x 1 root  root    13036 2008-02-27 08:22 aconnect
-rwxr-xr-x 1 root  root    10552 2007-08-15 10:34 acpi
-rwxr-xr-x 1 root  root     3800 2008-04-14 03:51 acpi_fakekey
-rwxr-xr-x 1 root  root     7536 2008-04-19 00:19 acpi_listen
-rwxr-xr-x 1 root  root     3576 2008-04-29 07:57 addpart
-rwxr-xr-x 1 root  root    20808 2008-01-03 18:02 addr2line
-rwxr-xr-x 1 root  root   489704 2008-10-09 17:02 adept_batch

```

Ignoring, for the moment, that `ls` can sort its results by size, we could use `sort` to sort this list by file size as well:

```

[me@linuxbox ~]$ ls -l /usr/bin | sort -nr -k 5 | head
-rwxr-xr-x 1 root  root   8234216 2008-04-07 17:42 inkscape
-rwxr-xr-x 1 root  root   8222692 2008-04-07 17:42 inkview
-rwxr-xr-x 1 root  root   3746508 2008-03-07 23:45 gimp-2.4
-rwxr-xr-x 1 root  root   3654020 2008-08-26 16:16 quanta
-rwxr-xr-x 1 root  root   2928760 2008-09-10 14:31 gdbtui
-rwxr-xr-x 1 root  root   2928756 2008-09-10 14:31 gdb
-rwxr-xr-x 1 root  root   2602236 2008-10-10 12:56 net
-rwxr-xr-x 1 root  root   2304684 2008-10-10 12:56 rpcclient
-rwxr-xr-x 1 root  root   2241832 2008-04-04 05:56 aptitude
-rwxr-xr-x 1 root  root   2202476 2008-10-10 12:56 smbcacls

```

Many uses of `sort` involve the processing of *tabular data* such as the results of the `ls`

command above. If we apply database terminology to the table above we would say that each row is a *record* and that each record consists of multiple *fields* such as the file attributes, link count, file name, file size and so on. `sort` is able to process individual fields. In database terms, we are able to specify one or more *key fields* to use as *sort keys*. In the example above, we specify the `n` and `r` options to perform a reverse numerical sort and specify `-k 5` to cause `sort` to use the fifth field as the key for sorting.

The `k` option is very interesting and has many features, but first we need to talk about how `sort` defines fields. Let's consider a very simple text file consisting of a single line containing the author's name:

```
William    Shotts
```

By default, `sort` sees this line as having two fields. The first field contains the characters:

```
"William"
```

and the second field contains the characters:

```
"    Shotts"
```

meaning that whitespace characters (spaces and tabs) are used as delimiters between fields and that the delimiters are included in the field when sorting is performed.

Looking again at a line from our `ls` output, we can see that a line contains eight fields and that the fifth field is the file size:

```
-rwxr-xr-x 1 root  root  8234216 2008-04-07 17:42 inkscape
```

For our next series of experiments, let's consider the following file containing the history of three popular Linux distributions released from 2006 to 2008. Each line in the file has three fields, the distribution name, version number, and date of release in MM/DD/YYYY format:

```
SUSE      10.2  12/07/2006
Fedora    10     11/25/2008
SUSE      11.0   06/19/2008
Ubuntu    8.04   04/24/2008
Fedora    8      11/08/2007
SUSE      10.3   10/04/2007
Ubuntu    6.10   10/26/2006
```

```
Fedora      7      05/31/2007
Ubuntu      7.10   10/18/2007
Ubuntu      7.04   04/19/2007
SUSE        10.1   05/11/2006
Fedora      6       10/24/2006
Fedora      9       05/13/2008
Ubuntu      6.06   06/01/2006
Ubuntu      8.10   10/30/2008
Fedora      5       03/20/2006
```

Using a text editor (vim?), we'll enter this data and name the resulting file `distros.txt`.

Next, we'll try sorting the file and observe the results:

```
[me@linuxbox ~]$ sort distros.txt
Fedora      10     11/25/2008
Fedora      5       03/20/2006
Fedora      6       10/24/2006
Fedora      7       05/31/2007
Fedora      8       11/08/2007
Fedora      9       05/13/2008
SUSE        10.1   05/11/2006
SUSE        10.2   12/07/2006
SUSE        10.3   10/04/2007
SUSE        11.0   06/19/2008
Ubuntu      6.06   06/01/2006
Ubuntu      6.10   10/26/2006
Ubuntu      7.04   04/19/2007
Ubuntu      7.10   10/18/2007
Ubuntu      8.04   04/24/2008
Ubuntu      8.10   10/30/2008
```

Well, it mostly worked. The problem is the sorting of the Fedora version numbers. Since a "1" comes before a "5" in the character set, version "10" ends up at the top while version "9" falls to the bottom.

To fix this problem we are going to have to sort on multiple keys. We want to perform an alphabetic sort on the first field and then a numeric sort on the third field. `sort` allows multiple instances of the `-k` option so that multiple sort keys can be specified. In fact, a key may include a range of fields. If no range is specified (as has been the case with our previous examples), `sort` uses a key that begins with the specified field and extends to the end of the line. Here is the syntax for our multi-key sort:

```
[me@linuxbox ~]$ sort --key=1,1 --key=2n distros.txt
Fedora      5      03/20/2006
Fedora      6      10/24/2006
Fedora      7      05/31/2007
Fedora      8      11/08/2007
Fedora      9      05/13/2008
Fedora     10     11/25/2008
SUSE       10.1   05/11/2006
SUSE       10.2   12/07/2006
SUSE       10.3   10/04/2007
SUSE       11.0   06/19/2008
Ubuntu     6.06   06/01/2006
Ubuntu     6.10   10/26/2006
Ubuntu     7.04   04/19/2007
Ubuntu     7.10   10/18/2007
Ubuntu     8.04   04/24/2008
Ubuntu     8.10   10/30/2008
```

We use the long form of the option for interest, `-k 1,1 -k 2n` would be exactly equivalent. In the first instance of the key option, we specify a range of fields to include in the first key. Since we want to limit the sort to just the first field, we specified `1,1` which means, start at field one and end at field one. In the second instance, we specify `2n` which means that field two is sort key and that the sort should be numeric. An option letter may be included at the end of a key specifier to specify the type of sort to be performed. These option letters are the same as the global options for the `sort` program, `b` (ignore leading blanks), `n` (numeric sort), `r` (reverse sort), and so on.

The third field in our list contains a date in an inconvenient format for sorting. On computers, dates are usually formatted in `YYYY-MM-DD` order to make chronological sorting easy but ours are in the American format of `MM/DD/YYYY`. How can we sort this list in chronological order?

Fortunately, `sort` provides a way. The key option allows specification of *offsets* within fields so we can define keys within fields:

```
[me@linuxbox ~]$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt
Fedora      10     11/25/2008
Ubuntu     8.10   10/30/2008
SUSE       11.0   06/19/2008
Fedora      9      05/13/2008
Ubuntu     8.04   04/24/2008
Fedora      8      11/08/2007
Ubuntu     7.10   10/18/2007
SUSE       10.3   10/04/2007
Fedora      7      05/31/2007
```


Ubuntu	7.04	04/19/2007
SUSE	10.2	12/07/2006
Ubuntu	6.10	10/26/2006
Fedora	6	10/24/2006
Ubuntu	6.06	06/01/2006
SUSE	10.1	05/11/2006
Fedora	5	03/20/2006

By specifying `-k 3.7` we instruct `sort` to use a sort key that begins at the seventh character within the third field which corresponds to the start of the year. Likewise, we specify `-k 3.1` and `-k 3.4` to isolate the month and day portions of the date. We also add the `n` and `r` options to achieve a reverse numeric sort. The `b` option is included to suppress the leading spaces (whose numbers vary from line to line, thereby affecting the outcome of the sort) in the date field.

Some files that we will encounter don't use tabs and spaces as field delimiters, for example the `/etc/passwd` file:

```
[me@linuxbox ~]$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
```

The fields in this file are delimited with colons (:), so how would we sort this file using a key field? `sort` provides the `-t` option to define the field separator character. To sort the `passwd` file on the seventh field (the account's default shell) we could do this:

```
[me@linuxbox ~]$ sort -t ':' -k 7 /etc/passwd | head
me:x:1001:1001:Myself,,,:/home/me:/bin/bash
root:x:0:0:root:/root:/bin/bash
dhcp:x:101:102:./nonexistent:/bin/false
gdm:x:106:114:Gnome Display Manager:/var/lib/gdm:/bin/false
hplip:x:104:7:HPLIP system user,,,:/var/run/hplip:/bin/false
klog:x:103:104:./home/klog:/bin/false
messagebus:x:108:119:./var/run/dbus:/bin/false
polkituser:x:110:122:PolicyKit,,,:/var/run/PolicyKit:/bin/false
```

```
pulse:x:107:116:PulseAudio daemon,,,:/var/run/pulse:/bin/false
```

By specifying the colon character as the field separator, we can sort on the seventh field.

uniq

Compared to `sort`, the `uniq` program is a lightweight. `uniq` performs a seemingly trivial task. When given a sorted file (including standard input), it removes any duplicate lines and sends the results to standard output. It is often used in conjunction with `sort` to clean the output of duplicates.

Tip: While `uniq` is a traditional Unix tool often used with `sort`, the GNU version of `sort` supports a `-u` option which removes duplicates from the sorted output.

Let's make a text file to try this out:

```
[me@linuxbox ~]$ cat > foo.txt
a
b
c
a
b
c
```

Remember to type `Ctrl-d` to terminate standard input. Now, if we run `uniq` on our text file:

```
[me@linuxbox ~]$ uniq foo.txt
a
b
c
a
b
c
```

The results are no different from our original file, the duplicates were not removed. For `uniq` to actually do its job, its input must be sorted first:

```
[me@linuxbox ~]$ sort foo.txt | uniq
```

```
a
b
c
```

This is because `uniq` only removes duplicate lines which are adjacent to each other.

`uniq` has several options. Here are the common ones:

Option	Description
<code>-c</code>	Output a list of duplicate lines preceded by the number of times the line occurs.
<code>-d</code>	Only output repeated lines rather than unique lines.
<code>-f n</code>	Ignore <i>n</i> leading fields in each line. Fields are separated by whitespace as they are in <code>sort</code> , however unlike <code>sort</code> , <code>uniq</code> has no option for setting an alternate field separator.
<code>-i</code>	Ignore case during the line comparisons.
<code>-s n</code>	Skip (ignore) the leading <i>n</i> characters of each line.
<code>-u</code>	Only output unique lines. This is the default.

Here we see `uniq` used to report the number of duplicates found in our text file using the `-c` option:

```
[me@linuxbox ~]$ sort foo.txt | uniq -c
  2 a
  2 b
  2 c
```

Slicing And Dicing

The next three programs we will discuss are used to peel columns of text out of files and recombine them in useful ways.

cut

The `cut` program is used to extract a section of text from a line and output the extracted section to standard output. It can accept multiple file arguments or input from standard input.

Specifying the section of the line to be extracted is somewhat awkward and is specified using the following options:

Option	Description
<code>-c char_list</code>	Extract the portion of the line defined by <i>char_list</i> . The list may consist of one or more comma-separated character ranges.
<code>-f field_list</code>	Extract one or more fields from the line as defined by <i>field_list</i> . The list may contain one or more fields or field ranges separated by commas.
<code>-d delim_char</code>	When <code>-f</code> is specified, use <i>delim_char</i> as the field delimiting character. By default, fields must be separated by a single tab character.
<code>--complement</code>	Extract the entire line of text except for those portions specified by <code>-c</code> and/or <code>-f</code> .

As we can see, the way `CUT` extracts text is rather brittle and `CUT` is best used to extract text from files that are produced by other programs rather than text directly typed by humans. We'll take a look at our `distros.txt` file to see if it is "clean" enough to be a good specimen for our `CUT` examples. If we use `CAT` with the `-A` option, we can see if the file meets our requirements of tab separated fields:

```
[me@linuxbox ~]$ cat -A distros.txt
SUSE^I10.2^I12/07/2006$
Fedora^I10^I11/25/2008$
SUSE^I11.0^I06/19/2008$
Ubuntu^I8.04^I04/24/2008$
Fedora^I8^I11/08/2007$
SUSE^I10.3^I10/04/2007$
Ubuntu^I6.10^I10/26/2006$
Fedora^I7^I05/31/2007$
Ubuntu^I7.10^I10/18/2007$
Ubuntu^I7.04^I04/19/2007$
SUSE^I10.1^I05/11/2006$
Fedora^I6^I10/24/2006$
Fedora^I9^I05/13/2008$
Ubuntu^I6.06^I06/01/2006$
Ubuntu^I8.10^I10/30/2008$
Fedora^I5^I03/20/2006$
```

It looks good. No embedded spaces, just single tab characters between the fields. Since

the file uses tabs rather than spaces, we'll use the `-f` option to extract a field:

```
[me@linuxbox ~]$ cut -f 3 distros.txt
12/07/2006
11/25/2008
06/19/2008
04/24/2008
11/08/2007
10/04/2007
10/26/2006
05/31/2007
10/18/2007
04/19/2007
05/11/2006
10/24/2006
05/13/2008
06/01/2006
10/30/2008
03/20/2006
```

Since our `distros` file is tab-delimited, it is best to use `CUT` to extract fields rather than characters. This is because when a file is tab-delimited, it is unlikely that each line will contain the same number of characters and this makes calculating character positions within the line difficult or impossible. In our example above however, we now have extracted a field that luckily contains data of identical length so we can show how character extraction works by extracting the year from each line:

```
[me@linuxbox ~]$ cut -f 3 distros.txt | cut -c 7-10
2006
2008
2008
2008
2007
2007
2006
2007
2007
2007
2006
2006
2008
2006
2008
2006
```

By running `cut` a second time on our list, we are able to extract character positions seven through 10 which corresponds to the year in our date field. The `7-10` notation is an example of a range. The `cut` man page contains a complete description of how these can be specified.

Expanding Tabs

Our `distros.txt` file is ideally formatted for extracting fields using `cut`. But what if we wanted a file that could be fully manipulated with `cut` by characters rather than fields. This would require that we replace the tab characters within the file with the corresponding number of spaces. Fortunately, the GNU Coreutils package includes a tool for that. Named `expand`, this program accepts either one or more file arguments or standard input and outputs the modified text to standard output.

If we process our `distros.txt` file with `expand`, we can use the `cut -c` to extract any range of characters from the file. For example we could use the following command to extract the year of release from our list by expanding the file and using `cut` to extract every character from the twenty-third position to the end of the line:

```
[me@linuxbox ~]$ expand distros.txt | cut -c 23-
```

Coreutils also provides the `unexpand` program to substitute tabs for spaces.

When working with fields, it is possible to specify a different field delimiter rather than the tab character. Here we will extract the first field from the `/etc/passwd` file:

```
[me@linuxbox ~]$ cut -d ':' -f 1 /etc/passwd | head
root
daemon
bin
sys
sync
games
man
lp
mail
news
```

Using the `-d` option we are able to specify the colon character as the field delimiter.

paste

The `paste` command does the opposite of `cut`. Rather than extracting a column of text from a file, it adds one or more columns of text to a file. It does this by reading multiple files and combining the fields found in each file into a single stream on standard output. Like `cut`, `paste` accepts multiple file arguments and/or standard input. To demonstrate how `paste` operates, we will perform some surgery on our `distros.txt` file to produce a chronological list of releases.

From our earlier work with `sort`, we will first produce a list of distros sorted by date and store the result in a file called `distros-by-date.txt`:

```
[me@linuxbox ~]$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt > distros-by-date.txt
```

Next, we will use `cut` to extract the first two fields from the file (the distro name and version) and store that result in a file named `distro-versions.txt`:

```
[me@linuxbox ~]$ cut -f 1,2 distros-by-date.txt > distros-versions.txt
[me@linuxbox ~]$ head distros-versions.txt
Fedora      10
Ubuntu     8.10
SUSE       11.0
Fedora      9
Ubuntu     8.04
Fedora      8
Ubuntu     7.10
SUSE       10.3
Fedora      7
Ubuntu     7.04
```

The final piece of preparation is to extract the release dates and store them a file named `distro-dates.txt`:

```
[me@linuxbox ~]$ cut -f 3 distros-by-date.txt > distros-dates.txt
[me@linuxbox ~]$ head distros-dates.txt
11/25/2008
10/30/2008
06/19/2008
05/13/2008
04/24/2008
```

```
11/08/2007
10/18/2007
10/04/2007
05/31/2007
04/19/2007
```

We now have the parts we need. The final step is to use `paste` to put the column of dates ahead of the distro names and versions thus creating a chronological list. This is done simply by using `paste` and ordering its arguments in the desired arrangement:

```
[me@linuxbox ~]$ paste distros-dates.txt distros-versions.txt
11/25/2008 Fedora      10
10/30/2008 Ubuntu      8.10
06/19/2008 SUSE        11.0
05/13/2008 Fedora      9
04/24/2008 Ubuntu      8.04
11/08/2007 Fedora      8
10/18/2007 Ubuntu      7.10
10/04/2007 SUSE        10.3
05/31/2007 Fedora      7
04/19/2007 Ubuntu      7.04
12/07/2006 SUSE        10.2
10/26/2006 Ubuntu      6.10
10/24/2006 Fedora      6
06/01/2006 Ubuntu      6.06
05/11/2006 SUSE        10.1
03/20/2006 Fedora      5
```

join

In some ways, `join` is like `paste` in that it adds columns to a file, but it uses a unique way to do it. A *join* is an operation usually associated with *relational databases* where data from multiple *tables* is combined to form a desired result. The `join` program performs the same operation. It joins data from multiple files based on a shared key field.

To see how a join operation is used in a relational database, let's image a very small database consisting of two tables each containing a single record. The first table, called CUSTOMERS, has three fields: a customer number (CUSTNUM), the customer's first name (FNAME) and the customer's last name (LNAME):

CUSTNUM	FNAME	LNAME
=====	=====	=====

4681934 John Smith

The second table is called ORDERS and contains four fields: an order number (ORDERNUM), the customer number (CUSTNUM), the quantity (QUAN), and the item ordered (ITEM).

ORDERNUM	CUSTNUM	QUAN	ITEM
=====	=====	=====	=====
3014953305	4681934	1	Blue Widget

Note that both tables share the field CUSTNUM. This is important as it allows a relationship between the tables.

Performing a join operation would allow us to combine the fields in the two tables to achieve a useful result such as preparing an invoice. Using the matching values in the CUSTNUM fields of both tables, a join operation could produce the following:

FNAME	LNAME	QUAN	ITEM
=====	=====	=====	=====
John	Smith	1	Blue Widget

To demonstrate the `join` program, we'll need to make a couple of files with a shared key. To do this we will use our `distros-by-date.txt` file. From this, we will construct two files, one containing the release date (which will be our shared key for this demonstration) and the release name:

```
[me@linuxbox ~]$ cut -f 1,1 distros-by-date.txt > distros-names.txt
[me@linuxbox ~]$ paste distros-dates.txt distros-names.txt > distros-
key-names.txt
[me@linuxbox ~]$ head distros-key-names.txt
11/25/2008 Fedora
10/30/2008 Ubuntu
06/19/2008 SUSE
05/13/2008 Fedora
04/24/2008 Ubuntu
11/08/2007 Fedora
10/18/2007 Ubuntu
10/04/2007 SUSE
05/31/2007 Fedora
04/19/2007 Ubuntu
```

and the second file which contains the release dates and the version numbers:

```
[me@linuxbox ~]$ cut -f 2,2 distros-by-date.txt > distros-vernums.txt
[me@linuxbox ~]$ paste distros-dates.txt distros-vernums.txt > distros-key-vernums.txt
[me@linuxbox ~]$ head distros-key-vernums.txt
11/25/2008 10
10/30/2008 8.10
06/19/2008 11.0
05/13/2008 9
04/24/2008 8.04
11/08/2007 8
10/18/2007 7.10
10/04/2007 10.3
05/31/2007 7
04/19/2007 7.04
```

We now have two files with a shared key. It is important to point out that the files must be sorted on the key field for `join` to work properly.

```
[me@linuxbox ~]$ join distros-key-names.txt distros-key-vernums.txt
| head
11/25/2008 Fedora 10
10/30/2008 Ubuntu 8.10
06/19/2008 SUSE 11.0
05/13/2008 Fedora 9
04/24/2008 Ubuntu 8.04
11/08/2007 Fedora 8
10/18/2007 Ubuntu 7.10
10/04/2007 SUSE 10.3
05/31/2007 Fedora 7
04/19/2007 Ubuntu 7.04
```

Note also that, by default, `join` uses whitespace as the input field delimiter and a single space as the output field delimiter. This behavior can be modified by specifying options. See the `join` man page for details.

Comparing Text

It is often useful to compare versions of text files. For system administrators and software developers, this is particularly important. A system administrator may, for example need to compare an existing configuration file to a previous version to diagnose a system problem. Likewise, a programmer frequently needs to see what changes have been made to programs over time.

COMM

The `comm` program compares two text files and displays what lines are unique to each one and which lines they have in common. To demonstrate, we will create two nearly identical text files using `cat`:

```
[me@linuxbox ~]$ cat > file1.txt
a
b
c
d
[me@linuxbox ~]$ cat > file2.txt
b
c
d
e
```

Next, we will compare the two files using `comm`:

```
[me@linuxbox ~]$ comm file1.txt file2.txt
a
      b
      c
      d
e
```

As we can see, `comm` produces three columns of output. The first column contains lines unique to the first file argument, the second column, the lines unique to the second file argument. The third column contains the lines shared by both files. `comm` supports options in the form `-n` where `n` is either 1, 2 or 3. When used, these options specify which column(s) to suppress. For example, if we only wanted to output the lines shared by both files, we would suppress the output of columns one and two:

```
[me@linuxbox ~]$ comm -12 file1.txt file2.txt
b
c
d
```

diff

Like the `comm` program, `diff` is used to detect the differences between files. However

`diff` is a much more complex tool supporting many output formats and the ability to process large collections of text files at once. `diff` is often used by software developers to examine changes between different versions of program source code and thus has the ability to recursively examine directories of source code often referred to as *source trees*. One common use for `diff` is the creation of *diff files* or *patches* that are used by programs such as `patch` (which we'll discuss shortly) to convert one version of a file (or files) to another version.

If we use `diff` to look at our previous example files:

```
[me@linuxbox ~]$ diff file1.txt file2.txt
1d0
< a
4a4
> e
```

we see its default style of output, a terse description of the differences between the two files. In the default format, each group of changes is preceded by a *change command* in the form of *range operation range* to describe the positions and type of changes required to convert the first file to the second file:

Change	Description
<i>r1ar2</i>	Append the lines at the position <i>r2</i> in the second file to the position <i>r1</i> in the first file.
<i>r1cr2</i>	Change (replace) the lines at position <i>r1</i> with the lines at the position <i>r2</i> in the second file.
<i>r1dr2</i>	Delete the lines in the first file at position <i>r1</i> which would have appeared at range <i>r2</i> in the second file

In this format, a range is a comma separated list of the starting line and the ending line. While this format is the default (mostly for POSIX compliance and backward compatibility with traditional Unix versions of `diff`) it is not as widely used as other, optional formats. Two of the more popular formats are the *context format* and the *unified format*.

When viewed using the context format (the `-C` option) we will see this:

```
[me@linuxbox ~]$ diff -c file1.txt file2.txt
*** file1.txt      2008-12-23 06:40:13.000000000 -0500
--- file2.txt      2008-12-23 06:40:34.000000000 -0500
```

```

*****
*** 1,4 ****
- a
  b
  c
  d
--- 1,4 ----
  b
  c
  d
+ e

```

The output begins with the names of the two files and their timestamps. The first file is marked with asterisks and the second file is marked with dashes. Throughout the remainder of the listing these markers will signify their respective files. Next we see groups of changes including the default number of surrounding context lines. In the first group we see:

```
*** 1,4 ***
```

which indicates lines one through four in the first file. Later we see:

```
--- 1,4 ---
```

which indicates lines one through four in the second file. Within a change group, lines begin with one of four indicators:

Indicator	Meaning
blank	A line shown for context. It does not indicate a difference between the two files.
-	A line deleted. This line will appear in the first file but not in the second file.
+	A line added. This line will appear in the second file but not in the first file.
!	A line changed. The two versions of the line will be displayed each in their respective sections of the change group.

The unified format is similar to the context format but more concise. It is specified with the `-u` option:

```

[me@linuxbox ~]$ diff -u file1.txt file2.txt
--- file1.txt      2008-12-23 06:40:13.000000000 -0500

```

```
+++ file2.txt      2008-12-23 06:40:34.000000000 -0500
@@ -1,4 +1,4 @@
-a
 b
 c
 d
+e
```

The most notable difference between the context and unified formats is the elimination of the duplicated lines of context making the results of the unified format shorter than the context format. In our example above, we see file timestamps like those of the context format followed by the string @@ -1,4 +1,4 @@. This indicates the lines in the first file and the lines in the second file described in the change group. Following this are the lines themselves with the default three lines of context. Each line starts with one of three possible characters:

Character	Meaning
blank	This line is shared by both files.
-	This line was removed from the first file.
+	This line was added to the first file.

patch

The patch program is used to apply changes to text files. It accepts output from diff and is generally used to convert older version of files into newer versions. Let's consider a famous scenario. The Linux kernel is developed by a large, loosely organized team of contributors who submit a constant stream of small changes to the source code. The kernel consists of several million lines of code while the changes that are made by one contributor at one time are quite small. To pass these changes around, it makes no sense for a contributor to send each developer an entire kernel source tree each time a small change is made. Instead a diff file is submitted that contains the change from the previous version of the kernel to the new version with contributor's changes. The receiver then uses the patch program to apply the change to their own source tree. Using diff/patch offers two significant advantages:

1. The diff file is very small compared to the full size of the source tree.
2. The diff file concisely shows the change being made allowing reviewers of the patch to quickly evaluate it.

Of course, diff/patch will work on any text file, not just source code. It would be equally applicable to configuration files or any other text.

To prepare a diff file for use with `patch`, the GNU documentation (see Further Reading below) suggests using `diff` as follows:

```
diff -Naur old_file new_file > diff_file
```

Where `old_file` and `new_file` are either single files or are directories containing files. The `r` option supports recursion of a directory tree.

Once the diff file has been created, we can apply it to patch the old file into the new file:

```
patch < diff_file
```

Using our test file, we can demonstrate:

```
[me@linuxbox ~]$ diff -Naur file1.txt file2.txt > patchfile.txt
[me@linuxbox ~]$ patch < patchfile.txt
patching file file1.txt
[me@linuxbox ~]$ cat file1.txt
b
c
d
e
```

In this example we created a diff file named `patchfile.txt` and then used the `patch` program to apply the patch. Note that we did not have to specify a target file to `patch` as the diff file (in unified format) already contains the file names in the header. Once the patch is applied, we can see that `file1.txt` now matches `file2.txt`.

`patch` has a large number of options and there are additional utility programs that can be used to analyze and edit patches.

Editing On The Fly

Our experience with text editors has been largely *interactive*, meaning that we manually move a cursor around then type our changes. However, there are *non-interactive* ways to edit text as well. It's possible, for example, to apply a set of changes to multiple files with a single command.

tr

The `tr` program is used to *transliterate* characters. We can think of this a sort of character-based search-and-replace operation. Transliteration is the process of changing characters from one alphabet to another. For example, converting characters from lowercase to uppercase is transliteration. We can perform such a conversion with `tr` as follows:

```
[me@linuxbox ~]$ echo "lowercase letters" | tr a-z A-Z
LOWERCASE LETTERS
```

As we can see, `tr` operates on standard input and outputs its results on standard output. `tr` accepts two arguments: a set of characters to convert from and a corresponding set of characters to convert to. Character sets may be expressed in one of three ways:

1. An enumerated list. For example: ABCDEFGHIJKLMNOPQRSTUVWXYZ
2. A character range. For example A-Z. Note that this method is sometimes subject to the same issues as other commands due to the locale collation order and thus should be used with caution.
3. POSIX character classes. For example `[:upper:]`.

In most cases, both character sets should be of equal length, however it is possible for the first set to be larger than the second particularly in the case where we wish to convert multiple characters to a single character:

```
[me@linuxbox ~]$ echo "lowercase letters" | tr [:lower:] A
AAAAAAAAA AAAAAAA
```

In addition to transliteration, `tr` allows characters to be simply deleted from the input stream. Earlier in this chapter we discussed the problem of converting MS-DOS text files to Unix style text. To perform this conversion, carriage return characters need to be removed from the end of each line. This can be performed with `tr` as follows:

```
tr -d '\r' < dos_file > unix_file
```

where *dos_file* is the file to be converted and *unix_file* is the result. This form of the command uses the escape sequence `\r` to represent the carriage return character. To see a complete list of the sequences and character classes `tr` supports try:

```
[me@linuxbox ~]$ tr --help
```

ROT13

One amusing use of `tr` is to perform *ROT13 encoding* of text. ROT13 is a trivial type of encryption based on a simple substitution cipher. Calling ROT13

encryption is being generous, “text obfuscation” is more accurate. It is used sometimes in email and such to obscure potentially offensive content. The method simply moves each character thirteen places up the alphabet. Since this is half way up the possible twenty-six characters, performing the algorithm a second time on the text restores it to its original form. To perform this encoding with `tr`:

```
echo "secret text" | tr a-zA-Z n-za-mN-ZA-M
frperg grkg
```

Performing the same procedure a second time results in the translation:

```
echo "frperg grkg" | tr a-zA-Z n-za-mN-ZA-M
secret text
```

A number of email programs and USENET news readers support ROT13 encoding. As usual, the Wikipedia contains a good article on the subject:

<http://en.wikipedia.org/wiki/ROT13>

`tr` can perform another trick, too. Using the `-s` option, `tr` can “squeeze” (delete) repeated instances of a character:

```
[me@linuxbox ~]$ echo "aaabbbccc" | tr -s ab
abccc
```

Here we have a string containing repeated characters. By specifying the set “ab” to `tr`, we eliminate the repeated instances of the letters in the set while leaving the character missing from the set (“c”) unchanged. Note that the repeating characters must be adjoining. If they are not:

```
[me@linuxbox ~]$ echo "abcabcabc" | tr -s ab
abcabcabc
```

the squeezing will have no effect.

sed

The name `sed` is short for *stream editor*. It performs text editing on a stream of text, either a set of specified files or standard input. `sed` is a powerful and somewhat complex program (there are entire books about it) and we will not completely cover it here.

In general, the way that `sed` works is that it is given either a single editing command (on the command line) or the name of a script file containing multiple commands and it then performs these commands upon each line in the stream of text. Here is a very simple example of `sed` in action:

```
[me@linuxbox ~]$ echo "front" | sed 's/front/back/'  
back
```

In this example, we produce a one word stream of text using `echo` and pipe it into `sed`. `sed`, in turn, carries out the instruction `s/front/back/` upon the text in the stream and produces the output “back” as a result. We can also recognize this command as resembling the “substitution” (search and replace) command in `vi`. In fact, it resembles it a lot.

Commands in `sed` each begin with a single letter. In the example above the substitution command is represented by the letter `s` and is followed by the search and replace strings separated by the slash character as a delimiter. The choice of the delimiter character is arbitrary. By convention, the slash character is often used but `sed` will accept any character that immediately follows the command as the delimiter. We could perform the same command this way:

```
[me@linuxbox ~]$ echo "front" | sed 's_ front_ back_ '  
back
```

By using the underscore character immediately after the command, it becomes the delimiter. The ability to set the delimiter can be used to make commands more readable, as we shall see.

Most commands in `sed` may be preceded by an *address* which specifies which line(s) of the input stream is to be edited. If the address is omitted, then the editing command is carried out on every line in the input stream. The simplest form of address is a line number. We can add one to our example:

```
[me@linuxbox ~]$ echo "front" | sed '1s/front/back/'  
back
```

Adding the address `1` to our command causes our substitution to be performed on the first line of our one-line input stream. If we specify another number:

```
[me@linuxbox ~]$ echo "front" | sed '2s/front/back/'
front
```

we would see that the editing is not carried out since our input stream does not have a line two.

Addresses may be expressed in many ways. Here are the most common:

Address	Description
<i>n</i>	A line number where <i>n</i> is a positive integer.
\$	The last line.
<i>/regexp/</i>	A POSIX basic regular expression. Note that the regular expression is delimited by slash characters. Optionally, the regular expression may be delimited by an alternate character of one's own choosing by specifying the expression with <code>\cregexpc</code> , where <i>c</i> is the alternate character.
<i>addr1,addr2</i>	A range of lines from <i>addr1</i> to <i>addr2</i> inclusive. Addresses may be any of the single address forms above.
<i>first~step</i>	Match the line represented by the number <i>first</i> , then each subsequent line at <i>step</i> intervals. For example <code>1~2</code> refers to each odd numbered line, <code>5~5</code> refers to the fifth line and every fifth line thereafter.
<i>addr1,+n</i>	Match <i>addr1</i> and the following <i>n</i> lines.
<i>addr!</i>	Match all lines except <i>addr</i> which may be any of the forms above.

We'll demonstrate different kinds of addresses using the `distros.txt` file from earlier in this chapter. First, a range of line numbers:

```
[me@linuxbox ~]$ sed -n '1,5p' distros.txt
SUSE      10.2  12/07/2006
Fedora    10    11/25/2008
SUSE      11.0  06/19/2008
Ubuntu    8.04  04/24/2008
Fedora    8     11/08/2007
```

In this example, we print the a range of lines starting with line one and continuing until line five. To do this, we use the `p` command which causes a matched line to be simply

printed. For this to be effective however, we must include the option `-n` (the no auto-print option) to cause `sed` not to print every line by default.

Next, we'll try a regular expression:

```
[me@linuxbox ~]$ sed -n '/SUSE/p' distros.txt
SUSE      10.2  12/07/2006
SUSE      11.0  06/19/2008
SUSE      10.3  10/04/2007
SUSE      10.1  05/11/2006
```

By including the slash-delimited regular expression `/SUSE/`, we are able isolate the lines containing it in much the same manner as `grep`.

Finally, we'll try negation by adding an `!` to the address:

```
[me@linuxbox ~]$ sed -n '/SUSE/!p' distros.txt
Fedora    10    11/25/2008
Ubuntu    8.04  04/24/2008
Fedora    8     11/08/2007
Ubuntu    6.10  10/26/2006
Fedora    7     05/31/2007
Ubuntu    7.10  10/18/2007
Ubuntu    7.04  04/19/2007
Fedora    6     10/24/2006
Fedora    9     05/13/2008
Ubuntu    6.06  06/01/2006
Ubuntu    8.10  10/30/2008
Fedora    5     03/20/2006
```

Here we see the expected result, all of the lines in the file except the ones matched by the regular expression.

So far, we've looked at two of the `sed` editing commands, `s` and `p`. Here is a more complete list of the basic editing commands:

Command	Description
<code>=</code>	Output current line number.
<code>d</code>	Delete the current line.
<code>p</code>	Print the current line. By default, <code>sed</code> prints every line and only edits lines that match a specified address within the file. The default

	behavior can be overridden by specifying the <code>-n</code> option.
<code>q</code>	Exit <code>sed</code> without processing any more lines. If the <code>-n</code> option is not specified, output the current line.
<code>Q</code>	Exit <code>sed</code> without processing any more lines.
<code>s/regexp/replacement/</code>	Substitute the contents of <i>replacement</i> wherever <i>regexp</i> is found. <i>replacement</i> may include the special character <code>&</code> which is equivalent to the text matched by <i>regexp</i> . In addition, <i>replacement</i> may include the sequences <code>\1</code> through <code>\9</code> which are the contents of the corresponding subexpressions in <i>regexp</i> . For more about this, see the discussion of <i>back references</i> below. After the trailing slash following <i>replacement</i> , an optional flag may be specified to modify the <code>S</code> command's behavior.
<code>y/set1/set2</code>	Perform transliteration converting characters from <i>set1</i> to the corresponding characters in <i>set2</i> . Note that unlike <code>tr</code> , <code>sed</code> requires that both sets be of the same length.

The `s` command is by far the most commonly used editing command. We will demonstrate just some of its power by performing an edit on our `distros.txt` file. We discussed before how the date field in `distros.txt` was not in a “computer-friendly” format. While the date is formatted `MM/DD/YYYY`, it would be better (for ease of sorting) if the format were `YYYY-MM-DD`. To perform this change on the file by hand would be both time-consuming and error prone, but with `sed`, this change can be performed in one step:

```
[me@linuxbox ~]$ sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)\
)$/\3-\1-\2/' distros.txt
SUSE      10.2  2006-12-07
Fedora    10     2008-11-25
SUSE      11.0  2008-06-19
Ubuntu    8.04  2008-04-24
Fedora    8      2007-11-08
SUSE      10.3  2007-10-04
Ubuntu    6.10  2006-10-26
Fedora    7      2007-05-31
```

Ubuntu	7.10	2007-10-18
Ubuntu	7.04	2007-04-19
SUSE	10.1	2006-05-11
Fedora	6	2006-10-24
Fedora	9	2008-05-13
Ubuntu	6.06	2006-06-01
Ubuntu	8.10	2008-10-30
Fedora	5	2006-03-20

Wow! Now that is an ugly looking command. But it works. In one step we have changed the date format in our file. This is an example of why regular expressions are sometimes jokingly referred to as a “write-only” medium. We can write them but we sometimes cannot read them. Before we are tempted to run away in terror from this command, let’s look at how it was constructed. First, we know that the command will have this basic structure:

```
sed 's/regexp/replacement/' distros.txt
```

Our next step is to figure out a regular expression that will isolate the date. Since it is in MM/DD/YYYY format and appears at the end of the line, we can use an expression like this:

```
[0-9]{2}/[0-9]{2}/[0-9]{4}$
```

which matches two digits, a slash, two digits, a slash, four digits, and the end of line. So that takes care of *regexp*, but what about *replacement*? To handle that, we must introduce a new regular expression feature that appears in some applications that use BRE. This feature is called *back references* and works like this: if the sequence `\n` appears in *replacement* where *n* is a number from one to nine, the sequence will refer to the corresponding subexpression in the preceding regular expression. To create the subexpressions, we simply enclose them in parentheses like so:

```
([0-9]{2})/([0-9]{2})/([0-9]{4})$
```

We now have three subexpressions. The first contains the month, the second contains the day of the month, and the third contains the year. Now we can construct *replacement* as follows:

```
\3-\1-\2
```

which gives us the year, a dash, the month, a dash, and the day.

Now, our command looks like this:

```
sed 's/([0-9]{2})/([0-9]{2})/([0-9]{4})$/\3-\1-\2/' distros.txt
```

We only have two remaining problems. The first is the extra slashes in our regular expression will confuse `sed` when it tries to interpret the `S` command, and the second is that since `sed`, by default, accepts only basic regular expressions, several of the characters in our regular expression will be taken as literals rather than meta characters as we intend. We can solve both these problems with a liberal application of backslashes to escape the offending characters:

```
sed 's/\([0-9]\{2\}\)\(\([0-9]\{2\}\)\)\(\([0-9]\{4\}\)\)$/\3-\1-\2/' distros.txt
```

And there you have it!

Another feature of the `S` command is the optional flags that may follow the replacement string. The most important of these is the `g` flag which instructs `sed` to apply the search and replace globally to a line and not just to the first instance as is the default. Here is an example:

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/'
aaaBbbccc
```

We see that the replacement was performed, but only to the first instance of the letter “b” while the remaining instances were left unchanged. By adding the `g` flag, we are able to change all the instances:

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/g'
aaaBBBccc
```

So far, we have only given `sed` single commands via the command line. It is also possible to construct more complex commands in a script file using the `-f` option. To

demonstrate, we will use `sed` with our `distros.txt` file to build a report. Our report will feature a title at the top, our modified dates, and all the distribution names converted to upper case. To do this we will need to write a script, so we'll fire up our text editor and enter the following:

```
# sed script to produce Linux distributions report

1 i\  
\  
Linux Distributions Report\  
  
s/\([0-9]\{2\}\)\.\(\([0-9]\{2\}\)\.\(\([0-9]\{4\}\)\)\$/\3-\1-\2/  
y/abcdefghijklmnopqrstuvwxyZ/ABCDEFHIJKLMNOPQRSTUVWXYZ/
```

We will save our `sed` script as `distros.sed` and run it like this:

```
[me@linuxbox ~]$ sed -f distros.sed distros.txt

Linux Distributions Report

SUSE      10.2  2006-12-07
FEDORA    10    2008-11-25
SUSE      11.0  2008-06-19
UBUNTU    8.04  2008-04-24
FEDORA    8     2007-11-08
SUSE      10.3  2007-10-04
UBUNTU    6.10  2006-10-26
FEDORA    7     2007-05-31
UBUNTU    7.10  2007-10-18
UBUNTU    7.04  2007-04-19
SUSE      10.1  2006-05-11
FEDORA    6     2006-10-24
FEDORA    9     2008-05-13
UBUNTU    6.06  2006-06-01
UBUNTU    8.10  2008-10-30
FEDORA    5     2006-03-20
```

As we can see, our script produces the desired results, but how does it do it? Let's take another look at our script. We'll use `cat` to number the lines:

```
[me@linuxbox ~]$ cat -n distros.sed
 1 # sed script to produce Linux distributions report
 2
```



```
3 1 i\  
4 \  
5 Linux Distributions Report\  
6  
7 s/\([0-9]\{2\}\)\(\([0-9]\{2\}\)\)\(\([0-9]\{4\}\)\)$/\3-\1-\2/  
8 y/abcdefghijklmnopqrstuvwxyZ/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

Line one of our script is a *comment*. Like many configuration files and programming languages on Linux systems, comments begin with the # character and are followed by human-readable text. Comments can be placed anywhere in the script (however not within commands themselves) and are helpful to any humans who might need to identify and/or maintain the script.

Line two is a blank line. Like comments, blank lines may be added to improve readability.

Lines three through six contain text to be inserted at the address 1, the first line of the input. The `i` command is followed by the sequence backslash-carriage return to produce an escaped carriage return or what is called a *line continuation character*. This sequence, which can be used in many circumstances including shell scripts, allows a carriage return to be embedded in a stream of text without signaling the interpreter (in this case `sed`) that the end of the line has been reached. The `i` and likewise the `a` and `c` commands allow multiple lines of text so long as each line, except the last, ends with a line continuation character. The sixth line of our script is actually the end of our inserted text and ends with a plain carriage return rather than a line continuation character, signaling the end of the inserted text.

Note: A line continuation character is formed by a backslash followed *immediately* by a carriage return. No intermediary spaces are permitted.

Line seven is our search and replace command. Since it is not preceded by an address, each line in the input stream is subject to its action.

Line eight performs transliteration of the lowercase letters into uppercase letters. Note that unlike `tr`, the `y` command in `sed` does not support character ranges (for example, `[a-z]`) nor does it support POSIX character classes. Again, since the `y` command is not preceded by an address, it applies to every line in the input stream.

People Who Like sed Also Like...

`sed` is a very capable program able to perform fairly complex editing tasks to streams of text though it is most often used for simple one line tasks rather than long scripts. Many users prefer other tools for larger tasks. The most popular of these are `awk` and `perl`. These go beyond mere tools like the programs covered here and extend into the realm of complete programming languages. `perl`, in particular, is often used in place of shell scripts for many system management and administration tasks, as well as, being a very popular medium for web development. `awk` is a little more specialized. Its specific strength is its ability to manipulate tabular data. In some ways it resembles `sed` in that `awk` programs normally processes text files line-by-line using a scheme similar to the `sed` concept of an address followed by an action. While both `awk` and `perl` are outside the scope of this book, they are very good skills for the Linux command line user.

aspell

The last tool we will look at is `aspell`, an interactive spelling checker. The `aspell` program is the successor to an earlier program named `ispell` and can be used, for the most part, as a drop-in replacement. While the `aspell` program is mostly used by other programs that require spell checking capability, it can also be used very effectively as a stand alone tool from the command line. It has the ability to intelligently check various type of text files including HTML documents, C/C++ programs, email messages and other kinds of specialized texts.

To spell check a text file containing simple prose, it could be used like this:

```
aspell check textfile
```

where *textfile* is the name of the file to check. As practical example, let's create a simple text file named `foo.txt` containing some deliberate spelling errors:

```
[me@linuxbox ~]$ cat > foo.txt  
The quick brown fox jimped over the laxy dog.
```

Next we'll check the file using `aspell`:

```
[me@linuxbox ~]$ aspell check foo.txt
```

As `aspell` is interactive in the check mode, we will see a screen like this:

```
The quick brown fox jimped over the laxy dog.
1) jumped          6) wimped
2) gimped          7) camped
3) comped         8) humped
4) limped         9) impede
5) pimped        0) umped
i) Ignore        I) Ignore all
r) Replace      R) Replace all
a) Add         l) Add Lower
b) Abort       x) Exit
?
```

At the top of the display we see our text with a suspiciously spelled word highlighted. In the middle, we see ten spelling suggestions numbered zero through nine followed by a list of other possible actions. Finally, at the very bottom we see a prompt ready to accept our choice.

If we press the `1` key, `aspell` replaces the offending word with the word “jumped” and moves on to the next misspelled word which is “laxy”. If we select the replacement “lazy”, `aspell` replaces it and terminates. Once `aspell` has finished, we can examine our file and see that the misspellings have been corrected:

```
[me@linuxbox ~]$ cat foo.txt
The quick brown fox jumped over the lazy dog.
```

Unless told otherwise via the command line option `--dont-backup`, `aspell` creates a backup file containing the original text by appending the extension `.bak` to the file name.

Showing off our `sed` editing prowess, we’ll put our spelling mistakes back in so we can reuse our file:

```
[me@linuxbox ~]$ sed -i 's/lazy/laxy/; s/jumped/jimped/' foo.txt
```

The `sed` option `-i` tells `sed` to edit the file “in-place” meaning that rather than sending the edited output to standard output, it will re-write the file with the changes applied. We also see demonstrated here the ability to place more than one editing command on the line by separating them with a semicolon.

Next, we’ll look at how `aspell` can handle different kinds of text files. Using a text editor (such as `vim`, but the adventurous may want to try `sed`) we will add some HTML markup to our file:

```
<html>
  <head>
    <title>Mispelled HTML file</title>
  </head>
  <body>
    <p>The quick brown fox jumped over the laxy dog.</p>
  </body>
</html>
```

Now, if we try to spell check our modified file, we run into a problem. If we do it this way:

```
[me@linuxbox ~]$ aspell check foo.txt
```

we’ll get this:

```
<html>
  <head>
    <title>Mispelled HTML file</title>
  </head>
  <body>
    <p>The quick brown fox jumped over the laxy dog.</p>
  </body>
</html>
████████████████████████████████████████████████████████████████████████████████
1) HTML          4) Hamel
2) ht ml        5) Hamil
3) ht-ml       6) hotel

i) Ignore       I) Ignore all
r) Replace      R) Replace all
a) Add         1) Add Lower
```

```
b) Abort                                     x) Exit
████████████████████████████████████████████
?
```

`aspell` will see the contents of the HTML tags as misspelled. This problem can be overcome by including the `-H` (HTML) checking mode option like this:

```
[me@linuxbox ~]$ aspell -H check foo.txt
```

which will result in this:

```
<html>
  <head>
    <title>Mispelled HTML file</title>
  </head>
  <body>
    <p>The quick brown fox jumped over the laxy dog.</p>
  </body>
</html>
████████████████████████████████████████████
1) Mi spelled          6) Misapplied
2) Mi-spelled         7) Miscalled
3) Misspelled         8) Respelled
4) Dispelled          9) Misspell
5) Spelled            0) Misled
i) Ignore             I) Ignore all
r) Replace            R) Replace all
a) Add                l) Add Lower
b) Abort              x) Exit
████████████████████████████████████████████
?
```

The HTML is ignored and only the non-markup portions of the file are checked. In this mode the contents of HTML tags are ignored and not checked for spelling. However, the contents of ALT tags, which benefit from checking, are checked in this mode.

Note: By default, `aspell` will ignore URLs and email addresses in text. This behavior can be overridden with command line options. It is also possible to set which markup tags are checked and skipped. See the `aspell` man page for details.

Summing Up

In this chapter, we have looked at a few of the many command line tools that operate on text. In the the next chapter, we will look a several more. Admittedly, it may not seem immediately obvious how or why you might use some of these tools on a day-to-day basis, though we have tried to show some semi-practical examples of their use. What we will find in later chapters is that these tools form the basis of a tool set that is used to solve a host of practical problems. This will be particularly true when we get into shell scripting, where these tools will really show their worth.

Further Reading

The GNU Project website contains many online guides to the tools discussed in this chapter.

From the Coreutils package:

<http://www.gnu.org/software/coreutils/manual/coreutils.html#Output-of-entire-files>

<http://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-sorted-files>

<http://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-fields-within-a-line>

<http://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-characters>

From the Diffutils package:

http://www.gnu.org/software/diffutils/manual/html_mono/diff.html

sed:

<http://www.gnu.org/software/sed/manual/sed.html>

aspell:

<http://aspell.net/man-html/index.html>

There are many other online resources for sed, in particular:

<http://www.grymoire.com/Unix/Sed.html>

<http://sed.sourceforge.net/sed1line.txt>

Also try googling “sed one liners”, “sed cheat sheets”

Extra Credit

There are a few more interesting text manipulation commands worth investigating. Among these are: `split` (split files into pieces), `csplit` (split files into pieces based on context), and `sdiff` (side-by-side merge of file differences.)

22 - Formatting Output

We continue our look at text related tools in this chapter, focusing on programs that are used to format output rather than changing the text itself. These tools are often used to prepare text for eventual printing, a subject that we will cover in the next chapter. The programs that we will cover in this chapter include:

- `nl` – Number lines
- `fold` – Wrap each line to a specified length
- `fmt` – A simple text formatter
- `pr` – Prepare text for printing
- `printf` – Format and print data
- `groff` – A document formatting system

Simple Formatting Tools

We'll look at some of the simple formatting tools first. These programs are mostly single purpose, and a bit unsophisticated in what they do, but can be used for small tasks and parts of multiple process pipelines and scripts.

`nl` – Number Lines

The `nl` program is a rather arcane tool used to perform a simple task. It numbers lines. In its simplest use, it resembles `cat -n`:

```
[me@linuxbox ~]$ nl distros.txt | head
1      SUSE      10.2  12/07/2006
2      Fedora   10     11/25/2008
3      SUSE      11.0  06/19/2008
4      Ubuntu   8.04  04/24/2008
5      Fedora   8      11/08/2007
6      SUSE      10.3  10/04/2007
7      Ubuntu   6.10  10/26/2006
```

8	Fedora	7	05/31/2007
9	Ubuntu	7.10	10/18/2007
10	Ubuntu	7.04	04/19/2007

Like `cat`, `n1` can accept either multiple files as command line arguments or standard input. However, `n1` has a number of options and supports a primitive form of markup to allow more complex kinds of numbering.

`n1` supports a concept called “logical pages” when numbering. This allows `n1` to reset (start over) the numerical sequence when numbering. Using options, it is possible to set the starting number to a specific value and, to a limited extent, its format. A logical page is further broken down into a header, body, and footer. Within each of these sections, line numbering may be reset and/or its style changed. If `n1` is given multiple files, it treats them as a single stream of text. Sections in the text stream are indicated by the presence of some rather odd-looking markup added to the text:

Markup	Meaning
<code>\:\:\:</code>	Start of logical page header
<code>\:\:</code>	Start of logical page body
<code>\:</code>	Start of logical page footer

Each of the above markup elements must appear on alone on its own line. When encountered by `n1`, each markup element, and the line it occurs on, is deleted from the text stream.

Here are the common options for `n1`:

Option	Meaning
<code>-b style</code>	Set body numbering to <i>style</i> , where <i>style</i> is one of the following: <code>a</code> = number all lines <code>t</code> = number only non-blank lines. This is the default. <code>n</code> = none <code>pregexp</code> = number only lines matching basic regular expression <i>regexp</i> .
<code>-f style</code>	Set footer numbering to <i>style</i> . Default is <code>n</code> (none).
<code>-h style</code>	Set header numbering to <i>style</i> . Default is <code>n</code> (none).
<code>-i number</code>	Set page numbering increment to <i>number</i> . Default is one.
<code>-n format</code>	Sets numbering format to <i>format</i> , where <i>format</i> is: <code>ln</code> = left justified, without leading zeros. This is the default.

	rn = right justified without leading zeros rz = right justified with leading zeros
-p	Do not reset page numbering at the beginning of each logical page.
-s <i>string</i>	Add <i>string</i> to the end of each line number to create a separator. Default is a single tab character.
-v <i>number</i>	Set first line number of each logical page to <i>number</i> . Default is one.
-w <i>width</i>	Set width of the line number field to <i>width</i> . Default is six.

Admittedly, we probably won't be numbering lines that often, but we can use `n1` to again look at how we can combine multiple tools to perform more complex tasks. We will build on our work in the previous chapter to produce a Linux distributions report. Since we will be using `n1`, it will be useful to include its header/body/footer markup. To do this, we will add it to the `sed` script from the last chapter. Using our text editor, we will change the script as follows and save it as `distros-n1.sed`:

```
# sed script to produce Linux distributions report

1 i\
\\:\\:\\:\
\
Linux Distributions Report\
\
Name      Ver.  Released\
----      ----  -\
\\:\\:
s/\([0-9]\{2\}\)\v\([0-9]\{2\}\)\v\([0-9]\{4\}\)$/\3-\1-\2/
$ a\
\\:\
\
End Of Report
```

The script now inserts the `n1` logical page markup and adds a footer at the end of the report. Note that we had to double up the backslashes in our markup because they are normally interpreted as a quoting character (they escape things) by `sed`.

Next, we'll produce our enhanced report by combining `sort`, `sed`, and `n1`:

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-n1.sed | n1
```

```
Linux Distributions Report

Name      Ver.      Released
----      -
1  Fedora  5         2006-03-20
2  Fedora  6         2006-10-24
3  Fedora  7         2007-05-31
4  Fedora  8         2007-11-08
5  Fedora  9         2008-05-13
6  Fedora  10        2008-11-25
7  SUSE     10.1      2006-05-11
8  SUSE     10.2      2006-12-07
9  SUSE     10.3      2007-10-04
10 SUSE     11.0      2008-06-19
11 Ubuntu  6.06      2006-06-01
12 Ubuntu  6.10      2006-10-26
13 Ubuntu  7.04      2007-04-19
14 Ubuntu  7.10      2007-10-18
15 Ubuntu  8.04      2008-04-24
16 Ubuntu  8.10      2008-10-30

End Of Report
```

Our report is the result of our pipeline of commands. First, we sort the list by distribution name and version (fields one and two), then we process the results with `sed`, adding the report header (including the logical page markup for `n1`) and footer. Finally, we process the result with `n1`, which, by default, only numbers the lines of the text stream that belong to the body section of the logical page.

We can repeat the command and experiment with different options for `n1`. Some interesting ones are:

```
n1 -n rz
```

and

```
n1 -w 3 -s ' '
```

fold – Wrap Each Line To A Specified Length

Folding is the process of breaking lines of text at a specified width. Like our other commands, `fold` accepts either one or more text files or standard input. If we send `fold` a simple stream of text, we can see how it works:

```
[me@linuxbox ~]$ echo "The quick brown fox jumped over the lazy dog."
| fold -w 12
The quick br
own fox jump
ed over the
lazy dog.
```

Here we see `fold` in action. The text sent by the `echo` command is broken into segments specified by the `-w` option. In this example we specify a line width of twelve characters. If no width is specified, the default is eighty characters. Notice how the lines are broken regardless of word boundaries. The addition of the `-s` option will cause `fold` to break the line at the last available space before the line width is reached:

```
[me@linuxbox ~]$ echo "The quick brown fox jumped over the lazy dog."
| fold -w 12 -s
The quick
brown fox
jumped over
the lazy
dog.
```

fmt – A Simple Text Formatter

The `fmt` program folds text too, and a lot more. It accepts either files or standard input and performs paragraph formatting on the text stream. Basically it fills and joins lines in text while preserving blank lines and indentation.

To demonstrate, we'll need some text. Let's lift some from the `fmt` info page:

```
`fmt' reads from the specified FILE arguments (or standard input
if none are given), and writes to standard output.
```

```
By default, blank lines, spaces between words, and indentation are
```

```
preserved in the output; successive input lines with different
indentation are not joined; tabs are expanded on input and introduced
on output.
```

```
`fmt' prefers breaking lines at the end of a sentence, and tries
to avoid line breaks after the first word of a sentence or before the
last word of a sentence. A "sentence break" is defined as either the
end of a paragraph or a word ending in any of `?.!', followed by two
spaces or end of line, ignoring any intervening parentheses or
quotes. Like TeX, `fmt' reads entire "paragraphs" before choosing
line breaks; the algorithm is a variant of that given by Donald E.
Knuth and Michael F. Plass in "Breaking Paragraphs Into Lines",
`Software--Practice & Experience' 11, 11 (November 1981), 1119-1184.
```

We'll copy this text into our text editor and save the file as `fmt-info.txt`. Now, let's say we wanted to reformat this text to fit a fifty character wide column. We could do this by processing the file with `fmt` and the `-w` option:

```
[me@linuxbox ~]$ fmt -w 50 fmt-info.txt | head
`fmt' reads from the specified FILE arguments
(or standard input if
none are given), and writes to standard output.
```

```
By default, blank lines, spaces between words,
and indentation are
preserved in the output; successive input lines
with different indentation are not joined; tabs
are expanded on input and introduced on output.
```

Well, that's an awkward result. Perhaps we should actually read this text, since it explains what's going on:

“By default, blank lines, spaces between words, and indentation are preserved in the output; successive input lines with different indentation are not joined; tabs are expanded on input and introduced on output.”

So, `fmt` is preserving the indentation of the first line. Fortunately, `fmt` provides an option to correct this:

```
[me@linuxbox ~]$ fmt -cw 50 fmt-info.txt
`fmt' reads from the specified FILE arguments
(or standard input if none are given), and writes
to standard output.
```

By default, blank lines, spaces between words, and indentation are preserved in the output; successive input lines with different indentation are not joined; tabs are expanded on input and introduced on output.

``fmt'` prefers breaking lines at the end of a sentence, and tries to avoid line breaks after the first word of a sentence or before the last word of a sentence. A "sentence break" is defined as either the end of a paragraph or a word ending in any of ``.?!'`, followed by two spaces or end of line, ignoring any intervening parentheses or quotes. Like TeX, ``fmt'` reads entire "paragraphs" before choosing line breaks; the algorithm is a variant of that given by Donald E. Knuth and Michael F. Plass in "Breaking Paragraphs Into Lines", *'Software--Practice & Experience'* 11, 11 (November 1981), 1119-1184.

Much better. By adding the `-C` option, we now have the desired result.

`fmt` has some interesting options:

Option	Description
<code>-C</code>	Operate in <i>crown margin</i> mode. This preserves the indentation of the first two lines of a paragraph. Subsequent lines are aligned with the indentation of the second line.
<code>-p string</code>	Only format those lines beginning with the prefix <i>string</i> . After formatting, the contents of <i>string</i> are prefixed to each reformatted line. This option can be used to format text in source code comments. For example, any programming language or configuration file that uses a <code>"#"</code> character to delineate a comment could be formatted by specifying <code>-p '#'</code> and only the comments will be formatted. See the example below.
<code>-S</code>	Split-only mode. In this mode, lines will only be split if needed to fit the specified column width. Short lines will not be joined to fill lines. This mode is useful when formatting text such as code where joining is not desired.
<code>-u</code>	Perform uniform spacing. This will apply traditional "typewriter style" formatting to the text. This means a single space between words and two spaces between sentences. This mode is useful for removing "justification," that is, text that has been padded with

22 - Formatting Output

	spaces to force alignment on both the left and right margins.
<code>-w <i>width</i></code>	Format text to fit within a column <i>width</i> characters wide. The default is 75 characters.

The `-p` option is particularly interesting. With it, we can format selected portions of a file provided that the lines to be formatted all begin with the same sequence of characters. Many programming languages use the pound sign (`#`) to indicate the beginning of a comment and thus can be formatted using this option. Let's create a file that simulates a program that uses comments:

```
[me@linuxbox ~]$ cat > fmt-code.txt
# This file contains code with comments.

# This line is a comment.
# Followed by another comment line.
# And another.

This, on the other hand, is a line of code.
And another line of code.
And another.
```

Our sample file contains comments which begin the string `#` (a `#` followed by a space) and lines of "code" which do not. Now, using `fmt`, we can format the comments and leave the code untouched:

```
[me@linuxbox ~]$ fmt -w 50 -p '# ' fmt-code.txt
# This file contains code with comments.

# This line is a comment. Followed by another
# comment line. And another.

This, on the other hand, is a line of code.
And another line of code.
And another.
```

Notice how the adjoining comment lines are joined while the blank lines and the lines that do not begin with the specified prefix are preserved.

`pr` – Format Text For Printing

The `pr` program is used to *paginate* text. When printing text, it is often desirable to

separate the pages of output with several lines of whitespace to provide a top and bottom margin for each page. Further, this whitespace can be used to insert a header and footer on each page.

We'll demonstrate `pr` by formatting our `distros.txt` file into a series of very short pages (only the first two pages shown):

```
[me@linuxbox ~]$ pr -l 15 -w 65 distros.txt

2008-12-11 18:27                distros.txt                Page 1

SUSE      10.2  12/07/2006
Fedora    10    11/25/2008
SUSE      11.0  06/19/2008
Ubuntu    8.04  04/24/2008
Fedora     8     11/08/2007

2008-12-11 18:27                distros.txt                Page 2

SUSE      10.3  10/04/2007
Ubuntu    6.10  10/26/2006
Fedora    7     05/31/2007
Ubuntu    7.10  10/18/2007
Ubuntu    7.04  04/19/2007
```

In this example we employ the `-l` option (for page length) and the `-w` option (page width) to define a “page” that is sixty-five columns wide and fifteen lines long. `pr` paginates the contents of the `distros.txt` file and separates each page with several lines of whitespace and creates a default header containing the file modification time, file name, and page number. The `pr` program provides many options to control page layout. We'll take a look at more of them when we come back to `pr` in the next chapter.

`printf` – Format And Print Data

Unlike the other commands in this chapter, the `printf` command is not used for pipelines (it does not accept standard input) nor does it find frequent application directly

on the command line (it's mostly used in scripts), so why is it important? Because it is so widely used.

`printf` (from the phrase, “print formatted”) was originally developed for the C programming language and has been implemented in many programming languages including the shell. In fact, in `bash`, `printf` is a builtin.

`printf` works like this:

`printf "format" arguments`

The command is given a string containing a format description which is then applied to a list of argument. The formatted result is sent to standard output. Here is a trivial example:

```
[me@linuxbox ~]$ printf "I formatted the string: %s\n" foo
I formatted the string: foo
```

The format string may contain literal text (like “I formatted the string:”), escape sequences (such as `\n`, a newline character), and sequences beginning with the `%` character which are called *conversion specifications*. In the above example, the conversion specification `%S` is used to format the string “foo” and place it in command’s output. Here it is again:

```
[me@linuxbox ~]$ printf "I formatted: %s as a string.\n" foo
I formatted: foo as a string.
```

As we can see, the `%S` conversion specification is replaced by the string “foo” in the command’s output. The `s` conversion is used to format string data. There are other specifiers for other kinds of data. This table lists the commonly used data types:

Specifier	Description
d	Format a number as a signed decimal integer.
f	Format and output a floating point number.
o	Format an integer as an octal number.
S	Format a string.
x	Format an integer as a hexadecimal number using lowercase a-f where needed.
X	Same as x but use upper case letters.

%	Print a literal % symbol (i.e. specify “%%”)
---	--

We’ll demonstrate the effect each of the conversion specifiers on the string “380”:

```
[me@linuxbox ~]$ printf "%d, %f, %o, %s, %x, %X\n" 380 380 380 380
380 380
380, 380.000000, 574, 380, 17c, 17C
```

Since we specified six conversion specifiers, we must also supply six arguments for `printf` to process. The six results show the effect of each specifier.

There are several optional components that may be added to the conversion specifier to adjust its output. A complete conversion specification may consist of the following:

```
%[flags][width][.precision]conversion_specification
```

Multiple optional components, when used, must appear in the order specified above to be properly interpreted. Here is a description of each:

Component	Description
<i>flags</i>	There are five different flags:
	# - use the “alternate format” for output. This varies by data type. For <code>o</code> (octal number) conversion, the output is prefixed with <code>0</code> . For <code>x</code> and <code>X</code> (hexadecimal number) conversions, the output is prefixed with <code>0x</code> or <code>0X</code> respectively.
	<code>0</code> – (a zero) pad the output with zeros. This means that the field will be filled with leading zeros as in “000380”.
	- - (a dash) left-align the output. By default, <code>printf</code> right-aligns output.
	‘ ‘ - (a space) produces a leading space for positive numbers.
	+ - (a plus sign) signs positive numbers. By default, <code>printf</code> only signs negative numbers.
<i>width</i>	A number specifying the minimum field width.
<i>.precision</i>	For floating point numbers, specify the number of digits of precision to be output after the decimal point. For string conversion, <i>precision</i> specifies the number of characters to output.

Here are some examples of different formats in action:

Argument	Format	Result	Notes
380	"%d"	380	Simple formatting of an integer.
380	"%#x"	0x17c	Integer formatted as a hexadecimal number using the "alternate format" flag.
380	"%05d"	00380	Integer formatted with leading zeros (padding) and a minimum field width of five characters.
380	"%05.5f"	380.00000	Number formatted as a floating point number with padding and five decimal places of precision. Since the specified minimum field width (5) is less than the actual width than the formatted number, the padding has no effect.
380	"%010.5f"	0380.00000	By increasing the minimum field width to 10 the padding is now visible.
380	"%+d"	+380	The + flag signs a positive number.
380	"%-d"	380	The - flag left aligns the formatting.
abcdefghijkl	"5s"	abcdefghijkl	A string formatted with a minimum field width.
abcdefghijkl	".5s"	abcde	By applying precision to a string, it is truncated.

Again, `printf` is used mostly in scripts where it is often employed to format tabular data, rather than on the command line directly. But we can still show how it can be used to solve various formatting problems. First, let's output some fields separated by tab characters:

```
[me@linuxbox ~]$ printf "%s\t%s\t%s\n" str1 str2 str3
str1 str2 str3
```

By inserting `\t` (the escape sequence for a tab) we achieve the desired effect. Next, some numbers with neat formatting:

```
[me@linuxbox ~]$ printf "Line: %05d %15.3f Result: %+15d\n" 1071
3.14156295 32589
Line: 01071          3.142 Result:          +32589
```

which shows the effect of minimum field width on the spacing of the fields. Or how about formatting a tiny web page:

```
[me@linuxbox ~]$ printf "<html>\n\t<head>\n\t\t<title>%s</title>\n
\t</head>\n\t<body>\n\t\t<p>%s</p>\n\t</body>\n</html>\n" "Page Tit
le" "Page Content"
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>
    <p>Page Content</p>
  </body>
</html>
```

Document Formatting Systems

So far, we have examined the simple text formatting tools. These are good for small, simple tasks, but what about larger jobs? One of the reasons that Unix became a popular operating system among technical and scientific users (aside from providing a powerful multitasking, multiuser environment for all kinds of software development) is that it offered tools that could be used to produce many types of documents, particularly scientific and other academic publications. In fact, as the GNU documentation describes, document preparation was instrumental to the development of Unix:

The first version of UNIX was developed on a PDP-7 which was sitting around Bell Labs. In 1971 the developers wanted to get a PDP-11 for further work on the operating system. In order to justify the cost for this system, they proposed that they would implement a document formatting system for the AT&T patents division. This first formatting program was a reimplement of McIllroy's `\roff`, written by J. F. Ossanna.

There are two main families of document formatters that dominate the field, those descended from the original `roff` program including `nroff` and `troff` and those based on Donald Knuth's T_EX (pronounced "tek") typesetting system. And yes, the dropped "E" in the middle is part of its name.

The name "roff" is derived from the term "run off" as in, "I'll run off a copy for you." The `nroff` program is used to format documents for output to devices that use monospaced fonts such as character terminals and typewriter-style of printers. At the time of its introduction, this included nearly all printing devices attached to computers. The later `troff` program formats documents for output on *typesetters*, devices used to produce "camera-ready" type for commercial printing. Most computer printers today are able to simulate the output of typesetters. The `roff` family also includes some other programs that are used to prepare portions of documents. These include `eqn` (for mathematical equations) and `tbl` (for tables).

The T_EX system (in stable form) first appeared in 1989 and has, to some degree, displaced `troff` as the tool of choice for typesetter output. We won't be covering T_EX here due both to its complexity (there are entire books about it) and to the fact that it is not installed by default on most modern Linux systems.

Tip: For those interested in installing T_EX on their system, check out the `texlive` package which can be found in most distribution repositories, and the LyX graphical content editor.

groff

`groff` is suite of programs containing the GNU implementation of `troff`. It also includes a script that is used to emulate `nroff` and the rest of the `roff` family as well.

While `roff` and its descendants are used to make formatted documents, they do it in a way that is rather foreign to modern users. Most documents today are produced using word processors that are able to perform both the composition and layout of a document in a single step. Prior to the advent of the graphical word processor, documents were often produced in a two-step process involving the use of a text editor to perform composition and a processor, such as `troff` to apply the formatting. Instructions for the formatting program were embedded into the composed text through the use of a markup language. The modern analog for such a process is the web page which is composed using a text editor of some kind and then rendered by a web browser using HTML as the markup language to describe the final page layout.

We're not going to cover `groff` in its entirety as many elements of its markup language deal with rather arcane details of typography, rather we will concentrate on one of its *macro packages* that remains in wide use. These macro packages condense many of its

low-level commands into a smaller set of high-level commands that make using `groff` much easier.

Let's, for a moment, consider the humble man page. It lives in the `/usr/share/man` directory as a `gzip` compressed text file. If we were to examine its uncompressed contents, we would see the following (the man page for `ls` in section 1 is shown):

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | head
.\" DO NOT MODIFY THIS FILE! It was generated by help2man 1.35.
.TH LS "1" "April 2008" "GNU coreutils 6.10" "User Commands"
.SH NAME
ls \- list directory contents
.SH SYNOPSIS
.B ls
[\fIOPTION\fR]... [\fIFILE\fR]...
.SH DESCRIPTION
.\" Add any additional description here
.PP
```

Compared to the man page in its normal presentation, we can begin to see a correlation between the markup language and its results:

```
[me@linuxbox ~]$ man ls | head
LS(1)                                User Commands                                LS(1)

NAME
    ls - list directory contents

SYNOPSIS
    ls [OPTION]... [FILE]...
```

The reason this is of interest is that man pages are rendered by `groff` using the `mandoc` macro package. In fact, we can simulate the `man` command with the following pipeline:

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc -T
ascii | head
LS(1)                                User Commands                                LS(1)

NAME
    ls - list directory contents
```

```
SYNOPSIS
  ls [OPTION]... [FILE]...
```

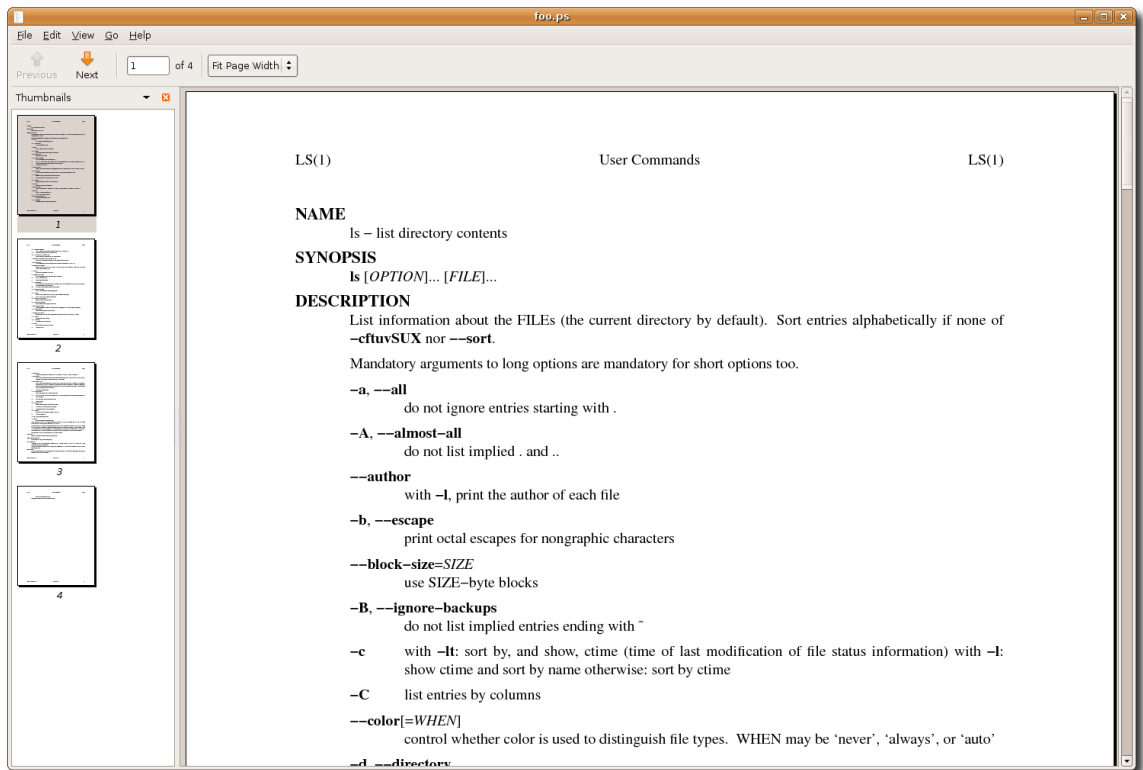
Here we use the `groff` program with the options to specify the `mandoc` macro package and the output driver for ASCII. `groff` can produce output in several formats. If no format is specified, then PostScript is output by default:

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc |
head
%!PS-Adobe-3.0
%%Creator: groff version 1.18.1
%%CreationDate: Thu Feb  5 13:44:37 2009
%%DocumentNeededResources: font Times-Roman
%%+ font Times-Bold
%%+ font Times-Italic
%%DocumentSuppliedResources: procset grops 1.18 1
%%Pages: 4
%%PageOrder: Ascend
%%Orientation: Portrait
```

We've talked a little about PostScript in earlier chapters and will again in the next chapter. PostScript is a page description language that is used to describe the contents of a printed page to a typesetter-like device. If we take the output of our command and store it to a file (assuming that we are using a graphical desktop with a `Desktop` directory):

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc >
~/Desktop/foo.ps
```

An icon for the output file should appear on the desktop. By double-clicking on the icon, a page viewer should start up and reveal the file in its rendered form:



What we see, is the man page for `ls` nicely typeset! In fact, it's possible to convert the PostScript file in to a PDF (*Portable Document Format*) file with this command:

```
[me@linuxbox ~]$ ps2pdf ~/Desktop/foo.ps ~/Desktop/ls.pdf
```

The `ps2pdf` program is part of the `ghostscript` package which is installed on most Linux systems that support printing.

Tip: Linux systems often include many command line programs for file format conversion. They are often named using the convention of *format2format*. Try using the command `ls /usr/bin/*[[:alpha:]]2[[:alpha:]]*` to identify them.

For our last exercise with `groff` we will revisit our old friend `distros.txt` once more. This time we will use the `tbl` program which is used to format tables to typeset our list of Linux distributions. To do this, we are going to use our earlier `sed` script to add markup to a text stream that we will feed to `groff`.

First, we need to modify our `sed` script to add the necessary requests that `tbl` requires. Using a text editor, we will change `distros.sed` to the following:

```
# sed script to produce Linux distributions report

1 i\
.TS\
center box;\
cb s s\
cb cb cb\
l n c.\
Linux Distributions Report\
=\
Name Version Released\
-
s/\([0-9]\{2\}\)\.\/\([0-9]\{2\}\)\.\/\([0-9]\{4\}\)\$/\3-\1-\2/
$a\
.TE
```

Note that for the script to work properly, care must be taken to see that the words “Name Version Released” are separated by tabs and not spaces. We’ll save the resulting file as `distros-tbl.sed`. `tbl` uses the `.TS` and `.TE` requests to start and end the table. The rows following the `.TS` request define global properties of the table, which, for our example, are centered horizontally on the page and surrounded by a box. The remaining lines of the definition describe the layout of each table row. Now, if we run our report-generating pipeline again with the new `sed` script, we’ll get the following :

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-tbl
.sed | groff -t -T ascii
```

Linux Distributions Report		
Name	Version	Released
Fedora	5	2006-03-20
Fedora	6	2006-10-24
Fedora	7	2007-05-31
Fedora	8	2007-11-08
Fedora	9	2008-05-13
Fedora	10	2008-11-25
SUSE	10.1	2006-05-11
SUSE	10.2	2006-12-07
SUSE	10.3	2007-10-04
SUSE	11.0	2008-06-19
Ubuntu	6.06	2006-06-01


```

|Ubuntu    6.10    2006-10-26 |
|Ubuntu    7.04    2007-04-19 |
|Ubuntu    7.10    2007-10-18 |
|Ubuntu    8.04    2008-04-24 |
|Ubuntu    8.10    2008-10-30 |
+-----+

```

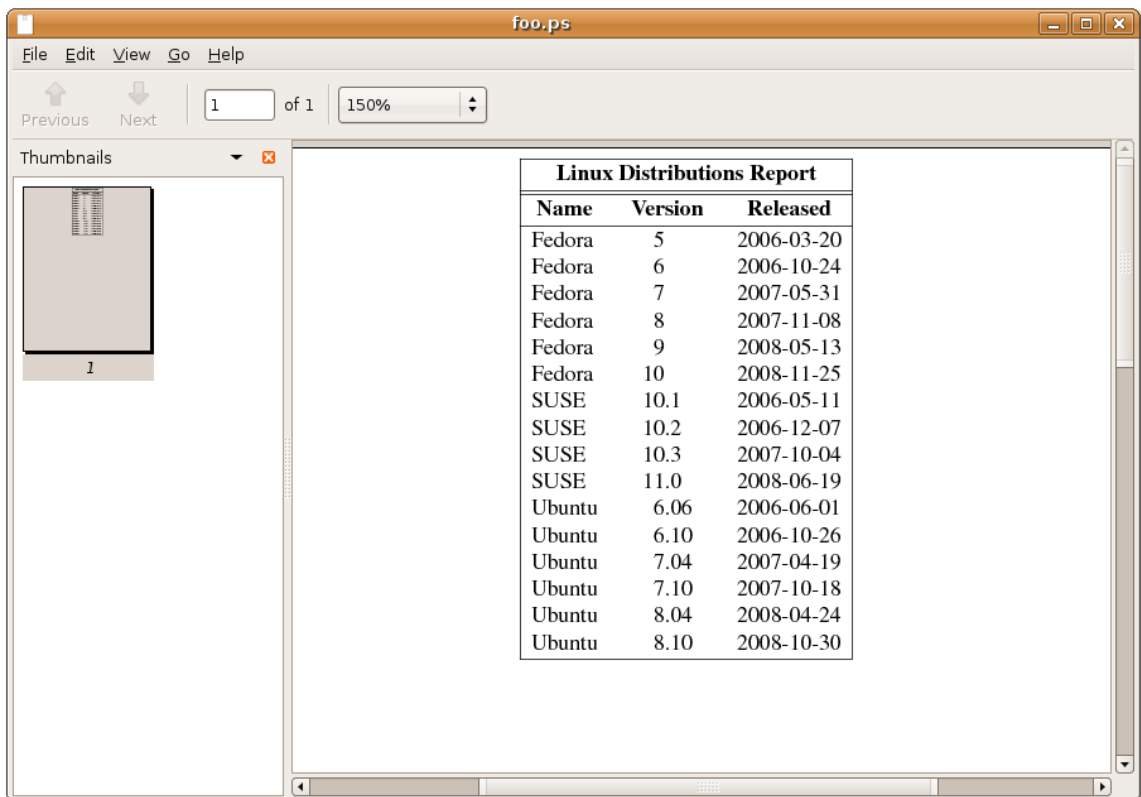
Adding the `-t` option to `groff`, instructs it to pre-process the text stream with `tbl`. Likewise, the `-T` option is used to output to ASCII rather than the default output medium, PostScript.

The format of the output is the best one can expect when limited to the capabilities of a terminal screen or typewriter-style printer. If we specify PostScript output and graphically view the resulting output, we get a much more satisfying result:

```

[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-tbl
.sed | groff -t > ~/Desktop/foo.ps

```



Summing Up

Given that text is so central to the character of Unix-like operating systems, it makes sense that there would be many tools that are used to manipulate and format text. As we have seen, there are! The simple formatting tools like `fmt` and `pr` will find many uses in scripts that produce short documents while `groff` (and friends) can be used to write books. We may never write a technical paper using command line tools (though there are many people who do!), but it's good to know that we could.

Further Reading

`groff` User's Guide

<http://www.gnu.org/software/groff/manual/>

Writing Papers With `nroff` Using `-me`:

<http://docs.freebsd.org/44doc/usd/19.memacros/paper.pdf>

`-me` Reference Manual

<http://docs.freebsd.org/44doc/usd/20.meref/paper.pdf>

`Tbl` – A Program To Format Tables:

<http://plan9.bell-labs.com/10thEdMan/tbl.pdf>

And, of course, try the following articles at Wikipedia:

<http://en.wikipedia.org/wiki/TeX>

http://en.wikipedia.org/wiki/Donald_Knuth

<http://en.wikipedia.org/wiki/Typesetting>

23 – Printing

After spending the last couple of chapters manipulating text, it's time to put it to paper. In this chapter, we'll look at the command line tools that are used to print files and control printer operation. We won't be looking at how to configure printing as that varies from distribution to distribution and is usually set up automatically during installation. Note that we will need a working printing configuration to perform the exercises in this chapter.

We will discuss the following commands:

- `pr` – Convert text files for printing
- `lpr` – Print files
- `a2ps` – Format files for printing on a PostScript printer
- `lpstat` – Show printer status information
- `lpq` – Show printer queue status
- `lprm` – Cancel print jobs

A Brief History Of Printing

To fully understand the printing features found in Unix-like operating systems, we first have to learn some history. Printing on Unix-like systems goes way back to the beginning of operating system itself. In those days, printers, and the way they were used was much different than today.

Printing In The Dim Times

Like the computers themselves, printers in the pre-PC era tended to be large, expensive and centralized. The typical computer user of 1980 worked at a terminal connected to a computer some distance away. The printer was located near the computer and was under the watchful eyes of the computer's operators.

When printers were expensive and centralized, as they often were in the early days of Unix, it was common practice for many users to share a printer. To identify print jobs

belonging to a particular user, a *banner page* was often printed at the beginning of each print job displaying the name of the user. The computer support staff would then load up a cart containing the day's print jobs and deliver them to the individual users.

Character-based Printers

The printer technology of that period was very different in two respects. First, printers of that period were almost always *impact printers*. Impact printers use a mechanical mechanism to strike a ribbon against the paper to form character impressions on the page. Two of the popular technologies of that time were *daisy wheel* printing and *dot-matrix* printing.

The second, and more important characteristic of early printers was that printers used a fixed set of characters that were intrinsic to the device itself. For example, a daisy wheel printer could only print the characters actually molded into the petals of the daisy wheel. This made printers of the period much like high-speed typewriters. As with most typewriters, they printed using monospaced (fixed width) fonts. This means that each character has the same width. Printing was done at fixed positions on the page and the printable area of a page contained a fixed number of characters. Printers, depending on the model, most likely printed ten characters per inch (CPI) horizontally and six lines per inch (LPI) vertically. Using this scheme, a US letter sheet of paper is eighty-five characters wide and sixty-six lines high. Taking into account a small margin on each side, eighty characters was considered the maximum width of a print line. This explains why terminal displays (and our terminal emulators) are normally eighty characters wide. It's to provide a WYSIWYG (*What You See Is What You Get*) view of printed output using a monospaced font.

Data is sent to a typewriter-like printer in a simple stream of bytes containing the characters to be printed. For example, to print an “a”, the ASCII character code 97 is sent. In addition, the low-numbered ASCII control codes provided a means of moving the printer's carriage and paper using codes for carriage return, line feed, form feed and the like. Using the control codes, it's possible to achieve some limited font effects such as bold face by having the printer print a character, backspace and print the character again to get a darker print impression on the page. We can actually witness this if we use `nroff` to render a man page and examine the output using `cat -A`:

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | nroff -man | cat
-A | head
LS(1)                                User Commands                                LS(1)
$
$
$
$
N^HNA^HAM^HME^HE$
```

```
ls - list directory contents$
$
S^HSY^HYN^HNO^HOP^HPS^HSI^HIS^HS$
  l^Hl^s^hs [_^HO^HP^HT^HI^HO^HN]... [_^HF^HI^HL^HE]...$
```

The ^H (Control-h) characters are the backspaces used to create the boldface effect. Likewise, we can also see a backspace/underscore sequence used to produce underlining.

Graphical Printers

The development of GUIs lead to major changes in printer technology. As computers moved to more picture-based displays, so to did printing move from character-based to graphical techniques. This was facilitated by the advent of the low-cost laser printer which, instead of printing fixed characters, could print tiny dots anywhere in the printable area of the page. This made printing proportional fonts (like those used by typesetters) and even photographs and high quality diagrams possible.

However, moving from a character-based scheme to a graphical scheme presented a formidable technical challenge. Here's why: the number of bytes needed to fill a page using a character-based printer can be calculated this way (assuming sixty lines per page each containing eighty characters):

$$60 \times 80 = 4800 \text{ bytes}$$

whereas, a three hundred dot per inch (DPI) laser printer (assuming a eight by ten inch print area per page) requires:

$$(8 \times 300) \times (10 \times 300) / 8 = 900000 \text{ bytes}$$

The need to send nearly one megabyte of data per page to fully utilize a laser printer was more than many of the slow PC networks could handle, so it was clear that a clever invention was needed.

That invention turned out to be the *page description language* (PDL). A page description language is a programming language that describes the contents of a page. Basically it says, "go to this position, draw the character 'a' in ten point Helvetica, go to this position..." until everything on the page is described. The first major PDL was *PostScript* from Adobe Systems which is still in wide use today. The PostScript language is a complete programming language tailored for typography and other kinds of graphics and imaging. It includes built-in support for thirty-five standard high quality fonts and the ability to accept additional font definitions at run time. At first, support for PostScript was built into printers themselves. This solved the data transmission problem. While the typical PostScript program was very verbose in comparison to the simple byte stream of character-based printers, it was much smaller than the number of bytes required to represent the entire printed page.

A *PostScript printer* accepted a PostScript program as input. The printer contained its own processor and memory (often times making the printer a more powerful computer than the computer it was attached to) and executed a special program called a *PostScript interpreter* which read the incoming PostScript program and *rendered* the results into the printer's internal memory thus forming the pattern of bits (dots) that would be transferred to the paper. The generic name for this process of rendering something into a large bit pattern (called a *bitmap*) is *raster image processor* or RIP.

As the years went by, both computers and networks became much faster. This allowed the RIP to move from the printer to the host computer. This permitted high quality printers to be much less expensive.

Many printers today still accept character-based streams, but many low-cost printers do not. They rely on the host computer's RIP to provide a stream of bits to print as dots. There are still some PostScript printers too.

Printing With Linux

Modern Linux systems employ two software suites to perform and manage printing. The first, CUPS (Common Unix Printing System) which provides print drivers and management of print jobs, and the second, Ghostscript, a PostScript interpreter, acts as a RIP.

CUPS manages printers by creating and maintaining print queues. As we recall from our history lesson above, Unix printing was originally designed to manage a centralized printer shared by multiple users. Since printers are slow by nature compared to the computers that are feeding them, printing systems need a way to schedule multiple print jobs and keep things organized. CUPS also has the ability to recognize different types of data (within reason) and can convert files to a printable form.

Preparing Files For Printing

As a command line user, we are mostly interested in printing text, though it is certainly possible to print other data formats as well.

pr – Convert Text Files For Printing

We looked at `pr` a little in the last chapter. This time we will examine some of its many options used in conjunction with printing. In our history of printing, we saw how character-based printers use monospaced fonts resulting in fixed numbers of characters per line and lines per page. `pr` is used to adjust text to fit on a specific page size with optional page headers and margins. Here is a summary of its most commonly used options:

Option	Description
<code>+first[:last]</code>	Output a range of pages starting with <i>first</i> and optionally ending with <i>last</i> .
<code>-columns</code>	Organize the content of the page into the number of columns specified by <i>columns</i> .
<code>-a</code>	By default, multicolumn output is listed vertically. By adding the <code>-a</code> (across) option, content is listed horizontally.
<code>-d</code>	Double-space output.
<code>-D "format"</code>	Format the date displayed in page headers using <i>format</i> . See the man page for the <code>date</code> command for a description of the format string.
<code>-f</code>	Use form feeds rather than carriage returns to separate pages.
<code>-h "header"</code>	In the center portion of the page header, use <i>header</i> rather than the name of the file being processed.
<code>-l length</code>	Set page length to <i>length</i> . Default is 66 (US letter at 6 lines per inch)
<code>-n</code>	Number lines.
<code>-o offset</code>	Create a left margin <i>offset</i> characters wide.
<code>-w width</code>	Set page width to <i>width</i> . Default is 72.

`pr` is often used in pipelines as a filter. In this example, we will produce a directory listing of `/usr/bin` and format it into paginated three column output using `pr`:

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 -w 65 | head

2009-02-18 14:00                                     Page 1

[
411toppm      apturl      bsd-write
a2p           ar          bsh
a2ps         arecord    btcflash
a2ps         arecordmidi bug-buddy
a2ps-lpr-wrapper ark         buildhash
```

Sending A Print Job To A Printer

The CUPS printing suite supports two methods of printing historically used on Unix-like system. One method, called Berkeley or LPD (used in the Berkeley Software Distribution version of Unix), uses the `lpr` program while the other method, called SysV (from the System V version of Unix) uses the `lp` program. Both programs do roughly the same thing. Choosing one over the other is a matter of personal taste.

`lpr` – Print Files (Berkeley Style)

The `lpr` program can be used to send files to the printer. It may also be used in pipelines as it accepts standard input. For example, to print the results of our multi-column directory listing above we could do this:

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 | lpr
```

and the report would be sent to the system's default printer. To send the file to a different printer, the `-P` option can be used like this:

```
lpr -P printer_name
```

where *printer_name* is the name of the desired printer. To see a list of printers known to the system:

```
[me@linuxbox ~]$ lpstat -a
```

Tip: Many Linux distributions allow you to define a “printer” that outputs files in PDF (Portable Document Format) format rather than printing on the physical printer. This is very handy for experimenting with printing commands. Check your printer configuration program to see if it supports this configuration. On some distributions, you may need to install additional packages to enable this capability.

Here are some of the common options for `lpr`:

Option	Description
<code>-# <i>number</i></code>	Sets number of copies to <i>number</i> .

-p	Print each page with a shaded header with the date, time, job name and page number. This so called “pretty print” option can be used when printing text files.
-P <i>printer</i>	Specifies the name of the printer used for output. If no printer is specified, the system’s default printer is used.
-r	Delete files after printing. This would be useful for programs that produce temporary printer output files.

lp – Print Files (System V Style)

Like lpr, lp accepts either files or standard input for printing. it differs from lpr in that it supports a different (and slightly more sophisticated) option set. Here are the common ones:

Option	Description
-d <i>printer</i>	Set the destination (printer) to <i>printer</i> . If no d option is specified, the system default printer is used.
-n <i>number</i>	Sets the number of copies to <i>number</i> .
-o landscape	Set output to landscape orientation.
-o fitplot	Scale the file to fit the page. This is useful when printing images such as JPEG files.
-o scaling= <i>number</i>	Scale file to <i>number</i> . The value of 100 fills the page. Values less than 100 are reduced, while values greater than 100 cause the file to be printed across multiple pages.
-o cpi= <i>number</i>	Set the output characters per inch to <i>number</i> . Default is 10.
-o lpi= <i>number</i>	Set the output lines per inch to <i>number</i> . Default is 6.
-o page-bottom= <i>points</i> -o page-left= <i>points</i> -o page-right= <i>points</i> -o page-top= <i>points</i>	Sets the page margins. Values are expressed in <i>points</i> , a unit of typographic measurement. There are 72 points to an inch.
-P <i>pages</i>	Specifies the list of pages. <i>pages</i> may be expressed as a comma separated list and/or a range. For example “1,3,5,7-10”

We'll produce our directory listing again, this time printing twelve CPI and eight LPI with a left margin of one half inch. Note that we have to adjust the `pr` options to account for the new page size:

```
[me@linuxbox ~]$ ls /usr/bin | pr -4 -w 90 -l 88 | lp -o page-left=36  
-o cpi=12 -o lpi=8
```

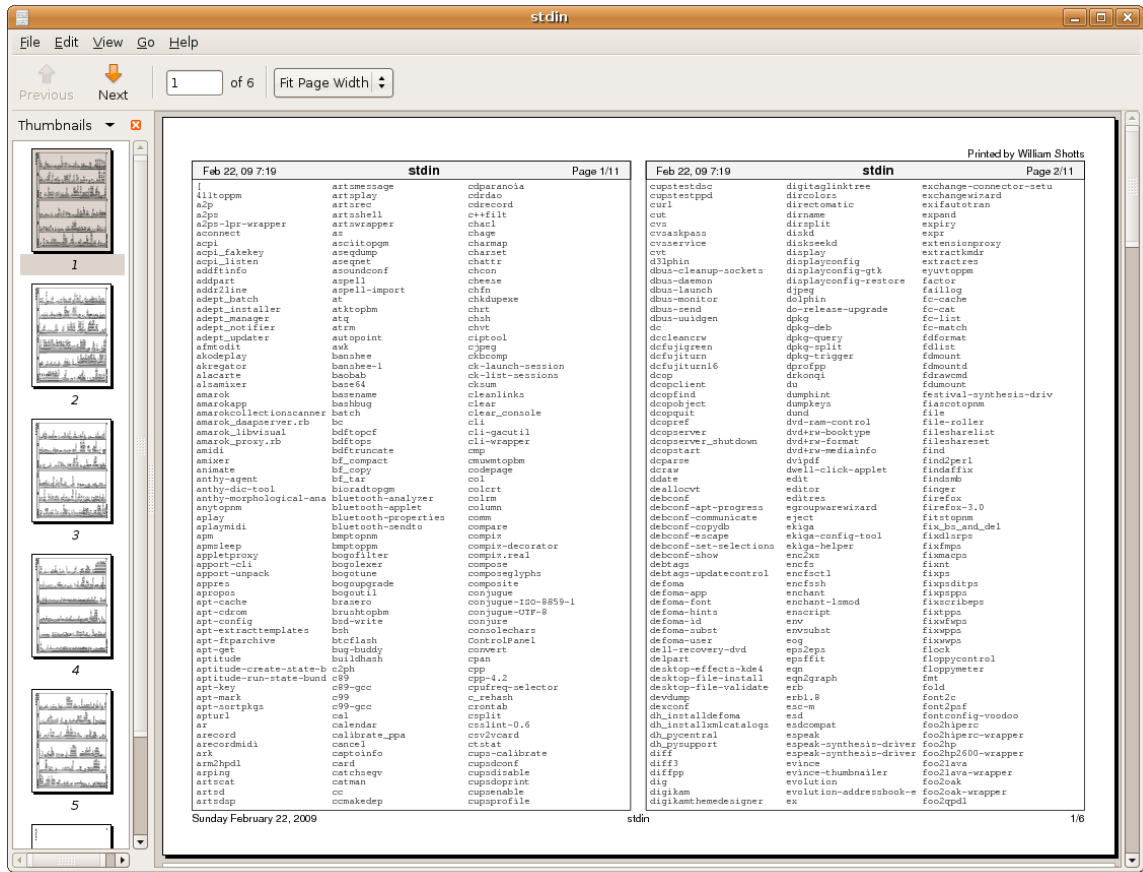
This pipeline produces a four column listing using smaller type than the default. The increased number of characters per inch allows us to fit more columns on the page.

Another Option: a2ps

The `a2ps` program is interesting. As we can surmise from its name, it's a format conversion program and it is, but it also much more. Its name originally meant "ASCII to PostScript" and it was used to prepare text files for printing on PostScript printers. Over the years however, the capabilities of the program have grown and now its name means "Anything to PostScript." While its name suggests a format conversion program, it is actually a printing program as its default output is to the system's default printer rather than standard output as one might normally expect. The program's default behavior is that of a "pretty printer" meaning that it improves the appearance of output. If we use the program to send its output to a PostScript file on our desktop:

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 -t | a2ps -o ~/Desktop/ls.ps -L  
66  
[stdin (plain): 11 pages on 6 sheets]  
[Total: 11 pages on 6 sheets] saved into the file `~/home/me/Desktop/  
ls.ps'
```

Here we filter the stream with `pr` using the `-t` option (omit headers and footers) and then with `a2ps` specifying an output file (`-o` option) and 66 lines per page (`-L` option) to match the output pagination of `pr`. If we view the resulting file with a suitable file viewer, we will see this:



As we can see, the default output layout is “two up” format. This causes the contents of two pages to be printed on each sheet of paper. a2ps applies nice page headers and footers too.

a2ps has a lot of options. Here is a summary:

Option	Description
--center-title <i>text</i>	Set center page title to <i>text</i> .
--columns <i>number</i>	Arrange pages into <i>number</i> columns. Default is 2.
--footer <i>text</i>	Set page footer to <i>text</i> .
--guess	Report the types of files have been given as arguments. Since a2ps tries to convert and format all types of data, this option can be useful for predicting what a2ps will do when given a particular file.

<code>--left-footer <i>text</i></code>	Set left page footer to <i>text</i> .
<code>--left-title <i>text</i></code>	Set left page title to <i>text</i> .
<code>--line-numbers=<i>interval</i></code>	Number lines of output every <i>interval</i> lines.
<code>--list=defaults</code>	Display default settings.
<code>--list=<i>topic</i></code>	Display settings for <i>topic</i> , where <i>topic</i> is one of the following: delegations (external programs that will be used to convert data), encodings, features, variables, media (paper sizes and the like), ppd (PostScript printer descriptions), printers, prologues (portions of code that are prefixed to normal output), style-sheets, and user options.
<code>--pages <i>range</i></code>	Print pages in range.
<code>--right-footer <i>text</i></code>	Set right page footer to <i>text</i> .
<code>--right-title <i>text</i></code>	Set right page title to <i>text</i> .
<code>--rows <i>number</i></code>	Arrange pages into <i>number</i> rows. Default is one.
<code>-B</code>	No page headers.
<code>-b <i>text</i></code>	Set page header to <i>text</i> .
<code>-f <i>size</i></code>	Use <i>size</i> point font.
<code>-l <i>number</i></code>	Set characters per line to <i>number</i> . This and the <code>-L</code> option (below) can be used to make files paginated with other programs such as <code>pr</code> fit correctly on the page.
<code>-L <i>number</i></code>	Set lines per page to <i>number</i> .
<code>-M <i>name</i></code>	Use media name. For example “A4”.
<code>-n <i>number</i></code>	Output <i>number</i> copies of each page.
<code>-o <i>file</i></code>	Send output to <i>file</i> . If <i>file</i> is specified as “-”, use standard output.
<code>-P <i>printer</i></code>	Use <i>printer</i> . If a printer is not specified, the system default printer is used.
<code>-R</code>	Portrait orientation.
<code>-r</code>	Landscape orientation.
<code>-T <i>number</i></code>	Set tab stops to every <i>number</i> characters.

<code>-u text</code>	Underlay (watermark) pages with <i>text</i> .
----------------------	---

This is just a summary. `a2ps` has several more.

Note: `a2ps` is still in active development. During my testing, I noticed various behavior differences on different distributions. On CentOS 4, output always went to standard output by default. On CentOS 4 and Fedora 10, output defaulted to A4 media despite the program being configured to use letter size media by default. I could overcome these issues by explicitly specifying the desired option. On Ubuntu 8.04, it performed as documented.

Also note that there is another output formatter that is useful for converting text into PostScript. Called `enscript`, it can perform many of the same kinds formatting and printing tricks, but unlike `a2ps`, it only accepts text input.

Monitoring And Controlling Print Jobs

As Unix printing systems are designed to handle multiple print jobs from multiple users, CUPS is designed to do the same. Each printer is given a *print queue* where jobs are parked until they can be *spooled* to the printer. CUPS supplies several command line programs that are used to manage printer status and print queues. Like the `lpr` and `lp` programs, these management programs are modeled after the corresponding programs from the Berkeley and System V printing systems.

lpstat – Display Print System Status

The `lpstat` program is useful for determining the names and availability of printers on the system. For example, if we had a system with both a physical printer (named “printer”) and a PDF virtual printer (named “PDF”), we could check their status like this:

```
[me@linuxbox ~]$ lpstat -a
PDF accepting requests since Mon 08 Dec 2008 03:05:59 PM EST
printer accepting requests since Tue 24 Feb 2009 08:43:22 AM EST
```

Further, we could determine a more detailed description of the print system configuration this way:

```
[me@linuxbox ~]$ lpstat -s
```

```
system default destination: printer
device for PDF: cups-pdf:/
device for printer: ipp://print-server:631/printers/printer
```

In this example, we see that “printer” is the system’s default printer and that it is a network printer using Internet Printing Protocol (ipp://) attached to a system named “print-server”.

The commonly useful options include:

Option	Description
-a [<i>printer...</i>]	Display the state of the printer queue for <i>printer</i> . Note that this is the status of the printer queue’s ability to accept jobs, not the status of the physical printers. If no printers are specified, all print queues are shown.
-d	Display the name of system’s default printer.
-p [<i>printer...</i>]	Display the status of the specified <i>printer</i> . If no printers are specified, all printers are shown.
-r	Display the status of the print server.
-s	Display a status summary.
-t	Display a complete status report.

lpq – Display Printer Queue Status

To see the status of a printer queue, the `lpq` program is used. This allows us to view the status of the queue and the print jobs it contains. Here is an example of an empty queue for a system default printer named “printer”:

```
[me@linuxbox ~]$ lpq
printer is ready
no entries
```

If we do not specify a printer (using the `-P` option), the system’s default printer is shown. If we send a job to the printer and then look at the queue, we will see it listed:

```
[me@linuxbox ~]$ ls *.txt | pr -3 | lp
request id is printer-603 (1 file(s))
```

```
[me@linuxbox ~]$ lpq
printer is ready and printing
Rank   Owner   Job     File(s)           Total Size
active me      603     (stdin)          1024 bytes
```

lprm / cancel – Cancel Print Jobs

CUPS supplies two programs used to terminate print jobs and remove them from the print queue. One is Berkeley style (`lprm`) and the other is System V (`cancel`). They differ slightly in the options they supports, but do basically the same thing. Using our print job above as an example, we could stop the job and remove it this way:

```
[me@linuxbox ~]$ cancel 603
[me@linuxbox ~]$ lpq
printer is ready
no entries
```

Each command has options for removing all the jobs belonging to a particular user, particular printer, and multiple job numbers. Their respective man pages have all the details.

Summing Up

We have seen in this chapter how the printers of the past influenced the design of the printing systems on Unix-like machines and how much control is available on the command line to control not only the scheduling and execution of print jobs, but also the various output options.

Further Reading

A good article on the PostScript page description language:

<http://en.wikipedia.org/wiki/PostScript>

The Common Unix Printing System (CUPS):

http://en.wikipedia.org/wiki/Common_Unix_Printing_System

<http://www.cups.org/>

The Berkeley and System V Printing Systems:

http://en.wikipedia.org/wiki/Berkeley_printing_system

http://en.wikipedia.org/wiki/System_V_printing_system

24 – Compiling Programs

In this chapter we will look at how to build programs by compiling source code. The availability of source code is the essential freedom that makes Linux possible. The entire ecosystem of Linux development relies on free exchange between developers. For many desktop users, compiling is a lost art. It used to be quite common, but today, distribution providers maintain huge repositories of pre-compiled binaries ready to download and use. At the time of this writing, the Debian repository (one of the largest of any of the distributions) contains almost twenty-three thousand packages.

So why compile software? There are two reasons:

1. **Availability.** Despite the number of pre-compiled programs in distribution repositories, some distributions may not include all the desired applications. In this case, the only way to get the desired program is to compile it from source.
2. **Timeliness.** While some distributions, specialize in cutting edge versions of programs, many do not. This means that in order to have the very latest version of a program, compiling is necessary.

Compiling software from source code can become very complex and technical, well beyond the reach of many users. However, many compiling tasks are quite easy and involve only a few steps. It all depends on the package. We will look at a very simple case in order to provide an overview of the process and a starting point for those who wish to undertake further study.

We will introduce one new command:

- `make` – Utility To Maintain Programs

What Is Compiling?

Simply put, compiling is the process of translating *source code* (the human-readable description of a program written by a programmer) into the native language of the computer's processor.

The computer's processor (or *CPU*) works at a very elemental level executing programs in what is called *machine language*. This is a numeric code that describes very small operations, such as “add this byte”, “point to this location in memory”, “copy this byte.”

Each of these instructions are expressed in binary (ones and zeros). The earliest computer programs were written using this numeric code and it explains why programmers who wrote it were said to smoke a lot, drink gallons of coffee, and wear thick glasses.

This problem was overcome by the advent of *assembly language*, which replaced the numeric codes with (slightly) easier to use character *mnemonics* such as CPY (for copy) and MOV (for move). Programs written in assembly language are processed into machine language by a program called an *assembler*. Assembly language is still used today for certain specialized programming tasks such as *device drivers* and *embedded systems*.

We next come to what are called, *high-level programming languages*. They are called this because they allow the programmer to be less concerned with the details of what the processor is doing and more with solving the problem at hand. The early ones (developed during the 1950s) included *FORTRAN* (designed for scientific and technical tasks) and *COBOL* (designed for business applications.) Both are still in limited use today.

While there are many popular programming languages, two predominate. Most programs written for modern systems are written in either *C* or *C++*. In the examples to follow, we will be compiling a *C* program.

Programs written in high-level programming languages are processed into machine language by processing them with another program called a *compiler*. Some compilers translate high-level instructions into assembly language and then an assembler is used to perform the final stage of translation into machine language.

A process often used in conjunction with compiling is called *linking*. There are many tasks in programming common to many programs. Take for instance, opening a file. Many programs perform this task, but it would be wasteful to have each program implement its own routine to open files. It makes more sense to have a single piece of programming that knows how to open files and allow all programs that need it to share it. This is accomplished by what are called *libraries*. They contain multiple *routines*, each performing some common task that multiple programs can share. If we look in the `/lib` and `/usr/lib` directories, we can see where many of them live. A program called a *linker* is used to form the connections between the output of the compiler and the libraries that the compiled program requires. The final result of this process is the *executable program file* ready for use.

Are All Programs Compiled?

No. As we have seen, there are programs such as shell scripts that do not require compiling. They are executed directly. These are written in what are known as *scripting* or *interpreted* languages. These languages have grown in popularity in recent years and

include *perl*, *python*, *PHP*, *ruby* and many others.

Scripted languages are executed by a special program called an *interpreter*. An interpreter inputs the program file and reads and executes each instruction contained within it. In general, interpreted programs execute much more slowly than compiled programs. This is due to the fact that each source code instruction in an interpreted program is translated every time it is carried out, whereas with a compiled program a source code instruction is only translated once and this translation is permanently recorded in the final executable file.

So why are interpreted languages so popular? For many programming chores, the results are “fast enough,” but the real advantage is that it is generally faster and easier to develop interpreted programs than compiled programs. Programs are usually developed in a repeating cycle of code, compile, test... As a program grows in size, the compilation phase of the cycle can become quite long. Interpreted languages remove the compilation step and thus speed up program development.

Compiling A C Program

Let’s compile something. Before we do that however, we’re going to need some tools, like the compiler, linker, and `make`. The C compiler used almost universally in the Linux environment is called `gcc` (GNU C Compiler) originally written by Richard Stallman. Most distributions do not install `gcc` by default. We can check to see if the compiler is present like this:

```
[me@linuxbox ~]$ which gcc
/usr/bin/gcc
```

The results in this example indicate that the compiler is installed.

Tip: Your distribution may have a meta-package (a collection of packages) for software development. If so, consider installing it if you intend to compile programs on your system. If your system does not provide a meta-package, try installing the `gcc` and `make` packages. On many distributions, this is sufficient to carry out the exercise below.

Obtaining The Source Code

For our compiling exercise, we are going to compile a program from the GNU project called `diction`. This is a handy little program that checks text files for writing quality and style. As programs go, its fairly small and easy to build.

Following convention, we're first going to create a directory for our source code named `src` and then download the source code into it using `ftp`:

```
[me@linuxbox ~]$ mkdir src
[me@linuxbox ~]$ cd src
[me@linuxbox src]$ ftp ftp.gnu.org
Connected to ftp.gnu.org.
220 GNU FTP server ready.
Name (ftp.gnu.org:me): anonymous
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd gnu/diction
250 Directory successfully changed.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
-rw-r--r--  1 1003  65534  68940 Aug 28  1998 diction-0.7.tar.gz
-rw-r--r--  1 1003  65534  90957 Mar 04  2002 diction-1.02.tar.gz
-rw-r--r--  1 1003  65534 141062 Sep 17  2007 diction-1.11.tar.gz
226 Directory send OK.
ftp> get diction-1.11.tar.gz
local: diction-1.11.tar.gz remote: diction-1.11.tar.gz
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for diction-1.11.tar.gz
(141062 bytes).
226 File send OK.
141062 bytes received in 0.16 secs (847.4 kB/s)
ftp> bye
221 Goodbye.
[me@linuxbox src]$ ls
diction-1.11.tar.gz
```

As we can see, source code is usually supplied in the form of a compressed tar file. Sometimes called a “*tarball*,” this file contains the *source tree*, or hierarchy of directories and files that comprise the source code. After arriving at the `ftp` site, we examine the list of tar files available and select the newest version for download. Using the `get` command within `ftp`, we copy the file from the `ftp` server to the local machine.

Once the tar file is downloaded, it must be unpacked. This is done with the `tar` program:

```
[me@linuxbox src]$ tar xzf diction-1.11.tar.gz
[me@linuxbox src]$ ls
diction-1.11      diction-1.11.tar.gz
```

Tip: The `diction` program, like all GNU Project software, follows certain standards for source code packaging. Most other source code available in the Linux ecosystem also follows this standard. One element of the standard is that by unpacking the source code tar file, a directory will be created containing the source tree and that this directory will be named `project-x.xx` thus containing both the project's name and version number. This scheme allows easy installation of multiple versions of the same program. However, it is often a good idea to examine the layout of the tree before unpacking it. Some projects will not create the directory, but rather deliver the files directly into the current directory. This will make a mess in your otherwise well-organized `src` directory. To avoid this, use the following command to examine the contents of the tar file:

```
tar tzvf tarfile | head
```

Examining The Source Tree

Unpacking the tar file results in the creation of a new directory named `diction-1.11`. This directory contains the source tree. Let's look inside:

```
[me@linuxbox src]$ cd diction-1.11
[me@linuxbox diction-1.11]$ ls
config.guess  diction.c      getopt.c      nl
config.h.in   diction.pot    getopt.h      nl.po
config.sub    diction.spec   getopt_int.h  README
configure     diction.spec.in  INSTALL      sentence.c
configure.in  diction.texi.in  install-sh    sentence.h
COPYING       en              Makefile.in   style.1.in
de            en_GB          misc.c        style.c
de.po         en_GB.po       misc.h        test
diction.1.in  getopt1.c      NEWS
```

In it, we see a number of files. Programs belonging to the GNU Project, as well as many others, will supply the documentation files `README`, `INSTALL`, `NEWS`, and `COPYING`. These files contain the description of the program, how to build and install it, and its licensing terms. It is always a good idea to carefully read the `README` and `INSTALL` files before attempting to build the program.

The other interesting files in this directory are the ones ending with the `.c` and `.h`:

```
[me@linuxbox diction-1.11]$ ls *.c
diction.c  getopt1.c  getopt.c  misc.c  sentence.c  style.c
```

```
[me@linuxbox diction-1.11]$ ls *.h
getopt.h  getopt_int.h  misc.h  sentence.h
```

The `.c` files contain the two C programs supplied by the package (`style` and `diction`) divided into modules. It is common practice for large programs to be broken into smaller and easier to manage pieces. The source code files are ordinary text and can be examined with `less`:

```
[me@linuxbox diction-1.11]$ less diction.c
```

The `.h` files are known as *header files*. These too, are ordinary text. Header files contain descriptions of the routines contained in a source code file or library. In order for the compiler to connect the modules, it must receive a description of all the modules needed to complete the entire program. Near the beginning of the `diction.c` file we see this line:

```
#include "getopt.h"
```

This instructs the compiler to read the file `getopt.h` as it reads the source code in `diction.c` in order to “know” about what’s in the `getopt.c`. The `getopt.c` file supplies routines that are shared by both the `style` and `diction` programs.

Above the `include` statement for `getopt.h` we see some other `include` statements such as these:

```
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

These also refer to header files, but these refer to header files that live outside the current source tree. They are supplied by the system to support the compilation of every program. If we look in `/usr/include` we can see them:

```
[me@linuxbox diction-1.11]$ ls /usr/include
```

The header files in this directory were installed when we installed the compiler.

Building The Program

Most programs build with a simple two command sequence:

```
./configure  
make
```

The `configure` program is a shell script that is supplied with the source tree. Its job is to analyze the *build environment*. Most source code is designed to be *portable*. That is, it is designed to build on more than one kind of Unix-like system. But in order to do that, the source code may need to undergo slight adjustments during the build to accommodate differences between systems. `configure` also checks to see that necessary external tools and components are installed. Let's run `configure`. Since `configure` is not located where the shell expects programs to be located, we must explicitly tell the shell its location by prefixing the command with `./` indicating that the program is located in the current working directory:

```
[me@linuxbox diction-1.11]$ ./configure
```

`configure` will output a lot of messages as it tests and configures the build. When it finishes, it will look something like this:

```
checking libintl.h presence... yes  
checking for libintl.h... yes  
checking for library containing gettext... none required  
configure: creating ./config.status  
config.status: creating Makefile  
config.status: creating diction.1  
config.status: creating diction.texi  
config.status: creating diction.spec  
config.status: creating style.1  
config.status: creating test/rundiction  
config.status: creating config.h  
[me@linuxbox diction-1.11]$
```

What's important here is that there are no error messages. If there were, the configuration failed and the program will not build until the errors are corrected.

We see `configure` created several new files in our source directory. The most important one is `Makefile`. `Makefile` is a configuration file that instructs the `make` program exactly how to build the program. Without it, `make` will refuse to run. `Makefile` is an ordinary text file, so we can view it:

```
[me@linuxbox diction-1.11]$ less Makefile
```

The `make` program takes as input a *makefile* (which is normally named `Makefile`) which describes the relationships and dependencies among the components that comprise the finished program.

The first part of the `makefile` defines variables that are substituted in later sections of the `makefile`. For example we see the line:

```
CC=          gcc
```

which defines the C compiler to be `gcc`. Later in the `makefile`, we see one instance where it get used:

```
diction:      diction.o sentence.o misc.o getopt.o getopt1.o
              $(CC) -o $@ $(LDFLAGS) diction.o sentence.o misc.o \
              getopt.o getopt1.o $(LIBS)
```

A substitution is performed here and the value `$(CC)` is replaced by `gcc` at run time.

Most of the `makefile` consists of lines such as the one above defining a *target*, in this case the executable file `diction`, and the files on which it is dependent. The remaining lines describe the command(s) needed to create the target from its components. We see in this example that the executable file `diction` (one of the final end products) depends on the existence of `diction.o`, `sentence.o`, `misc.o`, `getopt.o`, and `getopt1.o`. Later on in the `makefile` we see definitions of each of these as targets:

```
diction.o:    diction.c config.h getopt.h misc.h sentence.h
getopt.o:     getopt.c getopt.h getopt_int.h
getopt1.o:    getopt1.c getopt.h getopt_int.h
misc.o:       misc.c config.h misc.h
sentence.o:   sentence.c config.h misc.h sentence.h
style.o:      style.c config.h getopt.h misc.h sentence.h
```

However, we don't see any command specified for them. This is handled by a general target earlier in the file that describes the command used to compile any .C file into a .O file:

```
.c.o:
    $(CC) -c $(CPPFLAGS) $(CFLAGS) $<
```

This all seems very complicated. Why not simply list all the steps to compile the parts and be done with it? The answer to this will become clear in a moment. In the meantime, let's run `make` and build our programs:

```
[me@linuxbox diction-1.11]$ make
```

The `make` program will run, using the contents of `Makefile` to guide its actions. It will produce a lot of messages.

When it finishes, we will see all the targets are now present in our directory:

```
[me@linuxbox diction-1.11]$ ls
config.guess  de.po          en              install-sh     sentence.c
config.h      diction        en_GB           Makefile       sentence.h
config.h.in   diction.1     en_GB.mo       Makefile.in    sentence.o
config.log    diction.1.in  en_GB.po       misc.c         style
config.status diction.c     getopt1.c      misc.h         style.1
config.sub    diction.o     getopt1.o      misc.o         style.1.in
configure     diction.pot   getopt.c       NEWS           style.c
configure.in  diction.spec  getopt.h       nl             style.o
COPYING       diction.spec.in  getopt_int.h  nl.mo         test
de            diction.texi   getopt.o       nl.po
de.mo         diction.texi.in  INSTALL       README
```

Among the files, we see `diction` and `style`, the programs that we set out to build. Congratulations are in order! We just compiled our first programs from source code!

But just out of curiosity, let's run `make` again:

```
[me@linuxbox diction-1.11]$ make
make: Nothing to be done for `all'.
```

It only produces this strange message. What's going on? Why didn't it build the

program again? Ah, this is the magic of `make`. `make` only builds what needs building. With all of the targets present, `make` determined that there was nothing to do. We can demonstrate this by deleting one of the targets and running `make` again to see what it does. Let's get rid of one of the intermediate ones:

```
[me@linuxbox diction-1.11]$ rm getopt.o
[me@linuxbox diction-1.11]$ make
```

and we see that `make` rebuilds it and re-links the `diction` and `style` programs since they depend on the missing module. This behavior also points out another important feature of `make`: it keeps targets up-to-date. `make` insists that targets be newer than their dependencies. This makes perfect sense as a programmer will often update a bit of source code and then use `make` to build a new version of the finished product. `make` insures that everything that needs building based on the updated code is built. If we use the `touch` program to "update" one of the source code files, we can see this happen:

```
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me me 37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me me 33125 2007-03-30 17:45 getopt.c
[me@linuxbox diction-1.11]$ touch getopt.c
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me me 37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me me 33125 2009-03-05 06:23 getopt.c
[me@linuxbox diction-1.11]$ make
```

After `make` runs, we see that it has restored the target to being newer than the dependency:

```
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me me 37164 2009-03-05 06:24 diction
-rw-r--r-- 1 me me 33125 2009-03-05 06:23 getopt.c
```

This ability of `make` to intelligently build only that which needs building is a great benefit to programmers. While the time savings may not be very apparent with our small project, it is very significant with larger projects. Remember, the Linux kernel (a program that undergoes continuous modification and improvement) is several *million* lines of code.

Installing The Program

Well-packaged source code will often include a special `make` target called `install`. This target will install the final product in a system directory for use. Usually this directory is `/usr/local/bin`, the traditional location for locally built software. However, this directory is not normally writable by ordinary users, so we must become the superuser to perform the installation:

```
[me@linuxbox diction-1.11]$ sudo make install
```

After we perform the installation, we can check that the program is ready-to-go:

```
[me@linuxbox diction-1.11]$ which diction
/usr/local/bin/diction
[me@linuxbox diction-1.11]$ man diction
```

And there we have it!

Summing Up

In this chapter, we have seen how three simple commands:

```
./configure
```

```
make
```

```
make install
```

can be used to build many source code packages. We have also seen the important role that `make` plays in the maintenance of programs. The `make` program can be used for any task that needs to maintain a target/dependency relationship, not just compiling source code.

Further Reading

The Wikipedia has good articles on compilers and the `make` program:

<http://en.wikipedia.org/wiki/Compiler>

[http://en.wikipedia.org/wiki/Make_\(software\)](http://en.wikipedia.org/wiki/Make_(software))

The GNU Make Manual:

http://www.gnu.org/software/make/manual/html_node/index.html

Part 5 - Writing Shell Scripts

25 - Writing Your First Script

Throughout the preceding chapters we have assembled an arsenal of command line tools used to solve many kinds of computing problems. Even so, we can go further, having the computer do more of the work (and having us do less) by joining our tools together into programs of our own design. We do this by writing *shell scripts*.

What Are Shell Scripts?

In the simplest terms, a shell script is a file containing a series of commands. The shell reads this file and carries out the commands as though they been entered directly on the command line.

The shell is somewhat unique in that it is both a powerful command line interface to the system, and a scripting language interpreter. As we will see, most of the things that can be done on the command line can be done in scripts and most of the things that can be done in scripts can be done on the command line.

We have covered many shell features, but we have focused on those features most often used directly on the command line. The shell also provides a set of features usually (but not always) used when writing programs.

How To Write A Shell Script

To successfully create and run a shell script, we need to do three things:

1. **Write a script.** Shell scripts are ordinary text files. So we need a text editor to write them. The best text editors will provide *syntax highlighting* allowing us to see a color coded view of the elements of the script. Syntax highlighting will help us spot certain kinds of common errors. `vim`, `gedit`, `kate`, and many other editors are good candidates for writing scripts.
2. **Make the script executable.** The system is rather fussy about not letting any old text file be treated as a program, and for good reason! We need to set the script file's permissions to allow execution.
3. **Put the script somewhere the shell can find it.** The shell automatically searches certain directories for executable files when no explicit pathname is specified.

For maximum convenience, we will place our scripts in these directories.

Script File Format

In keeping with programming tradition, we'll create a "hello world" program to demonstrate an extremely simple script. So let's fire up our text editors and enter the following script:

```
#!/bin/bash
# This is our first script.
echo 'Hello World!'
```

The last line of our script is pretty familiar, just an `echo` command with a string argument. The second line is also familiar. It looks like a comment that we have seen used in many of the configuration files we have examined and edited. One thing about comments in shell scripts is that they may also appear at the end of lines like so:

```
echo 'Hello World!' # This is a comment too
```

Everything from the `#` symbol onward on the line is ignored. Like many things, this works on the command line too:

```
[me@linuxbox ~]$ echo 'Hello World!' # This is a comment too
Hello World!
```

Though comments are of little use on the command line, they will work.

The first line of our script is a little mysterious. It looks like it should be a comment since it starts with `#`, but it looks too purposeful to be just that. The `#!` character sequence is, in fact, a special construct called a *shebang*. The shebang is used to tell the system the name of the interpreter that should be used to execute the script that follows. Every shell script should include this as its first line.

Let's save our script file as `hello_world`.

Executable Permissions

The next thing we have to do is make our script executable. This is easily done using

chmod:

```
[me@linuxbox ~]$ ls -l hello_world
-rw-r--r-- 1 me      me      63 2009-03-07 10:10 hello_world
[me@linuxbox ~]$ chmod 755 hello_world
[me@linuxbox ~]$ ls -l hello_world
-rwxr-xr-x 1 me      me      63 2009-03-07 10:10 hello_world
```

There are two common permission settings for scripts, 755 for scripts that everyone can execute, and 700 for scripts that only the owner can execute. Note that scripts must be readable in order to be executed.

Script File Location

With the permissions set, we can now execute our script:

```
[me@linuxbox ~]$ ./hello_world
Hello World!
```

But in order for the script to run, we must precede the script name with an explicit path because if we don't, we get this:

```
[me@linuxbox ~]$ hello_world
bash: hello_world: command not found
```

Why is this? What makes our script different from other programs? As it turns out, nothing. Our script is fine. Its location is the problem. Back in Chapter 12, we discussed the `PATH` environment variable and its effect on how the system searches for executable programs. To recap, the system searches a list of directories each time it needs to find an executable program when no explicit path is specified. This is how the system knows to execute `/bin/ls` when we type `ls` at the command line. The `/bin` directory is one of the directories that the system automatically searches. The list of directories is held within an environment variable named `PATH`. The `PATH` variable contains a colon-separated list of directories to be searched. We can view the contents of `PATH`:

```
[me@linuxbox ~]$ echo $PATH
/home/me/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

Here we see our list of directories. If our script were located in any of the directories in the list, our problem would be solved. Notice the first directory in the list, `/home/me/bin`. Most Linux distributions configure the `PATH` variable to contain a `bin` directory in the user's home directory to allow users to execute their own programs. So if we create the `bin` directory and place our script within it, it should start to work like other programs:

```
[me@linuxbox ~]$ mkdir bin
[me@linuxbox ~]$ mv hello_world bin
[me@linuxbox ~]$ hello_world
Hello World!
```

And so it does.

If the `PATH` variable does not contain the directory, we can easily add it by including this line in our `.bashrc` file:

```
export PATH=~/.bin:$PATH
```

After this change is made, it will take effect in each new terminal session. To apply the change to the current terminal session, we must have the shell re-read the `.bashrc` file. This can be done by “sourcing” it:

```
[me@linuxbox ~]$ . .bashrc
```

The dot (`.`) command is a synonym for the `source` command, a shell builtin which reads a specified file of shell commands and treats it like input from the keyboard.

Note: Ubuntu automatically adds the `~/bin` directory to the `PATH` variable if the `~/bin` directory exists when the user's `.bashrc` file is executed. So on Ubuntu systems, if we create the `~/bin` directory and then log out and log in again, everything works.

Good Locations For Scripts

The `~/bin` directory is a good place to put scripts intended for personal use. If we write a script that everyone on a system is allowed to use, the traditional location is `/usr/local/bin`. Scripts intended for use by the system administrator are often

located in `/usr/local/sbin`. In most cases, locally supplied software, whether it be scripts or compiled programs, should be placed in the `/usr/local` hierarchy and not in `/bin` or `/usr/bin`. These directories are specified by the Linux Filesystem Hierarchy Standard to contain only files supplied and maintained by the Linux distributor.

More Formatting Tricks

One of the key goals of serious script writing is ease of *maintenance*, that is the ease at which a script may be modified by its author or others to adapt it to changing needs. One way easy maintenance is facilitated by making the script easy to read and understand.

Long Option Names

Many of the commands we have studied feature both short and long option names. For instance, the `ls` command has many options that can be expressed in either short or long form. For example:

```
[me@linuxbox ~]$ ls -ad
```

and:

```
[me@linuxbox ~]$ ls --all --directory
```

are equivalent commands. In the interests of reduced typing, short options are preferred when entering options on the command line, but when writing scripts, long options can provide improved readability.

Indentation And Line Continuation

When employing long commands, readability can be enhanced by spreading the command over several lines. In Chapter 18, we looked at a particularly long example of the `find` command:

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec  
chmod 0600 '{}' ';' \) -or \( -type d -not -perm 0711 -exec chmod  
0711 '{}' ';' \)
```

Obviously this command is a little hard to figure out at first glance. In a script this

command might be easier to understand if written this way:

```
find playground \  
  \( \  
    -type f \  
    -not -perm 0600 \  
    -exec chmod 0600 '{}' ';' \  
  ) \  
  -or \  
  \( \  
    -type d \  
    -not -perm 0711 \  
    -exec chmod 0711 '{}' ';' \  
  )
```

By using line continuations (backslash-linefeed sequences) and indentation, the logic of this complex command is more clearly described to the reader. This technique works on the command line too, though it is seldom used as it is very awkward to type and edit. One difference between a script and the command line is that the script may employ tab characters to achieve indentation whereas the command line cannot since tabs are used to activate completion.

Summing Up

In this first chapter of scripting, we have looked at how scripts are written and made to easily execute on our system.. We also saw how we may use various formatting techniques to improve the readability (and thus the maintainability) of our scripts. In future chapters, ease of maintenance will come up again and again as a central principle in good script writing.

Further Reading

For “Hello World” programs and examples in various programming languages, see:

http://en.wikipedia.org/wiki/Hello_world

This Wikipedia article talks more about the shebang mechanism:

[http://en.wikipedia.org/wiki/Shebang_\(Unix\)](http://en.wikipedia.org/wiki/Shebang_(Unix))

26 – Starting A Project

Starting with this chapter, we will begin a project to build a program. The purpose of this project is to see how various shell features are used to fashion programs and more importantly, fashion *good* programs.

The program we will write is a *report generator*. It will present various statistics about our system and its status and will produce this report in HTML format so we can view it with a web browser such as Firefox, or Konqueror.

Programs are usually built up in a series of stages, with each stage adding features and capabilities. The first stage of our program will produce a very minimal HTML page that contains no system information. That will come later.

First Stage: Minimal Document

The first thing we need to know is the format of a well-formed HTML document. It looks like this:

```
<HTML>
  <HEAD>
    <TITLE>Page Title</TITLE>
  </HEAD>
  <BODY>
    Page body.
  </BODY>
</HTML>
```

If we enter this into our text editor and save the file as `foo.html`, we can use the following URL in Firefox to view the file:

```
file:///home/username/foo.html
```

The first stage of our program will be able to output this HTML file to standard output. We can write a program to do this pretty easily. Let's start our text editor and create a new file named `~/bin/sys_info_page`:

```
[me@linuxbox ~]$ vim ~/bin/sys_info_page
```

and enter the following program:

```
#!/bin/bash

# Program to output a system information page

echo "<HTML>"
echo "    <HEAD>"
echo "        <TITLE>Page Title</TITLE>"
echo "    </HEAD>"
echo "    <BODY>"
echo "        Page body."
echo "    </BODY>"
echo "</HTML>"
```

Our first attempt at this problem contains a shebang, a comment (always a good idea) and a sequence of `echo` commands, one for each line of output. After saving the file, we'll make it executable and attempt to run it:

```
[me@linuxbox ~]$ chmod 755 ~/bin/sys_info_page
[me@linuxbox ~]$ sys_info_page
```

When the program runs, we should see the text of the HTML document displayed on the screen, as the `echo` commands in the script send their output to standard output. We'll run the program again and redirect the output of the program to the file `sys_info_page.html` so that we can view the result with a browser:

```
[me@linuxbox ~]$ sys_info_page > sys_info_page.html
```

So far, so good.

When writing programs, it's always a good idea to strive for simplicity and clarity. Maintenance is easier when a program is easy to read and understand. Not to mention, it can make the program easier to write by reducing the amount of typing. Our present version of the program works fine, but it could be simpler. We could actually combine all the `echo` commands into one, which will certainly make it easier to add more lines to the program's output. So, let's change our program to this:

```
#!/bin/bash

# Program to output a system information page

echo "<HTML>
  <HEAD>
    <TITLE>Page Title</TITLE>
  </HEAD>
  <BODY>
    Page body.
  </BODY>
</HTML>"
```

A quoted string may include newlines and therefore contain multiple lines of text. The shell will keep reading the text until it encounters the closing quotation mark. It works this way on the command line too:

```
[me@linuxbox ~]$ echo "<HTML>
>   <HEAD>
>           <TITLE>Page Title</TITLE>
>   </HEAD>
>   <BODY>
>           Page body.
>   </BODY>
> </HTML>"
```

The leading “>” character is the shell prompt contained in the PS2 shell variable. It appears whenever we are typing a multi-line statement into the shell. This feature is a little obscure right now, but later, when we cover multi-line programming statements, this feature will turn out to be quite handy.

Second Stage: Adding A Little Data

Now that our program can generate a minimal document let’s put some data in the report. To do this, we will make the following changes:

```
#!/bin/bash

# Program to output a system information page

echo "<HTML>
  <HEAD>
    <TITLE>System Information Report</TITLE>
```

```
        </HEAD>
        <BODY>
            <H1>System Information Report</H1>
        </BODY>
    </HTML>"
```

We added a page title and a heading to the body of the report.

Variables And Constants

There is an issue with our script however. Notice how the string “System Information Report” is repeated? That is redundant. Of course with our tiny script it’s not a problem, but let’s imagine that our script was really long and we had multiple instances of this string. If we wanted to change the title to something else, we would have to change it in multiple places which could be a lot of work. What if we could arrange the script so that the string only appeared once and not multiple times? That would make future maintenance of the script much easier. Here’s how we could do that:

```
#!/bin/bash
# Program to output a system information page
title="System Information Report"
echo "<HTML>
    <HEAD>
        <TITLE>$title</TITLE>
    </HEAD>
    <BODY>
        <H1>$title</H1>
    </BODY>
</HTML>"
```

By creating a *variable* named `title` and assigning it the value “System Information Report” we can take advantage of parameter expansion and place the string in multiple locations.

So, how do we create a variable? Simple, just use it. When the shell encounters one, it automatically creates it. This differs from many programming languages where variables must be explicitly *declared* or defined before use. The shell is very lax about this which can lead to some problems. For example, consider this scenario played out on the command line:

```
[me@linuxbox ~]$ foo="yes"
[me@linuxbox ~]$ echo $foo
yes
[me@linuxbox ~]$ echo $fool

[me@linuxbox ~]$
```

We first assign the value “yes” to the variable `foo`, then display its value with `echo`. Next we display the value of the variable name misspelled as “fool” and get a blank result. This is because the shell happily created the variable `fool` when it encountered it and gave it the default value of nothing, or empty. From this, we learn that we must pay attention to our spelling! It’s also important to understand what is really happening in this example. From our previous look at how the shell performs expansions, we know that the command:

```
[me@linuxbox ~]$ echo $foo
```

undergoes parameter expansion and results in:

```
[me@linuxbox ~]$ echo yes
```

Whereas the command:

```
[me@linuxbox ~]$ echo $fool
```

expands into:

```
[me@linuxbox ~]$ echo
```

The empty variable expands into nothing! This can play havoc with commands that require arguments. Here’s an example:

```
[me@linuxbox ~]$ foo=foo.txt
[me@linuxbox ~]$ foo1=foo1.txt
[me@linuxbox ~]$ cp $foo $fool
cp: missing destination file operand after `foo.txt'
```

```
Try `cp --help` for more information.
```

We assign values to two variables, `foo` and `foo1`. We then perform a `cp` but misspell the name of the second argument. After expansion, the `cp` command is only sent one argument though it requires two.

There are some rules about variable names:

1. Variable names may consist of alphanumeric characters (letters and numbers) and underscore characters.
2. The first character of a variable name must be either a letter or an underscore.
3. Spaces and punctuation symbols are not allowed.

The word “variable” implies a value that changes, and in many applications variables are used this way, however in our application, our variable `title` is used as a *constant*. A constant is just like a variable except its value does not change. In an application that performs geometric calculations, we might define a constant `PI` and assign it the value `3.1415` instead of using the number literally throughout our program. The shell makes no distinction between variables and constants, its mostly for the programmers convenience. A common convention is to use upper case letters to designate constants and lower case letters for true variables. We will modify our script to comply with this convention:

```
#!/bin/bash
# Program to output a system information page
TITLE="System Information Report For $HOSTNAME"
echo "<HTML>
  <HEAD>
    <TITLE>$TITLE</TITLE>
  </HEAD>
  <BODY>
    <H1>$TITLE</H1>
  </BODY>
</HTML>"
```

We also took the opportunity to jazz up our title by adding the value of the shell variable `HOSTNAME`. This is the network name of the machine.

Note: The shell actually does provide a way to enforce the immutability of

constants through the use of the `declare` builtin command with the `-r` (read-only) option. Had we assigned `TITLE` this way:

```
declare -r TITLE="Page Title"
```

the shell would prevent any subsequent assignment to `TITLE`. This feature is rarely used but it exists for very formal scripts.

Assigning Values To Variables And Constants

Here is where our knowledge of expansion really starts to pay off. As we have seen, variables are assigned values this way:

```
variable=value
```

where *variable* is the name of the variable and *value* is a string. Unlike some other programming languages, the shell does not care about the type of data assigned to a variable, it treats them all as strings. You can force the shell to restrict the assignment to integers by using the `declare` command with the `-i` option, but like setting variables as read-only, this is rarely done.

Note that in an assignment, there must be no spaces between the variable name, the equals sign, and the value. So what can the value consist of? Anything that we can expand into a string:

```
a=z                # Assign the string "z" to variable a.
b="a string"       # Embedded spaces must be within quotes.
c="a string and $a" # Other expansions such as variables can be
                  # expanded into the assignment.
d=$(ls -l foo.txt) # Results of a command.
e=$((5 * 7))      # Arithmetic expansion.
f="\t\ta string\n" # Escape sequences such as tabs and newlines.
```

Multiple variable assignments may be done on a single line:

```
a=5 b="a string"
```

During expansion, variable names may be surrounded by optional curly braces “{}”. This is useful in cases where a variable name becomes ambiguous due to its surrounding context. Here we try to change the name of a file from `myfile` to `myfile1` using a variable:


```
[me@linuxbox ~]$ filename="myfile"
[me@linuxbox ~]$ touch $filename
[me@linuxbox ~]$ mv $filename $filename1
mv: missing destination file operand after `myfile'
Try `mv --help' for more information.
```

This attempt fails because the shell interprets the second argument of the `mv` command as a new (and empty) variable. The problem can be overcome this way:

```
[me@linuxbox ~]$ mv $filename ${filename}1
```

By adding the surrounding braces, the shell no longer interprets the trailing `1` as part of the variable name.

We'll take this opportunity to add some data to our report, namely the date and time the report was created and the user name of the creator:

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

echo "<HTML>
  <HEAD>
    <TITLE>$TITLE</TITLE>
  </HEAD>
  <BODY>
    <H1>$TITLE</H1>
    <P>$TIME_STAMP</P>
  </BODY>
</HTML>"
```

Here Documents

We've looked at two different methods of outputting our text, both using the `echo` command. There is a third way called a *here document* or *here script*. A here document is an additional form of I/O redirection where we embed a body of text into our script and feed it into the standard input of a command. It works like this:

```
command << token
```

text

token

where *command* is the name of command that accepts standard input and *token* is a string used to indicate the end of the embedded text. We'll modify our script to use a here document:

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

cat << _EOF_
<HTML>
    <HEAD>
        <TITLE>$TITLE</TITLE>
    </HEAD>
    <BODY>
        <H1>$TITLE</H1>
        <P>$TIME_STAMP</P>
    </BODY>
</HTML>
_EOF_
```

Instead of using `echo`, our script now uses `cat` and a here document. The string `_EOF_` (meaning “End Of File”, a common convention) was selected as the token and marks the end of the embedded text. Note that the token must appear alone and there must be no trailing spaces on the line.

So what's the advantage of using a here document? It's mostly the same as `echo`, except that by default, single and double quotes within here documents lose their special meaning to the shell. Here is a command line example:

```
[me@linuxbox ~]$ foo="some text"
[me@linuxbox ~]$ cat << _EOF_
> $foo
> "$foo"
> '$foo'
> \ $foo
> _EOF_
some text
"some text"
```

```
'some text'  
$foo
```

As we can see the shell pays no attention to the quotation marks. It treats them as ordinary characters. This allows us to embed quotes freely within a here document. This could turn out to be handy for our report program.

Here documents can be used with any command that accepts standard input. In this example, we use a here document to pass a series of commands to the `ftp` program to retrieve a file from a remote FTP server:

```
#!/bin/bash  
  
# Script to retrieve a file via FTP  
  
FTP_SERVER=ftp.nl.debian.org  
FTP_PATH=/debian/dists/lenny/main/installer-i386/current/images/cdrom  
REMOTE_FILE=debian-cd_info.tar.gz  
  
ftp -n << _EOF_  
open $FTP_SERVER  
user anonymous me@linuxbox  
cd $FTP_PATH  
hash  
get $REMOTE_FILE  
bye  
_EOF_  
ls -l $REMOTE_FILE
```

If we change the redirection operator from “<<” to “<<-”, the shell will ignore leading tab characters in the here document. This allows a here document to be indented which can improve readability:

```
#!/bin/bash  
  
# Script to retrieve a file via FTP  
  
FTP_SERVER=ftp.nl.debian.org  
FTP_PATH=/debian/dists/lenny/main/installer-i386/current/images/cdrom  
REMOTE_FILE=debian-cd_info.tar.gz  
  
ftp -n <<- _EOF_  
    open $FTP_SERVER  
    user anonymous me@linuxbox
```

```
cd $FTP_PATH
hash
get $REMOTE_FILE
bye
_EOF_

ls -l $REMOTE_FILE
```

Summing Up

In this chapter, we started a project that will carry us through the process of building a successful script. We introduced the concept of variables and constants and how they can be employed. They are the first of many applications we will find for parameter expansion. We also looked at how to produce output from our script and various methods for embedding blocks of text.

Further Reading

For more information about HTML, see the following articles and tutorials:

<http://en.wikipedia.org/wiki/Html>

http://en.wikibooks.org/wiki/HTML_Programming

<http://html.net/tutorials/html/>

The `bash` man page includes a section entitled “HERE DOCUMENTS” which has a full description of this feature.

27 – Top-Down Design

As programs get larger and more complex, they become more difficult to design, code and maintain. As with any large project, it is often a good idea to break large complex tasks into a series of small, simple tasks. Let's imagine that we were trying to describe a common, everyday task, going to the market to buy food, to a person from Mars. We might describe the overall process as the following series of steps:

1. Get in car.
2. Drive to market.
3. Park Car.
4. Enter market.
5. Purchase food.
6. Return to car.
7. Drive home.
8. Park car.
9. Enter house.

However, a person from Mars is likely to need more detail. We could further break down the subtask "Park car" into this series of steps:

1. Find parking space.
2. Drive car into space.
3. Turn off motor.
4. Set parking brake.
5. Exit car.
6. Lock car.

The "Turn off motor" subtask could further be broken down into steps including "Turn off ignition" and "Remove ignition key" and so on until every step of the entire process of going to the market has been fully defined.

This process of identifying the top-level steps and developing increasingly detailed views of those steps is called *top-down design*. This technique allows us to break large complex tasks into many small, simple tasks. Top-down design is a common method of designing

programs and one that is well suited to shell programming in particular.

In this chapter, we will use top-down design to further develop our report generator script.

Shell Functions

Our script currently performs the following steps to generate the HTML document:

1. Open page.
2. Open page header.
3. Set page title.
4. Close page header.
5. Open page body.
6. Output page heading.
7. Output time stamp.
8. Close page body.
9. Close page.

For our next stage of development, we will add some additional tasks between steps 7 and 8. These will include:

- System uptime and load. This is the amount of time since the last shutdown or reboot and the average number of tasks currently running on the processor over several time intervals.
- Disk space. The overall use of space on the system's storage devices.
- Home space. The amount of storage space being used by each user.

If we had a command for each of these tasks, we could add them to our script simply through command substitution:

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

cat << _EOF_
<HTML>
    <HEAD>
```

```

        <TITLE>${TITLE}</TITLE>
    </HEAD>
    <BODY>
        <H1>${TITLE}</H1>
        <P>${TIME_STAMP}</P>
        $(report_uptime)
        $(report_disk_space)
        $(report_home_space)
    </BODY>
</HTML>
_EOF_

```

We could create these additional commands two ways. We could write three separate scripts and place them in directory listed in our PATH or we could embed the scripts within our program as *shell functions*. As we have mentioned before, shell functions are “mini-scripts” that are located inside of other scripts and can act as autonomous programs. Shell functions have two syntactic forms:

```
function name {
    commands
    return
}
```

and

```
name () {
    commands
    return
}
```

where *name* is the name of the function and *commands* are a series of commands contained within the function. Both forms are equivalent and may be used interchangeably. Below we see a script that demonstrates the use of a shell function:

```

1  #!/bin/bash
2
3  # Shell function demo
4
5  function funct {
6      echo "Step 2"
7      return
8  }
9
10 # Main program starts here
11
12 echo "Step 1"

```

```
13  funct
14  echo "Step 3"
```

As the shell reads the script, it ignores lines lines 1 through 11 as those lines consist of comments and the function definition. Execution begins at line 12 with an `echo` command. Line 13 *calls* the shell function `funct` and the shell executes the function just as it would any other command. Program control then moves to line 6 and the second `echo` command is executed. Line 7 is executed next. Its `return` command terminates the function and returns control to the program at the line following the function call (line 14) and the final `echo` command is executed. Note that, in order for function calls to be recognized as shell functions and not interpreted as the names of external programs, shell functions must appear in the script before they are called.

We'll add minimal shell functions to our script:

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

report_uptime () {
    return
}

report_disk_space () {
    return
}

report_home_space () {
    return
}

cat << _EOF_
<HTML>
  <HEAD>
    <TITLE>$TITLE</TITLE>
  </HEAD>
  <BODY>
    <H1>$TITLE</H1>
    <P>$TIME_STAMP</P>
    $(report_uptime)
    $(report_disk_space)
    $(report_home_space)
```



```
</BODY>
</HTML>
_EOF_
```

Shell function names follow the same rules as variables. A function must contain at least one command. The `return` command (which is actually optional) satisfies the requirement.

Local Variables

In the scripts we have written so far, all the variables (including constants) have been *global variables*. Global variables maintain their existence throughout the program. This is fine for many things, but it can sometimes complicate the use of shell functions. Inside shell functions, it is often desirable to have *local variables*. Local variables are only accessible within the shell function in which they are defined and cease to exist once the shell function terminates.

Having local variables allows the programmer to use variables with names that may already exist in the script globally or in other shell functions without having to worry about potential name conflicts.

Here is an example script that demonstrates how local variables are defined and used:

```
#!/bin/bash
# local-vars: script to demonstrate local variables
foo=0 # global variable foo
funct_1 () {
    local foo # variable foo local only to funct_1
    foo=1
    echo "funct_1: foo = $foo"
}
funct_2 () {
    local foo # variable foo local only to funct_2
    foo=2
    echo "funct_2: foo = $foo"
}
```

```
echo "global:  foo = $foo"
funct_1
echo "global:  foo = $foo"
funct_2
echo "global:  foo = $foo"
```

As we can see, local variables are defined by preceding the variable name with the word `local`. This creates a variable that is local only to the shell function in which it is defined. Once outside the shell function, the variable no longer exists. When we run this script we see the results:

```
[me@linuxbox ~]$ local-vars
global:  foo = 0
funct_1: foo = 1
global:  foo = 0
funct_2: foo = 2
global:  foo = 0
```

We see that the assignment of values to the local variable `foo` within both shell functions has no effect on the value of `foo` defined outside the functions.

This feature allows shell functions to be written so that they remain independent of each other and of the script in which they appear. This is very valuable as it helps prevent one part of a program from interfering with another. It also allows shell functions to be written so that they can be portable. That is, they may be cut and pasted from script to script as needed.

Keep Scripts Running

While developing our program, it is useful to keep the program in a runnable state. By doing this, and testing frequently, we can detect errors early in the development process. This will make debugging problems much easier. For example, if we run the program, make a small change, run the program again and there is a problem, it's very likely that the most recent change is the source of the problem. By adding the empty functions, called *stubs* in programmer-speak, we can verify the logical flow of our program at an early stage. When constructing a *stub*, it a good idea to include something that provides feedback to the programmer that shows the logical flow is being carried out. If we look at the output of our script now:

```
[me@linuxbox ~]$ sys_info_page
<HTML>
```

```
<HEAD>
  <TITLE>System Information Report For twin2</TITLE>
</HEAD>
<BODY>
  <H1>System Information Report For linuxbox</H1>
  <P>Generated 03/19/2009 04:02:10 PM EDT, by me</P>

</BODY>
</HTML>
```

we see that there are some blank lines in our output after the time stamp, but we can't be sure of the cause. If we change the functions to include some feedback:

```
report_uptime () {
    echo "Function report_uptime executed."
    return
}

report_disk_space () {
    echo "Function report_disk_space executed."
    return
}

report_home_space () {
    echo "Function report_home_space executed."
    return
}
```

and run the script again:

```
[me@linuxbox ~]$ sys_info_page
<HTML>
  <HEAD>
    <TITLE>System Information Report For linuxbox</TITLE>
  </HEAD>
  <BODY>
    <H1>System Information Report For linuxbox</H1>
    <P>Generated 03/20/2009 05:17:26 AM EDT, by me</P>
    Function report_uptime executed.
    Function report_disk_space executed.
    Function report_home_space executed.
  </BODY>
</HTML>
```

we now see that, in fact, our three functions are being executed.

With our function framework in place and working, it's time to flesh out some of the function code. First, the `report_uptime` function:

```
report_uptime () {
    cat <<- _EOF_
        <H2>System Uptime</H2>
        <PRE>$(uptime)</PRE>
        _EOF_
    return
}
```

It's pretty straightforward. We use a here document to output a section header and the output of the `uptime` command surrounded by `<PRE>` tags to preserve the formatting of the command. The `report_disk_space` function is similar:

```
report_disk_space () {
    cat <<- _EOF_
        <H2>Disk Space Utilization</H2>
        <PRE>$(df -h)</PRE>
        _EOF_
    return
}
```

This function uses the `df -h` command to determine the amount of disk space. Lastly, we'll build the `report_home_space` function:

```
report_home_space () {
    cat <<- _EOF_
        <H2>Home Space Utilization</H2>
        <PRE>$(du -sh /home/*)</PRE>
        _EOF_
    return
}
```

We use the `du` command with the `-sh` options to perform this task. This however is not a complete solution to the problem. While this will work on some systems (Ubuntu, for example), it will not work on others. The reason is that many systems set the permissions of home directories to prevent them from being world-readable, which is a reasonable security measure. On these system, the `report_home_space` function, as written,

will only work if our script is run with superuser privileges. A better solution would be if the script could adjust its behavior according to the privileges of the user. We will take this up in the next chapter..

Further Reading

The Wikipedia has many articles on software design philosophy. Here are a couple of good ones:

http://en.wikipedia.org/wiki/Top-down_design

<http://en.wikipedia.org/wiki/Subroutines>

28 – Flow Control: Branching With if

In the last chapter we were presented with a problem. How can we make our report generator script adapt to the privileges of the user running the script? The solution to this problem will require a way to “change directions” within our script based on a condition, or the results of a test. In programming terms, we need the program to *branch*.

Let’s consider a simple example of logic expressed in *pseudocode*, a simulation of a computer language, but intended for human consumption:

X = 5

If X = 5, then:

 Say “X equals 5.”

Otherwise:

 Say “X is not equal to 5.”

This is an example of a branch. Based on the condition, “Does X = 5?” do one thing, “Say X equals 5,” else do another thing “Say X is not equal to 5.”

if

Using the shell, we can code the logic above as follows:

```
x=5
if [ $x = 5 ]; then
    echo "x equals 5."
else
    echo "x does not equal 5."
fi
```

or enter it directly at the command line (slightly shortened):

```
[me@linuxbox ~]$ x=5
[me@linuxbox ~]$ if [ $x = 5 ]; then echo "equals 5"; else echo "not
equals 5."; fi
equals 5
[me@linuxbox ~]$ x=0
[me@linuxbox ~]$ if [ $x = 5 ]; then echo "equals 5"; else echo "not
equals 5."; fi
not equals 5
```

In this example, we execute the command twice. Once with the value of `x` set to 5 which results in the string “equals 5” being output and the second time with the value of `x` set to 0 which results in the string “not equals 5” being output.

The `if` statement has the following syntax:

```
if commands; then
    commands
[elif commands; then
    commands...]
[else
    commands]
fi
```

where *commands* is a list of commands. This is a little confusing at first glance. But before we can clear this up we have to look at how the shell evaluates the success or failure of a command.

Exit Status

Commands (including the scripts and shell functions we write) issue a value to the system when they terminate called an *exit status*. This value, which is a number in the range of 0 to 255 indicates the success or failure of the command’s execution. By convention, a value of zero indicates success and any other value indicates failure. The shell provides a parameter that we can use to examine the exit status. Here we see it in action:

```
[me@linuxbox ~]$ ls -d /usr/bin
/usr/bin
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ ls -d /bin/usr
ls: cannot access /bin/usr: No such file or directory
[me@linuxbox ~]$ echo $?
2
```

In this example, we execute the `ls` command twice. The first time, the command executes successfully. If we display the value of the parameter `$?`, we see that it is zero. We execute the `ls` command a second time producing an error and examine the parameter `$?` again. This time it contains a 2 indicating that the command encountered an error. Some commands use different exit status values to provide diagnostics for errors, while many commands simply exit with a value of one when they fail. Man pages often include a section entitled “Exit Status” describing what codes are used. However, a zero always indicates success.

The shell provides two extremely simple commands that do nothing except terminate with either a zero or one exit status. The `true` command always executes successfully and the `false` command always executes unsuccessfully:

```
[me@linuxbox ~]$ true
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ false
[me@linuxbox ~]$ echo $?
1
```

We can use these commands to see how the `if` statement works. What the `if` statement really does is evaluate the success or failure of commands:

```
[me@linuxbox ~]$ if true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if false; then echo "It's true."; fi
[me@linuxbox ~]$
```

The command `echo "It's true."` is executed when the command following `if` executes successfully and it is not executed when the command following `if` does not execute successfully. If a list of commands follows `if`, the last command in the list is evaluated:

```
[me@linuxbox ~]$ if false; true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if true; false; then echo "It's true."; fi
[me@linuxbox ~]$
```


Test

By far, the command used most frequently with `if` is `test`. The `test` command performs a variety of checks and comparisons. It has two equivalent forms:

```
test expression
```

and the more popular:

```
[ expression ]
```

where *expression* is an expression that is evaluated as either true or false. The `test` command returns an exit status of zero when the expression is true and one when the expression is false.

File Expressions

The following expressions are used to evaluate the status of files:

Expression	Is True If:
<i>file1</i> -ef <i>file2</i>	<i>file1</i> and <i>file2</i> have the same inode numbers (the two file names refer to the same file by hard linking.)
<i>file1</i> -nt <i>file2</i>	<i>file1</i> is newer than <i>file2</i> .
<i>file1</i> -ot <i>file2</i>	<i>file1</i> is older than <i>file2</i> .
-b <i>file</i>	<i>file</i> exists and is a block special (device) file.
-c <i>file</i>	<i>file</i> exists and is character special (device) file.
-d <i>file</i>	<i>file</i> exists and is a directory.
-e <i>file</i>	<i>file</i> exists.
-f <i>file</i>	<i>file</i> exists and is a regular file.
-g <i>file</i>	<i>file</i> exists and is set-group-ID.
-G <i>file</i>	<i>file</i> exists and is owned by the effective group ID.
-k <i>file</i>	<i>file</i> exists and has its “sticky bit” set.
-L <i>file</i>	<i>file</i> exists and is a symbolic link.
-O <i>file</i>	<i>file</i> exists and is owned by the effective user ID.
-p <i>file</i>	<i>file</i> exists and is a named pipe.
-r <i>file</i>	<i>file</i> exists and is readable (has readable permission for the effective user.)
-s <i>file</i>	<i>file</i> exists and has a length greater than zero.

<code>-S file</code>	<code>file</code> exists and is a network socket.
<code>-t fd</code>	<code>fd</code> is a file descriptor directed to/from the terminal. This can be used to determine if standard input/output/error is being redirected.
<code>-u file</code>	<code>file</code> exists and it is setuid.
<code>-w file</code>	<code>file</code> exists and is writable (has write permission for the effective user.)
<code>-x file</code>	<code>file</code> exists and is executable (has execute/search permission for the effective user.)

Here we have a script that demonstrates some of the file expressions:

```
#!/bin/bash
# test-file: Evaluate the status of a file
FILE=~/.bashrc
if [ -e "$FILE" ]; then
    if [ -f "$FILE" ]; then
        echo "$FILE is a regular file."
    fi
    if [ -d "$FILE" ]; then
        echo "$FILE is a directory."
    fi
    if [ -r "$FILE" ]; then
        echo "$FILE is readable."
    fi
    if [ -w "$FILE" ]; then
        echo "$FILE is writable."
    fi
    if [ -x "$FILE" ]; then
        echo "$FILE is executable/searchable."
    fi
else
    echo "$FILE does not exist"
    exit 1
fi
exit
```

The script evaluates the file assigned to the constant `FILE` and displays its results as the evaluation is performed. Two interesting things to note about this script. First, notice

how the parameter `$FILE` is quoted within the expressions. This is not required, but is a defense against the parameter being empty. If the parameter expansion of `$FILE` were to result in an empty value, it would result in a shell syntax error. Using the quotes around the parameter prevents this. Second, notice the presence of the `exit` commands near the end of the script. The `exit` command accepts a single, optional argument which becomes the script's exit status. When no argument is passed, the exit status defaults to zero. Using `exit` in this way allows the script to indicate failure if `$FILE` expands to the name of a nonexistent file. The `exit` command appearing on the last line of the script is there as a formality. When a script "runs off the end" (reaches end of file), it terminates with an exit status of zero by default anyway.

Similarly, shell functions can return an exit status by including a numeric argument to the `return` statement. If we were to convert the script above to a shell function to include it in a larger program, we could replace the `exit` commands with `return` statements and get the desired behavior:

```
test_file () {
    # test-file: Evaluate the status of a file

    FILE=~/.bashrc

    if [ -e "$FILE" ]; then
        if [ -f "$FILE" ]; then
            echo "$FILE is a regular file."
        fi
        if [ -d "$FILE" ]; then
            echo "$FILE is a directory."
        fi
        if [ -r "$FILE" ]; then
            echo "$FILE is readable."
        fi
        if [ -w "$FILE" ]; then
            echo "$FILE is writable."
        fi
        if [ -x "$FILE" ]; then
            echo "$FILE is executable/searchable."
        fi
    else
        echo "$FILE does not exist"
        return 1
    fi
}
```

String Expressions

The following expressions are used to evaluate strings:

Expression	Is True If...
<code>-n string</code>	The length of <i>string</i> is greater than zero.
<code>-z string</code>	The length of <i>string</i> is zero.
<code>string1 = string2</code> <code>string1 == string2</code>	<i>string1</i> and <i>string2</i> are equal. Single or double equal signs may be used.
<code>string1 != string2</code>	<i>string1</i> and <i>string2</i> are not equal.
<code>string1 > string2</code>	<i>string1</i> sorts after <i>string2</i> .
<code>string1 < string2</code>	<i>string1</i> sorts before <i>string2</i> .

Warning: the `>` and `<` expression operators must be quoted (or escaped with a backslash) when used with `test`. If they are not, they will be interpreted by the shell as redirection operators with potentially destructive results.

Here is a script that incorporates string expressions:

```
#!/bin/bash
# test-string: evaluate the value of a string
ANSWER=maybe
if [ -z "$ANSWER" ]; then
    echo "There is no answer." >&2
    exit 1
fi
if [ "$ANSWER" = "yes" ]; then
    echo "The answer is YES."
elif [ "$ANSWER" = "no" ]; then
    echo "The answer is NO."
elif [ "$ANSWER" = "maybe" ]; then
    echo "The answer is MAYBE."
else
    echo "The answer is UNKNONN."
fi
```

In this script, we evaluate the constant `ANSWER`. We first determine if the string is empty. If it is, we terminate the script and set the exit status to one. Notice the redirection that is applied to the `echo` command. This redirects the error message “There

is no answer.” to standard error which is the “proper” thing to do with error messages. If the string is not empty, we evaluate the value of the string to see if it is equal to either “yes”, “no”, or “maybe.” We do this by using `elif`, which is short for “else if.” By using `elif` we are able to construct a more complex logical test.

Integer Expressions

The following expressions are used with integers:

Expression	Is True If...
<i>integer1</i> -eq <i>integer2</i>	<i>integer1</i> is equal to <i>integer2</i> .
<i>integer1</i> -ne <i>integer2</i>	<i>integer</i> is not equal to <i>integer2</i> .
<i>integer1</i> -le <i>integer2</i>	<i>integer1</i> is less than or equal to <i>integer2</i> .
<i>integer1</i> -lt <i>integer2</i>	<i>integer1</i> is less than <i>integer2</i> .
<i>integer1</i> -ge <i>integer2</i>	<i>integer1</i> is greater than or equal to <i>integer2</i> .
<i>integer1</i> -gt <i>integer2</i>	<i>integer1</i> is greater than <i>integer2</i> .

And a script that demonstrates them:

```
#!/bin/bash
# test-integer: evaluate the value of an integer.
INT=-5
if [ -z "$INT" ]; then
    echo "INT is empty." >&2
    exit 1
fi
if [ $INT -eq 0 ]; then
    echo "INT is zero."
else
    if [ $INT -lt 0 ]; then
        echo "INT is negative."
    else
        echo "INT is positive."
    fi
    if [ $((INT % 2)) -eq 0 ]; then
        echo "INT is even."
    else
        echo "INT is odd."
```

```
fi
fi
```

The interesting part of the script is how it determines whether an integer is even or odd. By performing a modulo 2 operation on the number, which divides the number by two and returns the remainder, it can tell if the number is odd or even.

A More Modern Version Of test

Recent versions of `bash` include a compound command that acts as an enhanced replacement for `test`. It uses the following syntax:

```
[[ expression ]]
```

where, like `test`, *expression* is an expression that evaluates to either a true or false result. The `[[]]` command is very similar to `test` (it supports all of its expressions) but adds an important new string expression:

```
string1 =~ regex
```

which returns true if *string1* is matched by the extended regular expression *regex*. This opens up a lot of possibilities for performing such tasks as data validation. In our earlier example of the integer expressions, the script would fail if the constant `INT` contained anything except an integer. The script needs a way to verify that the constant contains an integer. Using `[[]]` with the `=~` string expression operator, we could improve the script this way:

```
#!/bin/bash
# test-integer2: evaluate the value of an integer.
INT=-5
if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [ $INT -eq 0 ]; then
        echo "INT is zero."
    else
        if [ $INT -lt 0 ]; then
            echo "INT is negative."
        else
            echo "INT is positive."
        fi
        if [ $((INT % 2)) -eq 0 ]; then
            echo "INT is even."
        else
```

```

        echo "INT is odd."
    fi
fi
else
    echo "INT is not an integer." >&2
    exit 1
fi

```

By applying the regular expression, we are able to limit the value of INT to only strings that begin with an optional minus sign followed by one or more numerals. This expression also eliminates the possibility of empty values.

Another added feature of `[[]]` is that the `==` operator supports pattern matching the same way pathname expansion does. For example:

```

[me@linuxbox ~]$ FILE=foo.bar
[me@linuxbox ~]$ if [[ $FILE == foo.* ]]; then
> echo "$FILE matches pattern 'foo.*'"
> fi
foo.bar matches pattern 'foo.*'

```

This makes `[[]]` useful for evaluating file and path names.

Combining Expressions

It's also possible to combine expressions to create more complex evaluations. Expressions are combined by using logical operators. We saw these many chapters ago when we learned about the `find` command. There are three logical operations for `test` and `[[]]`. They are AND, OR and NOT. `test` and `[[]]` use different operators to represent these operations :

Operation	test	[[]]
AND	-a	&&
OR	-o	
NOT	!	!

Here's an example of an AND operation. The following script determines if an integer is within a range of values:

```
#!/bin/bash

# test-integer3: determine if an integer is within a
# specified range of values.

MIN_VAL=1
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-[0-9]+$ ]]; then
    if [[ $INT -ge $MIN_VAL && $INT -le $MAX_VAL ]]; then
        echo "$INT is within $MIN_VAL to $MAX_VAL."
    else
        echo "$INT is out of range."
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

In this script, we determine if the value of integer `INT` is lies between the values of `MIN_VAL` and `MAX_VAL`. This is performed by a single use of `[[]]` which includes two expressions separated by the `&&` operator. We could have also coded this using `test`:

```
if [ $INT -ge $MIN_VAL -a $INT -le $MAX_VAL ]; then
    echo "$INT is within $MIN_VAL to $MAX_VAL."
else
    echo "$INT is out of range."
fi
```

The `!` negation operator reverses the outcome of an expression. It returns true if an expression is false and it returns false if an expression is true. In the following script, we modify the logic of our evaluation to find values of `INT` that are outside the specified range:

```
#!/bin/bash

# test-integer4: determine if an integer is outside a
# specified range of values.

MIN_VAL=1
```



```
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [[ ! ( $INT -ge $MIN_VAL && $INT -le $MAX_VAL ) ]]; then
        echo "$INT is outside $MIN_VAL to $MAX_VAL."
    else
        echo "$INT is in range."
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

We also include parentheses around the expression for grouping. If these were not included, the negation would only apply to the first expression and not the combination of the two. Coding this with `test` would be done this way:

```
if [ ! \( $INT -ge $MIN_VAL -a $INT -le $MAX_VAL \) ]; then
    echo "$INT is outside $MIN_VAL to $MAX_VAL."
else
    echo "$INT is in range."
fi
```

Since all expressions and operators used by `test` are treated as command arguments by the shell (unlike `[[]]`), characters which have special meaning to `bash`, such as `<`, `>`, `(`, and `)`, must be escaped.

Seeing that `test` and `[[]]` do roughly the same thing, which is better? `test` is traditional (and part of POSIX) whereas `[[]]` is specific to `bash`. Its important to know how to use `test` since it is very widely used but `[[]]` is clearly more useful and is easier to code.

Portability Is The Hobgoblin Of Little Minds

If you talk to “real” Unix people, you quickly discover that many of them don’t like Linux very much. They regard it as impure and unclean. One argument you encounter is how everything should be “portable.” This means that if you write a script it should be able to run, unchanged, on any Unix-like system.

Unix people have good reason to believe this. Having seen what proprietary extensions to commands and shells did to the Unix world before POSIX, they are naturally wary of the effect of Linux on their beloved OS.

But portability has a serious downside. It prevents progress. It requires that things are always done using the lowest common denominator techniques. In the case of shell programming, it means making everything compatible with `sh`, the original Bourne shell.

This is the excuse that proprietary vendors use to justify their proprietary extensions, only they call them “innovations.” But they are really lock-in devices for their customers.

The GNU tools have no such restrictions. They encourage portability by supporting standards and by being universally available. You can install `bash` and the other GNU tools on almost any kind of system, even Windows, without cost. So feel free to use all the features of `bash`. It’s *really* portable.

Control Operators: Another Way To Branch

`bash` provides two control operators that can perform branching. The `&&` (AND) and `||` (OR) operators work like the similar logical operators in the `[[]]` command. This is the syntax:

```
command1 && command2
```

and

```
command1 || command2
```

The behavior of these is important to understand. With the `&&` operator *command1* is executed and *command2* is executed if, *and only if*, *command1* is successful. With the `||` operator, *command1* is executed and *command2* is executed if, *and only if*, *command1* is unsuccessful.

In practical terms, it means that we can do something like this:

```
[me@linuxbox ~]$ mkdir temp && cd temp
```

This will create a directory named `temp`, and if it succeeds, the current working directory will be changed to `temp`. The second command is attempted only if the `mkdir` command is successful. Likewise, a command like this:

```
[me@linuxbox ~]$ [ -d temp ] || mkdir temp
```

will test for the existence of the directory `temp`, and only if the test fails, will the directory be created. This type of construct is very handy in scripts for handling errors (a subject we will discuss more in later chapters.) For example, in a script we could do this:

```
[ -d temp ] || exit 1
```

If the script requires the directory `temp`, and it does not exist, then the script will terminate with an exit status of one.

Summing Up

We started out this chapter with a question. How could we make our `sys_info_page` script detect if the user had permission to read all the home directories? With our knowledge of `if`, we can solve the problem by adding this code to the `report_home_space` function:

```
report_home_space () {
    if [[ $(id -u) -eq 0 ]]; then
        cat <<- _EOF_
            <H2>Home Space Utilization (All Users)</H2>
            <PRE>$(du -sh /home/*)</PRE>
        _EOF_
    else
        cat <<- _EOF_
            <H2>Home Space Utilization ($USER)</H2>
            <PRE>$(du -sh $HOME)</PRE>
        _EOF_
    fi
    return
}
```

We evaluate the output of the `id` command. With the `-u` option, `id` outputs the numeric user ID number of the effective user. The superuser is always zero and every other user is a number greater than zero. Knowing this, we can construct two different here documents, one taking advantage of superuser privileges, and the other, restricted to the user's own home directory.

We going to take a break from the `sys_info_page` program, but don't worry. We'll be back. In the meantime, we'll cover some topics that we'll need when we resume our

work.

Further Reading

There are several sections of the `bash` man page that further detail the topics covered in this chapter:

- Lists (covers the control operators `|` and `&&`)
- Compound Commands (covers `[[]` and `if`)
- CONDITIONAL EXPRESSIONS
- SHELL BUILTIN COMMANDS (covers `test`)

Further, the Wikipedia has a good article on the concept of pseudocode:

<http://en.wikipedia.org/wiki/Pseudocode>

29 – Reading Keyboard Input

The scripts we have written so far lack a common feature found in most computer programs – *interactivity*. That is, the ability of the program to interact with the user. While many programs don't need to be interactive, they are smart enough to get the job done on their own, some programs benefit from being able to accept input directly from the user. Take for example this script from the previous chapter:

```
#!/bin/bash
# test-integer2: evaluate the value of an integer.
INT=-5
if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [ $INT -eq 0 ]; then
        echo "INT is zero."
    else
        if [ $INT -lt 0 ]; then
            echo "INT is negative."
        else
            echo "INT is positive."
        fi
        if [ $((INT % 2)) -eq 0 ]; then
            echo "INT is even."
        else
            echo "INT is odd."
        fi
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

Each time we want to change the value of `INT` we have to edit the script. It would be much better if the script could ask the user for a value. In this chapter, we will begin to look at how we can add this capability to our programs.

read – Read Values From Standard Input

The `read` builtin is used to read a single line of standard input. This command can be used to read keyboard input or, when redirection is employed, read a line of data from a file. The command has the following syntax:

```
read [-options] [variable...]
```

where *options* is one or more of the available options listed below and *variable* is the name of one or more variables used to hold the input value. If no variable name is supplied the shell variable `REPLY` contains the line of data.

Basically, `read` assigns fields from standard input to the specified variables. If we modify our integer evaluation script to use `read`, it might look like this:

```
#!/bin/bash

# read-integer: evaluate the value of an integer.

echo -n "Please enter an integer -> "
read int

if [[ "$int" =~ ^-?[0-9]+$ ]]; then
    if [ $int -eq 0 ]; then
        echo "$int is zero."
    else
        if [ $int -lt 0 ]; then
            echo "$int is negative."
        else
            echo "$int is positive."
        fi
        if [ $((int % 2)) -eq 0 ]; then
            echo "$int is even."
        else
            echo "$int is odd."
        fi
    fi
else
    echo "Input value is not an integer." >&2
    exit 1
fi
```

We use `echo` with the `-n` option (which suppresses the trailing newline on output) to display a prompt, then use `read` to input a value for the variable `int`. Running this script results in this:

```
[me@linuxbox ~]$ read-integer
Please enter an integer -> 5
5 is positive.
5 is odd.
```

read can assign input to multiple variables:

```
#!/bin/bash

# read-multiple: read multiple values from keyboard

echo -n "Enter one or more values > "
read var1 var2 var3 var4 var5

echo "var1 = '$var1'"
echo "var2 = '$var2'"
echo "var3 = '$var3'"
echo "var4 = '$var4'"
echo "var5 = '$var5'"
```

In this script we assign and display up to five values. Notice how read behaves when given different numbers of values:

```
[me@linuxbox ~]$ read-multiple
Enter one or more values > a b c d e
var1 = 'a'
var2 = 'b'
var3 = 'c'
var4 = 'd'
var5 = 'e'
[me@linuxbox ~]$ read-multiple
Enter one or more values > a
var1 = 'a'
var2 = ''
var3 = ''
var4 = ''
var5 = ''
[me@linuxbox ~]$ read-multiple
Enter one or more values > a b c d e f g
var1 = 'a'
var2 = 'b'
var3 = 'c'
var4 = 'd'
var5 = 'e f g'
```

If `read` receives fewer than the expected number, the extra variables are empty, while an excessive amount of input results in the final variable containing all of the extra input.

If no variables are listed after the `read` command, a shell variable, `REPLY` will be assigned all the input:

```
#!/bin/bash

# read-single: read multiple values into default variable

echo -n "Enter one or more values > "
read

echo "REPLY = '$REPLY'"
```

Running this script results in this:

```
[me@linuxbox ~]$ read-single
Enter one or more values > a b c d
REPLY = 'a b c d'
```

Options

`read` supports the following options:

Option	Description
<code>-a array</code>	Assign the input to <i>array</i> starting with index zero. We will cover arrays in an upcoming chapter.
<code>-d delimiter</code>	The first character in the string <i>delimiter</i> is used to indicate end of input rather than a newline character.
<code>-e</code>	Use Readline to handle input. This permits input editing in the same manner as the command line.
<code>-n num</code>	Read <i>num</i> characters of input rather than an entire line.
<code>-p prompt</code>	Display a prompt for input using the string <i>prompt</i> .
<code>-r</code>	Raw mode. Do not interpret backslash characters as escapes.
<code>-s</code>	Silent mode. Do not echo characters to the display as they are typed. This is useful when inputting passwords

	and other confidential information.
<code>-t <i>seconds</i></code>	Timeout. Terminate input after <i>seconds</i> . <code>read</code> returns a non-zero exit status if an input times out.
<code>-u <i>fd</i></code>	Use input from file descriptor <i>fd</i> rather than standard input.

Using the various options, we can do interesting things with `read`. For example, with the `-p` option, we can provide a prompt string:

```
#!/bin/bash
# read-single: read multiple values into default variable
read -p "Enter one or more values > "
echo "REPLY = '$REPLY'"
```

With the `-t` and `-s` options we can write a script that reads “secret” input and times out if the input is not completed in a specified time:

```
#!/bin/bash
# read-secret: input a secret pass phrase
if read -t 10 -sp "Enter secret pass phrase > " secret_pass; then
    echo -e "\nSecret pass phrase = '$secret_pass'"
else
    echo "Input timed out" >&2
    exit 1
fi
```

The script prompts the user for a secret pass phrase and waits ten seconds for input. If the entry is not completed within the specified time, the script exits with an error. Since the `-s` option is included, the characters of the pass phrase are not echoed to the display as they are typed.

IFS

Normally, the shell performs word splitting on the input provided to `read`. As we have seen, this means the multiple words separated by one or more spaces become separate

items on the input line and are assigned to separate variables by `read`. This behavior is configured by a shell variable named `IFS` (for Internal Field Separator.) The default value of `IFS` contains a space, a tab, and a newline character, each of which will separate items from one another.

We can adjust the value of `IFS` to control the separation of fields input to `read`. For example, the `/etc/passwd` file contains lines of data that use the colon character as a field separator. By changing the value of `IFS` to a single colon, we can use `read` to input the contents of `/etc/passwd` and successfully separate fields into different variables. Here we have a script that does just that:

```
#!/bin/bash
# read-ifs: read fields from a file
FILE=/etc/passwd
read -p "Enter a user name > " user_name
file_info=$(grep "^$user_name:" $FILE)
if [ -n "$file_info" ]; then
    IFS=":" read user pw uid gid name home shell <<< "$file_info"
    echo "User = '$user'"
    echo "UID = '$uid'"
    echo "GID = '$gid'"
    echo "Full Name = '$name'"
    echo "Home Dir. = '$home'"
    echo "Shell = '$shell'"
else
    echo "No such user '$user_name'" >&2
    exit 1
fi
```

This script prompts the user to enter the user name of an account on the system then displays the different fields found in the user's record in the `/etc/passwd` file. The script contains two interesting lines. The first is:

```
file_info=$(grep "^$user_name:" $FILE)
```

This line assigns the results of a `grep` command to the variable `file_info`. The regular expression used by `grep` assures that the user name will only match a single line in the `/etc/passwd` file.

The second interesting line is this one:

```
IFS=":" read user pw uid gid name home shell <<< "$file_info"
```

The line consists of three parts, a variable assignment, a `read` command with a list of variable names as arguments, and a strange new redirection operator. We'll look at the variable assignment first.

The shell allows one or more variable assignments to take place before a command. These assignments alter the environment for the command that follows. The effect of the assignment is temporary, only changing the environment for the duration of the command. In our case, the value of `IFS` is changed to a colon character. Alternately, we could have coded it this way:

```
OLD_IFS="$IFS"
IFS=":"
read user pw uid gid name home shell <<< "$file_info"
IFS="$OLD_IFS"
```

where we store the value of `IFS`, assign a new value, perform the `read` command, then restore `IFS` to its original value. Clearly, placing the variable assignment in front of the command is a more concise way of doing the same thing.

The `<<<` operator indicates a *here string*. A here string is like a here document, only shorter, consisting of a single string. In our example, the line of data from the `/etc/passwd` file is fed to the standard input of the `read` command. We might wonder why this rather oblique method was chosen rather than:

```
echo "$file_info" | IFS=":" read user pw uid gid name home shell
```

Well, there's a reason...

You Can't Pipe read

While the `read` command normally takes input from standard input, you cannot do this:

```
echo "foo" | read
```

One would expect this to work, but it does not. The command will appear to succeed but the `REPLY` variable will always be empty. Why is this?

The explanation has to do with the way the shell handles pipelines. In `bash` (and other shells such as `sh`) pipelines create *subshells*. These are copies of the shell and its environment which are used to execute the command in the pipeline. In our example above, `read` is executed in a subshell.

Subshells in Unix-like systems create copies of the environment for the processes to use while they execute. When the processes finishes, the copy of the environment is destroyed. This means that a *subshell can never alter the*

environment of its parent process. `read` assigns variables which then become part of the environment. In the example above, `read` assigns the value “foo” to the variable `REPLY` in its subshell’s environment, but when the command exits, the subshell and its environment are destroyed and the effect of the assignment is lost.

Validating Input

With our new ability to have keyboard input to our programs comes an additional programming challenge, validating input. Very often the difference between a well written program and a poorly written one is in the program’s ability to deal with the unexpected. Quite often the unexpected appears in the form of bad input. We’ve done a little of this with our evaluation programs in the previous chapter where we checked the value of integers and screened out empty values and non-numeric characters. It is important to perform these kinds of programming checks every time a program receives input to guard against invalid data. This is especially important for programs that are shared by multiple users. Omitting these safeguards in the interests of economy might be excused if a program is to be used once and only by the author to perform some special task, but even then, if the program performs dangerous tasks such as deleting files, it might be wise to include data validation, just in case.

Here we have an example program that validates various kinds of input:

```
#!/bin/bash

# read-validate: validate input

invalid_input () {
    echo "Invalid input '$REPLY'" >&2
    exit 1
}

read -p "Enter a single item > "

# input is empty (invalid)
[[ -z $REPLY ]] && invalid_input

# input is multiple items (invalid)
[[ $(echo $REPLY | wc -w) -gt 1 ]] && invalid_input

# is input a valid file name?
if [[ $REPLY =~ ^[-[:alnum:]\.\_]+$ ]]; then
    echo "'$REPLY' is a valid file name."
    if [[ -e $REPLY ]]; then
```

```
        echo "And file '$REPLY' exists."
    else
        echo "However, file '$REPLY' does not exist."
    fi

    # is input a floating point number?
    if [[ $REPLY =~ ^-?[:digit:]*\.[[:digit:]]+$ ]]; then
        echo "'$REPLY' is a floating point number."
    else
        echo "'$REPLY' is not a floating point number."
    fi

    # is input an integer?
    if [[ $REPLY =~ ^-?[:digit:]+$ ]]; then
        echo "'$REPLY' is an integer."
    else
        echo "'$REPLY' is not an integer."
    fi
else
    echo "The string '$REPLY' is not a valid file name."
fi
```

This script prompts the user to enter an item. The item is subsequently analyzed to determine its contents. As we can see, it makes use of many of the scripting concepts that we have covered thus far, including shell functions, `[[]]`, the control operator `&&`, and `if`, as well as, a healthy dose of regular expressions.

Menus

A common type of interactivity is called *menu-driven*. In menu-driven programs, the user is presented with a list of choices and is asked to choose one. For example, we could imagine a program that presented the following:

```
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit

Enter selection [0-3] >
```

Using what we learned from writing our `sys_info_page` program, we can construct a menu-driven program to perform the tasks on the above menu:

```
#!/bin/bash

# read-menu: a menu driven system information program

clear
echo "
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter selection [0-3] > "

if [[ $REPLY =~ ^[0-3]$ ]]; then
    if [[ $REPLY == 0 ]]; then
        echo "Program terminated."
        exit
    fi
    if [[ $REPLY == 1 ]]; then
        echo "Hostname: $HOSTNAME"
        uptime
        exit
    fi
    if [[ $REPLY == 2 ]]; then
        df -h
        exit
    fi
    if [[ $REPLY == 3 ]]; then
        if [[ $(id -u) -eq 0 ]]; then
            echo "Home Space Utilization (All Users)"
            du -sh /home/*
        else
            echo "Home Space Utilization ($USER)"
            du -sh $HOME
        fi
        exit
    fi
else
    echo "Invalid entry." >&2
    exit 1
fi
```

This script is logically divided into two parts. The first part displays the menu and inputs the response from the user. The second part identifies the response and carries out the selected action. Notice the use of the `exit` command in this script. It is used here to prevent the script from executing unnecessary code after an action has been carried out. The presence of multiple exit points in a program is generally a bad idea (it makes

program logic harder to understand) but it works in this script.

Summing Up

In this chapter, we took our first steps toward inactivity allowing users to input data into our programs via the keyboard. Using the techniques presented thus far, it is possible to write many useful programs, such as specialized calculation programs or easy-to-use front ends for arcane command line tools. In the next chapter we will build on the menu driven program concept to make it even better.

Extra Credit

It is important to study the programs in this chapter carefully and have a complete understanding of the way they are logically structured as the programs to come will be increasing complex. As an exercise, rewrite the programs in this chapter using the `test` command rather than the `[[]]` compound command. Hint: use `grep` to evaluate the regular expressions and evaluate its exit status. This will be good practice.

Further Reading

The Bash Reference Manual contains a chapter on builtins which includes the `read` command:

<http://www.gnu.org/software/bash/manual/bashref.html#Bash-Builtins>

30 – Flow Control: Looping With while / until

In the previous chapter, we developed a menu-driven program to produce various kinds of system information. The program works, but it still has a significant usability problem. It only executes a single choice and then terminates. Even worse, if an invalid selection is made, the program terminates with an error without giving the user an opportunity to try again. It would be better if we could somehow construct the program so that it could repeat the menu display and selection over and over until the user chooses to exit the program.

In this chapter, we will look at a programming concept called *looping* which can be used to make portions of programs repeat. The shell provides three compound commands for looping. We will look at two of them in this chapter and the third in a later one.

Looping

Daily life is full of repeated activities. Going to work each day, walking, slicing a carrot are all tasks that involve repeating a series of steps. Let's consider slicing a carrot. If we express this activity in pseudocode, it might look something like this:

1. get cutting board
2. get knife
3. place carrot on cutting board
4. lift knife
5. advance carrot
6. slice carrot
7. if entire carrot sliced, then quit, else go to step 4

Steps 4 through 7 form a *loop*. The actions within the loop are repeated until the condition, "entire carrot sliced" is reached.

while

bash can express a similar idea. Let's say we wanted to display five numbers in

sequential order from one to five. a `bash` script could be constructed as follows:

```
#!/bin/bash

# while-count: display a series of numbers

count=1

while [ $count -le 5 ]; do
    echo $count
    count=$((count + 1))
done
echo "Finished."
```

When executed, this script performs the following:

```
[me@linuxbox ~]$ while-count
1
2
3
4
5
Finished.
```

The syntax of the `while` command is:

```
while commands; do commands; done
```

Like `if`, `while` evaluates the the exit status of a list of commands. As long as the exit status is zero it performs the commands inside the loop. In the script above, the variable `COUNT` is created and assigned a initial value of 1. The `while` command evaluates the exit status of the `test` command. So long as the `test` command returns an exit status of zero the commands within the loop are executed. At the end of each cycle, the `test` command is repeated. After six iterations of the loop, the value of `COUNT` has increased to six and the `test` command no longer returns an exit status of zero and the loop terminates. The program continues with the next statement following the loop.

We can use a *while loop* to improve the `read-menu` program from the previous chapter:

```
#!/bin/bash

# while-menu: a menu driven system information program
```

```
DELAY=3 # Number of seconds to display results

while [[ $REPLY != 0 ]]; do
  clear
  cat <<- _EOF_
    Please Select:

    1. Display System Information
    2. Display Disk Space
    3. Display Home Space Utilization
    0. Quit

  _EOF_
  read -p "Enter selection [0-3] > "

  if [[ $REPLY =~ ^[0-3]$ ]]; then
    if [[ $REPLY == 1 ]]; then
      echo "Hostname: $HOSTNAME"
      uptime
      sleep $DELAY
    fi
    if [[ $REPLY == 2 ]]; then
      df -h
      sleep $DELAY
    fi
    if [[ $REPLY == 3 ]]; then
      if [[ $(id -u) -eq 0 ]]; then
        echo "Home Space Utilization (All Users)"
        du -sh /home/*
      else
        echo "Home Space Utilization ($USER)"
        du -sh $HOME
      fi
      sleep $DELAY
    fi
  else
    echo "Invalid entry."
    sleep $DELAY
  fi
done
echo "Program terminated."
```

By enclosing the menu in a while loop we are able to have the program repeat the menu display after each selection. The loop continues as long as REPLY is not equal to "0" and the menu is displayed again giving the user the opportunity to make another selection. At the end of each action, a sleep command is executed so the program will pause for a specified number of seconds to allow the results of the selection to be seen before the screen is cleared and the menu is redisplayed. Once REPLY is equal to "0", indicating

the “quit” selection, the loop terminates and execution continues with the line following done.

Breaking Out Of A Loop

bash provides two builtin commands that can be used to control program flow inside loops. The `break` command immediately terminates a loop and program control resumes with the next statement following the loop. The `continue` command causes the remainder to the loop to be skipped and program control resumes with the next iteration of the loop. Here we see a version of the `while-menu` program incorporating both `break` and `continue`:

```
#!/bin/bash

# while-menu2: a menu driven system information program

DELAY=3 # Number of seconds to display results

while true; do
    clear
    cat <<- _EOF_
        Please Select:

            1. Display System Information
            2. Display Disk Space
            3. Display Home Space Utilization
            0. Quit

    _EOF_
    read -p "Enter selection [0-3] > "

    if [[ $REPLY =~ ^[0-3]$ ]]; then
        if [[ $REPLY == 1 ]]; then
            echo "Hostname: $HOSTNAME"
            uptime
            sleep $DELAY
            continue
        fi
        if [[ $REPLY == 2 ]]; then
            df -h
            sleep $DELAY
            continue
        fi
        if [[ $REPLY == 3 ]]; then
            if [[ $(id -u) -eq 0 ]]; then
                echo "Home Space Utilization (All Users)"
                du -sh /home/*
            fi
        fi
    fi
done
```

```
                else
                    echo "Home Space Utilization ($USER)"
                    du -sh $HOME
                fi
                sleep $DELAY
                continue
            fi
            if [[ $REPLY == 0 ]]; then
                break
            fi
        else
            echo "Invalid entry."
            sleep $DELAY
        fi
    done
    echo "Program terminated."
```

In this version of the script, we set up an *endless loop* (one that never terminates on its own) by using the `true` command to supply an exit status to `while`. Since `true` will always exit with a exit status of zero, the loop will never end. This is a surprisingly common scripting technique. Since the loop will never end on its own, it's up to the programmer to provide some way to break out of the loop when the time is right. In this script, the `break` command is used to exit the loop when the “0” selection is chosen. The `continue` command has been included at the end of the other script choices to allow for more efficient execution. By using `continue`, the script will skip over code that is not needed when a selection is identified. For example, if the “1” selection is chosen and identified, there is no reason to test for the other selections.

until

The `until` command is much like `while`, only instead of exiting a loop when a non-zero exit status is encountered, it does the opposite. An *until loop* continues until it receives a zero exit status. In our `while-count` script, we continued the loop as long as the value of the `COUNT` variable was less than or equal to five. We could get the same result by coding the script with `until`:

```
#!/bin/bash

# until-count: display a series of numbers

count=1

until [ $count -gt 5 ]; do
```

```
    echo $count
    count=$((count + 1))
done
echo "Finished."
```

By changing the test expression to `$count -gt 5`, `until` will terminate the loop at the correct time. The decision as to which loop to use, `while` or `until` is usually a matter of choosing the one that allows the clearest test to be written.

Reading Files With Loops

`while` and `until` can process standard input. This allows files to be processed with `while` and `until` loops. In the following example, we will display the contents of `distros.txt` file used in earlier chapters:

```
#!/bin/bash

# while-read: read lines from a file

while read distro version release; do
    printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
           $distro \
           $version \
           $release
done < distros.txt
```

To redirect a file to the loop, we place the redirection operator after the `done` statement. The loop will use `read` to input the fields from the redirected file. The `read` command will exit after each line with a zero exit status until the end-of-file is reached. At that point, it will exit with a non-zero exit status thereby terminating the loop. It is also possible to pipe standard input into a loop:

```
#!/bin/bash

# while-read2: read lines from a file

sort -k 1,1 -k 2n distros.txt | while read distro version release; do
    printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
           $distro \
           $version \
           $release
done
```

```
done
```

Here we take the output of the `sort` command and display the stream of text. However, it must be remembered that since a pipe will execute the loop in a subshell, any variables created or assigned within the loop will be lost when the loop terminates.

Summing Up

With the introduction of loops, and our previous encounters with branching, subroutines and sequences, we have covered the major types of flow control used in programs. `bash` has some more tricks up its sleeve but they are refinements on these basic concepts.

Further Reading

The Bash Guide for Beginners from the Linux Documentation Project has some more examples of while loops:

http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_09_02.html

The Wikipedia has an article on loops which is part of a larger article on flow control:

http://en.wikipedia.org/wiki/Control_flow#Loops

31 – Troubleshooting

With our scripts becoming more complex, its time to take a look at what happens when things go wrong and our scripts don't do what what we want. In this chapter, we'll look at some of the common kinds of errors that occur in scripts and describe a few useful techniques that can be used to track down and eradicate problems.

Syntactic Errors

One general class of errors are *syntactic*. Syntactic errors involve mis-typing some element of shell syntax. In most cases, these kinds of errors will lead to the shell refusing to execute the script.

For the following the discussions, we will use this script for demonstrating common types of errors:

```
#!/bin/bash
# trouble: script to demonstrate common errors
number=1
if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
```

As written, this script runs successfully:

```
[me@linuxbox ~]$ trouble
Number is equal to 1.
```

Missing Quotes

If we edit our script and remove the trailing quote from the argument following the first `echo` command:

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ]; then
    echo "Number is equal to 1.
else
    echo "Number is not equal to 1."
fi
```

watch what happens:

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 10: unexpected EOF while looking for
matching `"'
/home/me/bin/trouble: line 13: syntax error: unexpected end of file
```

It generates two errors. Interestingly, the line numbers reported are not where the missing quote was removed but rather much later in the program. We can see why if we follow the program after the missing quote. `bash` will continue looking for the closing quote until it finds one, which it does immediately after the second `echo` command. `bash` becomes very confused after that and the syntax of the `if` command is broken because the `fi` statement is now inside a quoted (but open) string.

In long scripts, this kind of error can be quite hard to find. Using an editor with syntax highlighting will help. If a complete version of `vim` is installed, syntax highlighting can be enabled by entering the command:

```
:syntax on
```

Missing Or Unexpected Tokens

Another common mistake is forgetting to complete a compound command such as `if` or

`while`. Let's look at what happens if we remove the semicolon after the `test` in the `if` command:

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ] then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
```

The result is this:

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 9: syntax error near unexpected token
`else'
/home/me/bin/trouble: line 9: `else'
```

Again, the error message points to a error that occurs later than the actual problem. What happens is really pretty interesting. As we recall, `if` accepts a list of commands and evaluates the exit code of the last command in the list. In our program, we intend this list to consist of a single command, `[`, a synonym for `test`. The `[` command takes what follows it as list of arguments. In our case, three arguments: `$number`, `=`, and `]`. With the semicolon removed, the word `then` is added to the list of arguments which is syntactically legal. The following `echo` command is legal too. It's interpreted as another command in the list of commands that `if` will evaluate for an exit code. The `else` is encountered next but, it's out of place since the shell recognizes it as a *reserved word* (a word that has special meaning to the shell) and not the name of a command, hence the error message.

Unanticipated Expansions

It's possible to have errors that only occur sometimes in a script. Sometimes the script will run fine and other times it will fail because of results of an expansion. If we return our missing semicolon and change the value of `number` to an empty variable, we can demonstrate:

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=

if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
```

Running the script with this change results in the output:

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 7: [: =: unary operator expected
Number is not equal to 1.
```

We get this rather cryptic error message followed by the output of the second `echo` command. The problem is the expansion of the `number` variable within the `test` command. When the command:

```
[ $number = 1 ]
```

undergoes expansion with `number` being empty, the result is this:

```
[ = 1 ]
```

which is invalid and the error is generated. The `=` operator is a binary operator (it requires a value on each side) but the first value is missing so the `test` command expects a unary operator (such as `-z`) instead. Further, since the `test` failed (because of the error), the `if` command receives a non-zero exit code and acts accordingly and the second `echo` command is executed.

This problem can be corrected by adding quotes around the first argument in the `test` command:

```
[ "$number" = 1 ]
```

Then when expansion occurs, the result will be this:

```
[ "" = 1 ]
```

which yields the correct number of arguments. In addition to empty strings, quotes should be used in cases where a value could expand into a multi-word strings such as with file names containing embedded spaces.

Logical Errors

Unlike syntactic errors, *logical errors* do not prevent a script from running. The script will run, but it will not produce the desired result. due to a problem with its logic.. There are countless numbers of possible logical errors, but here are a few of the most common kinds found in scripts:

1. **Incorrect conditional expressions.** It's easy to incorrectly code an if/then/else and have the wrong logic carried out. Sometimes the logic will be reversed or it will be incomplete.
2. **“Off by one” errors.** When coding loops that employ counters, it is possible to overlook that the loop may require that counting start with zero rather than one for the count to conclude at the correct point. These kinds of errors result in either a loop “going off the end” by counting too far, or else missing the last iteration of the loop by terminating one iteration too soon.
3. **Unanticipated situations.** Most logic errors result from a program encountering data or situations that were unforeseen by the programmer. This can also include unanticipated expansions, such as a file name that contains embedded spaces that expands into multiple command arguments rather than a single file name.

Defensive Programming

It is important to verify assumptions when programming. This means a careful evaluation of the exit status of programs and commands that are used by a script. Here is an example based on a true story. An unfortunate system administrator wrote a script to perform a maintenance task on an important server. The script, contained the following two lines of code:

```
cd $dir_name  
rm *
```

There is nothing intrinsically wrong with these two lines, so long as the the directory named in the variable `dir_name` exists. But what happens if it does not? In that case, the `cd` command fails and script continues to the next line and deletes the files in the current working directory. Not the desired outcome at all! The hapless administrator destroyed an important part of the server because of this design decision.

Let's look at some ways this design could be improved. First, it might be wise to make the execution of `rm` contingent on the success of `cd`:

```
cd $dir_name && rm *
```

This way if the `cd` command fails, the `rm` command is not carried out. This is better, but still leaves open the possibility that the variable `dir_name` is empty which would result in the files in the user's home directory being deleted. This could be avoided by also checking to see that `dir_name` actually contains the name of an existing directory:

```
[[ -d $dir_name ]] && cd $dir_name && rm *
```

Often it is best to terminate the script with an error when an situation such as the one above occurs:

```
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
        rm *
    else
        echo "cannot cd to '$dir_name'" >&2
        exit 1
    fi
else
    echo "no such directory: '$dir_name'" >&2
    exit 1
fi
```

Here we check both the name to see that it is that of an existing directory and the success of the `cd` command. If either fails, a descriptive error message is sent to standard error and the script terminates with an exit status of one to indicate a failure.

Verifying Input

A general rule of good programming is if a program accepts input, it must be able to deal

with anything it receives. This usually means that input must be carefully screened to insure that only valid input is accepted for further processing. We saw an example of this in the previous chapter when we studied the `read` command. One script contained the following test to verify a menu selection:

```
[[ $REPLY =~ ^[0-3]$ ]]
```

This test is very specific. It will only return a zero exit status if the string returned by the user is a numeral in the range of zero to three. Nothing else will be accepted. Sometimes these sorts of tests can be very challenging to write but the effort is needed to produce a high quality script.

Design Is A Function Of Time

When I was a college student studying industrial design, a wise professor described how the amount of design performed on a project was determined by the amount of time given to the designer. If you were given five minutes to design a device “that kills flies”, you designed a flyswatter. If you were given five months, you might come up with a laser-guided “anti-fly system” instead.

The same principle applies to programming. Sometimes a “quick and dirty” script will do if it’s only going to be used once and only used by the programmer. Those kinds of scripts are common and should be developed quickly to make the effort economical. Such scripts don’t need a lot of comments and defensive checks. On the other hand, if a script is intended for *production use*, that is, a script that will be used over and over for an important task or by multiple users, it needs more careful development.

Testing

Testing is an important step in every kind of software development including scripts. There is a saying in the open source world, “release early, release often” which reflects this fact. By releasing early and often, software gets more exposure to use and testing. When developing scripts, testing should be done early and often, for experience has shown that bugs are much easier to find and much less expensive to fix if they are found early in the development cycle.

In our previous discussion of stubs, we saw how they can be used to verify program flow. From the earliest stages of script development, they are a valuable technique to check the progress of our work.

Let's look at the file deletion problem above and see how this could be coded for easy testing. Testing the original fragment of code would be dangerous since its purpose is to delete files, but we could modify the code to make the test safe:

```
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
        echo rm * # TESTING
    else
        echo "cannot cd to '$dir_name'" >&2
        exit 1
    fi
else
    echo "no such directory: '$dir_name'" >&2
    exit 1
fi
exit # TESTING
```

Since the error conditions already output useful messages, we don't have to add any. The most important change is to place an `echo` command just before the `rm` command to allow the command and its expanded argument list to be displayed rather than the command actually being executed. This change allows safe execution of the code. At the end of the code fragment we place an `exit` command to conclude the test and prevent any other part of the script from being carried out. The need for this will vary according to the design of the script.

We also include some comments that act as “markers” for our test-related changes. These can be used to help find and remove the changes when testing is complete.

Test Cases

To perform useful testing, it's important to develop and apply good *test cases*. This is done by carefully choosing input data or operating conditions that reflect *edge* and *corner* cases. In our code fragment (which is very simple), we want to know how the code performs under three specific conditions:

1. `dir_name` contains the name of an existing directory
2. `dir_name` contains the name of a non-existent directory
3. `dir_name` is empty

By performing the test with each of these conditions, good *test coverage* is achieved.

Just as with design, testing is a function of time as well. Not every script feature needs to be extensively tested. It's really a matter of determining what is most important. Our code fragment, since it could be so potentially destructive if it malfunctioned, deserves

careful consideration during both its design and testing.

Debugging

If testing reveals a problem with a script, the next step is debugging. “A problem” usually means that the script is in some way not performing to the programmers expectations. If this is the case, we need to carefully determine exactly what the script is actually doing and why. Finding bugs can sometimes involve a lot of detective work.

A well designed script will try and help. It should be programmed defensively to detect abnormal conditions and provide useful feedback to the user. Sometimes however problems are really strange and unexpected and more involved techniques are required.

Finding The Problem Area

In some scripts, particularly long ones, it is sometimes useful to isolate the area of the script that is related to the problem. This won't always be the actual error, but isolation will often provide insights into the actual cause. One technique that can be used to isolate code is by “commenting out” sections a script. For example, our file deletion fragment could be modified to see if the removed section was related to an error:

```
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
        rm *
    else
        echo "cannot cd to '$dir_name'" >&2
        exit 1
    fi
# else
# echo "no such directory: '$dir_name'" >&2
# exit 1
fi
```

By placing comment symbols at the beginning of each line in a logical section of a script, we prevent that section from being executed. Testing can be performed again to see if the removal of the code has any impact on the behavior of the bug.

Tracing

Often, bugs are really cases of unexpected logical flow within a script. That is, portions of the script are either never being executed or are being executed in the wrong order or at the wrong time. To view the actual flow of the program, we use a technique called *tracing*.

One tracing method involves placing informative messages in a script that display the location of execution. We can add messages to our code fragment:

```
echo "preparing to delete files" >&2
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
        echo "deleting files" >&2
        rm *
    else
        echo "cannot cd to '$dir_name'" >&2
        exit 1
    fi
else
    echo "no such directory: '$dir_name'" >&2
    exit 1
fi
echo "file deletion complete" >&2
```

We send the messages to standard error to separate them from normal output. We also do not indent the lines containing the messages to make them easier to find when it's time to remove them.

Now when the script is executed, it's possible to see that the file deletion has been performed:

```
[me@linuxbox ~]$ deletion-script
preparing to delete files
deleting files
file deletion complete
[me@linuxbox ~]$
```

bash also provides a method of tracing implemented by the `-x` option and the `set` command with the `-x` option. Using our earlier `trouble` script, we can activate tracing for the entire script by adding the `-x` option to the first line:

```
#!/bin/bash -x

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ]; then
    echo "Number is equal to 1."
```



```
else
    echo "Number is not equal to 1."
fi
```

When executed, the results look like this:

```
[me@linuxbox ~]$ trouble
+ number=1
+ '[' 1 = 1 ']'
+ echo 'Number is equal to 1.'
Number is equal to 1.
```

With tracing enabled, we see the commands being performed with expansions applied. The leading plus signs indicate the display of the trace to distinguish them from lines of regular output. The plus sign is the default character for trace output. It is contained in the PS4 (prompt string 4) shell variable. The contents of this variable can be adjusted to make the prompt more useful. Here we modify the contents of the variable to include the current line number in the script where the trace is performed. Note that the single quotes are required to prevent expansion until the prompt is actually used:

```
[me@linuxbox ~]$ export PS4='$LINENO + '
[me@linuxbox ~]$ trouble
5 + number=1
7 + '[' 1 = 1 ']'
8 + echo 'Number is equal to 1.'
Number is equal to 1.
```

To perform a trace on a selected portion of a script rather than the entire script, we can use the `set` command with the `-x` option:

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

set -x # Turn on tracing
if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
```

```
fi
set +x # Turn off tracing
```

We use the `set` command with the `-x` option to activate tracing and the `+x` option to deactivate tracing. This technique can be used to examine multiple portions of a troublesome script.

Examining Values During Execution

It is often useful, along with tracing, to display the content of variables to see the internal workings of a script while it is being executed. Applying additional `echo` statements will usually do the trick:

```
#!/bin/bash
# trouble: script to demonstrate common errors
number=1
echo "number=$number" # DEBUG
set -x # Turn on tracing
if [ $number = 1 ]; then
    echo "Number is equal to 1."
else
    echo "Number is not equal to 1."
fi
set +x # Turn off tracing
```

In this trivial example, we simply display the value of the variable `number` and mark the added line with a comment to facilitate its later identification and removal. This technique is particularly useful when watching the behavior of loops and arithmetic within scripts.

Summing Up

In this chapter we looked at just a few of the problems that can crop up during script development. Of course, there are many more. The techniques described here will enable finding most common bugs. Debugging is a fine art that develops with experience, both in knowing how to avoid bugs (testing constantly throughout development) and finding bugs (effective use of tracing.)

Further Reading

The Wikipedia has a couple of short articles on syntactic and logical errors:

http://en.wikipedia.org/wiki/Syntax_error

http://en.wikipedia.org/wiki/Logic_error

There are many online resources for the technical aspects of `bash` programming:

<http://woledge.org:8000/BashPitfalls>

<http://tldp.org/LDP/abs/html/gotchas.html>

http://www.gnu.org/software/bash/manual/html_node/Reserved-Word-Index.html

Eric Raymond's *The Art of Unix Programming* is a great resource for learning the basic concepts found in well-written Unix programs. Many of these ideas apply to shell scripts as well:

<http://www.faqs.org/docs/artu/>

<http://www.faqs.org/docs/artu/ch01s06.html>

For really heavy-duty debugging, there is the Bash Debugger:

<http://bashdb.sourceforge.net/>

32 – Flow Control: Branching With case

In this chapter we continue with our look at flow control. In Chapter 29, we constructed some simple menus and built the logic used to act on a user's selection. To do this we used a series of `if` commands to identify the which of the possible choices has been selected. This type of problem appears frequently in programming, so much so that a flow control mechanism is found in many programming (including the shell) to handle multiple-choice decisions.

case

The `bash` multiple-choice compound command is called `case`. It has the following syntax:

```
case word in
    [pattern [| pattern]...) commands ;;]...
esac
```

If we look at the `read-menu` program from Chapter 29, we see the logic used to act on a user's selection:

```
#!/bin/bash
# read-menu: a menu driven system information program

clear
echo "
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter selection [0-3] > "

if [[ $REPLY =~ ^[0-3]$ ]]; then
    if [[ $REPLY == 0 ]]; then
```

```

        echo "Program terminated."
        exit
    fi
    if [[ $REPLY == 1 ]]; then
        echo "Hostname: $HOSTNAME"
        uptime
        exit
    fi
    if [[ $REPLY == 2 ]]; then
        df -h
        exit
    fi
    if [[ $REPLY == 3 ]]; then
        if [[ $(id -u) -eq 0 ]]; then
            echo "Home Space Utilization (All Users)"
            du -sh /home/*
        else
            echo "Home Space Utilization ($USER)"
            du -sh $HOME
        fi
        exit
    fi
else
    echo "Invalid entry." >&2
    exit 1
fi

```

Using `CASE`, we can replace this logic with something simpler:

```

#!/bin/bash

# case-menu: a menu driven system information program

clear
echo "
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter selection [0-3] > "

case $REPLY in
    0)    echo "Program terminated."
          exit
          ;;

```

```
1)  echo "Hostname: $HOSTNAME"
    uptime
    ;;
2)  df -h
    ;;
3)  if [[ $(id -u) -eq 0 ]]; then
    echo "Home Space Utilization (All Users)"
    du -sh /home/*
    else
    echo "Home Space Utilization ($USER)"
    du -sh $HOME
    fi
    ;;
*)  echo "Invalid entry" >&2
    exit 1
    ;;
esac
```

The `case` command looks at the value of *word*, in our case the value of the `REPLY` variable, and then attempts to match it against one of the specified *patterns*. When a match is found, the *commands* associated with the specified pattern are executed. After a match is found, no further matches are attempted.

Patterns

The patterns used by `case` are the same as those used by pathname expansion. Patterns are terminated with a “)” character. Here are some valid patterns:

Pattern	Description
a)	Matches if <i>word</i> equals “a”
[[:alpha:]])	Matches if <i>word</i> is a single alphabetic character.
???)	Matches if <i>word</i> is exactly three characters long.
*.txt)	Matches if <i>word</i> ends with the characters “.txt”
*)	Matches any value of <i>word</i> . It is good practice to include this as the last pattern in a <code>case</code> command to catch any values of <i>word</i> that did not match a previous pattern, that is, to catch any possible invalid values.

Here is an example of patterns at work:

```
#!/bin/bash

read -p "enter word > "

case $REPLY in
    [[:alpha:]]      echo "is a single alphabetic character." ;;
    [ABC][0-9])     echo "is A, B, or C followed by a digit." ;;
    ???)           echo "is three characters long." ;;
    *.txt)         echo "is a word ending in '.txt'" ;;
    *)             echo "is something else." ;;
esac
```

It is also possible to combine multiple patterns using the vertical bar character as a separator. This creates an “or” conditional pattern. This is useful for, such things as, handling both upper and lower case characters. For example:

```
#!/bin/bash

# case-menu: a menu driven system information program

clear
echo "
Please Select:

A. Display System Information
B. Display Disk Space
C. Display Home Space Utilization
Q. Quit
"
read -p "Enter selection [A, B, C or Q] > "

case $REPLY in
    q|Q) echo "Program terminated."
        exit
        ;;
    a|A) echo "Hostname: $HOSTNAME"
        uptime
        ;;
    b|B) df -h
        ;;
    c|C) if [[ $(id -u) -eq 0 ]]; then
        echo "Home Space Utilization (All Users)"
        du -sh /home/*
        else
        echo "Home Space Utilization ($USER)"
        du -sh $HOME
        fi
esac
```

```
        ;;
*)      echo "Invalid entry" >&2
        exit 1
        ;;
esac
```

Here, we modify the `case-menu` program to use letters instead of digits for menu selection. Notice how the new patterns allow for entry of both upper and lower case letters.

Summing Up

The `case` command is a handy addition to our bag of programming tricks. As we will see in the next chapter, it's the perfect tool for handling certain types of problems.

Further Reading

The *Bash Reference Manual* section on Conditional Constructs describes the `case` command in detail:

<http://tiswww.case.edu/php/chet/bash/bashref.html#SEC21>

The *Advanced Bash-Scripting Guide* provides some further examples of `case` applications:

<http://tldp.org/LDP/abs/html/testbranch.html>

33 – Positional Parameters

One feature that has been missing from our programs is the ability to accept and process command line options and arguments. In this chapter, we will examine the shell features that allow our programs to get access to the contents of the command line.

Accessing The Command Line

The shell provides a set of variables called *positional parameters* that contain the individual words on the command line. The variables are named 0 through 9. They can be demonstrated this way:

```
#!/bin/bash

# posit-param: script to view command line parameters

echo "
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"
```

A very simple script that displays the values of the variables \$0-\$9. When executed with no command line arguments:

```
[me@linuxbox ~]$ posit-param

$0 = /home/me/bin/posit-param
```

```
$1 =  
$2 =  
$3 =  
$4 =  
$5 =  
$6 =  
$7 =  
$8 =  
$9 =
```

Even when no arguments are provided, `$0` will always contain the first item appearing on the command line which is the name of the program being executed. When arguments are provided, we see the results:

```
[me@linuxbox ~]$ posit-param a b c d  
  
$0 = /home/me/bin/posit-param  
$1 = a  
$2 = b  
$3 = c  
$4 = d  
$5 =  
$6 =  
$7 =  
$8 =  
$9 =
```

Note: You can actually access more than nine parameters using parameter expansion. To specify a number greater than nine, surround the number in braces. For example `${10}`, `${55}`, `${211}` and so on.

Determining The Number of Arguments

The shell also provides a variable, `$#` that yields the number of arguments on the command line:

```
#!/bin/bash  
  
# posit-param: script to view command line parameters  
  
echo "  
Number of arguments: $#
```

```
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"
```

The result:

```
[me@linuxbox ~]$ posit-param a b c d

Number of arguments: 4
$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =
$9 =
```

shift – Getting Access To Many Arguments

But what happens when we give the program a large number of arguments such as this:

```
[me@linuxbox ~]$ posit-param *

Number of arguments: 82
$0 = /home/me/bin/posit-param
$1 = addresses.ldif
$2 = bin
$3 = bookmarks.html
$4 = debian-500-i386-netinst.iso
$5 = debian-500-i386-netinst.jigdo
$6 = debian-500-i386-netinst.template
$7 = debian-cd_info.tar.gz
$8 = Desktop
```

```
$9 = dirlist-bin.txt
```

On this example system the wildcard `*` expands into eighty-two arguments, so how can we process that many? The shell provides a method, albeit it clumsy, to do this. The `shift` command causes all of the parameters to “move down one” each time it is executed. In fact, by using `shift` it is possible to get by with only one parameter (in addition to `$0`, which never changes):

```
#!/bin/bash
# posit-param2: script to display all arguments
count=1
while [[ $# -gt 0 ]]; do
    echo "Argument $count = $1"
    count=$((count + 1))
    shift
done
```

Each time `shift` is executed, the value of `$2` is moved to `$1`, the value of `$3` is moved to `$2` and so on. The value of `$#` is also reduced by one.

In the `posit-param2` program we create a loop that evaluates the number of arguments remaining and continues as long as there is at least one. We display the current argument, increment the variable `COUNT` with each iteration of the loop to provide a running count of the number of argument processed, and finally execute a `shift` to load `$1` with the next argument. Here is the program at work:

```
[me@linuxbox ~]$ posit-param2 a b c d
Argument 1 = a
Argument 2 = b
Argument 3 = c
Argument 4 = d
```

Simple Applications

Even without `shift`, it’s possible to write useful applications using positional parameters. By way of example, here is a simple file information program:

```
#!/bin/bash

# file_info: simple file information program

PROGNAME=$(basename $0)

if [[ -e $1 ]]; then
    echo -e "\nFile Type:"
    file $1
    echo -e "\nFile Status:"
    stat $1
else
    echo "$PROGNAME: usage: $PROGNAME file" >&2
    exit 1
fi
```

This program displays the file type (determined by the `file` command) and the file status (from the `stat` command) of a specified file. One interesting feature of this program is the `PROGNAME` variable. It is given the value resulting from the `basename $0` command. The `basename` command removes the leading portion of a pathname leaving on the base name of a file. In our example, `basename` removes the leading portion of the pathname contained in the `$0` parameter, the full pathname of our example program. This value is useful when constructing messages such as the usage message at the end of the program. By coding it this way, the script can be renamed and the message automatically adjusts to contain the name of the program.

Using Positional Parameters With Shell Functions

Just as positional parameters are used to pass arguments to shell scripts, they can also be used to pass arguments to shell functions. To demonstrate, we will convert the `file_info` script into a shell function:

```
file_info () {

    # file_info: function to display file information

    if [[ -e $1 ]]; then
        echo -e "\nFile Type:"
        file $1
        echo -e "\nFile Status:"
        stat $1
    else
        echo "$FUNCNAME: usage: $FUNCNAME file" >&2
        return 1
    fi
}
```

```
    fi  
}
```

Now, if a script that incorporates the `file_info` shell function calls the the function with a file name argument, the argument will be passed to the function.

With this capability, one can write many useful shell functions that can not only be used in scripts, but also within the `.bashrc` file.

Notice that the `PROGNAME` variable was changed to the shell variable `FUNCNAME`. The shell automatically updates this variable to keep track of the currently executed shell function. Note that `$0` always contains the full pathname of the first item on the command line (i.e. the name of the program) and does not contain the name of the shell function as one might expect.

Handling Positional Parameters *En Masse*

It is sometimes useful to manage all the positional parameters as a group. For example we might want to write a “wrapper” around another program. This means that we create a script or shell function that simplifies the execution of another program. The wrapper supplies a list of arcane command line options and then passes a list of arguments to the lower level program.

The shell provides two special parameters for this purpose. They both expand into the complete list of positional parameters but differ in rather subtle ways. They are:

Parameter	Description
<code>\$*</code>	Expands into the list of positional parameters starting with 1. When surrounded by double quotes, it expands into a double quoted string containing all of the positional parameters each separated by the first character of the IFS shell variable (by default a space character.)
<code>\$@</code>	Expands into the list of positional parameters starting with 1. When surrounded by double quotes, it expands each positional parameter into a separate word surrounded by double quotes.

Here is a script that shows them in action:

```
#!/bin/bash  
  
# posit-params3 : script to demonstrate $* and $@
```

```
print_params () {
    echo "\$1 = $1"
    echo "\$2 = $2"
    echo "\$3 = $3"
    echo "\$4 = $4"
}

pass_params () {
    echo -e "\n" '$* :'; print_params $*
    echo -e "\n" '"$*" :'; print_params "$*"
    echo -e "\n" '$@ :'; print_params @$@
    echo -e "\n" '"$@" :'; print_params "$@"
}

pass_params "word" "words with spaces"
```

In this rather convoluted program, we create two arguments, “word” and “words with spaces” and pass them to the `pass_params` function. That function, in turn, passes them on to the `print_params` function using each of the four methods available with the special parameters `$!` and `$@`. When executed, the script reveals the differences:

```
[me@linuxbox ~]$ posit-param3
```

```
$* :
$1 = word
$2 = words
$3 = with
$4 = spaces

"$*" :
$1 = word words with spaces
$2 =
$3 =
$4 =

$@ :
$1 = word
$2 = words
$3 = with
$4 = spaces

"$@" :
$1 = word
$2 = words with spaces
$3 =
$4 =
```

With our arguments, both `$!` and `$@` produce a four word result:

```
word words with spaces
```

`"$*" produces a one word result:`

```
"word words with spaces"
```

`"$@" produces a two word result:`

```
"word" "words with spaces"
```

which matches our actual intent. The lesson to take from this is, even though the shell provides four different ways of getting the list of positional parameters, `"$@"` is by far the most useful for most situations because it preserves the integrity of each positional parameter.

A More Complete Application

After a long hiatus, we are going to resume work on our `sys_info_page` program. Our next addition will add several command line options to the program as follows:

- **Output file.** We will add an option to specify a name for a file to contain the program's output. It will be specified as either `-f file` or `--file file`.
- **Interactive mode.** This option will prompt the user for an output file name and will determine if the specified file already exists. If it does, the user will be prompted before the existing file is overwritten. This option will be specified by either `-i` or `--interactive`.
- **Help.** Either `-h` or `--help` may be specified to cause the program to output an informative usage message.

Here is the code needed to implement the command line processing:

```
usage () {
    echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
    return
}

# process command line options

interactive=
filename=

while [[ -n $1 ]]; do
    case $1 in
        -f | --file)          shift
                             filename=$1
    esac
done
```



```

        -i | --interactive)    ;;
                               interactive=1
        -h | --help)          ;;
                               usage
                               exit
        *)                    ;;
                               usage >&2
                               exit 1
                               ;;
    esac
    shift
done

```

First we add a shell function called `usage` to display a message when the help option is invoked or an unknown option is attempted.

Next we begin the processing loop. This loop continues while the positional parameter `$1` is not empty. At the bottom of the loop we have a `shift` command to advance the positional parameters to ensure that the loop will eventually terminate.

Within the loop, we have a `case` statement that examines the current positional parameter to see if it matches any of the supported choices. If a supported parameter is found it is acted upon. If not, then the usage message is displayed and the script terminates with an error.

The `-f` parameter is handled in an interesting way. When detected, it causes an additional `shift` to occur which advances positional parameter `$1` to be the file name argument supplied to the `-f` option.

We next add the code to implement the interactive mode:

```

# interactive mode

if [[ -n $interactive ]]; then
    while true; do
        read -p "Enter name of output file: " filename
        if [[ -e $filename ]]; then
            read -p "'$filename' exists. Overwrite? [y/n/q] > "
                case $REPLY in
                    Y|y) break
                        ;;
                    Q|q) echo "Program terminated."
                            exit
                        ;;
                    *) continue
                esac
        fi
    done
fi

```

```
fi
done
fi
esac
;;
```

If the `interactive` variable is not empty then an endless loop is started containing the file name prompt and subsequent existing file handling code. If the desired output file already exists, the user is prompted to either overwrite, choose another file name, or quit the program. If the user chooses to overwrite an existing file, a `break` is executed to terminate the loop. Notice how the `case` statement only detects if the user chooses to overwrite or quit. Any other choice causes the loop to continue and prompt the user again.

In order to implement the output file name feature, we must first convert the existing page writing code into a shell function for reasons that will become clear in a moment:

```
write_html_page () {
    cat <<- _EOF_
    <HTML>
        <HEAD>
            <TITLE>$TITLE</TITLE>
        </HEAD>
        <BODY>
            <H1>$TITLE</H1>
            <P>$TIME_STAMP</P>
            $(report_uptime)
            $(report_disk_space)
            $(report_home_space)
        </BODY>
    </HTML>
    _EOF_
    return
}

# output html page

if [[ -n $filename ]]; then
    if touch $filename && [[ -f $filename ]]; then
        write_html_page > $filename
    else
        echo "$PROGNAME: Cannot write file '$filename'" >&2
        exit 1
    fi
else
    write_html_page
```

```
fi
```

The code that handles the logic of `-f` option appears at the bottom of the listing above. In it, we test for the existence of a file name and, if found, a test is performed to see if the file is indeed writable. To do this, a `touch` is performed followed by a test to determine if the resulting file is a regular file. These two tests take care of situations where an invalid pathname is input (`touch` will fail) and if the file already exists, that it's not a directory.

As we can see, the `write_html_page` function is called to perform the actual generation of the page. Its output is either directed to standard output (if the variable `filename` is empty) or redirected to the specified file.

Summing Up

With the addition of positional parameters, we can now write pretty functional scripts. and for simple, repetitive tasks, positional parameters make it possible to write really useful shell functions that can be placed in a user's `.bashrc` file.

Our `sys_info_page` program has grown in complexity and sophistication. Here is a complete listing with the most recent changes highlighted:

```
#!/bin/bash

# sys_info_page: program to output a system information page

PROGNAME=$(basename $0)
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

report_uptime () {
    cat <<- _EOF_
        <H2>System Uptime</H2>
        <PRE>$(uptime)</PRE>
    _EOF_
    return
}

report_disk_space () {
    cat <<- _EOF_
        <H2>Disk Space Utilization</H2>
        <PRE>$(df -h)</PRE>
    _EOF_
    return
}
```

```
}

report_home_space () {
    if [[ $(id -u) -eq 0 ]]; then
        cat <<- _EOF_
            <H2>Home Space Utilization (All Users)</H2>
            <PRE>$(du -sh /home/*)</PRE>
        _EOF_
    else
        cat <<- _EOF_
            <H2>Home Space Utilization ($USER)</H2>
            <PRE>$(du -sh $HOME)</PRE>
        _EOF_
    fi
    return
}

usage () {
    echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
    return
}

write_html_page () {
    cat <<- _EOF_
        <HTML>
            <HEAD>
                <TITLE>$TITLE</TITLE>
            </HEAD>
            <BODY>
                <H1>$TITLE</H1>
                <P>$TIME_STAMP</P>
                $(report_uptime)
                $(report_disk_space)
                $(report_home_space)
            </BODY>
        </HTML>
    _EOF_
    return
}

# process command line options

interactive=
filename=

while [[ -n $1 ]]; do
    case $1 in
        -f | --file)
            shift
            filename=$1
            ;;
    esac
done
```

```
        -i | --interactive)    interactive=1
                                ;;
        -h | --help)          usage
                                exit
                                ;;
        *)                    usage >&2
                                exit 1
                                ;;
    esac
    shift
done

# interactive mode

if [[ -n $interactive ]]; then
    while true; do
        read -p "Enter name of output file: " filename
        if [[ -e $filename ]]; then
            read -p "'$filename' exists. Overwrite? [y/n/q] > "
                case $REPLY in
                    Y|y) break
                        ;;
                    Q|q) echo "Program terminated."
                            exit
                            ;;
                    *) continue
                        ;;
                esac
        fi
    done
fi

# output html page

if [[ -n $filename ]]; then
    if touch $filename && [[ -f $filename ]]; then
        write_html_page > $filename
    else
        echo "$PROGNAME: Cannot write file '$filename'" >&2
        exit 1
    fi
else
    write_html_page
fi
```

We're not done yet. There are still more things we can do and improvements we can make.

Further Reading

The *Bash Hackers Wiki* has a good article on positional parameters:

<http://bash-hackers.org/wiki/doku.php/scripting/posparams>

The *Bash Reference Manual* has an article on the special parameters including `$*` and `$@`:

<http://www.gnu.org/software/bash/manual/bashref.html#Special-Parameters>

34 – Flow Control: Looping With for

In this, the final chapter on flow control, we will look at another of the shell’s looping constructs. The *for loop*, differs from the *while* and *until* loops in that it provides a means of processing sequences during a loop. This turns out to be very useful when programming and the *for* loop is a very popular construct in *bash* scripting.

A *for* loop is implemented, naturally enough, with the `for` command. In modern versions of *bash*, `for` is available in two forms.

for: Traditional Shell Form

The original `for` command’s syntax is:

```
for variable [in words]; do
    commands
done
```

Where *variable* is the name of a variable that will increment during the execution of the loop, *words* is an optional list of items that will be sequentially assigned to *variable*, and *commands* are the commands that are to be executed on each iteration of the loop.

The `for` command is useful on the command line. We can easily demonstrate how it works:

```
[me@linuxbox ~]$ for i in A B C D; do echo $i; done
A
B
C
D
```

In this example, `for` is given a list of four words. “A”, “B”, “C”, and “D.” With a list of four words, the loop is executed four times. Each time the loop is executed, a word is assigned to the variable `i`. Inside the loop, we have an `echo` command that displays the value of `i` to show the assignment. As with the `while` and `until` loops, the `done` keyword closes the loop.

The really powerful feature of `for` is the number of interesting ways we can create the list of words. For example through pathname expansion:

```
[me@linuxbox ~]$ for i in distros*.txt; do echo $i; done
distros-by-date.txt
distros-dates.txt
distros-key-names.txt
distros-key-vernums.txt
distros-names.txt
distros-newdates.txt
distros-orig.txt
distros.txt
distros-vernums.txt
distros-versions.txt
```

Or command substitution:

```
#!/bin/bash

# longest_word : find longest string in a file

while [[ -n $1 ]]; do
    if [[ -r $1 ]]; then
        max_word=
        max_len=
        for i in $(strings $1); do
            len=$(echo $i | wc -c)
            if (( len > max_len )); then
                max_len=$len
                max_word=$i
            fi
        done
        echo "$1: '$max_word' ($max_len characters)"
    fi
    shift
done
```

In this example, we look for the longest string found within a file. When passed one or more files names on the command line, this program uses the `strings` program to generate a list of readable text “words” in each file. The `for` loop processes each word in turn and determines if the current word is the longest found so far. When the loop concludes, the longest one is displayed.

If the optional `in words` portion of the command is omitted, `for` defaults to processing the positional parameters. We will modify our `longest_word` script to use

this method:

```
#!/bin/bash

# longest_word2 : find longest string in a file

for i; do
    if [[ -r $i ]]; then
        max_word=
        max_len=
        for j in $(strings $i); do
            len=$(echo $j | wc -c)
            if (( len > max_len )); then
                max_len=$len
                max_word=$j
            fi
        done
        echo "$i: '$max_word' ($max_len characters)"
    fi
done
```

As we can see, we have changed the outermost loop to use `for` in place of `while`. By omitting the list of words in the `for` command, the positional parameters are used instead. Inside the loop, previous instances of the variable `i` have been changed to the variable `j`. The use of `shift` has also been eliminated.

Why i?

You may have noticed that the variable `i` was chosen for the each of the `for` loop examples above. Why was this? No good reason actually, besides tradition. The variable used with `for` can be any valid variable, but `i` is the most common.

The basis of this tradition comes from the Fortran programming language. In Fortran, undeclared variables starting with the letters I, J, K, L, and M are automatically typed as integers while variables whose names begin with any other letter are typed as real (numbers with decimal fractions.) This behavior led programmers to use the variables I, J, and K for loop variables since it was less work to just use them when a temporary variable (as loop variables often are) was needed.

It also led to the following Fortran-based witticism:

“GOD is real, unless declared integer.”

for: C Language Form

Recent versions of `bash` have added a second form of `for` command syntax, one that resembles the form used in the C programming language. Many other languages support this form, as well:

```
for (( expression1; expression2; expression3 )); do
    commands
done
```

where *expression1*, *expression2*, and *expression3* are arithmetic expressions and *commands* are the commands to be performed during each iteration of the loop.

In terms of behavior, this form is equivalent to the following construct:

```
(( expression1 ))
while (( expression2 )); do
    commands
    (( expression3 ))
done
```

expression1 is used to initialize conditions for the loop, *expression2* is used to determine when the loop is finished and *expression3* is carried out at the end of each iteration of the loop.

Here is a typical application:

```
#!/bin/bash

# simple_counter : demo of C style for command

for (( i = 0; i < 5; i = i + 1 )); do
    echo $i
done
```

When executed, it produces the following output

```
[me@linuxbox ~]$ simple_counter
0
1
2
3
4
```

In this example, *expression1* initializes the variable `i` with the value of zero, *expression2* allows the loop to continue as long as the value of `i` remains less than five and

expression3 increments the value of *i* by one each time the loop repeats.

The C language form of `for` is useful anytime a numeric sequence is needed. We will see several applications for this in the next two chapters.

Summing Up

With our knowledge of the `for` command, we will now apply some improvements to the `report_home_space` function in our `sys_info_page` script. Currently, the function looks like this:

```
report_home_space () {
    if [[ $(id -u) -eq 0 ]]; then
        cat <<- _EOF_
            <H2>Home Space Utilization (All Users)</H2>
            <PRE>$(du -sh /home/*)</PRE>
            _EOF_
    else
        cat <<- _EOF_
            <H2>Home Space Utilization ($USER)</H2>
            <PRE>$(du -sh $HOME)</PRE>
            _EOF_
    fi
    return
}
```

Next, we will rewrite it to provide more detail for each user's home directory including the total number of files and subdirectories in each:

```
report_home_space () {
    local format="%8s%10s%10s\n"
    local i dir_list total_files total_dirs total_size user_name
    if [[ $(id -u) -eq 0 ]]; then
        dir_list=/home/*
        user_name="All Users"
    else
        dir_list=$HOME
        user_name=$USER
    fi
    echo "<H2>Home Space Utilization ($user_name)</H2>"
    for i in $dir_list; do
```

```
total_files=$(find $i -type f | wc -l)
total_dirs=$(find $i -type d | wc -l)
total_size=$(du -sh $i | cut -f 1)

echo "<H3>$i</H3>"
echo "<PRE>"
printf "$format" "Dirs" "Files" "Size"
printf "$format" "----" "-----" "-----"
printf "$format" $total_dirs $total_files $total_size
echo "</PRE>"
done
return
}
```

This rewrite applies much of what we have learned so far. We still test for the superuser, but instead of performing the complete set of actions as part of the `if`, we set some variables used later in a `for` loop. We have added several local variables to the function and made use of `printf` to format some of the output.

Further Reading

The *Advanced Bash-Scripting Guide* has a chapter on loops with a variety of examples using `for`:

<http://tldp.org/LDP/abs/html/loops1.html>

The *Bash Reference Manual* describes the looping compound commands including `for`:

<http://www.gnu.org/software/bash/manual/bashref.html#Looping-Constructs>

35 – Strings And Numbers

Computer programs are all about working with data. In past chapters, we have focused on processing data at the file level. However there are many programming problems that need to be solved at a lower level using smaller units of data such as strings and numbers.

In this chapter we will look at several shell features that are used to manipulate strings and numbers. The shell provides a variety of parameter expansions that perform string operations. In addition to arithmetic expansion (which we touched upon in Chapter 8), there is a common command line program called `bc` which performs higher level math.

Parameter Expansion

While parameter expansion came up in Chapter 8, we did not cover it in detail. The reason being most parameter expansions are used in scripts rather than the command line. We have already worked with some forms of parameter expansion. Our use of shell variables is one form. The shell provides many more that manipulate strings and shell variables.

Basic Parameters

The simplest form of parameter expansion is reflected in the ordinary use of variables. For example:

```
$a
```

when expanded, becomes whatever the variable `a` contains. Simple parameters may also be optionally surrounded by braces:

```
${a}
```

This has no effect on the expansion, but is required if the variable is adjacent to other text which may confuse the shell:

```
[me@linuxbox ~]$ a="foo"
[me@linuxbox ~]$ echo "$a_file"
```

If we perform this sequence, the result will be nothing because the shell will try to expand a variable named `a_file` rather than `a`. This problem can be solved by adding braces:

```
[me@linuxbox ~]$ echo "${a}_file"
foo_file
```

We have also seen that the positional parameters greater than 9 can be accessed by surrounding the number in braces. For example, to access the eleventh positional parameter, we can do this:

```
${11}
```

Expansions To Manage Empty Variables

Several parameter expansions deal with nonexistent and empty variables. These expansions are handy for handling missing positional parameters and assigning default values.

`${parameter:-word}`

If *parameter* is unset (i.e. does not exist) or is empty, this expansion results in the value of *word*. If *parameter* is not empty, the expansion results in the value of *parameter*.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:-"default value"}
default value
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:-"default value"}
bar
```

`${parameter:=word}`

If *parameter* is unset or empty, this expansion results in the value of *word*. In addition, the value of *word* is assigned to *parameter*. If *parameter* is not empty, the expansion results in the value of *parameter*.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:="default value"}
default value
[me@linuxbox ~]$ echo $foo
default value
[me@linuxbox ~]$ foo=bar
```

```
[me@linuxbox ~]$ echo ${foo:="default value"}
bar
[me@linuxbox ~]$ echo $foo
bar
```

Note: Positional and other special parameters cannot be assigned this way.

`${parameter:?word}`

If *parameter* is unset or empty, this expansion causes the script to exit with an error and the contents of *word* are sent to standard error. If *parameter* is not empty, the expansion results in the value of *parameter*.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:? "parameter is empty"}
bash: foo: parameter is empty
[me@linuxbox ~]$ echo $?
1
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:? "parameter is empty"}
bar
[me@linuxbox ~]$ echo $?
0
```

`${parameter:+word}`

If *parameter* is unset or empty, the expansion results in nothing. If *parameter* is not empty, the value of *word* is substituted for *parameter*, however, the value of *parameter* is not changed.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:+ "alternate value"}

[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:+ "alternate value"}
alternate value
```

Expansions That Return Variable Names

The shell has the ability to return the names of variables. This is used in some rather exotic situations.

```
`${!prefix*}`  
`${!prefix@}`
```

This expansion returns the names of existing variables with names beginning with `prefix`. Here we list all the variables in the environment with names that begin with `BASH`:

```
[me@linuxbox ~]$ echo `${!BASH*}`  
BASH BASH_ARGC BASH_ARGV BASH_COMMAND BASH_COMPLETION  
BASH_COMPLETION_DIR BASH_LINENO BASH_SOURCE BASH_SUBSHELL  
BASH_VERSINFO BASH_VERSION
```

String Operations

There is a large set of expansions that can be used to operate on strings. Many of these expansions are particularly well-suited for operations on path names.

```
`${#parameter}`
```

Expands into the length of the string contained by *parameter*. Normally *parameter* is a string, however if *parameter* is either `@` or `*`, then the expansion results in the number of positional parameters.

```
[me@linuxbox ~]$ foo="This string is long."  
[me@linuxbox ~]$ echo "'$foo' is `${#foo}` characters long."  
'This string is long.' is 20 characters long.
```

```
`${parameter:offset}`  
`${parameter:offset:length}`
```

This expansion is used to extract a portion of the string contained in *parameter*. The extraction begins *offset* characters from the beginning of the string and continues until the end of the string unless the *length* is specified.

```
[me@linuxbox ~]$ foo="This string is long."  
[me@linuxbox ~]$ echo `${foo:5}`  
string is long.  
[me@linuxbox ~]$ echo `${foo:5:6}`
```



```
string
```

If the value of *offset* is negative, it is taken to mean from the end of the string rather than the beginning. Note that negative values must be preceded by a space to prevent confusion with the `${parameter:-word}` expansion. *length*, if present, must not be less than zero.

If *parameter* is `@`, the result of the expansion is *length* positional parameters, starting at *offset*.

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo ${foo: -5}
long.
[me@linuxbox ~]$ echo ${foo: -5:2}
lo
```

```
${parameter#pattern}
${parameter##pattern}
```

This expansion removes a leading portion of the string contained in *parameter* defined by *pattern*. *pattern* is a wildcard pattern like those used in pathname expansion. The difference in the two forms is the `#` form removes the shortest match, while the `##` form removes the longest match.

```
[me@linuxbox ~]$ foo=file.txt.zip
[me@linuxbox ~]$ echo ${foo#*.}
txt.zip
[me@linuxbox ~]$ echo ${foo##*.}
zip
```

```
${parameter%pattern}
${parameter%%pattern}
```

These expansions are the same as the `#` and `##` expansions above except they remove text from the end of the string contained in *parameter* rather than the beginning.

```
[me@linuxbox ~]$ foo=file.txt.zip
[me@linuxbox ~]$ echo ${foo%. *}
file.txt
[me@linuxbox ~]$ echo ${foo%%.*}
file
```

```
${parameter/pattern/string}
${parameter//pattern/string}
${parameter/#pattern/string}
${parameter/%pattern/string}
```

This expansion performs a search and replace upon the contents of *parameter*. If text is found matching wildcard *pattern*, it is replaced with the contents of *string*. In the normal form, only the first occurrence of *pattern* is replaced. In the *//* form, all occurrences are replaced. The */#* form requires that the match occur at the beginning of the string and the */%* form requires the match to occur at the end of the string. *string* may be omitted which causes the text matched by *pattern* to be deleted.

```
[me@linuxbox ~]$ foo=JPG.JPG
[me@linuxbox ~]$ echo ${foo/JPG/jpg}
jpg.JPG
[me@linuxbox ~]$ echo ${foo//JPG/jpg}
jpg.jpg
[me@linuxbox ~]$ echo ${foo/#JPG/jpg}
jpg.JPG
[me@linuxbox ~]$ echo ${foo/%JPG/jpg}
JPG.jpg
```

Parameter expansion is a good thing to know. The string manipulation expansions can be used as substitutes for other common commands such as `sed`, and `cut`. Expansions improve the efficiency of scripts by eliminating the use of external programs. As an example, we will modify the `longest_word` program discussed in the previous chapter to use the parameter expansion `${#j}` in place of the command substitution `$(echo $j | wc -c)` and its resulting subshell like so:

```
#!/bin/bash

# longest_word3 : find longest string in a file

for i; do
    if [[ -r $i ]]; then
        max_word=
        max_len=
        for j in $(strings $i); do
            len=${#j}
            if (( len > max_len )); then
                max_len=$len
                max_word=$j
            fi
        done
    fi
done
```

```

        echo "$i: '$max_word' ($max_len characters)"
    fi
    shift
done

```

Next, we will compare the efficiency of the two versions by using the `time` command:

```

[me@linuxbox ~]$ time longest_word2 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38
characters)

real    0m3.618s
user    0m1.544s
sys     0m1.768s
[me@linuxbox ~]$ time longest_word3 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38
characters)

real    0m0.060s
user    0m0.056s
sys     0m0.008s

```

The original version of the script takes 3.618 seconds to scan the text file while the new version, using parameter expansion, takes only 0.06 seconds, a very significant improvement.

Arithmetic Evaluation And Expansion

We looked at arithmetic expansion in Chapter 8. It is used to perform various arithmetic operations on integers. Its basic form is:

```
$(expression)
```

where *expression* is a valid arithmetic expression.

This is related to the compound command `(())` used for arithmetic evaluation (truth tests) we encountered in Chapter 28.

In previous chapters, we saw some of the common types of expressions and operators. Here we will look at a more complete list.

Number Bases

Back in Chapter 10, we got a look at octal (base 8) and hexadecimal (base 16) numbers. In arithmetic expressions, the shell supports integer constants in any base.

Notation	Description
<i>number</i>	By default, numbers without any notation are treated as decimal (base 10) integers.
<i>0number</i>	In arithmetic expressions, numbers with a leading zero are considered octal.
<i>0xnumber</i>	Hexadecimal notation
<i>base#number</i>	<i>number</i> is in <i>base</i>

Some examples:

```
[me@linuxbox ~]$ echo $((0xff))
255
[me@linuxbox ~]$ echo $((2#11111111))
255
```

In the examples above we print the value of the hexadecimal number `ff` (the largest two digit number) and the largest eight digit binary (base 2) number.

Unary Operators

There are two unary operators, the `+` and `-` which are used to indicate if a number is positive or negative respectively. For example, `-5`.

Simple Arithmetic

The ordinary arithmetic operators are listed in the table below:

Operator	Description
<code>+</code>	Addition
<code>-</code>	Subtraction
<code>*</code>	Multiplication
<code>/</code>	Integer division
<code>**</code>	Exponentiation
<code>%</code>	Modulo (remainder)

Most of these are self-explanatory, but integer division and modulo deserve further

discussion.

Since the shell's arithmetic only operates on integers, the results of division are always whole numbers:

```
[me@linuxbox ~]$ echo $(( 5 / 2 ))  
2
```

This makes the determination of a remainder in a division operation more important:

```
[me@linuxbox ~]$ echo $(( 5 % 2 ))  
1
```

By using the division and modulo operators, we can determine that the five divided by two results in two with a remainder of one.

Calculating the remainder is useful in loops. It allows an operation to be performed at specified intervals during the loop's execution. In the example below, we display a line of numbers highlighting each multiple of five:

```
#!/bin/bash  
  
# modulo1 : demonstrate the modulo operator  
  
for ((i = 0; i <= 20; i = i + 1)); do  
    remainder=$((i % 5))  
    if (( remainder != 0 )); then  
        printf "%d " $i  
    else  
        printf "<%d> " $i  
    fi  
done  
printf "\n"
```

When executed, the results look like this:

```
[me@linuxbox ~]$ modulo1  
<0> 1 2 3 4 <5> 6 7 8 9 <10> 11 12 13 14 <15> 16 17 18 19 <20>
```

Assignment

Although its uses may not be immediately apparent, arithmetic expressions may perform assignment. We have performed assignment many times, though in a different context. Each time we give a variable a value, we are performing assignment. We can also do it within arithmetic expressions:

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo $foo

[me@linuxbox ~]$ if (( foo = 5 ));then echo "It is true."; fi
It is true.
[me@linuxbox ~]$ echo $foo
5
```

In the example above, we first assign an empty value to the variable `foo` and verify that it is indeed empty. Next, we perform an `if` with the compound command `((foo = 5))`. Doing this does two interesting things: 1. it assigns the value of five to the variable `foo` and 2. it evaluates to true because the assignment was successful.

Note: It is important to remember the exact meaning of the `=` in the expression above. A single `=` performs assignment. `foo = 5` says “make `foo` equal to five,” while `==` evaluates equivalence. `foo == 5` says “does `foo` equal five?” This can be very confusing because the `test` command accepts a single `=` for string equivalence. Another reason to use the more modern `[[]]` and `(())` compound commands in place of `test`.

In addition to the `=`, the shell also provides notations that perform some very interesting assignments:

Notation	Description
<i>parameter = value</i>	Simple assignment. Assigns <i>value</i> to <i>parameter</i> .
<i>parameter += value</i>	Addition. Equivalent to <i>parameter = parameter + value</i>
<i>parameter -= value</i>	Subtraction. Equivalent to <i>parameter = parameter - value</i>
<i>parameter *= value</i>	Multiplication. Equivalent to <i>parameter = parameter * value</i>
<i>parameter /= value</i>	Integer division. Equivalent to <i>parameter = parameter / value</i>

<i>parameter</i> %= <i>value</i>	Modulo. Equivalent to <i>parameter</i> = <i>parameter</i> % <i>value</i>
<i>parameter</i> ++	Variable post-increment. Equivalent to <i>parameter</i> = <i>parameter</i> + 1 (however, see discussion below)
<i>parameter</i> --	Variable post-decrement. Equivalent to <i>parameter</i> = <i>parameter</i> - 1
++ <i>parameter</i>	Variable pre-increment. Equivalent to <i>parameter</i> = <i>parameter</i> + 1
-- <i>parameter</i>	Variable pre-decrement. Equivalent to <i>parameter</i> = <i>parameter</i> - 1

These assignment operators provide a convenient shorthand for many common arithmetic tasks. Of special interest are the increment (++) and decrement (--) operators which increase or decrease the value of their parameters by one. This style of notation is taken from the C programming language and has been incorporated by several programming languages including bash.

The operators may appear either at the front of a parameter or at the end. While they both either increment or decrement the parameter by one, the two placements have a subtle difference. If placed at the front of the parameter, the parameter is incremented (or decremented) before the parameter is returned. If placed after, the operation is performed *after* the parameter is returned. This is rather strange, but it is the intended behavior. Here is a demonstration:

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((foo++))
1
[me@linuxbox ~]$ echo $foo
2
```

If we assign the value of one to the variable `foo` and then increment it with the ++ operator placed after the parameter name, `foo` is returned with the value of one. However if we look at the value of the variable a second time, we see the incremented value. If we place the ++ operator in front of the parameter, we get the more expected behavior:

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((++foo))
```

```
2
[me@linuxbox ~]$ echo $foo
2
```

For most shell applications, prefixing the operator will be the most useful.

The ++ and -- operators are often used in conjunction with loops. We will make some improvements to our modulo script to tighten it up a bit:

```
#!/bin/bash

# modulo2 : demonstrate the modulo operator

for ((i = 0; i <= 20; ++i )); do
    if ((i % 5) != 0); then
        printf "%d " $i
    else
        printf "<%d> " $i
    fi
done
printf "\n"
```

Bit Operations

One class of operators manipulate numbers in an unusual way. They operate at the bit level. These are used for certain kinds of low level tasks, often involving setting or reading bit-flags.

Operator	Description
~	Bitwise negation. Negate all the bit in a number.
<<	Left bitwise shift. Shift all the bits in a number to the left.
>>	Right bitwise shift. Shift all the bits in a number to the right.
&	Bitwise AND. Perform an AND operation on all the bits in two numbers.
	Bitwise OR. Perform an OR operation on all the bits in two numbers.
^	Bitwise XOR. Perform an exclusive OR operation on all the bits in two numbers.

Note that there are also corresponding assignment operators (for example <<=) for all

but bitwise negation.

Here we will demonstrate producing a list of powers of two using the left bitwise shift operator:

```
[me@linuxbox ~]$ for ((i=0;i<8;++i)); do echo $((1<<i));done
1
2
4
8
16
32
64
128
```

Logic

As we discovered in Chapter 28, the `(())` compound command supports a variety of comparison operators. There are a few more that can be used to evaluate logic. Here is the complete list:

Operator	Description
<code><=</code>	Less than or equal to
<code>>=</code>	Greater than or equal to
<code><</code>	Less than
<code>></code>	Greater than
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code>&&</code>	Logical AND
<code> </code>	Logical OR
<code>expr1?expr2:expr3</code>	Comparison (ternary) operator. If expression <i>expr1</i> evaluates to be non-zero (arithmetic true) then <i>expr2</i> , else <i>expr3</i> .

When used for logical operations, expressions follow the rules of arithmetic logic, that is, expressions that evaluate as zero are considered false while non-zero expressions are considered true. The `(())` compound command maps the results into the shell's normal exit codes:

```
[me@linuxbox ~]$ if ((1)); then echo "true"; else echo "false"; fi
true
[me@linuxbox ~]$ if ((0)); then echo "true"; else echo "false"; fi
false
```

The strangest of the logical operators is the *ternary operator*. This operator (which is modeled after the one in the C programming language) performs a standalone logical test. It can be used as a kind of if/then/else statement. It acts on three arithmetic expressions (strings won't work) and if the first expression is true (or non-zero) the second expression is performed. Otherwise the third expression is performed. We can try this on the command line:

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
1
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
0
```

Here we see a ternary operator in action. This example implements a toggle. Each time the operator is performed, the value of the variable `a` switches from zero to one or vice versus.

Please note that performing assignment within the expressions is not straightforward. When attempted, `bash` will declare an error:

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?a+=1:a-=1))
bash: ((: a<1?a+=1:a-=1: attempted assignment to non-variable (error token is "-=1")
```

This problem can be mitigated by surrounding the assignment expression with parentheses:

```
[me@linuxbox ~]$ ((a<1?(a+=1):(a-=1)))
```

Next, we see a more complete example of using arithmetic operators in a script that produces a simple table of numbers:

```
#!/bin/bash

# arith-loop: script to demonstrate arithmetic operators

finished=0
a=0
printf "a\ta**2\t a**3\n"
printf "=\t====\t====\n"

until ((finished)); do
    b=$((a**2))
    c=$((a**3))
    printf "%d\t%d\t%d\n" $a $b $c
    ((a<10?++a:(finished=1)))
done
```

In this script, we implement an until loop based on the value of the `finished` variable. Initially the variable is set to zero (arithmetic false) and we continue the loop until it becomes non-zero. Within the loop, we calculate the square and cube of the counter variable `a`. At the end of the loop, the value of the counter variable is evaluated and if it is less than ten (the maximum number of iterations,) it is incremented by one, else the variable `finished` is given the value of one, making `finished` arithmetically true thereby terminating the loop. Running the script gives this result:

```
[me@linuxbox ~]$ arith-loop
a      a**2  a**3
=      ====  ====
0      0      0
1      1      1
2      4      8
3      9      27
4      16     64
5      25     125
6      36     216
7      49     343
8      64     512
9      81     729
10     100    1000
```

bc – An Arbitrary Precision Calculator Language

We have seen how the shell can handle all manner of integer arithmetic, but what if we need to perform higher math or even just use floating point numbers? The answer is we can't. At least not directly with the shell. To do this, we need to use an external program.

There are several approaches we can take. Embedding perl or awk programs is one possible solution, but unfortunately outside the scope of this book.

Another approach is to use a specialized calculator program. One such program is found on most Linux systems is called `bc`.

The `bc` program reads a file written in its own C-like language and executes it. A `bc` script may be a separate file or it may be read from standard input. The `bc` language supports quite a few features including variables, loops, and programmer-defined functions. We won't cover `bc` entirely here, just enough to get a taste. `bc` is well-documented by its man page.

Let's start with a simple example. We'll write a `bc` script to add two plus two:

```
/* A very simple bc script */  
2 + 2
```

The first line of the script is a comment. `bc` uses the same syntax for comments as the C programming language. Comments, which may span multiple lines, begin with `/*` and end with `*/`.

Using `bc`

If we save the `bc` script above as `foo.bc`, we can run it this way:

```
[me@linuxbox ~]$ bc foo.bc  
bc 1.06.94  
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software  
Foundation, Inc.  
This is free software with ABSOLUTELY NO WARRANTY.  
For details type `warranty'.  
4
```

If we look carefully, we can see the result at the very bottom, after the copyright message. This message can be suppressed with the `-q` (quiet) option.

`bc` can also be used interactively:

```
[me@linuxbox ~]$ bc -q  
2 + 2
```

```
4  
quit
```

When used interactively, we simply type the calculations we wish to perform and the results are immediately displayed. The `bc` command `quit` ends the interactive session.

It is also possible to pass a script to `bc` via standard input:

```
[me@linuxbox ~]$ bc < foo.bc  
4
```

The ability to take standard input means that we can use here documents, here strings, and pipes to pass scripts. This is a here string example:

```
[me@linuxbox ~]$ bc <<< "2+2"  
4
```

An Example Script

As an real-world example, we will construct a script that performs a common calculation, monthly loan payments. In the script below, we use a here document to pass a script to `bc`:

```
#!/bin/bash  
  
# loan-calc : script to calculate monthly loan payments  
  
PROGRAMNAME=$(basename $0)  
  
usage () {  
    cat <<- EOF  
    Usage:  
  
    $PROGRAMNAME principal interest months  
  
    where:  
  
    principal is the amount of the loan  
    interest is the APR as a number (7% = 0.07)  
    months is the length of the loan's term.  
EOF  
}
```

```
        EOF
    }
    if (($# != 3)); then
        usage
        exit 1
    fi

    principal=$1
    interest=$2
    months=$3

    bc <<- EOF
        scale = 10
        i = $interest / 12
        p = $principal
        n = $months
        a = p * ((i * ((1 + i) ^ n)) / (((1 + i) ^ n) - 1))
        print a, "\n"
    EOF
```

When executed, the results look like this:

```
[me@linuxbox ~]$ loan-calc 135000 0.0775 180
1270.7222490000
```

This example calculates the monthly payment for a \$135,000 loan at 7.75% APR for 180 months (15 years.) Notice the precision of the answer. This is determined by the value given to the special `scale` variable in the `bc` script. A full description of the `bc` scripting language is provided by the `bc` man page. While its mathematical notation is slightly different from that of the shell (`bc` more closely resembles `C`), most of it will be quite familiar based on what we have learned so far.

Summing Up

In this chapter, we have learned about many of the little things that can be used to get the “real work” done in scripts. As our experience with scripting grows, the ability to effectively manipulate strings and numbers will prove extremely valuable. Our `loan-calc` script demonstrates that even simple scripts can be created to do some really useful things.

Extra Credit

While the basic functionality of the `loan-calc` script is in place, the script is far from complete. For extra credit, try improving the `loan-calc` script with the following features:

- Full verification of the command line arguments
- A command line option to implement an “interactive” mode that will prompt the user to input the principal, interest rate, and term of the loan.
- A better format for the output.

Further Reading

The *Bash Hackers Wiki* has a good discussion of parameter expansion:

<http://bash-hackers.org/wiki/doku.php/syntax/pe>

The *Bash Reference Manual* covers this too:

<http://www.gnu.org/software/bash/manual/bashref.html#Shell-Parameter-Expansion>

The *Wikipedia* has a good article describing bit operations:

http://en.wikipedia.org/wiki/Bit_operation

and an article on ternary operations:

http://en.wikipedia.org/wiki/Ternary_operation

as well as a description of the formula for calculating loan payments use in our `loan-calc` script:

http://en.wikipedia.org/wiki/Amortization_calculator

36 – Arrays

In the last chapter, we looked at how the shell can manipulate string and numbers. So far, the data types we have looked at are known in computer science circles as *scalar variables*, that is, variables that contain a single value.

In this chapter, we will look at another kind of data structure called an *array* which holds multiple values. Arrays are a feature of virtually every programming language. The shell supports them too, though in a rather limited fashion. Even so, they can be very useful for solving programming problems.

What Are Arrays?

Arrays are variables that hold more than one value at a time. Arrays are organized like a table. Let's consider a spreadsheet as an example. A spreadsheet acts like a *two-dimensional array*. It has both rows and columns and individual cells in the spreadsheet can be located according to its row and column address. An array behaves the same way. An array has cells, which are called *elements*, and each element contains data. Elements are accessed by an address called an *index* or *subscript* which is used to locate an individual element in the array.

Most programming languages support *multi-dimensional arrays*. A spreadsheet is an example of a multi-dimensional array with two dimensions, width and height. Many languages support arrays with an arbitrary number of dimensions, though two and three dimensional arrays are probably the most commonly used.

Arrays in `bash` are limited to a single dimension. We can think of them being like a spreadsheet with but a single column. Even with this limitation, there are many applications for them. Array support first appeared in `bash` version 2. The original Unix shell program, `sh` did not support arrays at all.

Creating An Array

Arrays variables are named just like other `bash` variables and are created automatically when they are accessed just like other `bash` variables. Here is an example:


```
[me@linuxbox ~]$ a[1]=foo
[me@linuxbox ~]$ echo ${a[1]}
foo
```

Here we see an example of both assignment and access of an array element. With the first command, element 1 of array `a` is assigned the value “foo”. The second command displays the stored value of element 1. The use of braces in the second command is required to prevent the shell from attempting pathname expansion on the name of the array element.

An array can also be created with the `declare` command:

```
[me@linuxbox ~]$ declare -a a
```

Using the `-a` option, this example of `declare` creates the array `a`.

Assigning Values To An Array

Values may be assigned in one of two ways. Single values may be assigned using the following syntax:

```
name[subscript]=value
```

where *name* is the name of the array, *subscript* is an integer (or arithmetic expression) greater than or equal to zero. Note that the first element of an array is subscript zero, not one. *value* is a string or integer assigned to the array element.

Multiple values may be assigned using the following syntax:

```
name=(value1 value2 ...)
```

where *name* is the name of the array and *value...* are values assigned sequentially to elements of the array starting with element zero. For example, if we wanted to assign abbreviated days-of-the-week to the array `days`, we could do this:

```
[me@linuxbox ~]$ days=(Sun Mon Tue Wed Thu Fri Sat)
```

It is also possible to assign values to specific element by specifying a subscript for each value:

```
[me@linuxbox ~]$ days=([0]=Sun [1]=Mon [2]=Tue [3]=Wed [4]=Thu
```

```
[5]=Fri [6]=Sat)
```

Accessing Array Elements

So what are arrays good for? Just as many data management tasks can be performed with a spreadsheet program, many programming tasks can be performed with arrays

Let's consider a simple data gathering and presentation example. We will construct a script that examines the .modification times of the files in a specified directory. From this data, our script will output a table showing at what time-of-day the files were last modified. Such a script could be used to determine when a system is most active. This script, called `hours`, produces this result:

```
[me@linuxbox ~]$ hours .
Hour  Files Hour  Files
-----
00    0    12    11
01    1    13     7
02    0    14     1
03    0    15     7
04    1    16     6
05    1    17     5
06    6    18     4
07    3    19     4
08    1    20     1
09   14    21     0
10    2    22     0
11    5    23     0

Total files = 80
```

We execute the `hours` program specifying the current directory as the target. It produces a table showing for each hour of the day (0-23) how many files were last modified. The code to produce this is as follows:

```
#!/bin/bash

# hours : script to count files by modification time

usage () {
    echo "usage: $(basename $0) directory" >&2
}

}
```

```
# Check that argument is a directory
if [[ ! -d $1 ]]; then
    usage
    exit 1
fi

# Initialize array
for i in {0..23}; do hours[i]=0; done

# Collect data
for i in $(stat -c %y "$1"/* | cut -c 12-13); do
    j=${i/#0}
    ((++hours[j]))
    ((++count))
done

# Display data
echo -e "Hour\tFiles\tHour\tFiles"
echo -e "----\t-----\t----\t-----"
for i in {0..11}; do
    j=$((i + 12))
    printf "%02d\t%d\t%02d\t%d\n" $i ${hours[i]} $j ${hours[j]}
done
printf "\nTotal files = %d\n" $count
```

The script consists of one function (`usage`) and a main body with four sections. In the first section, we check that there is a command line argument and that it is a directory. If it is not, we display the usage message and exit.

The second section initializes the array `hours`. It does this by assigning each element a value of zero. There is no special requirement to prepare arrays prior to use, but our script needs to ensure that no element is empty. Note the interesting way the loop is constructed. By employing brace expansion (`{0..23}`) we are able to easily generate a sequence of words for the `for` command.

The next section gathers the data by running the `stat` program on each file in the directory. We use `cut` to extract the two digit hour from the result. Inside the loop, we remove any leading zero from the hour field. A leading zero will trick the shell into thinking the value is an octal number. Next, we increment the value of the array element corresponding with the hour-of-the-day. Finally we increment a counter (`COUNT`) to track the total number of files in the directory.

The last section of the script displays the contents of the array. We first output a couple of header lines and then enter a loop that produces two columns of output. Lastly we output the final tally of files.

Array Operations

When writing scripts it is good to learn techniques to manage and operate on arrays. Such things as deleting arrays, determining their size, sorting, etc. have many applications in scripting.

Outputting The Entire Contents Of An Array

The subscripts `*` and `@` can be used to get every element in an array. Like their use with positional parameters, the `@` notation is the more useful of the two. Here is a demonstration:

```
[me@linuxbox ~]$ animals=("a dog" "a cat" "a fish")
[me@linuxbox ~]$ for i in ${animals[*]}; do echo $i; done
a
dog
a
cat
a
fish
[me@linuxbox ~]$ for i in ${animals[@]}; do echo $i; done
a
dog
a
cat
a
fish
[me@linuxbox ~]$ for i in "${animals[*]}"; do echo $i; done
a dog a cat a fish
[me@linuxbox ~]$ for i in "${animals[@]}"; do echo $i; done
a dog
a cat
a fish
```

We create the array `animals` and assign it three two-word strings. We then execute four loops to see the affect of word-splitting on the array contents. The behavior of notations `${animals[*]}` and `${animals[@]}` are identical until they are quoted. The `*` notation results in a single word containing the the array’s contents, while the `@` notation results in three words which matches the arrays “real” contents.

Determining The Number Of Array Elements

Using parameter expansion, we can determine the number of elements in an array in much the same way as finding the length of a string. Here is an example:

```
[me@linuxbox ~]$ a[100]=foo
[me@linuxbox ~]$ echo ${#a[@]} # number of array elements
1
[me@linuxbox ~]$ echo ${#a[100]} # length of element 100
3
```

We create array `a` and assign the string “foo” to element 100. Next, we use parameter expansion to examine the length of the array using the `@` notation. Finally we look at the length of element 100 which contains the string “foo”. It is interesting to note that while we assigned our string to element 100, bash only reports that there is one element in the array. This differs from the behavior of some other languages where the unused elements of the array (elements 0-99) would be initialized with empty values and counted.

Finding The Subscripts Used By An Array

As bash allows arrays to contain “gaps” in the assignment of subscripts, it is sometimes useful to determine which elements actually exist. This can be done with a parameter expansion using the following forms:

```
${!array[*]}
${!array[@]}
```

where `array` is the name of an array variable. Like the other expansions that use `*` and `@`, the `@` form enclosed in quotes is the most useful as it expands into separate words:

```
[me@linuxbox ~]$ foo=( [2]=a [4]=b [6]=c )
[me@linuxbox ~]$ echo "${foo[@]}"
a b c
[me@linuxbox ~]$ echo "${!foo[@]}"
2 4 6
```

Adding Elements To The End Of An Array

Even knowing the number of elements in array is no help if we need to append values to the end of an array since the values return by the `*` and `@` notations do not tell us the maximum array index in use. Fortunately the shell provides us with a solution. By using the `+=` assignment operator, we can automatically append values to the end of an array. Here we assign three values to the array `foo` and then append three more.

```
[me@linuxbox ~]$ foo=(a b c)
```

```
[me@linuxbox ~]$ echo ${foo[@]}
a b c
[me@linuxbox ~]$ foo+=(d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
```

Sorting An Array

Just as with spreadsheets, it is often necessary to sort the values in a column of data. The shell has no direct way of doing this, but it's not hard to do with a little coding:

```
#!/bin/bash

# array-sort : Sort an array

a=(f e d c b a)

echo "Original array: ${a[@]}"
a_sorted=$(for i in "${a[@]"; do echo $i; done | sort)
echo "Sorted array:  ${a_sorted[@]}"
```

When executed, the script produces this:

```
[me@linuxbox ~]$ array-sort
Original array: f e d c b a
Sorted array:  a b c d e f
```

The script operates by copying the contents of the original array (**a**) into a second array (**a_sorted**) with a tricky piece of command substitution. This basic technique can be used to perform many kinds of operations on the array by changing the design of the pipeline.

Note that the script will only work on arrays containing simple (one word) strings. Strings containing embedded spaces will be broken into separate array elements by word-splitting unless the IFS variable is modified to prevent it. By making the modifications below, this code can now handle any array contents, not just the simple example we see here:

```
#!/bin/bash
```

```
# array-sort : Sort an array

a=(f e d c b a)

echo "Original array: ${a[@]}"
IFS_OLD="$IFS" IFS=$'\n'
a_sorted=$(for i in "${a[@]}"; do echo $i; done | sort)
IFS="$IFS_OLD"
echo "Sorted array:  ${a_sorted[@]}"
```

When we change the IFS variable so that it contains a single newline, embedded spaces are no longer word delimiters and do not trigger word-splitting.

Deleting An Array

To delete an array, use the `unset` command:

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ unset foo
[me@linuxbox ~]$ echo ${foo[@]}

[me@linuxbox ~]$
```

`unset` may also be used to delete single array elements:

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ unset 'foo[2]'
[me@linuxbox ~]$ echo ${foo[@]}
a b d e f
```

In this example, we delete the third element of the array, subscript 2. Remember, arrays start with subscript zero not one! Notice also that the array element must be quoted to prevent the shell from performing pathname expansion.

Interestingly, the assignment of an empty value to an array does not empty its contents:

```
[me@linuxbox ~]$ foo=(a b c d e f)
```

```
[me@linuxbox ~]$ foo=  
[me@linuxbox ~]$ echo ${foo[@]}  
b c d e f
```

Any reference to an array variable without a subscript refers to element zero of the array:

```
[me@linuxbox ~]$ foo=(a b c d e f)  
[me@linuxbox ~]$ echo ${foo[@]}  
a b c d e f  
[me@linuxbox ~]$ foo=A  
[me@linuxbox ~]$ echo ${foo[@]}  
A b c d e f
```

Summing Up

If we search the `bash` man page for the word “array,” we find many instances of where `bash` makes use of array variables. Most of these are rather obscure, but they may provide occasional utility in some special circumstances. In fact, the entire topic of arrays is rather under-utilized in shell programming owing largely to the fact that the traditional Unix shell programs (such as `sh`) lacked any support for arrays. This lack of popularity is unfortunate because arrays are widely used in other programming languages and provide a powerful tool for solving many kinds of programming problems.

Arrays and loops have a natural affinity and are often used together. The

```
for ((expr; expr; expr))
```

form of loop is particularly well-suited to calculating array subscripts.

Further Reading

[http://en.wikipedia.org/wiki/Scalar \(computing\)](http://en.wikipedia.org/wiki/Scalar_(computing))

37 – Exotica

In this, the final chapter of our journey, we will look at some odds and ends. While we have certainly covered a lot of ground in the previous chapters, there are many `bash` features that we have not covered. Most are fairly obscure, and useful mostly to those integrating `bash` into a Linux distribution. There are a few however that, while not in common use, are helpful for certain programming problems. We will cover them here.

Group Commands And Subshells

`bash` allows commands to be grouped together. This can be done in one of two ways, either with a *group command* or with a *subshell*. Here are examples of the syntax of each:

Group Command:

```
{ command1; command2; [command3; ...] }
```

Subshell:

```
(command1; command2; [command3;...])
```

The two forms differ in that a group command surrounds its commands with braces and a subshell uses parentheses. It is important to note that, due to the way `bash` implements group commands, the braces must be separated from the commands by a space and the last command must be terminated with either a semicolon or a newline prior to the closing brace.

So what are group commands and subshells good for? While they have an important difference (which we will get to in a moment) they are both used to manage redirection. Let's consider a script segment that performs redirections on multiple commands:

```
ls -l > output.txt
echo "Listing of foo.txt" >> output.txt
cat foo.txt >> output.txt
```

This is pretty straightforward. Three commands with their output redirected to a file

name `output.txt`. Using a group command, we could code this as follows:

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } > output.txt
```

Using a subshell is similar:

```
(ls -l; echo "Listing of foo.txt"; cat foo.txt) > output.txt
```

Using this technique, we have saved ourselves some typing, but where this really shines is with pipelines. When constructing a pipeline of commands, it is often useful to combine the results of several commands into a single stream. Group commands and subshells make this easy:

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } | lpr
```

Here we have combined the output of our three commands and piped them into the input of `lpr` to produce a printed report.

Process Substitution

While they look similar and can both be used to combine streams for redirection, there is an important difference between group commands and subshells. Whereas a group command executes all of its commands in the current shell, a subshell, as the name suggests, executes its commands in a child copy of the current shell. This means that the environment is copied and a given to a new instance of the shell. When the subshell exits, the copy of the environment is lost so any changes made to the subshell's environment (including variable assignment) is lost as well. So in most cases, unless a script requires a subshell, group commands are preferred over invoking subshells. It is both faster and requires less memory.

We saw an example of the subshell environment problem in Chapter 29 when we discovered that including a `read` command in a pipeline does not work as we might intuitively expect. To recap, if we construct a pipeline like this:

```
echo "foo" | read  
echo $REPLY
```

The contents of the `REPLY` variable are empty because the `read` command was executed in a subshell and its copy of `REPLY` was destroyed when the subshell terminated.

Because commands in pipelines are always executed in subshells, any command that assigns variables will encounter this issue. Fortunately, the shell provides an exotic form of expansion called *process substitution* that can be used to work around this problem.

Process substitution is expressed in two ways:

For processes that produce standard output:

```
<(list)
```

and processes that intake standard input:

```
>(list)
```

where *list* is a list of commands.

To solve our problem with `read`, we can employ process substitution like this:

```
read < <(echo "foo")
echo $REPLY
```

Process substitution allows us to treat the output of a subshell as an ordinary file for purposes of redirection. In fact, since it is a form of expansion, we can examine its real value:

```
[me@linuxbox ~]$ echo <(echo "foo")
/dev/fd/63
```

By using `echo` to view the result of the expansion, we see that the output of the subshell is being provided by a file named `/dev/fd/63`.

Process substitution is often used with loops containing `read`. Here is an example of a `read` loop that processes the contents of a directory listing created by a subshell:

```
#!/bin/bash

# pro-sub : demo of process substitution

while read attr links owner group size date time filename; do
    cat <<- EOF
        Filename:  $filename
```

```
        Size:      $size
        Owner:     $owner
        Group:     $group
        Modified:  $date $time
        Links:     $links
        Attributes: $attr

        EOF
done < <(ls -l | tail -n +2)
```

The loop executes `read` for each line of a directory listing. The listing itself is produced on the final line of the script. This line redirects the output of the process substitution into the standard input of the loop. The `tail` command is included in the process substitution pipeline to eliminate the first line of the listing which is not needed.

When executed, the script produces output like this:

```
[me@linuxbox ~]$ pro_sub | head -n 20
Filename:  addresses.ldif
Size:      14540
Owner:     me
Group:     me
Modified:  2009-04-02 11:12
Links:     1
Attributes: -rw-r--r--

Filename:  bin
Size:      4096
Owner:     me
Group:     me
Modified:  2009-07-10 07:31
Links:     2
Attributes: drwxr-xr-x

Filename:  bookmarks.html
Size:      394213
Owner:     me
Group:     me
```

Traps

In Chapter 11 we saw how programs can respond to signals. We can add this capability to our scripts too. While the scripts we have written so far have not needed this capability (owing to the fact that they have very short execution times and do not create temporary files), larger and more complicated scripts may benefit from having a signal

handling routine.

When we design a large, complicated script, it is important to consider what happens if the user logs off or shuts down the computer while the script is running. When such an event occurs, a signal will be sent to all affected processes and, in turn, the programs representing those processes can perform actions to ensure a proper and orderly termination of the program. Let's say for example that we wrote a script that created a temporary file during its execution. In the course of good design, we would have the script delete the file when the script finished its work. It would be good too, to have the script delete the file if a signal were received indicating that the program was going to be terminated prematurely.

`bash` provides a mechanism for this purpose known as a *trap*. Traps are implemented with the appropriately named builtin command, `trap`. `trap` uses the following syntax:

```
trap argument signal [signal...]
```

where *argument* is a string which will be read and treated as a command and *signal* is the specification of a signal that will trigger the execution of interpreted command.

Here is a simple example:

```
#!/bin/bash

# trap-demo : simple signal handling demo

trap "echo 'I am ignoring you.'" SIGINT SIGTERM

for i in {1..5}; do
    echo "Iteration $i of 5"
    sleep 5
done
```

This script defines a trap that will execute an `echo` command each time either the `SIGINT` or `SIGTERM` signals are received while the script is running. Execution of the program looks like this when the user attempts to stop the script by typing `Ctrl-C`:

```
[me@linuxbox ~]$ trap-demo
Iteration 1 of 5
Iteration 2 of 5
I am ignoring you.
Iteration 3 of 5
I am ignoring you.
Iteration 4 of 5
Iteration 5 of 5
```

As we can see, each time the user attempts to interrupt the program, the message is printed instead.

Constructing a string to form a useful sequence of commands can be awkward, so it is common practice to specify a shell function as the command. In this example a separate shell function is specified for each signal to be handled:

```
#!/bin/bash

# trap-demo2 : simple signal handling demo

exit_on_signal_SIGINT () {
    echo "Script interrupted." 2>&1
    exit 0
}

exit_on_signal_SIGTERM () {
    echo "Script terminated." 2>&1
    exit 0
}

trap exit_on_signal_SIGINT SIGINT
trap exit_on_signal_SIGTERM SIGTERM

for i in {1..5}; do
    echo "Iteration $i of 5"
    sleep 5
done
```

This script features two `trap` commands, one for each signal. Each trap, in turn, specifies a shell function to be executed when the particular signal is received. Note the inclusion of an `exit` command in each of the signal handling functions. Without an `exit`, the script would continue after completing the function.

When the user presses `Ctrl-C` during the execution of this script, the results looks like this:

```
[me@linuxbox ~]$ trap-demo2
Iteration 1 of 5
Iteration 2 of 5
Script interrupted.
```

Temporary Files

One reason signal handlers are included in scripts is to remove temporary files that the script may create to hold intermediate results during execution. There is something of an art to naming temporary files. Traditionally, programs on Unix-like systems create their temporary files in the `/tmp` directory, a shared directory intended for such files. However, since the directory is shared, this poses certain security concerns, particularly for programs running with superuser privileges. Aside from the obvious step of setting proper permissions for files exposed to all users of the system, it is important for temporary files to have non-predictable file names. This avoids an exploit known as a *temp race attack*. One way to create a non-predictable (but still descriptive) name is to do something like this:

```
tempfile=/tmp/$(basename $0)$$.$RANDOM
```

This will create a file name consisting of the program's name followed by its process ID (PID) followed by a random integer. Note however that the `$RANDOM` shell variable only returns a value in the range of 1-32767 which is not a very large range in computer terms, so a single instance of the variable is not sufficient to overcome a determined attacker.

A better way is to use the `mktemp` program (not to be confused with the `mktemp` standard library function) to both name and create the temporary file. The `mktemp` program accepts a template as an argument that is used build the file name. The template should include a series of "X" characters which are replaced by a corresponding number of random letters and numbers. The longer the series of "X" characters, the longer the series of random characters. Here is an example:

```
tempfile=$(mktemp /tmp/foobar.$$XXXXXXXXXX)
```

This creates a temporary file and assigns its name to the variable `tempfile`. The "X" characters in the template are replaced with random letters and numbers so that the final file name (which, in this example, also includes the expanded value of the special parameter `$$` to obtain the PID) might be something like:

```
/tmp/foobar.6593.U0ZuvM6654
```

While the `mktemp` man page states that `mktemp` makes a temporary file name, it also creates the file as well.

For scripts that are executed by regular users, it may be wise to avoid the use of the `/tmp` directory all together and create a directory for temporary files within the user's home directory with a line of code such as this

```
[[ -d $HOME/tmp ]] || mkdir $HOME/tmp
```

Asynchronous Execution

It is sometimes desirable to perform more than one task at the same time. We have seen how all modern operating systems are at least multitasking if not multi-user as well. Scripts can be constructed to behave in a multitasking fashion.

Usually this involves launching a script that in turn launches one or more child scripts that perform some additional task while the parent script continues to run. However, when a series of scripts runs this way, there can be problems keeping the parent and child coordinated, that is, what if the parent or child is dependent on the other and one script must wait for the other to finish its task before finishing its own?

`bash` has a builtin command to help manage *asynchronous execution* such as this. The `wait` command causes a parent script to pause until a specified process (i.e. the child script) finishes.

`wait`

We will demonstrate the `wait` command first. To do this, we will need two scripts, a parent script:

```
#!/bin/bash
# async-parent : Asynchronous execution demo (parent)
echo "Parent: starting..."
echo "Parent: launching child script..."
async-child &
pid=$!
echo "Parent: child (PID= $pid) launched."
echo "Parent: continuing..."
sleep 2
echo "Parent: pausing to wait for child to finish..."
wait $pid
echo "Parent: child is finished. Continuing..."
echo "Parent: parent is done. Exiting."
```

and a child script:


```
#!/bin/bash

# async-child : Asynchronous execution demo (child)

echo "Child: child is running..."
sleep 5
echo "Child: child is done.  Exiting."
```

In this example, we see that the child script is very simple. The real action is being performed by the parent. In the parent script, the child script is launched and put into the background. The process ID of the child script is recorded by assigning the `pid` variable with the value of the `#!` shell parameter which will always contain the process ID of the last job put into the background.

The parent script continues and then executes a `wait` command with the PID of the child process. This causes the parent script to pause until the child script exits, at which point, the parent script concludes.

When executed, the parent and child scripts produce the following output:

```
[me@linuxbox ~]$ async-parent
Parent: starting...
Parent: launching child script...
Parent: child (PID= 6741) launched.
Parent: continuing...
Child: child is running...
Parent: pausing to wait for child to finish...
Child: child is done.  Exiting.
Parent: child is finished.  Continuing...
Parent: parent is done.  Exiting.
```

Named Pipes

In most Unix-like systems it is possible to create a special type of file called a *named pipe*. Named pipes are used to create a connection between two processes and can be used just as any other type of file. They are not that popular but they're good to know about.

There is a common programming architecture called *client-server* which can make use of a communication method such as named pipes, as well as other kinds of *interprocess communication* such as network connections.

The most widely used type of client-server system is, of course, is a web browser communicating with a web server. The web browser acts as the client and makes

requests to the server and the server responds to the browser with web pages.

Named pipes behave like files but actually form first-in first-out (FIFO) buffers. As with ordinary (unnamed pipes) data goes in one end and emerges out the other. With named pipes, it is possible to set up something like this:

```
process1 > named_pipe
```

and

```
process2 < named_pipe
```

and it will behave as if:

```
process1 | process2
```

Setting Up A Named Pipe

First, we must create a named pipe. This is done using the `mkfifo` command:

```
[me@linuxbox ~]$ mkfifo pipe1
[me@linuxbox ~]$ ls -l pipe1
prw-r--r-- 1 me  me    0 2009-07-17 06:41 pipe1
```

Here we use `mkfifo` to create a named pipe called `pipe1`. Using `ls`, we examine the file and see that the first letter in the attributes field is “p” indicating that it is a named pipe.

Using Named Pipes

To demonstrate how the named pipe works we will need two terminal windows (or alternately, two virtual consoles.) In the first terminal we enter a simple command and redirect its output to the named pipe:

```
[me@linuxbox ~]$ ls -l > pipe1
```

After pressing the Enter key, the command will appear to hang. This is because there is nothing yet receiving data from the other end of the pipe. When this occurs, it is said that the pipe is *blocked*. This condition will clear once we attach a process to the other end and it begins to read input from the pipe. Using the second terminal window, we enter this command:

```
[me@linuxbox ~]$ cat < pipe1
```

and the directory listing produced from the first terminal window appears in the second terminal as the output from the `cat` command. The `ls` command in the first terminal successfully completes once it is no longer blocked.

Summing Up

Well, that completes our journey. The only thing left to do now is practice, practice, practice. Even though we covered a lot of ground in our trek, we barely scratched the surface as far as the command line goes. There are still thousands of command line programs left to be discovered and enjoyed. Start digging around in `/usr/bin` and you'll see!

Further Reading

The “Compound Commands” section of the bash man page contains a full description of group command and subshell notations.

The EXPANSIONS section of the bash man page contains a subsection of process substitution.

The Advanced Bash-Scripting Guide also has a discussion of process substitution:

<http://tldp.org/LDP/abs/html/process-sub.html>

Linux Journal has two good articles on named pipes. The first from September 1997:

<http://www.linuxjournal.com/article/2156>

and the second from March 2009:

<http://www.linuxjournal.com/content/using-named-pipes-fifos-bash>

Index

Stallman, Richard.....7, 9Torvalds, Linus.....7