Installing and Administering Linux

Second Edition

Linda McKinnon

Al McKinnon

Gearhead Press™

Wiley Computer Publishing

John Wiley & Sons, Inc.

Publisher: Robert Ipsen
Editor: Ben Ryan
Consulting Editor: Donis Marshall
Managing Editor: Angela Smith
Text Design & Composition: D&G Limited, LLC

This book is printed on acid-free paper. ♾

Published by John Wiley & Sons, Inc., New York

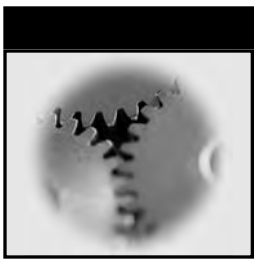
Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

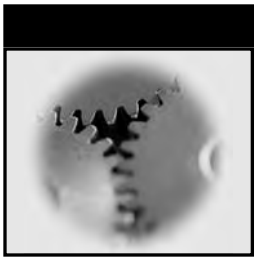
ISBN: 0-471-20884-1

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1



Contents



Introduction

Welcome to the world of Linux! And welcome especially to the second edition of *Installing and Administering Linux*. You are now in the trenches with us here at Gearhead Press and John Wiley & Sons. But fear not, you are in excellent company. Well, we think so, anyway.

We do not know how many Linux books you already own. We do not have any idea how many you have already picked up and put down here in the store or library. But here is a book that you can really use. You can use it in two ways. First, you can sit down at your personal computer with your chosen version of Linux and install it. Then, you can go step-by-step through a complete introductory course in basic Linux administration. This book has everything you need to go from a beginner to an intermediate-level user or beginning-level administrator. It has introductory material, organized "lectures," lots of examples, laboratory exercises you can do on your own system, and even quizzes you can take to test your comprehension. The information is all here. Or, you might already be an advanced beginner or intermediate administrator and need an orderly and progressive reference book. This book fills that need, too. Plus, it has the kind of real-world tips you can use on your home system or on the job.

Installing and Administering Linux, 2nd Edition might seem to focus on the Red Hat distribution (because that is the version we show you how to install in Chapter 2), but it teaches you concepts that are applicable to *all* Linux flavors and even to all UNIX flavors: FreeBSD, AIX, HPUX, you name it. So it is really "version independent." If you are working on a friend's FreeBSD system or on the company's IBM AIX system and you need to know what basic command to use for whatever purpose, you can still use this book. If you do not find exactly what you want in that case, we expect you will get a good lead.

Overview of the Book and Technology

We wrote this book for several reasons:

- The book contains material that is comparable to what you would find in a good introductory Linux course. Take a look at the price on the cover. We guarantee that the price is absolutely tiny compared to what you would pay at any technical institute, college, or university for a comparable course.
- The book also makes a great companion for an introductory course. In fact, the first edition of this book was actually adopted by more than one technical school as their "Intro Linux" course.
- This book will help you on the road to Linux certification. Unfortunately, it does not contain EVERYTHING you will need to know, but it provides a pretty good foundation for the next levels. Yes, be assured that it contains concepts, lab exercises, and quiz questions that are comparable to those you might eventually find on a Linux certification test. We are not interested in teaching you stuff just to teach you stuff. We want to help you move ahead.
- Most importantly, this book reflects what our professional colleagues and students have requested for years: a clear and easily read text that introduces and explains what they need to know NOW to get rolling with Linux.

One of the first things you will notice about "Installing and Administering Linux" is that it focuses on the "command line," similar to the classic UNIX or even to DOS, if you are familiar with the world before Microsoft Windows. In other words, we will not emphasize the use of Linux's X Window System, the *graphical user interface* (GUI) to Linux. Do not get the

wrong impression, though. Linux has some excellent window manager applications, from the basic "twm" to the more complete and very quickly developing "KDE" and "GNOME" window environments. No, the reason we focus on the command line is because once you get a feel for it, it is faster and more universally applicable from an administration standpoint. Even in the GUIs, you will probably be doing something at the command line eventually. "Be not afraid," for command line concepts and the commands themselves are easily learned with a little practice.

As we mentioned earlier, the material that you will find in this book is "version independent." In this case, it is really version and flavor independent because its concepts and commands apply to most UNIXes and to most Linuxes, from the earliest versions to the latest. This book is a basic-to-intermediate book; these are basic-to-intermediate concepts, and the basic concepts have not changed very much.

We have tried to make *Installing and Administering Linux, 2nd Edition* as up-to-date as possible. That is why we take the time every year or so to revise it. You will see, however, that the concepts found in it have not changed, and the commands and utilities have changed only for the better.

How This Book Is Organized

If you are familiar at all with the first edition of *Installing and Administering Linux, 2nd Edition,* then you will quickly notice how the organization of the book has changed significantly. We have "re-chunked" the material and introduced several new concepts and many new examples. Why? The original book followed the flow of a typical introductory course: doing a wide sweep of concepts on a basic level, then returning to certain concepts and exploring them in more detail. That was fine. But as a reference tool, the book was not as effective as it could have been. So, for this second edition, we regrouped the material and then filled in some blanks in the concepts and in the flow of discussion. We find this second edition to be a little more complete and comprehensive (for an introductory book, that is).

A lot of introductory courses begin by telling you a little about the technology you are dealing with—its origins and where it fits into the technological world. We do the same. In Chapter 1, we explain that Linux was not always there. We show you the evolution of a project called "Multics" and how Multics yielded UNIX. Then, we follow UNIX until Linux is born. We mention some Linux/UNIX concepts that you might or might not know, such as "free software" for example. We also hint at the future of Linux.

Chapter 2 shows you how to install Red Hat Linux on your *Integrated Drive Electronics* (IDE) personal computer. This installation is the most common type of Linux installation. In the first edition of *Installing and Administering Linux,* we put the installation chapter at the end. We were interested primarily in teaching Linux concepts first, because the various Linux flavors have slightly different installation procedures. Our readers and reviewers, however, suggested that we move the installation chapter forward for a better "start at the very beginning" approach. We are happy to oblige. Now, if you need to know how to install Linux on a system other than IDE, then we recommend our companion book called *Upgrading and Customizing Linux*, which covers installation in more depth on different systems and via different methods.

In Chapter 3, we show you how to "become somebody" in Linux. You will learn how to log in and log out. You will create users and groups and find out how to assign and change passwords. Then, just before we teach you a few common commands, we teach you *about* commands—what they all have in common with respect to syntax and how to find out more about them. Then, we show you some very basic getting around and communicating commands.

Chapters 4 through 6 delve into file and directory manipulation. You will learn about the file types supported by Linux; how to create, remove, and navigate the directories; how to examine and manipulate files; how to print files; and how to alter permissions on files for a modicum of file security.

In Chapter 7, we will introduce the concept of a "shell." You have likely seen shells in action before, but no one ever called them that. We will show you how the Linux shells interact with you and with Linux to make life easier and more efficient for you and for the system you are using.

The shell concepts will set the stage for the basic (Chapter 8) and advanced (Chapter 9) Linux utilities that will enable you to search, find, and manipulate Linux commands, files, and the data within files. Once you have mastered these utilities, a lot of the mysteries behind files and data will disappear—and your ability to perform administration functions will progress quickly.

Chapter 10 presents both a break and a catch-up. In this chapter, we introduce the almost universal (to Linux/UNIX users, anyway) text editor called "vi." If you aspire to higher-level Linux expertise, then knowledge of vi or other simple yet powerful Linux text editors will be mandatory. In fact, with your new knowledge of vi, you will be able to return to earlier exercises, if you wish, and do them so much faster. Needless to say, moving to the next chapters will be easier, too.

In Chapter 11, you will be introduced to the various environments in which your Linux system operates. We tend to think that our systems just operate the way they do, and that is that. This chapter will introduce you to variables, the environments created and colored by those variables, how your environments are built during system bootup and login, and how you can alter all those components to make your system work your way.

For those who are curious about what really goes on behind the scenes in a computer system, Chapter 12 will prove fascinating and potentially very powerful. We will introduce you to the system processes without which your system simply would not operate. You will discover how Linux/UNIX enables you to actually control how those processes work (something that other major operating systems do not enable you to do). In fact, you will even recognize some of the concepts and symptoms at work (or not at work) when other operating systems hang on you.

We introduce you to some basic system automation in Chapter 13 when we discuss shell programming. In that chapter, we even provide you with actual real-world scripts that you can use to make your system more responsive and more powerful while at the same time alerting you to, if not actually preventing, system overloads or other problems.

In Chapter 15, we provide a very basic introduction to Linux's GUIs. For some, this introduction might seem too basic and trivial. But there are also some tips and tricks that you might find handy, whether you are already familiar with Linux GUIs or not.

One of the most remarkable aspects of Linux is how it has grown up courtesy of the Internet and other media. So, we finish the book with Chapter 16, an up-to-date (if there really can be such a thing in a library or store-shelf book) compendium of Internet and other sources of information. If you need to know where to go to get applications, a newer kernel, X Window system information, or whatever, this chapter will lead you to it.

When we began to rewrite *Installing and Administering Linux*, we thought that it might be more fun to introduce a cast of characters about whom all this Linux activity revolved—imaginary people who might learn and illustrate the concepts and who might help keep the material lighter. We had read "Don Quixote" several years ago and were inspired by it from several standpoints, among which are 1) the cast of characters, 2) no matter what happened, Don Quixote had that indomitable and intrepid spirit (a trait we can all use in this fast-paced and often confusing world), 3) some of the predicaments and adventures, not to mention the actions and reactions of Don and his cohorts, were outrageous (not unlike some we get into ourselves), and 4)

they all combine to illustrate the beauty and strength of humanity and philosophy.

Therefore, we adopted some names and basic traits and then constructed, in our minds and on our pages, Rueful Figures, Inc., a company located in Spain that was dedicated to growing citrus fruit (to pay the bills) and to doing noble and chivalrous deeds. Where we can, we use RFI staff to show the reader how to perform the actions behind the concepts. We hope you enjoy them; we had fun creating them.

Who Should Read This Book?

We wrote this book for the busy system administrator or network engineer who aims to be up and running in short order. We want it to be fast-paced and version-independent—a reference that features insights we have gathered from our experiences in the trenches as system administrators, professional trainers, and consultants.

This book is written from a Linux system professional's perspective and highlights key similarities and differences between Linux and other UNIX systems to minimize the learning curve. We have provided time-tested exercises and corresponding quizzes with answers at the end of nearly every chapter plus helpful notes, warnings, and workarounds throughout the book. This information is all intended to accelerate the direct development of basic Linux skills as well as Linux system administration skills.

If you work your way through this book, you will learn to:

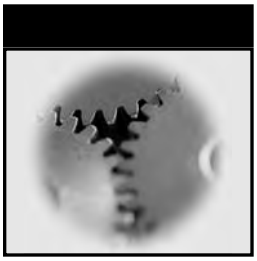
- Install Linux on an IDE system
- Log into and out of a Linux system as a root and ordinary user
- Create user accounts, group accounts, and passwords
- Execute common Linux commands
- Manage Linux files, directories, and processes
- Use common Linux utilities
- Use the vi text editor
- Customize the Linux work environment
- Use and customize a Linux X Window system environment

This book is also ideally suited to the Linux newcomer. That person could be a newcomer to computer systems or someone who has operated with that "other GUI operating system" but who is curious about a powerful and reliable operating system whose popularity is growing daily.

Vast experience, however, is not necessary to understand and enjoy it. Advanced system administration concepts will likely be mentioned but might not be covered in any great depth because they are beyond the scope of this introductory course. In the meantime, you will still be introduced to exercises and questions you can expect to encounter if you pursue Linux certification.

Summary

We hope you will enjoy this introduction to the Linux adventure. The book is a good and progressive course unto itself as well as an organized reference work. Once you have worked your way through it, you will have enough background to take on any intermediate Linux course or text and to take on any lower-level system administration or customer support.



Acknowledgments

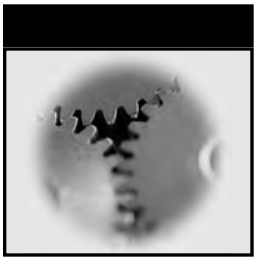
I have many to thank. First and foremost, thanks to Linda for the inspiration to write this book. Thanks to the many, many Linux developers all over the world. What a lesson in international cooperation we have all learned from you. Thanks also to all our colleagues and to all the students we have met over the past few years who've expressed the need for a progressive introduction to Linux and also for a book they could refer to when they need it. We also thank the reviewers and readers of the first edition of this book, who made the thoughtful comments and suggestions that led to this "new and improved" version. And thanks to the editors at Gearhead and Wiley, whose questions and comments kept us on our toes and made us account for all the ideas, claims, and instructions found herein. Thanks to Miguel de Cervantes Saavedra (1547-1616), whose "Don Quixote of La Mancha" was the inspiration for the characters you will meet in this book. If you have not read "Don Quixote" and (especially) if your spirit needs a lift, we wholeheartedly recommend it. Finally, as sentimental as it sounds, we still remember our beloved old dog Keeler, who spent her last few months watching us "pound out and mouse click" our way through the first edition. Her ghost lingers over this edition, too.

This book is dedicated to those who need or want a stable and robust operating system that is constantly and transparently undergoing improvement and to those Internet-linked free spirits who strive daily to make it even better.

—Al McKinnon

Thanks to all those who came to me and said "Please write that down for us." Without you, I would never have embarked on this literary tour. A special thanks goes to my peers who shared their views and to my students who shared their ideas and needs. During the past seven years, I have taught both basic and advanced Linux/UNIX curricula to more than 1,200 individuals. You know who you are; you guided my choice of topics. This book is dedicated to you and to those who we have yet to convert from that other operating system.

—Linda McKinnon



About the Authors

Linda McKinnon has more than 10 years of experience as a successful trainer and network engineer specializing in assisting major companies throughout the USA and Canada in network architecture, systems administration and RFP procurement. In addition to being sought-after for her Linux expertise, Ms. McKinnon teaches large-scale server technology including the IBM RS/6000 SP and pSeries systems. On the software side, she teaches WebSphere Application Server Administration as well as WebSphere Commerce Suite implementation."

Al McKinnon is a professional trainer and engineer who assists clients throughout North America with network auditing, optimization and troubleshooting, management and migration. Mr. McKinnon is also a technical author and editor for IBM and other clients, and he has extensive knowledge of networking technologies implemented on Linux, Windows 9x, Windows NT/2000, IBM AIX, Novell Netware and IBM OS/2 Warp Server.

Al and Linda are both headquartered in Calgary, Canada, where the prairie meets the foothills of the Canadian Rockies.

Linux Origins

This chapter presents a brief history of UNIX and Linux followed by an introduction to the major features of Linux. Although the current incarnation of Linux and its respective shells, programs, and other processes have been written from scratch, they have also evolved from previous versions of Linux and other UNIX flavors (HP-UX by Hewlett Packard, Solaris by Sun Microsystems, and others). Knowledge of the history of these operating systems can help you predict the usefulness of a command, program, or function or more easily troubleshoot unanticipated results.

A Brief History of UNIX

UNIX development was initiated in 1969 by several researchers working at Bell Laboratories in New Jersey. Bell Laboratories had previously (1964–1968) worked with General Electric and the *Massachusetts Institute of Technology* (MIT) on the development of a multi-user, time-sharing operating system called Multics (Multiplexed Information and Computing System). Thus, the project involved a partnership between private industry

and a respected academic institution. As the project progressed, however, Multics was proving more difficult and more expensive than the partners had anticipated. Deadlines were slipping, and the partners found they had differing philosophies and goals. So, in early 1969, Bell Labs withdrew from the Multics project.

Bell Labs researchers who had worked on Multics (Ken Thompson, Dennis Ritchie, Douglas McIlroy, Joseph Ossanna, and others) still wanted to develop an operating system for their own and Bell Labs' programming, job control, and resource usage needs. So, in addition to the comparatively routine activities they returned to, they also informally searched for an in-house alternative to the Multics project. Bell's executive managers, still somewhat scorched by the Multics project and burdened by internal politicking, however, were not immediately receptive to subsequent proposals.

During this same year, Thompson (later helped by Ritchie) wrote a simple space travel program called, naturally enough, Space Travel—first for Multics and then for the inflexible and inadequate GECOS operating system on a GE 635 computer. Space Travel on the GE 635 was unsatisfactory. That is, the program was jerky, hard to control, and expensive from a CPU-per-dollar standpoint. Thompson soon found another computer, an already obsolete DEC PDP-7, with acceptable display capabilities. But from a programming standpoint, Space Travel was still a challenge—it was written with macros for the assembler on the GE 635 computer and then sent through a postprocessor that produced paper tapes that were in turn fed into the PDP-7.

Thompson and Ritchie soon developed a new kind of primitive kernel for the PDP-7 plus a text editor, an assembler, a command interpreter (which we refer to as a shell), some rudimentary utilities, and a structured file system (including directories, special files that described system devices, inodes to describe file attributes, and so on). Many of these features were influenced by their previous work on the Multics project. Therefore, the PDP-7 achieved standalone capability. Almost all development thus far had been in DEC assembler language, although they had begun to develop a high-level language called B that is based on the existing BCPL language. B had been used to write a PDP-7 compiler.

In 1970, the developers requested and received a new computer, a DEC PDP-11, which had a new type of 16-bit processor and thus required alterations to the kernel code (this code was their Version 1). B was used to develop a PDP-7 to PDP-11 cross-compiler. Their colleague, Brian Kernighan, suggested that the new operating system be called Unix (UNIX would not be all capital letters for years to come, but we will talk more

about that later) as a sort of pun on Multics and as a way to contrast the simplicity of this new *operating system* (OS) with the complexity of Multics.

UNIX Is Born

In a supportive and collegial environment, the researchers continued the development of their portable (as defined in the preceding section), simple as possible, high-level language operating system that they called the Unix Time-Sharing System. The B language continued to be developed, too, and its name was later changed to C. C enabled them to create not only applications but also system programs that could be ported to new architectures as long as compilers existed for those architectures. By 1973, they were even able to rewrite the Unix kernel in C for what they called Version 4 (V4). They had now risen above the hardware-specific assembler language environment in which Unix had been born in 1969. The Unix operating system was different: The developers kept the kernel as close to an essential input/output multiplexer as possible but built a collection of add-on programming tools that carried out login and logout, command interpretation, file naming, console activity, and more via a vast array of functions. No longer did an application built with a hardware-specific language have to do it all. Applications could now do solely what they were intended to do (for example, text processing) and could rely on a high-level language operating system to carry out *input/output* (I/O), file manipulation, printing, handling several users at once, networking, inter-process communication, and so forth.

In 1973, AT&T/Bell recognized that hardware—especially telephone system hardware—and applications were changing and that Unix would have to become a standard interface between the two realms. Then, as hardware or hardware vendors changed, applications could continue functioning because Unix could be ported to accommodate the changing hardware.

From 1974 to 1977, Unix source code was licensed inexpensively or for free to universities. AT&T, going back to a 1956 United States-AT&T antitrust consent decree, was forbidden to sell software commercially. Instead, they had to make such computer technology licensable to and by the public. Unix V6, released in 1975, was especially popular. It was free and it was great for teaching about operating systems because it was distributed with its source code, and students could follow along after they became familiar with the C language. It was great for program developers, too. They could write new applications—in C, for example—and then take full advantage of the Unix operating system. DEC had provided thousands of PDP-11s to universities across the United States at very little cost.

The perfect match of Unix and PDP-11 was thus (accidentally) repeated many times. This ground was fertile for further Unix development.

UNIX Becomes Commercialized

In 1979, AT&T announced that it planned to commercialize Unix, an announcement that came true in 1983 with the release of its Unix System V. Meanwhile, the University of California at Berkeley, among others, began developing its own version of Unix based primarily on Unix V7. Berkeley was also implementing the *Transmission Control Protocol/Internet Protocol* (TCP/IP) networking protocol suite that had been based on its *Berkeley Systems Distribution* (BSD) of Unix. The Internet itself was being developed and expanded, sponsored by the U.S. Department of Defense Advanced Research Projects Agency (that is why the original Internet was called the ARPAnet).

The upshot was the first major proliferation of different versions of Unix in the early 1980s:

- AT&T was developing its System V Unix.
- Berkeley was actively developing BSD Unix.
- Sun Microsystems (founded by Berkeley Ph.D. recipient Bill Joy) produced its own BSD-based Unix called SunOS.
- Microsoft and the *Santa Cruz operation* (SCO) were already distributing XENIX.
- Hewlett-Packard developed HP-UX for its workstations.
- DEC released ULTRIX.
- IBM developed versions of Unix for its PCs, PS/2s, and System/370s. (Later, in 1986, IBM would develop AIX first for the RT 6150 and then for the RS 6000.)

So, a former strength of Unix, AT&T's free source code, was now becoming a sort of weakness because so many new Unix versions were being developed based on the pre-commercial V7, depending on the needs of the respective developers who were not following any type of standard.

UNIX Moves Toward Standardization

In 1984, in an effort to promote open architecture standards that emphasized Unix as the operating system, several European vendors (among them, Siemens, Amdahl, and Philips) formed X/Open. American compa-

nies (including AT&T, DEC, and Sun) and Japanese companies (such as Hitachi and Wang) later augmented the membership. The open aspects they promoted included the development of systems that allowed applications to be easily ported among them, that were interoperable, and that allowed users to work on the various systems without extensive retraining. X/Open was interested in using member consensus to adopt and integrate open standards to ensure that new products conformed to them, but not to actually write them. (In 1987, X/Open was incorporated as a limited company, the X/Open Company Limited, whose shareholders were several worldwide information technology suppliers.)

In 1988, AT&T bought a large portion of Sun Microsystems; therefore, AT&T and Sun could then collaborate on future versions of Unix. The purchase in effect merged AT&T's System V and Sun's OS. In response, nine other vendors (IBM, DEC, HP, Bull, Nixdorf, Siemens, Hitachi, Philips, and Apollo) formed a consortium called the *Open Software Foundation* (OSF); membership ballooned to more than 200 firms by 1991.

The OSF's approach was to produce a Unix operating system and other software and then license the use of its products to its members and others. In fact, in late 1990 they produced what they claimed was the first open UNIX operating system, called OSF/1. OSF/1 was based on Carnegie Mellon University's Mach, a Unix operating system with original ties to BSD Unix. Mach was developed by Carnegie Mellon from 1985 to 1994. After the introduction of OSF/1, however, only DEC adopted it outright; other OSF members—such as IBM and HP—adopted only parts of it.

OSF also developed the Motif GUI guideline and toolkit for further development of the X Window System. The X Window System and the X networking protocol had been under development at MIT since 1984. (We will discuss X again in Chapter 15, "Linux Documentation and Support.")

As a countermeasure to OSF, AT&T and Sun allied themselves with yet more and different vendors to create the *Unix International* (UI) trade association. Members of that association—from computer manufacturers and software developers to consultants and academics and even government agencies—would advise AT&T regarding future development of System V UNIX, although AT&T would retain its proprietary control. By 1991, UI's membership reached more than 200 from a dozen countries or so. Although some overlap existed between the lower-rank members of OSF and UI, no such overlap existed among principal members or sponsors.

Let's step back a bit because of the relevance to Linux. The year 1987 saw the birth of MINIX, a much smaller Unix-like operating system written by professor Andrew S. Tanenbaum of Vrije Universiteit in Amsterdam,

Holland. This OS was written from scratch—meaning that it contained no AT&T code—for university instruction. MINIX is mentioned again in the next section because it was Linus Torvalds' springboard to Linux.

In 1993, Novell purchased AT&T et al.'s UNIX Systems Laboratories, which included the source code to System VR4 (System V, Release 4), the amalgamation of System V, BSD, and XENIX as well as the UNIX trademark. Novell then negotiated with X/Open to give X/Open the UNIX trademark. This deal gave X/Open the power to bless any new operating system as a version of UNIX if it met X/Open standards. One of those standards is XPG4.2 (X/Open Portability Guide 4.2), which includes several *Portable Operating System Interface for UNIX* (POSIX) standards developed by the *Institute of Electrical and Electronic Engineers* (IEEE) for operating system interfaces. (For example, POSIX.1 is the application program interface standard for the C language, and POSIX.2 is the shell and utility interface standard.) Later in 1995, Novell sold its UNIXWare (including the source code) to SCO, which continues to develop UNIX to the present day.

In this brief history of UNIX, a great deal has been left out. UNIX development has continued on many fronts in the 1980s and 1990s and into the new millennium. To cover it all and to do it justice would require an entire book. But before proceeding, we should mention that X/Open and OSF eventually merged, forming the Open Group in 1996. The Open Group still strives to promote, develop, and license open standards software, especially UNIX.

Multics—Epilogue

You might wonder what happened to Multics. It was, indeed, eventually developed. The first Multics system was unveiled by MIT and the remaining Multics partners in 1969—late and not without problems. GE had jurisdiction over future Multics development, which it sold—along with its entire computer business—to Honeywell Corporation in 1970. Nevertheless, Multics systems were installed in several locations.

Development during the 1970s included a new storage system, using the new concepts of logical and physical volumes to give it better recovery capabilities. Also, in 1977, the first commercial relational database was installed on a Multics system at Honeywell in Phoenix. Multics was used by MIT, the U.S. Air Force, GM, Ford, and the University of Southwest Louisiana and was purchased by a large number of organizations in Europe in the 1980s.

In 1985, Honeywell canceled further development of Multics. Over the next three years, however, they made several attempts to revive it, as did a few other companies who wanted to buy Multics. In 1988, Honeywell transferred maintenance of Multics to the University of Calgary, Canada. That university set up a corporation called ACTC Technologies, which was eventually renamed Perigon Systems. Perigon was acquired by the CGI Group in late 1998. We called CGI in the late summer of 2000 and were told that they had just put their last Multics system to sleep. Later, in the fall of 2000, we heard that Canada's Department of National Defence retired the last-known (well, rumored, anyway) Multics system, apparently located in Halifax, Nova Scotia, Canada.

Although that seems to be the end of Multics systems, we would just like to acknowledge and congratulate those who were involved in its development, deployment, configuration, administration, and maintenance for all those years. Bravo, and thank you. Without your courage and pioneering spirit, who knows what the state of *information technology* (IT) would be today?

History of Linux

Any history of Linux would be remiss if it did not mention MINIX, because it was on a MINIX USENET newsgroup bulletin board (comp .os.minix) that Linus Torvalds posted his now-legendary notices, one of which we quote from later in this section. As mentioned, MINIX was a small UNIX-like operating system written by Professor Tanenbaum. Like several other versions of UNIX, MINIX was written from scratch, with no AT&T code, for university instruction. It is useful for anyone who wants to learn the basics of UNIX operation. It is available free on the Internet at www.cs.vu.nl/~ast/minix.html. We suggest that you visit the site. You will find out how MINIX has evolved and learn about the two current versions (MINIX 2.0 for Intel CPUs from 8088 to Pentium and MINIX 1.5 for Intel, Macintosh, Amiga, Atari, and SPARC). You will also find out about the special copyright on MINIX by publisher Prentice Hall. MINIX is basically a type of public domain property.

In 1991, Linus Torvalds, a student at the University of Helsinki in Finland, created Linux. He wanted to develop an operating system that would exceed MINIX's modest standards. In August 1991, shortly after creating his Linux version 0.01, he published the following message to the comp.os.minix newsgroup:

> Hello everybody out there using MINIX—I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386 (486) AT clones. This has been brewing since April, and is starting to get ready.
>
> I'd like any feedback on things like/dislike in minix, as my OS resembles it somewhat. Any suggestions are welcome, but I won't promise I'll implement them :-)

According to Torvalds, "I'd guess the first version (v. 0.01) went out in the middle of September '91." The first official working version, v. 0.02, was made available in October 1991. To read the recollections of Torvalds, visit Web sites such as www.li.org/linuxhistory.php.

Many additions and revisions followed, but the first complete, bug-free version, v. 1.0, was released in March 1994. These version numbers correspond to the kernel version only, not to versions of Linux distributions as applied by the respective manufacturers. The latest kernel version, as of this writing, is 2.4.2-2. Reasonably current kernel version timelines are found at www.linux-history.org/kernel/ and at www.memalpha.cx/Linux/Kernel/.

One of the most remarkable aspects of Linux's history is that it is a true child of the Internet. Torvalds posted his original notice and request for help over the Internet, and most of the improvements to Linux have come from more than 100 programmers from all over the world, courtesy of the Internet. You can also download Linux versions from the Internet at any time.

Linux Features

To quote from Linux Online!, "Linux is a free UNIX-type operating system." It is a POSIX implementation, which means it meets Open Group POSIX standards (described previously in this chapter). It enables multitasking, simultaneous multiple users, the sharing of system libraries for efficiency, TCP/IP networking, virtual memory and swap spaces, and other UNIX OS features. Users or administrators can use a GUI or the command line.

Linux enables you to set up Internet or intranet services, and many use it for setting up Internet firewalls. In fact, because Linux does not require steep licensing fees and can be used on relatively inexpensive (and used) equipment, it is becoming a favorite of *Internet service providers* (ISPs). In addition, Linux can accommodate existing Microsoft Windows applications and can be dual-booted with Windows operating systems. It can also be integrated into existing multi-vendor networks—especially UNIX-

based ones—because of their similarities. Soon, Linux will be incorporated into mobile, handheld computing devices.

Free Software

Linux is free software, meaning that it is distributed under the terms of the GNU General Public License developed by Richard Stallman. (To read about this ex-MIT software developer and his "free software as in freedom of speech, not free beer" philosophy, we invite you to visit his Web site at www.gnu.org/philosophy/free-software-for-freedom.html.) This free software does *not* mean that the Linux kernel, other associated software, or entire distributions are the same kind of free as you associate with public domain. It is also not shareware. Basically, *free software* means that you can use the software for any purpose; study it to see how it is built and how it works; adapt and improve it; and redistribute it free or for a price. But you cannot restrict the software after you have redistributed it (that is, you must distribute it under the GNU GPL, too), and you must provide the source code just as it was supplied to you.

Sometimes, you will see references to *open source software*. This software might be equivalent to free software or it might mean a free download with copyright restrictions. If you find yourself faced with this terminology, it is best to investigate further.

Mix and Match

At least 40 full-feature Linux distributions are available in English, andmore than 20 are available in other languages. There are also 35 special versions available for embedded systems, control systems, and so on (for further information, see Chapter 15, "Linux Documentation and Support"). Here, we define a *distribution* as an amalgamation of the Linux kernel with other associated software. The 60-odd distributions are not all the same. Each has a different focus and slightly different features aimed at a specific user audience (besides the obvious fact that the French editions, for example, are for a French audience). Each distribution has different perceived strengths and weaknesses.

In addition, you do not have to use a shrink-wrapped distribution; instead, you can download the kernel of your choice. For example, instead of using a stable, tested kernel, you might want to live on the edge with a less-tested, potentially less-stable kernel. Or, you might want to combine a

certain kernel with different applications or other special features (such as RAID support or integration with a specific network) than would normally be found with a standard distribution.

Linux enables you to mix and match, use an existing distribution, modify a distribution, or create your own distribution. You can purchase copyrighted software for Linux from software developers (new or established). Or, you can obtain software from the Free Software Foundation (www.gnu.org/fsf), which follows the GNU General Public License. Remember, GNU GPL software might not always be free (as in no cost), but it is always free as in source code and has no restrictions on the user, as described previously.

Hardware and Software Compatibility

Linux runs on Intel-compatible PCs, Alpha computers from Digital Equipment Corporation, and Sun's SPARC computers. Linux can also talk to proprietary databases from IBM, Oracle, Sybase, Informix, and Pick, and open source databases such as PostgresSQL and MySQL. A wide variety of office applications run on Linux (StarOffice and Corel WordPerfect, to name two). Several GUI interfaces are available in open source but are restricted, such as KDE, or free, such as GNOME. As mentioned, an introductory tour of the X Window System and some X Window manager applications appear in Chapter 14, "The Linux X Window System."

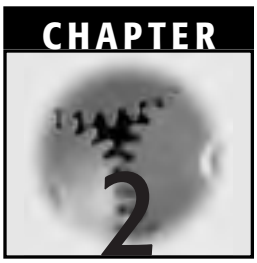
Further Developments

Today, there are more than 80 UNIXes, from AIX to XENIX. Many are undergoing continual development (such as Linux, SCO UNIX, FreeBSD, and AIX), while some have stagnated and are no longer supported (for example, Carnegie Mellon University's Project Mach). Carnegie Mellon, however, still maintains a Mach Web site at www-2.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/mach.html.

According to a Gartner Group-Dataquest survey conducted in the third quarter of 1998, UNIX workstations accounted for 67.4 percent of the worldwide workstation market with respect to revenues. Shipments of UNIX increased 14.6 percent between the third quarter of 1997 and the third quarter of 1998, also in terms of revenues. In terms of units sold, UNIX workstations accounted for approximately half, indicating that UNIX systems dominate the middle to high-end markets. This situation is a much better position than one would believe based on impressions obtained from the mainstream media.

Meanwhile, Linux has been gaining more friends everywhere. It has been endorsed by top industry vendors—IBM, Dell Computer, Oracle, Hewlett-Packard, SAP, and so on. According to Network World (www.nwfusion.com/power2000/power-strlinux/power-strlinux.html), "it was the second-most-shipped server operating system in 1999, behind Windows NT, with market shares of 24.6% vs. 38.1%, according to market research firm IDC. Linux out-shipped Novell NetWare and all combined Unix flavors, each of which had less than 20% market share. IDC predicts that Windows and Linux will continue to be No. 1 and 2, respectively, into 2004."

All are drawn by Linux's low cost, reliability, flexibility, and robust performance. Linux has also cracked specialty applications and the handheld market. More ISPs are moving to it, too. A purchaser for a large U.S. enterprise, however, stated that he will not recommend going to Linux. Why? Because it is free software, there is "no one to sue" if something goes wrong. Sheeesh.



Installing Linux from a CD-ROM

In this chapter, we discuss getting ready for an installation on an *Integrated Drive Electronics* (IDE) system, booting from CD-ROM, the necessity for a boot disk and rescue disks, and then some troubleshooting. Then, we walk through a fairly typical Linux installation on an also fairly typical IDE system.

You will encounter a few concepts, commands, and procedures here that we do not explain in detail until later in the book. If we fail to provide you with sufficient cross-references, we apologize. But you should be able to get by using the table of contents and the index. In our previous edition, we left the installation procedure until the end of the book, but we received many suggestions to move it forward.

First, though, if you are going to follow along, then you will need a copy of Linux. Linux can be obtained from several sources, including your local software vendor, from the various distribution manufacturers directly (say, via the Internet), or on CD-ROMs included with instruction books such as this one. If you have lots of bandwidth and a CD-ROM burner, you can actually download the ISO images directly from several vendors' Web sites (see Chapter 15, "Linux Documentation and Support," for a list of Linux distributions), such as Red Hat.

The installation menus are dynamic and might change order for various distribution versions. The installation in the chapter uses Red Hat Linux version 7.1, which implements a Linux kernel level of 2.4.2-2. All the Red Hat distributions starting at version 6.X offer a GUI installation option. We will bypass the GUI installation and intentionally guide you through a text menu installation for three reasons:

- There is (unfortunately) still a high failure rate for GUI installations.
- Our objective is to present you with a more consistent view of a Linux installation procedure that has been in place since the very early versions. Thus, if you are installing an earlier version, then this book can still assist you.
- The text menu installation includes some menus that you would not normally see in the GUI install and thus gives you a more technical view of the overall procedure.

By definition, "Linux is a UNIX-like operating system kernel that it can be freely distributed." Technically, Linux is not freeware nor is it in the public domain. Linux is protected by both a code of honor and the *GNU General Public License* (GNU GPL). Essentially, this situation means that you should not change the source code of the kernel and then release it under a more restrictive license. Any changes or modifications made to the Linux kernel are governed by the GPL as well. In other words, you can get the code from anywhere and legally use it. You can even resell it. But you have to supply the source code, too, just like it was supplied to you.

Installing Linux on an IDE System

As we said previously, in this section we will install Linux IDE architecture. IDE is an IBM PC *Industry Standard Architecture* (ISA). It is generally less expensive than SCSI and is simpler to configure. We have drawn the disk drive configurations (hard disk drive and CD-ROM drive, too) in Figure 2.1 for you to study the partitions and mount point data.

Troubleshooting the IDE Installation

If you have trouble with the installation, stop and research what you are doing. Based on our experience, most problems involve hard drive config-

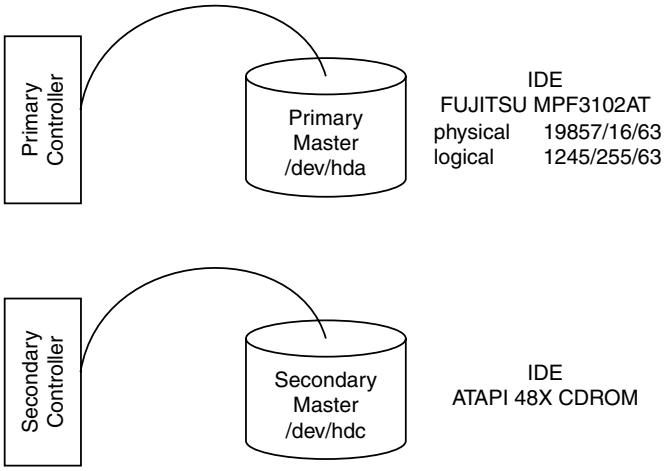


Figure 2.1 IDE disk drive configurations.

urations or hardware compatibility, followed by network card and bad driver configuration.

With respect to drives, the problem is often a lack of documentation on either the jumper or the termination settings. If you have a pair of IDE drives or a hard drive and CD-ROM combination, then it is often (actually, it is usually) a master/slave configuration issue. If you suspect that is your case, then make sure that all the IDE devices are compatible with each other. In other words, do not mix older IDE technology with the newer enhanced IDE technology. If they work together, then you are lucky. We have a saying in the industry that says, "Are we being lucky or smart?"

Network card and driver combinations can also be troublesome. In our opinion, Linux developers have made amazing progress in this area over the past two years. Three years ago, there were very few network cards supported in Linux—and now there are many. The newer network cards are PCI and therefore are self-configuring. When it works, it is great. Of course, how successfully this technology works with your system is entirely dependent on the BIOS that you have installed in your system. If the BIOS is buggy or old, then it could create a very difficult situation. Do not ignore the DOS utilities that we will tell you about soon in this chapter. Also, do not spend very much time on a network card that does not work without checking out the IRQ, I/O port, and memory address space that it is using (PCI or no PCI).

Watch for the following nasty behavior: You can ping the card internally, but it will not respond to a DHCP server. This situation is typical of a resource conflict, usually memory space. To check these issues, go directly to the */proc* directory and list the contents of all files called *iomem*, *ioports*, *irq*, and *net*. If the network card is not even present in these files, then the IRQ on the network card is no doubt set to something that is already used by another device. If you cannot even ping the card internally, then the driver is probably not functioning correctly. If you do not get the "link light" on the back of the network card, then it is probably the wrong driver or the network card itself. We recommend that you only spend so much time with a network card that will not work and then go out and buy one. For example, the last D-Link network card that I purchased was less than $20. Your time (and state of mind) is likely worth more than that. Barring that approach, you can find many Linux drivers on the Internet at www .driversguide.com, for example.

Video configuration is also a huge concern—so huge that we dedicated an entire chapter to it in our companion book, *Customizing and Upgrading Linux.* Please refer to that book for advice on video theory and video configuration.

Initiating a Linux Installation

Technically, all Linux CDs are bootable. Whether your Linux CD will boot your system depends upon whether the CMOS on your PC is set correctly. For example, most PCs are configured to boot from floppy disk first and hard disk second. You have to change the boot sequence to CD-ROM first, floppy second, and hard disk third. If it still fails to boot, then your BIOS or hard disk controller probably does not support booting from a CD-ROM, in which case you will have to create a boot diskette. You can create a boot diskette from either a DOS/Windows or UNIX session. You should have two diskettes available to accomplish this task.

CD Booting Problem Determination

Sometimes it appears that Linux cannot see or boot from IDE CD-ROMs. If this problem occurs, you should try to determine why Linux could not see the IDE device. It is possible that the BIOS cannot see or read an IDE channel because the BIOS is limited. This is probably the case if the system was functioning properly before under a different operating system. The solu-

tion is to either upgrade the BIOS or create a floppy boot diskette. We would recommend that you create a floppy boot diskette because it is easier. Another reason might be that the system might not be configured correctly. This situation is usually the case if you have taken the opportunity to add or move drives around in the system. Check the master/slave jumpers on all the drives and devices. If the drives are all connected to the same IDE controller, then one device should be configured as the master and the other device should be the slave. You will have to locate the documentation for the individual IDE device to find out how the jumpers should be set, because these instructions vary for different devices and for various manufacturers.

Another more interesting problem is dealing with a CD-ROM that will not load automatically after the installation. To get the system up and running, you can enter the following statement at the LILO prompt during the boot sequence:

```
boot: linux hdx=cdrom
```

x is the IDE letter that Linux specifies for the drive. This letter will vary depending on the IDE bus to which the drive is attached. This scenario might also cause you some problems during boot time. Make a note of it, and if the system also fails to boot after the installation, then revisit your cabling and master/slave configuration.

Installation Boot Disks and Linux Rescue Disks

With some Linux versions—those before the equivalent of Red Hat 6.x, for example—to provide additional support for *Personal Computer Memory Card Internal Association* (PCMCIA) and *Session Message Block* (SMB) installation, you have to create additional diskettes ahead of time. If you will be installing an earlier version here, then you will be prompted for these diskettes and they must be readily at hand. With newer versions of Red Hat Linux, however, the installation procedure has been changed, so this issue is no longer present.

The method for creating a Linux install boot disk depends on the operating system you have to begin with and whether you want to use that operating system to create the boot disk. We will discuss two methods in the next section:

- Creation from within a DOS/Windows operating system
- Creation from within a UNIX system

Boot diskettes made using DOS/Windows or Linux will behave the same way; thus, choosing one over the other makes no difference.

Meanwhile, if you are prompted to insert a blank diskette, which will be used to create a customized rescue diskette, we highly recommend that you do so.

You can create a rescue disk after the installation is finished should you have skipped this step during the installation. To create a post installation rescue diskette, go to the */lib/modules* directory and use the following command to create a rescue diskette customized for the system:

```
#cd /lib/modules
#mkbootdisk  --device  /dev/fd0  kernel_number
```

`kernel_number` will be similar in format to this one: 2.4.2-2. Although you choose to install a newer kernel, you must insert the appropriate version numbers for the level of kernel you are *currently running*.

You should always have a rescue diskette on hand that is specific for each system. For example, it is considered bad practice to use a rescue diskette from a system with a kernel level of 2.2.14-5 to rescue a system that is supposed to be at a kernel level of 2.4.2-2. The current kernel level of any system can be determined by observing the login prompt or by using the `uname a` command at the system prompt.

You can also create a rescue diskette directory from the Red Hat CD by going to the */images* directory and issuing the following command:

```
G:\>
G:\>cd images
G:\images>dd  if=rescue.img  of=/dev/fd0  bs=1440  count=1
```

Creating the Install Boot Disk from a DOS/Windows System

To create a boot diskette from DOS/Windows, we will use a program called `rawrite`. In this example session, we insert the Red Hat Linux product CD into the CD-ROM drive of a DOS or Windows system. The `rawrite` utility will be in a directory called *dosutils*. The boot file that you need is *boot.img*. Your listing of the *images* directory might not be identical,

but should resemble the one found in Step 1, as follows. When you run `rawrite`, it will prompt you for the *boot.img* file to transfer to the boot diskette. Use the following instructions to create a boot diskette from a DOS or Windows system:

1. Put the Red Hat CD in the CD-ROM drive. Change to the */images* directory and list the contents.

```
G:
G:>cd \images
G:\images>dir
Volume in drive G is Red Hat Linux_i3
 Volume Serial Number is 3BC3-AD45

 Directory of G:\images

08/30/00  06:40p        <DIR>          .
08/30/00  06:44p        <DIR>          ..
08/30/00  06:48p                   485 TRANS.TBL
08/30/00  06:39p             1,474,560 boot.img
08/30/00  06:39p             1,474,560 bootnet.img
08/30/00  06:40p        <DIR>          de
08/30/00  06:40p               860,160 drivers.img
08/30/00  06:40p        <DIR>          es
08/30/00  06:40p        <DIR>          fr
08/30/00  06:40p        <DIR>          it
08/30/00  06:40p               147,456 paride.img
08/30/00  06:39p             1,474,560 pcmcia.img
08/30/00  06:40p        <DIR>          pxeboot
08/30/00  06:40p        <DIR>          sv
              14 File(s)      5,431,781 bytes
                                      0 bytes free
```

2. Run the `rawrite` utility from the *\dosutils* directory and transfer the *boot.img* file to the floppy disk.

```
G:\dosutils\rawrite
Enter disk image source file name:  boot.img
Enter target diskette drive:  a:
Please insert a formatted diskette into drive A: and press -ENTER- :
  <CR>
```

   The result will be a bootable floppy with the following files on it. You can use the floppy to boot the system and to initiate the installation process.

```
Volume in drive A is LINUX BOOT
 Volume Serial Number is 2410-07EF

 Directory of A:\

08/30/00  06:39p                 5,860 LDLINUX.SYS
08/30/00  06:39p                   425 SYSLINUX.CFG
08/30/00  06:39p                 4,807 TEMPLATE.IMG
08/30/00  06:39p               789,543 INITRD.IMG
08/30/00  06:39p               600,265 VMLINUZ
08/30/00  06:39p                   751 BOOT.MSG
08/30/00  06:39p                   653 EXPERT.MSG
08/30/00  06:39p                   859 GENERAL.MSG
08/30/00  06:39p                   860 PARAM.MSG
08/30/00  06:39p                   506 RESCUE.MSG
08/30/00  06:39p                   545 SNAKE.MSG
              11 File(s)      1,405,074 bytes
                                 49,664 bytes free
```

Creating the Install Boot Diskette from a UNIX System

In a UNIX-based environment, you can use the `dd` utility to create a boot diskette. First, you have to mount the CD to get to the *images* directory where the *boot.img* file resides. The `dd` utility is a UNIX utility, so it is not available as such on the Linux CD (eventually it will be available on your new Linux system, but for now you have to use another Linux/UNIX system).

1. Put the Red Hat CD in the CD-ROM drive. Mount the CD-ROM and change to the */mnt/cdrom* directory and list its contents:

```
Red Hat Linux release 7.0 (Guinness)
Kernel 2.4.2-2 on an i686
login: freston
Password:
Last login: Thu Jun  7 11:48:36 from 8.3.105.200
[freston@HostA freston]$ su - root
Password:
[root@HostA /root]# mount /dev/cdrom
[root@HostA /root]# mount
/dev/hda1 on / type ext2 (rw)
none on /proc type proc (rw)
usbdevfs on /proc/bus/usb type usbdevfs (rw)
none on /dev/pts type devpts (rw,gid=5,mode=620)
automount(pid469) on /misc type autofs
(rw,fd=5,pgrp=469,minproto=2,maxproto=3)
```

```
/dev/hdc on /mnt/cdrom type iso9660 (ro,nosuid,nodev)
[root@HostA /root]# cd /mnt/cdrom
[root@HostA cdrom]# ls -l
total 63
-rw-r--r--    10 root     root        18385 Sep  7  1999 COPYING
-rw-r--r--    10 root     root         4730 Aug 25  2000 README
-rw-r--r--     5 root     root        25206 Aug 27  2000 RELEASE-NOTES
-rw-r--r--    10 root     root         1908 Sep 25  1999 RPM-GPG-KEY
drwxr-xr-x     4 root     root         2048 Aug 30  2000 RedHat
-r--r--r--     1 root     root          465 Aug 30  2000 TRANS.TBL
-r--r--r--     1 root     root         2048 Aug 30  2000 boot.cat
drwxr-xr-x     6 root     root         4096 Aug 30  2000 dosutils
drwxr-xr-x     8 root     root         2048 Aug 30  2000 images
```

2. The boot file that you need is *boot.img*. Change to the */mnt/cdrom/images* directory and list the contents of that directory to confirm that the *boot.img* file is present. Your listing of the `images` directory might not be identical, but it should resemble the following:

```
[root@HostA cdrom]# cd /mnt/cdrom/images
[root@HostA images]# ls -l
total 5333
-r--r--r--    1 root     root          485 Aug 30  2000 TRANS.TBL
-rw-r--r--    2 root     root      1474560 Aug 30  2000 boot.img
-rw-r--r--    2 root     root      1474560 Aug 30  2000 bootnet.img
drwxr-xr-x    2 root     root         2048 Aug 30  2000 de
-rw-r--r--    2 root     root       860160 Aug 30  2000 drivers.img
drwxr-xr-x    2 root     root         2048 Aug 30  2000 es
drwxr-xr-x    2 root     root         2048 Aug 30  2000 fr
drwxr-xr-x    2 root     root         2048 Aug 30  2000 it
-rw-r--r--    2 root     root       147456 Aug 30  2000 paride.img
-rw-r--r--    2 root     root      1474560 Aug 30  2000 pcmcia.img
drwxr-xr-x    2 root     root         2048 Aug 30  2000 pxeboot
drwxr-xr-x    2 root     root         2048 Aug 30  2000 sv
```

3. Run the `dd` utility to transfer the *boot.img* file to the floppy diskette. Provide the *boot.img* file and the floppy disk designation as arguments to `dd` so it will transfer the appropriate files to the boot diskette.

```
[root@HostA images]# dd if=boot.img  of=/dev/fd0
2880+0 records in
2880+0 records out
```

The result will be a bootable floppy with the following files on it that you can use to boot the system with and initiate the installation process. For your information, please note that although we used the `dd` utility here, the files are identical to those created with `rawrite` in DOS.

```
 [root@HostA /root]# mount /dev/fd0
[root@HostA /root]# cd /mnt/floppy
[root@HostA floppy]# ls -l
total 1401
-rwxr-xr-x    1 root     root          955 Apr  8 22:37 boot.msg
-rwxr-xr-x    1 root     root          658 Apr  8 22:37 expert.msg
-rwxr-xr-x    1 root     root         1202 Apr  8 22:37 general.msg
-rwxr-xr-x    1 root     root       768551 Apr  8 22:37 initrd.img
-r-xr-xr-x    1 root     root         6192 Apr  8 22:37 ldlinux.sys
-rwxr-xr-x    1 root     root          862 Apr  8 22:37 param.msg
-rwxr-xr-x    1 root     root          506 Apr  8 22:37 rescue.msg
-rwxr-xr-x    1 root     root          608 Apr  8 22:37 syslinux.cfg
-rwxr-xr-x    1 root     root       652144 Apr  8 22:37 vmlinuz
```

TIP **If you are installing Red Hat like we are here, Red Hat has reported certain problems with boot images. Consider downloading the latest recommended boot images. For example, Red Hat has them available at www.redhat.com/errata. If you are determined to perform a GUI-assisted installation and your video is not working, consider downloading a newer *boot.img* file. That might resolve the problem.**

Invoking the Linux Installation Program

The Linux installation program consists of a series of dialog boxes sometimes called *screens* or *windows*. The dialog boxes present information to you and request information and decisions from you. The two most common methods used to invoke the installation program are as follows:

Inserting the Linux CD-ROM into the CD drive and booting the system. The system CMOS must be set to facilitate booting from the CD-ROM drive (see *Initiating a Linux Installation* earlier in this chapter).

Inserting a bootable floppy disk into the floppy disk drive and booting the system. The system CMOS must be set to facilitate booting from the floppy disk drive.

The installation chronicled in this chapter presumes that you will use the floppy boot method. In fact, you will eventually see a dialog box that instructs you to insert the CD.

We knew beforehand that the system we are installing Linux on in this chapter would require a network connection to the Internet. Therefore, we obtained some *Internet protocol* (IP) addresses before we started.

Entering Information in the Installation Program

As you install Linux, you will notice several different ways to enter information into the dialog boxes. These methods are as follows:

Text input. You will encounter dialog boxes in which you are to type information, usually on a dashed line. This information insertion method is used in conjunction with other methods such as buttons, toggles, and so on.

Check boxes. These are spaces defined by brackets or boxes that you will either select or deselect by pressing the space bar.

Buttons. These are generally square boxes that you select by pressing the Tab key and then the Enter key.

Scroll and select. Some dialog boxes contain lists of components, packages, or default services from which you will choose by scrolling up or down and selecting. You will use combinations of Tab and space keys, followed by pressing Enter at the end of the dialog boxes.

We present this list here for two reasons. First, there are the insertion methods you have to use, and the listing described earlier serves as a guide. Second and more importantly, the list helps you focus on the issue at hand rather than the collection of information. If you occasionally make a mistake, take heart—you will not be the first or the last to do so.

Insert your CD and boot the system. If you have trouble booting, read the following paragraphs. If you do not experience trouble booting the system, proceed to Step 1.

Stepping through the Installation

Our screens do not necessarily reflect what you will see on your system. They will, however, give you an excellent idea of what information you

will require for your installation. We will explain default behavior and our rationale for choosing specific options. It is our intention that you will be able to make informed decisions when performing your own installation.

Step 1. Boot Message

If you were to walk away from this first screen without selecting anything here, the installation program would proceed to boot the system on its own and would default to the GUI installation mode. We are specifically going to demonstrate the *text* mode installation (in other words, the `type: text <Enter>` option in the screen facsimile shown in Figure 2.2).

Install or upgrade . . . graphical mode. This option is the default and will launch a GUI install panel. Be aware that this choice has a high video failure rate because it does not interpret all video cards and monitors correctly. If you choose this option and it fails, we recommend that you immediately reboot and use the text option.

Install or upgrade . . . text mode. This option is provided for practical reasons to facilitate the installation of the majority of systems with no graphical support. Remember, though, that not all systems that are destined to be servers will have graphical support. If that is what your system is to be, you might *have* to select this option.

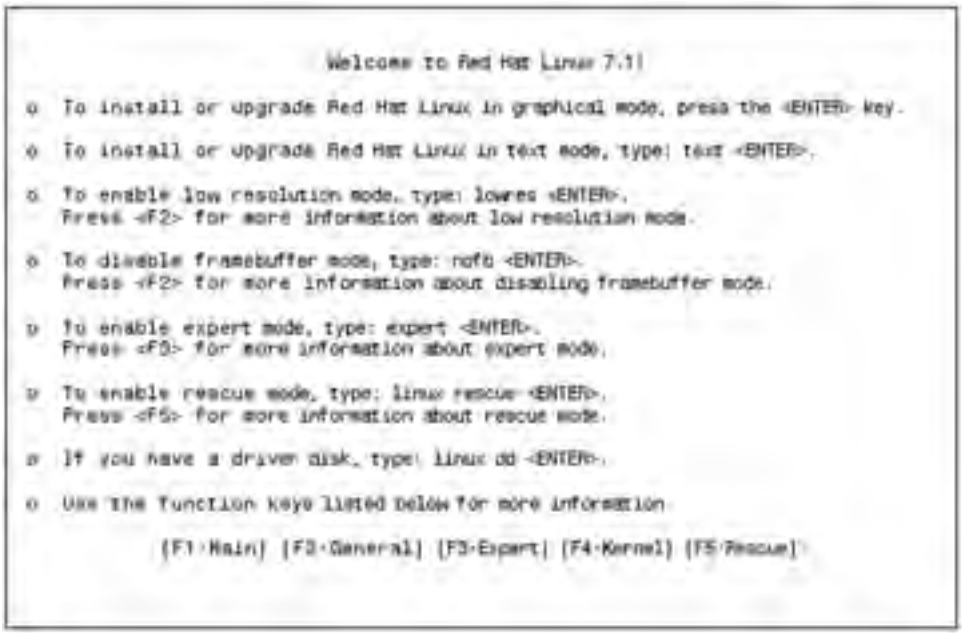


Figure 2.2 Installation options.

Enable expert mode. This option is not for the faint of heart but is sometimes necessary (for example, when Autoprobe does not appear to be discovering your card settings correctly). These dialog boxes will enable you to supply the necessary information for all card settings, such as IRQ, I/O, and DMA.

Enable rescue mode. This feature will enable you to boot a system from a Linux CD. This option is a last resort if you cannot boot the system and you do not have a rescue diskette.

All the rest of the dialog boxes we present in this chapter will presume that you have chosen text mode. We chose text mode.

Step 2. Select Language

Whatever language you choose in the screen shown in Figure 2.3 will become the default language used for the installation of the Linux operating system. You can add another language later should you require a multilingual system. We chose `English`.

Step 3. Specify Keyboard

Do not be creative here. The question in Figure 2.4 is a technical question about keyboard drivers. If you answer this question incorrectly, then issues might arise when you try to enable and map keys. Not all keyboards

Figure 2.3 Selecting an installation language.

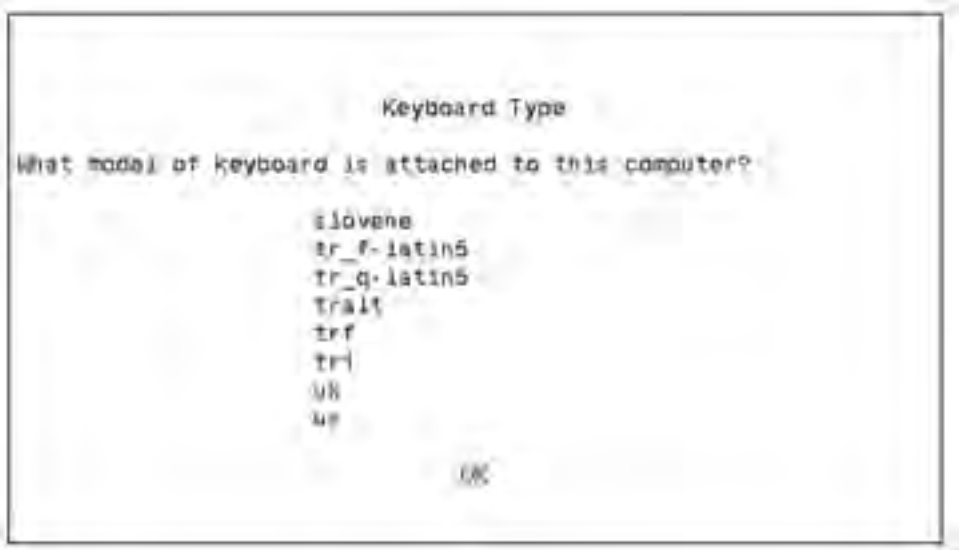


Figure 2.4 Keyboard type.

are created equal. You can, however, choose the closest one here and adjust it later when the system is up and running. We chose US.

Step 4. Welcome

Choose OK for the message shown in Figure 2.5 and go to the next screen.

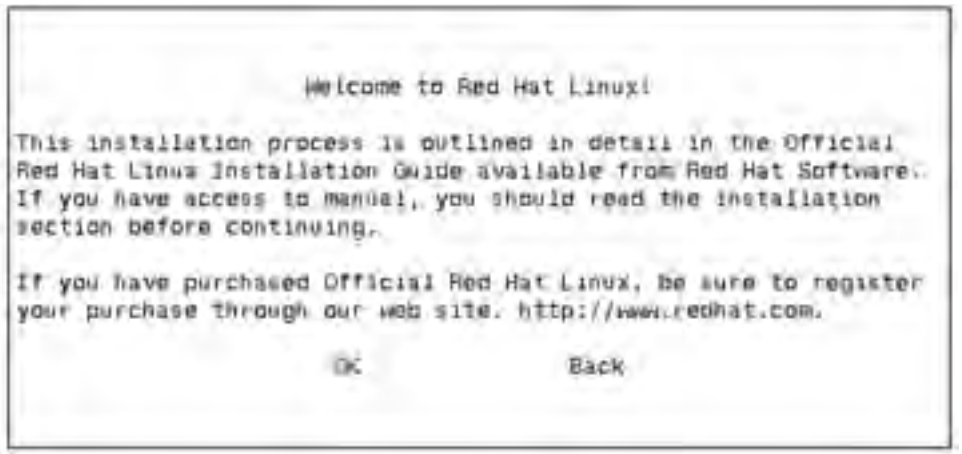


Figure 2.5 Welcome message.

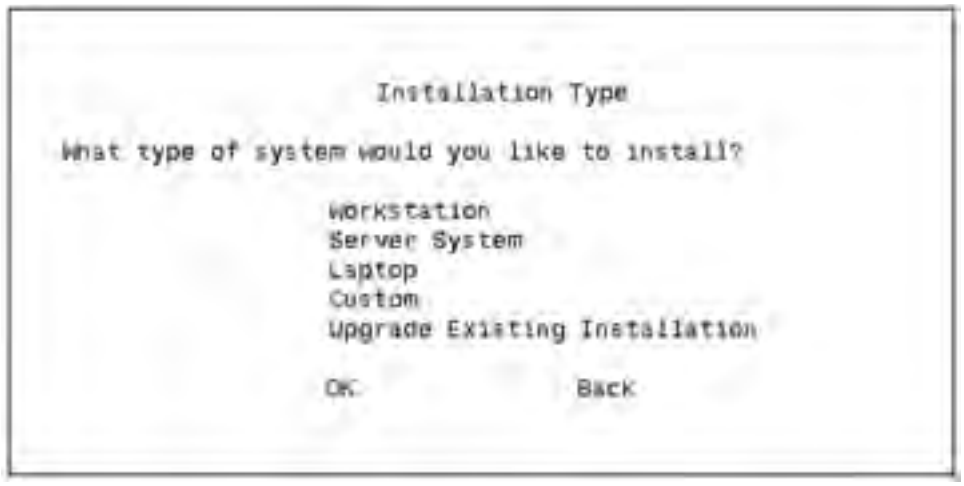


Figure 2.6 Installation type.

Step 5. Specify Installation Type

We will demonstrate a custom installation (see Figure 2.6), which will allow you to specify exactly which packages and utilities that you need. Just as a point of interest should you decide to install absolutely all of the packages: You will have to create a 2.0-GB system partition to hold the files, which does not include room for any data. Before we continue, though, please consider the following comments.

Among the most popular features of contemporary Linux systems are the new GNOME and KDE desktops. The one you choose should depend on what you want to do. If you are experimenting, try to install one and then the other. GNOME can run KDE applications and vice versa. Consider your choice between the two as making one dominant over the other. Of course, because the industry is somewhat political, the KDE development group has different mandates than the GNOME development group. Thus, we have two different factions creating desktops and utilities for such desktops.

Notice that both the GNOME and KDE choices here are workstation solutions. In other words, the default suggested combination of packages and utilities are well suited for a desktop system such as that for an end user. An end user here would also include a code developer or system administrator, whereas the server solution offered here would install a set of packages and utilities that you would not necessarily want on a workstation. For your information, if you choose `Server System`, no X server will be installed. This situation is a change from previous version behavior.

If you wish to have an X server for a server class system, choose `Select individual packages` during the installation or use the `rpm` utility after the installation is complete.

Finally, they have added a choice for laptop computers in response to so many users trying to get Linux up and running on their laptops only to find out that they were missing many of the required packages. The default installation for a laptop now includes those packages, such as a PCMCIA slot and power and TFT video support utilities.

There is also an upgrade path available. We highly recommend that you use this feature to upgrade a Linux distribution or Linux kernel to the next level. This situation is far preferable to attempting to recompile a kernel. Recompiling kernels is a specialized task that we are going to address in another chapter in this book.

To repeat, we chose `Custom`.

Step 6. Partitioning

We are going to choose `Continue` (see Figure 2.7) to use automatic partitioning to show you how the system will look after the installation. The automatic partitioning will create a small partition called `/boot`, which will contain the boot files for the system. All other files and directories will go into one large partition called `/`.

Please be reminded again that this installation is using Red Hat 7.1, which handles this situation differently from the previous Red Hat version 7.0. With RH 7.0, all the files—including the boot files—would go into one large partition called `/`.

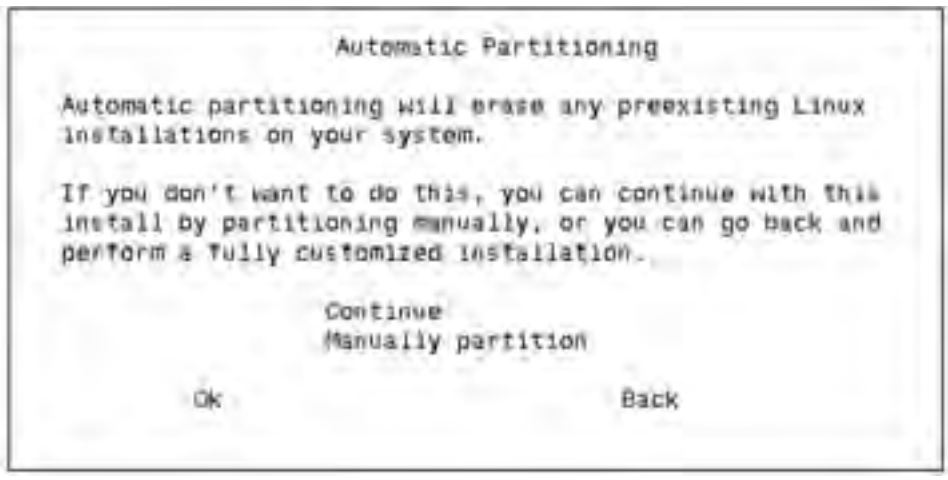


Figure 2.7 Hard disk partitioning.

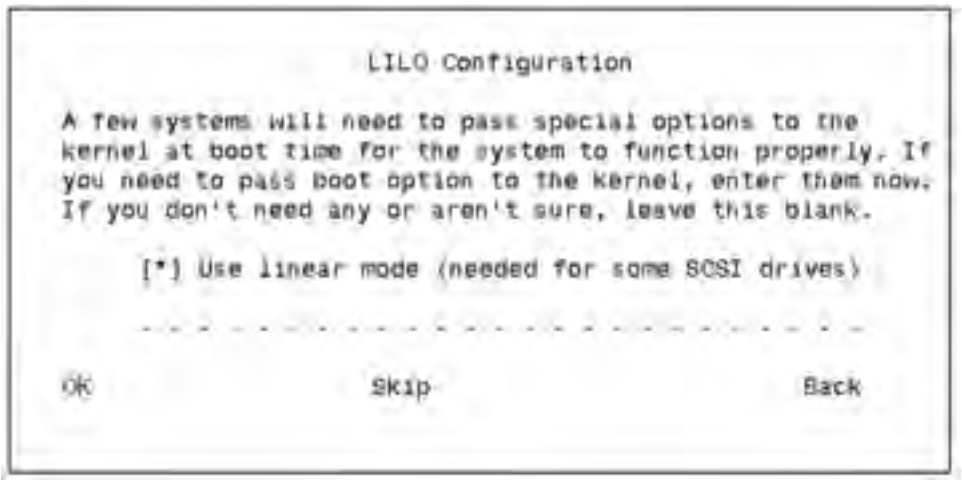


Figure 2.8 LILO configuration—boot options.

Step 7. Installing and Configuring LILO

The first LILO configuration screen (Figure 2.8) will appear and will ask about the use of linear mode. As it clearly says, it pertains to SCSI drives only—and we will address this issue in our SCSI installation after this section. On older versions of the Red Hat distribution, it did not state that it was specific to SCSI installations. If you were to choose to enable this feature, it would have no effect—nor would it cause any problems. We chose to remove the default `*` on `Use linear mode . . ..`

We plan to use LILO to manage the disk (see Figure 2.9). Therefore, we are going to choose the Master Boot Record (MBR) option. If there were some other alternate operating system partitions on the disk, the Linux configuration program would see them and you could choose to add them to your */etc/lilo.conf* file as boot choices (see Figure 2.10). Because we are only installing the one Linux system, we will let this choice default to booting Linux. If your system is a production system, we recommend that no other operating systems should share the disk. This situation will primarily simplify the maintenance, backup, and recovery of the system. In other working environments, however, such as at home, you might appreciate having more than one operating system from which to choose.

Step 8. Network Configuration

We are going to use a static IP address in the screen shown in Figure 2.11. But, if this system is going to be a print server or will run any services for other hosts, it should receive its own static IP address so that it will not

Figure 2.9 LILO configuration—bootloader options.

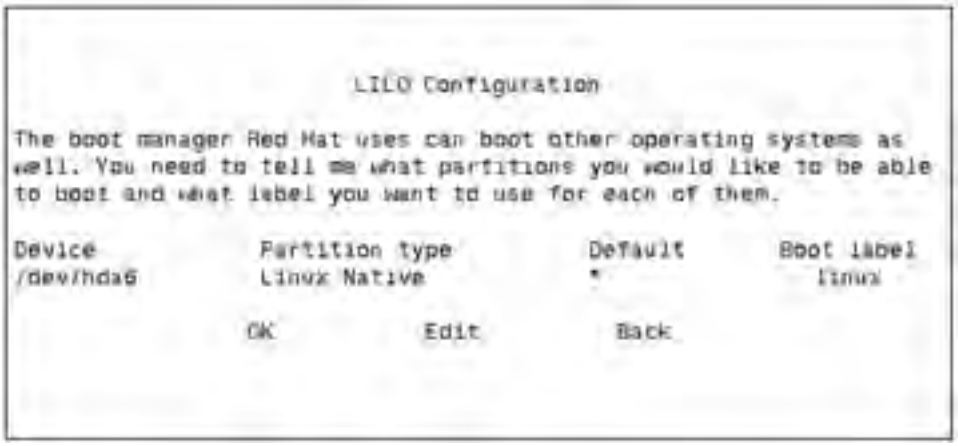


Figure 2.10 Boot partitions.

change. If the address changes, then other hosts on the network might not be capable of finding the service. Another issue to examine is whether any software on this host is not sensitive to having the IP address changed. Because we knew this machine would join an existing network and would eventually access the Internet, the IP addressing information we provided was as follows:

Disable bootp/dhcp

IP address: 192.168.2.205

Netmask: 255.255.255.0

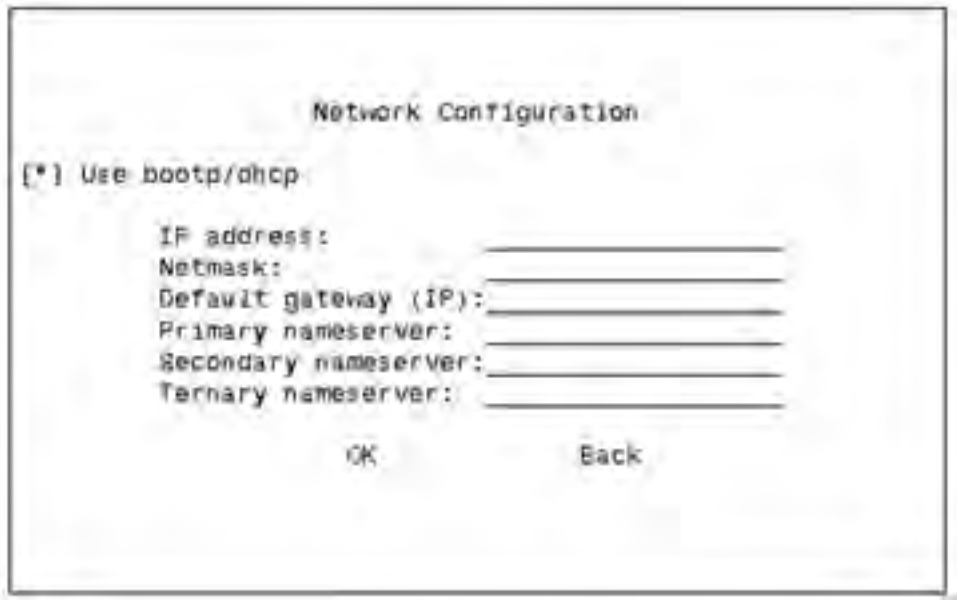


Figure 2.11 Network configuration.

Default gateway: 192.168.1.1

Primary nameserver: 24.64.2.33

Secondary nameserver: 24.64.2.34

You can revisit this menu at any time after the installation via the `/usr/sbin/netconf` or `/usr/sbin/netcfg` utilities. Please note that there is no opportunity during the installation to configure an ISP Internet connection. You can configure this option after the installation by using the `/usr/sbin/internet-config` utility.

Step 9. Firewall Configuration

Firewall configuration (see Figure 2.12) is a new feature added to the Red Hat 7.1 distribution. This feature will not appear on the Red Hat 7.0 or previous equivalent distributions. This feature is a response to concerns that users had about security exposure on the Linux system itself. This particular feature set does *not* create a firewall appliance in the sense that this system can be used as a firewall for other systems. This feature set just provides some firewall protection to the system on which it is being installed. Now, about these Security Levels: For example, if you choose `Medium` as an option, it will prevent Telnet access *to* the system but will *not* prevent Telnet connections *through* the system if the system is now or eventually

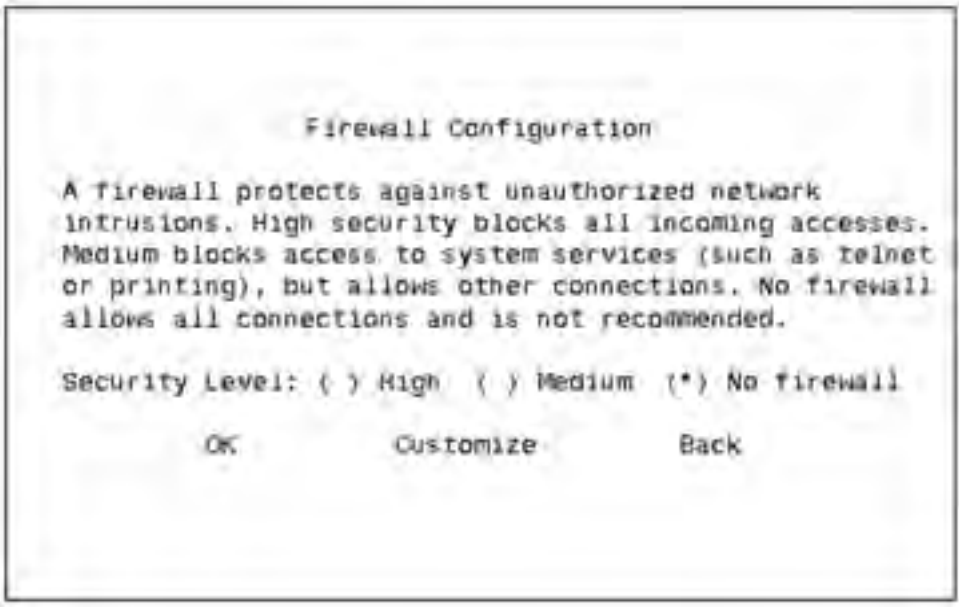


Figure 2.12 Firewall configuration.

equipped with two network cards. Linux can be used on firewalls, but this goal is not the objective of this dialog box.

Because in our shop we already have a proper firewall in place and we wish to have easy Telnet access to this system from various points on the network so that we can manage it remotely, we chose `No firewall`.

If the system you are installing Linux on is to be at the end of ISP service, we would recommend `Medium` or `High` so that it would not to be as easy a target for hackers. Meanwhile, you can reconfigure this information after the installation via the `/usr/sbin/firewall-config` utility.

Step 10. Mouse Selection

The system immediately asks you to configure the mouse if it senses one, as shown in Figure 2.13. If you have a three-button mouse, you will be better equipped to take advantage of the X Window system (please refer to Chapter 14, "The Linux X Window System") without confusion. For example, on a three-button mouse, the left button is copy; the middle button is paste; and the right button is used for context. On a two-button mouse, the left button is copy and the right button is paste. If you have a two-button mouse and you choose `Emulate 3-buttons`, then the left button will be copy, the right will be context, and clicking the left and right buttons together will be equivalent to the middle button. We have a two-button mouse. We chose `Generic - 2 button Mouse (PS/2)` and `Emulate`

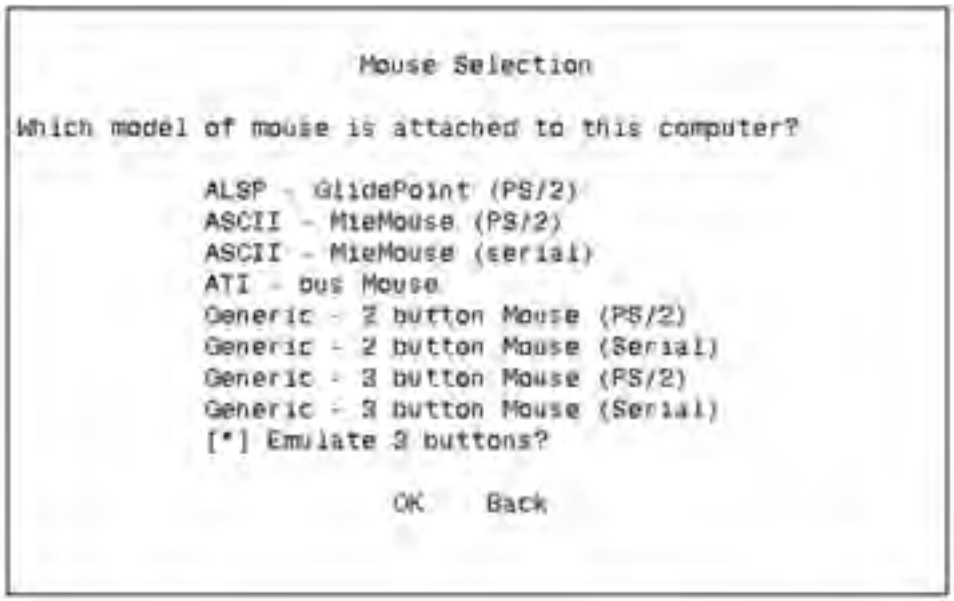


Figure 2.13 Mouse selection.

`three buttons`. You can revisit this information after the installation via `/usr/sbin/mouseconfig`.

Step 11. Language Support

We chose `English (USA)` in the screen shown in Figure 2.14 because we wish to have those code pages installed on this system. Installing more language support uses up significant space on your disk. All message files, code pages, and related files—such as screen fonts for the X Windows system—will be installed along with the language support.

If you are going to install additional or different language support here, give some thought to what you are going to use for keyboard support. Language support and keyboard support generally go hand in hand. This information can also be reconfigured after the installation via `/usr/sbin/locale_config`.

Step 12. Time Zone Configuration

Somewhere in your hardware setup plan, way back at the beginning, should have been instructions to set the time properly in the system CMOS. Does your CMOS time function on your system handle *Greenwich Mean Time* (GMT)? If it does and you are going to use the system itself as a source of time, then set the time in the CMOS and use the hardware clock set to GMT option here. Then, it will convert the time set in your CMOS to

Figure 2.14 Language support.

the local time. If your CMOS date and time does not have the GMT capability and you set the time in your CMOS to the local time, then do not set the feature in the screen shown in Figure 2.15. The decision to use the hard-

Figure 2.15 Time zone selection.

ware clock will ultimately have to do with whether or not this system will be a standalone entity with respect to time. For example, standalones are often used by developers or by system administrators as utility systems. If the system is going to be a server, it might have to participate in a network by using the *Network Time Protocol* (NTP). Then, it would be appropriate to make use of the Time Zone Selection. Plan for this situation ahead of time. We cannot tell you how messed up a database server can get if the time is set in the past. Not many database applications can recover from time changes gracefully. We chose *not* to use the hardware clock because we have the local time set in our CMOS, but we did choose `Canada/Mountain` (a little local bias, eh?). This information can be reconfigured after the installation via `/usr/sbin/timeconfig`.

Step 13. Root Password

Remember to choose a good password (see Figure 2.16) and then ensure that you remember it. Linux has rules and guidelines for proper passwords. Reviewing them would be worthwhile. Check the `man` pages (we discuss these manual pages for commands in Chapter 3, "Getting Started Using the Linux System") for `passwd` and the Red Hat online documentation for guidelines. This password can be reset after the installation by first logging in as `root` and using the `passwd` utility.

Figure 2.16 Specifying the root user password.

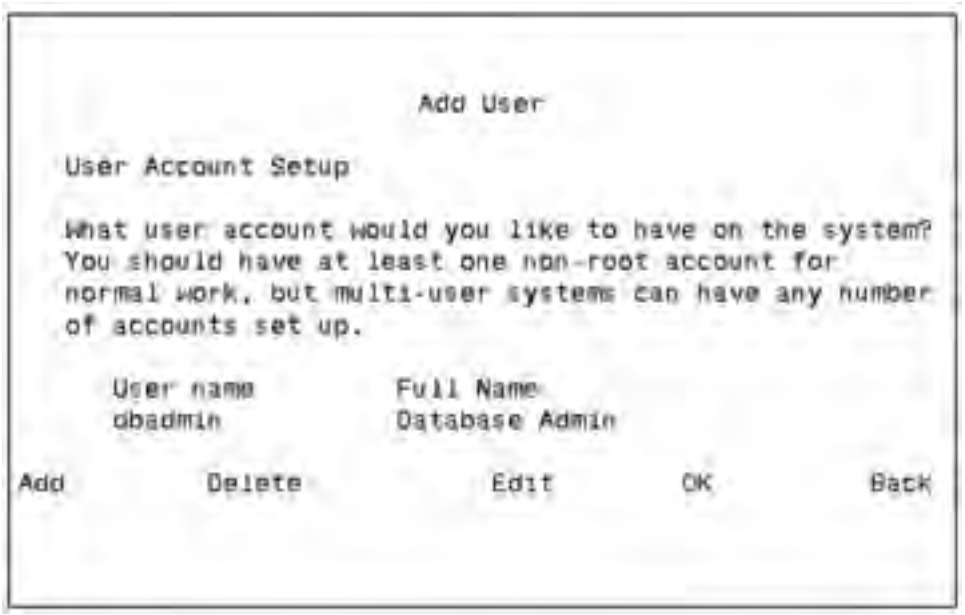


Figure 2.17 Adding non-root users.

Step 14. Add Non-Root Users

The installation process will enable you to add users at this time (see Figure 2.17). You do not need to add all system users now, though, because you can revisit this task at any time after completing the installation. The reason for the Add User dialog here is to enable you to log into the system for the first time as an ordinary user and not necessarily as a root user. Technically, it is not proper form to log into the system as root. Therefore, for those who play only by the rules, the opportunity is here. We chose to create a `dbadmin` user at this time because we knew we were going to need this one. We have plans to create all of our users after the installation is validated and ready for system administration. Another reason to create a non-root user is if you have to have Telnet access to the system after installation. By default, the Telnet daemon will not let you log in directly as root. You have to log in as a regular user and then `su` to root. Therefore, if you only have Telnet access immediately after the installation, you will require an additional userid.

Step 15. Authentication Configuration

The two default selections shown in Figure 2.18 will ensure that you are compatible with other systems on the network. Configuring for *Network Information Services* (NIS), *Lightweight Directory Access Protocol* (LDAP), and

Figure 2.18 Authentication.

Kerberos support is available for advanced configurations. These advanced configurations are almost mandatory in large distributed server environments. These capabilities have always been available for Linux, but now they are configurable during the installation process. We are going to use the defaults here: `Use Shadow Password` and `Enable MD5 Passwords`, in order to be compatible with our other operating systems on the network. This information can be reconfigured after the installation via `/usr/sbin/authconfig`.

Step 16. Package Group Selection

Components are groups of software packages that provide an overall service or feature to the system. The components shown in Figure 2.19 do not display all the possible selections, only the ones we suggest installing for this exercise. You might note that the dialog box on your screen is a little smaller than the one shown; use the arrow key to reveal more components. We installed these components:

- Printer support
- X Window System
- GNOME
- DOS/Windows Connectivity
- Games
- Networked workstation

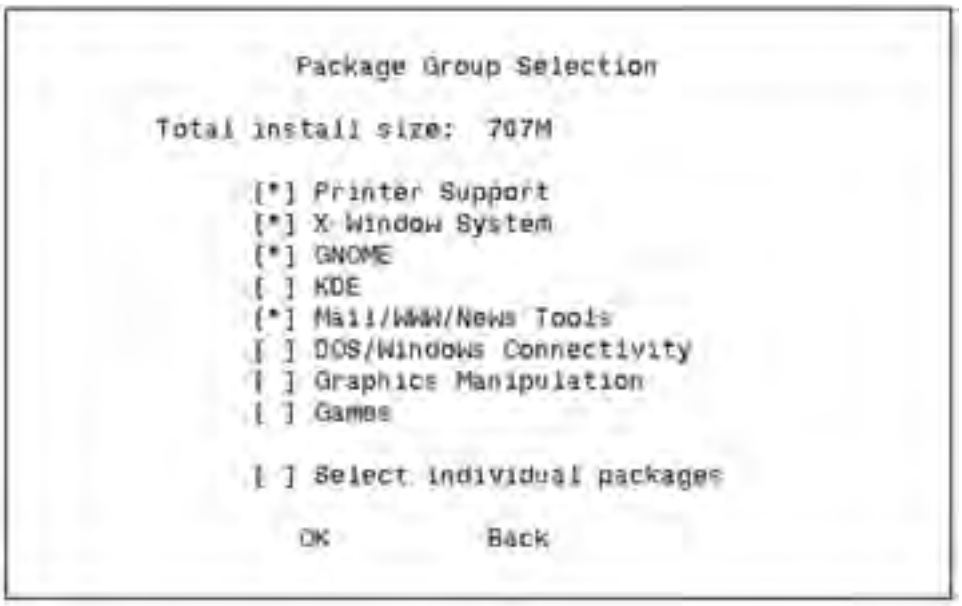


Figure 2.19 Selecting packages to install.

- NFS Server
- Kernel development
- Utilities

We could have added KDE, too, at least for comparison purposes. Normally, either GNOME or KDE is installed. Support for additional features can be added after the installation by using `rpm`, the Red Hat Package Manager. All of the associated services for various run levels can be managed after the installation using either `/usr/sbin/chkconfig` or `/usr/sbin/ntsysv` to modify the services.

There is potential here for frustrating errors. For example, if you had chosen to install a GNOME Workstation, these packages would be different. You would get networking capability but you might not have Telnet or *File Transfer Protocol* (FTP) functionality, which might be important to you. Normally, you would get only Telnet, FTP, and NFS if you had chosen to do a Server installation (remember, we chose Custom). Now is your chance to add all functionalities by choosing `Select individual packages`. It is much easier to add functionalities here than to install and configure them later after the installation.

We have worked with earlier versions of Red Hat, so we were used to working with utilities such as `linuxconf` and `XFree86Config`. We consider those packages to be absolute *musts* for a system administrator. Do

not trust the `Utilities` function. Go into `Select individual packages` under `Applications/System` and `User Interface/X` and go get the other two. While you are there, check for all your Telnet, FTP, and other essential daemons as well. Some daemons that were standard before are not anymore with the newer distributions. Also, although they can be added afterward, it is easier to deal with them here.

If you scroll down through these dialog boxes, you will eventually see the `Everything` choice at the bottom of the component list. We mentioned earlier that choosing to install everything will result in 2.0 GB of files and directories on your system. If you are performance-minded, you might want to be somewhat selective at this stage. If performance is not a primary consideration, you might want to choose all components and thus all of the packages.

WARNING **The installation program will tell you if you have inadequate disk space. In the meantime, do not forget your planning and do not make disk space too tight in the `/` and `/usr` file system partitions. Finally, be sure to avoid removing mandatory packages, such as kernel support.**

If you choose to install a particular package that uses another package as a prerequisite, the installation process will bring that to your attention, as shown in Figure 2.20 and will install all required software for you. We recommend `Install packages to satisfy dependencies`.

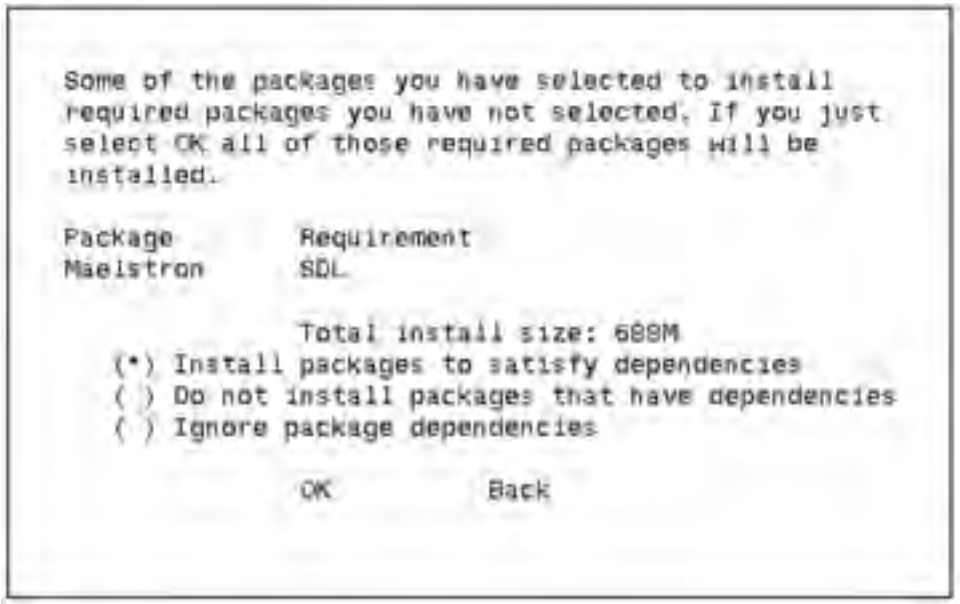


Figure 2.20 Adding packages to satisfy dependencies.

Figure 2.21 X probing for a video card and X server.

Step 17. Probing for the Video Card and X Server

In our case, the installation process has conducted a PCI Probe and found our video adapter (see Figure 2.21). We recommend at this point that you make a note of the video card and X server response and move along. Our experience is that this action will fail on most systems most of the time. We have an entire chapter dedicated to configuring the video on a system. Remain focused on other features and functionality at this point and move through the video configuration, accepting all of the suggested defaults. Apparently, we were having a good day (and we hope you do, too) and it recognized our video chip set. The video configuration can be reconfigured after the installation via `/usr/bin/X11/Xconfigurator`. Choose `OK`.

Step 18. Installation

A log file called */tmp/install.log* will remain after the installation so you can review what files and packages were installed (see Figure 2.22). This log can also serve as good documentation in the event that you ever have to re-create or recover a system without a proper backup system.

Figure 2.22 Installation begins.

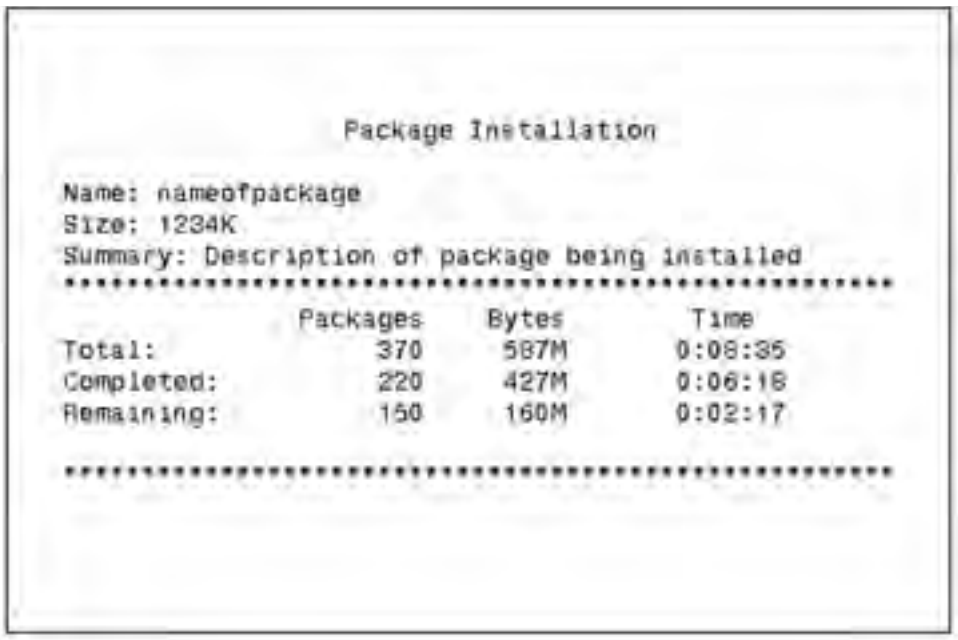


Figure 2.23 Installation progress.

A progress screen (see Figure 2.23) will keep you informed as to the status of the file copy process. Throughout the installation process, you will see the installation status dialog box, which will show you the progress of the installation of individual packages and programs as well as the overall progress of the total Linux installation. Note in the top half that as the individual packages install, the installation program will provide the package name, the size of the package, and a brief description of the package.

On the bottom, the program keeps track of the space used by the install packages as well as the remaining space required. Moreover, it tracks the total, elapsed, and remaining installation time. This point might be a good time for you to take a break while the installation proceeds.

If the second CD is required, it will be requested (see Figure 2.24). Do not worry if the second CD is not required; it just means that you did not choose to install any packages from that CD. If the CD-ROM does not run immediately, give it some time and try again.

Step 19. Create a Custom Bootdisk

Although you can create a custom boot disk at any time, we recommend that you make one now (see Figure 2.25). Did you know that these rescue disks are not generic but are specific to each system? Please, do it.

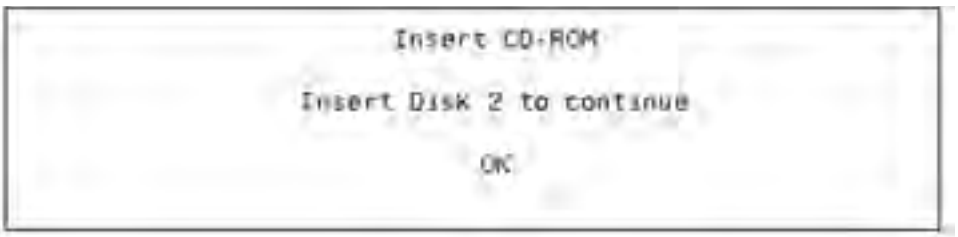


Figure 2.24 Request for the second CD-ROM.

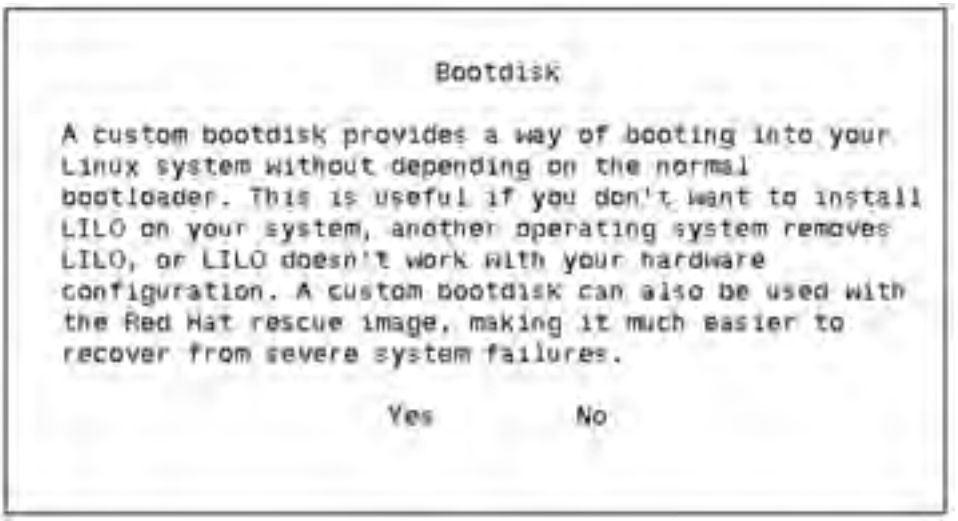


Figure 2.25 Creating a custom boot disk.

In response to the screen shown in Figure 2.26, insert a blank floppy disk and choose `OK`. The screen shown in Figure 2.27 will appear.

Step 20. Monitor Setup

Although scores of monitor names appear on the screen shown in Figure 2.28, your monitor might not show up in the list. If your monitor is not there, you can choose `Custom` here. If you choose `Custom`, it will try to test the configuration later. In the event your video fails, do not worry and move on. We will show you how to configure the monitor when we show you how to configure the video in another chapter. Again, we were having a good day; the Gateway Vivitron 17 was in the list.

Step 21. Back to Video Configuration

At this point (see Figure 2.29), our installation might not reflect what your installation process is doing because these menus are dynamic with

Figure 2.26 Inserting a blank floppy.

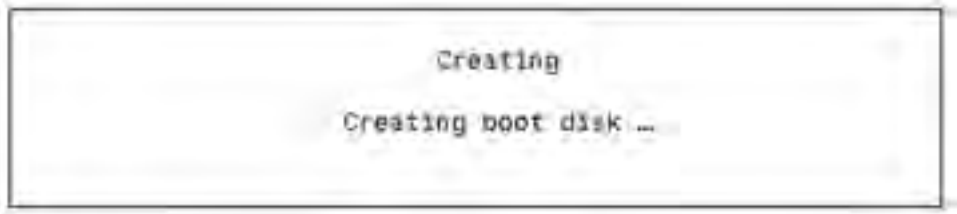


Figure 2.27 Inserting a blank floppy.

respect to previous responses made. In our case, it probed and located a video card and is now prompting us to indicate how much memory is available or on the video card. Again, we emphasize that you should move on, and we will reconfigure the video and monitor later using a better utility. Our video card has 4 MB of memory on it. So, naturally, we chose `4 mb`.

The only way that you would know which clockchip you had (see Figure 2.30) was to read the documentation that came with the video card. We are choosing to configure the video later, however, so choose `No Clockchip Setting` and move on. Once the system is running, you can run `/usr/bin/X11/SuperProbe` to identify the video chipset.

Although we are going to configure the video setting later, you have to choose at least one in the screen shown in Figure 2.31. We recommend that if you choose anything, choose at least these: `8 bit: 800x600, 1024x768, 16 bit: 800x600, 1024x768`. These are representative of the lowest common denominator; that is, they are what most monitors can handle.

Figure 2.28 Selecting monitor type.

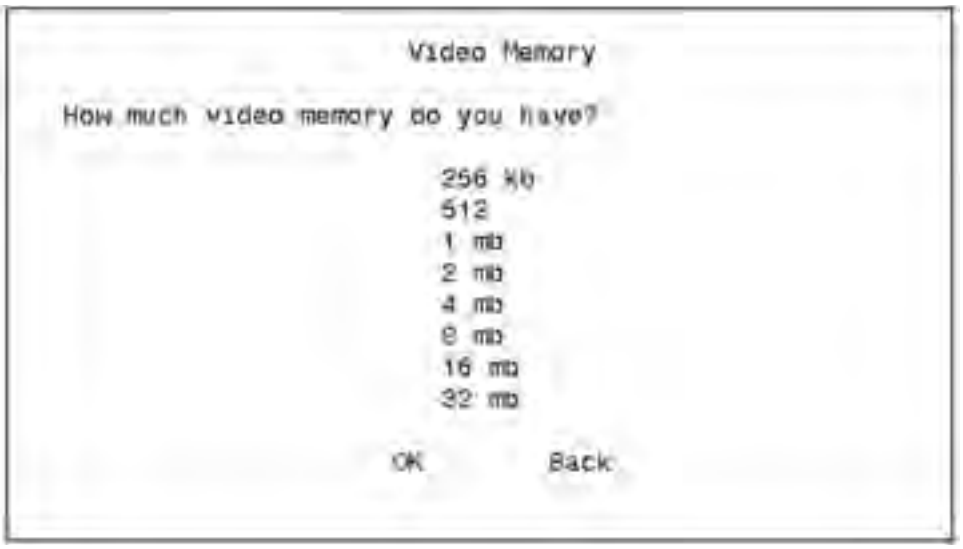


Figure 2.29 Specifying available video memory.

Step 22. X Window Configuration

You can choose OK in the screen shown in Figure 2.32 just for fun at this point, but if your video is not going well, this choice will definitely not

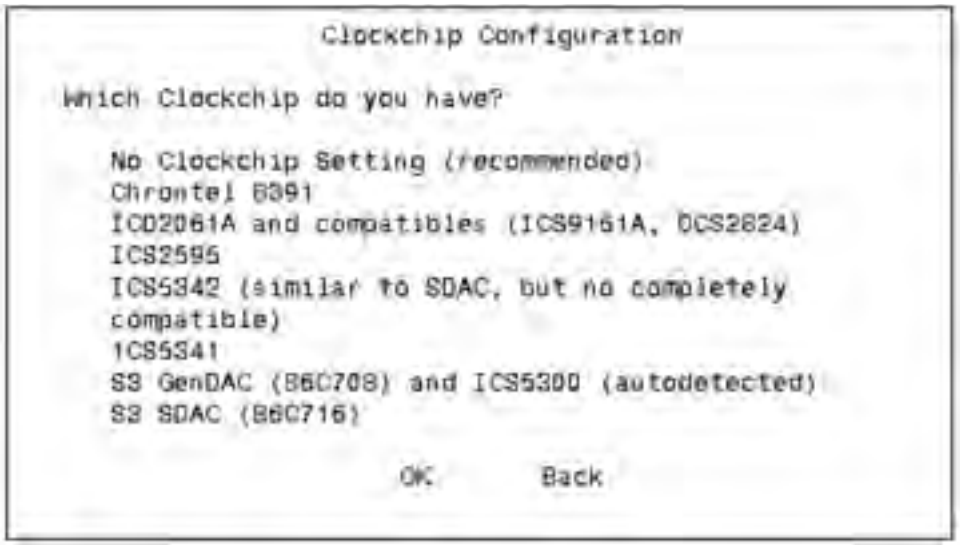


Figure 2.30 Clockchip specification.

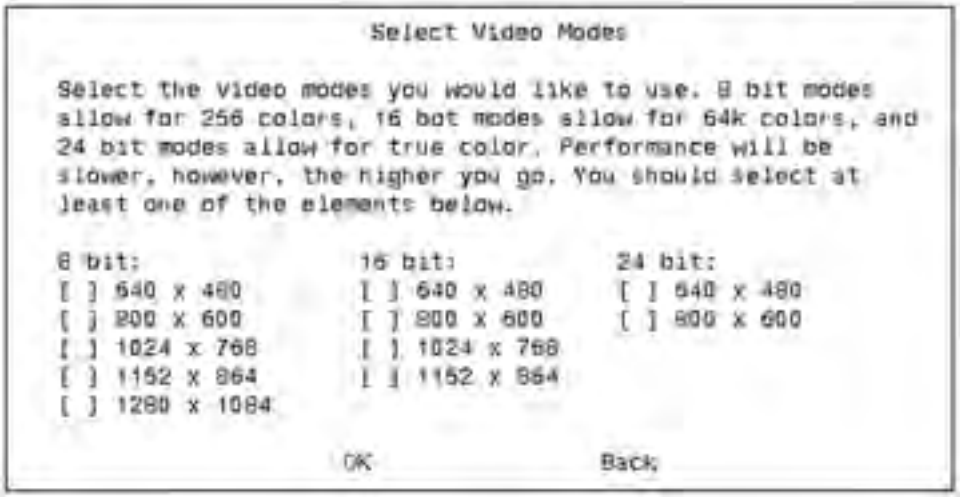


Figure 2.31 Video mode preferences.

work. If it works, then choose `Yes`. Only choose `Yes` if you do not see any irregularities. A typical irregularity would be a box or small images hanging around the mouse pointer or black lines on the screen where they should not be. Respond with `Skip` if there are any irregularities.

WARNING **You might also be presented with a screen asking you whether you would like the X Window desktop to automatically start when the system starts. We highly recommend that you say no in case the desktop hangs the system. You can always start the X Window desktop at any time by entering startx at the command line after you log into the system.**

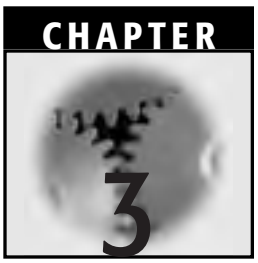
Figure 2.32 Testing the X Window configuration.

Figure 2.33 Configuration is complete.

Step 23. Complete

Congratulations! Your installation is complete (see Figure 2.33).

CHAPTER



Getting Started Using the Linux System

Initial access to any operating system is gained by logging in and out, so that is where we will begin here. The chapter continues with Linux/UNIX commands for adding users, adding and changing passwords, and communicating with all or some selected users on a Linux system. We include information on Linux/UNIX syntax and end with a description of several tools that we find useful and hope that you will, too.

Logging In and Out

Because Linux, like other UNIX systems, is a multi-user system, a basic level of security is implemented to control access. Although passwords might only be optional on some UNIX systems, they are *mandatory* on Linux systems. The system administrator sets the initial password for each user, but users can change their own passwords after they first log into the system.

NOTE **The term Linux/UNIX is used here to describe commands that work on Linux as well as on any UNIX-based system.**

Logging In

After you have booted your system and are ready to log in, the screen displays the `hostname login` prompt (`hostname` refers to the name given to the computer during the installation of Linux). At this point, you enter your username:

```
hostname login: username
```

You are then presented with the following message:

```
Password:
```

Obviously, this prompt tells you to provide your password by typing it. For security purposes, the password is not echoed back to (displayed on) the screen as it is entered. If the password matches your records and this is the first time you have logged into the system, the system replies with a system prompt. An example of such a prompt follows:

```
[username@hostname "home dir"]$
```

The prompt shown has two components: The first `username@hostname` indicates the name of the user and the system to which they are logged in, and `home dir` indicates that immediately after the login, the user has been placed into his or her home directory in the file system. Then, every time the user changes directories in the file system, the name of the new directory (that is, the present working directory or pwd) will appear as part of the prompt in the place where `home dir` is presently displayed. That is why, later in this book, you might see other terms in the prompt. One common term will be `username`, another indication that the user is situated in his or her home directory (the absolute name of which is commonly */home/username*; the prompt will only take the last part of the directory name). These system prompts can be customized. We will discuss customizing them later in Chapter 11, "Shell Variables and the User Environment," when we discuss the user environment.

Meanwhile, if you have logged in and out of the system previously, the system instead displays the following:

```
Last login: day date time on login location
[username@hostname "home dir"]$
```

In either case, the $ prompt indicates that the system is now ready for you to enter additional commands.

Logging Out

To log out, you have a choice of three methods: Entering the `logout` or `exit` commands or simultaneously pressing the combination Ctrl-D. The first choice looks like the following:

```
[username@hostname "home dir"]$ logout<Enter>
```

The `logout` command works only when you are in your own login shell, which is where you generally conduct your business anyway. (For more information about the login shell, see Chapter 11.) If you should discover that `logout` does not work, use the `exit` command as follows:

```
[username@hostname "home dir"]$ exit<Enter>
```

Alternatively, press Ctrl-D (repeatedly if necessary). Eventually, you will reach your login shell (where you can issue the `logout` command), or you will just be logged out of the system.

NOTE **If $ is part of the prompt, you are in your login shell.**

Once you have logged out, no matter which of the three logout methods you have used, Linux provides the same response:

```
distributorname Linux releasenumber
kernelnumber on a CPUname
hostname login:
```

Creating User Accounts and Passwords

On any Linux system, the system's root user or the system administrator creates user accounts and initial passwords. Ordinary users cannot create user accounts and cannot change the passwords of other user accounts.

If you have Linux installed on a home system or in a smaller network environment, you might not have a designated system administrator. Therefore, please note that when we refer to a system administrator, we

mean someone with root privileges. At Rueful Figures Inc., our sample organization, Freston the Wizard is the system administrator.

User Accounts

The first task in creating a user account, or record, is to specify a username. Linux imposes no restrictions on usernames, although we recommend that you do *not* use blank spaces, unprintable characters, or even mixtures of upper-case and lower-case characters in them. Such names can cause problems not only at login but also when you perform certain system administration tasks.

In addition, system administrators are usually advised beforehand by their respective organization about:

- The format of usernames (such as an abbreviated form of the user's name and the company name)
- The names of those who will soon be joining the organization's system (in other words, the names of new users)
- What organizational group(s) the new users are to join
- What their security clearances will be
- What file systems they will be able to access

The syntax for user account creation is simple:

```
[root@hostname /root]# useradd newusername<Enter>
```

or

```
[root@hostname /root]# adduser newusername<Enter>
```

Why are there two commands? Each comes from a different version of UNIX, and Linux developers did not want to play favorites. You can use either command; both accomplish the same thing.

(The `useradd` and `adduser` commands are examples of linking, in which two command names are linked to one command. Linking is not restricted to commands, however, but we are getting a little ahead of ourselves. We will discuss the linking function in more detail in Chapter 5, "Using Files in Linux.")

If you have made a mistake creating a user account (it happens), or if one or more users have left your organization, then you will have to delete the user account. Use the following syntax:

```
[root@hostname /root]# userdel username<Enter>
```

Meanwhile, to see what else you can do with `useradd` and `adduser`, check their `man` pages (we will discuss `man` pages in detail later in this chapter).

Group Accounts

Often, when user accounts are added to an organization's database, they are immediately assigned to specific user group accounts within that organization. That way, their activities and access to organizational resources can be quickly configured for security and convenience. If the system administrators assign user accounts to group accounts during the creation of the user account, they will include the `-g` option with the `adduser/useradd` command.

But, in order to add a username to a group account, the group account must have been created first. To create a group account, use the following syntax:

```
# groupadd [-g gid [-o] ] [-r] [-f] group<Enter>
```

The `-g` option is used if the administrator wants to assign a specific group number. Otherwise, Linux will assign the lowest available unique number greater than 500 that has not yet been assigned. Numbers 0 to 499 are reserved for system accounts. The `-o` option is used if the system administrator wants to advise Linux that the assigned group ID number is not unique. The `-r` option is used to indicate that the account is to be a system account. The `-f` is used to tell Linux to halt command execution if such a group already exists.

The `group` argument specifies the name of the new group.

The system administrator can check the correlation of accounts by examining the */etc/passwd* or */etc/group* files.

If the user account is to be added to more than one group, that can be done at the time of user account creation with the `-G` option. It can also be done later by using the `usermod` command as follows:

```
# usermod -G additionalgroupname username<Enter>
```

Similarly, `usermod` can be used to change initial group membership, as such:

```
# usermod -g newinitialgroupname username<Enter>
```

If a system administrator intends to delete a group account, he or she must first clear it of its component user accounts. Then, the following command is issued:

```
# groupdel groupname<Enter>
```

Example 3.1 Creating a Group Account

Originally, Don Quixote requested that Rueful Figures, Inc., staff members be grouped according to Table 3.1.

Example 3.2 illustrates this simultaneous user account creation with group account assignment.

Freston has complied and created the groups. Here is how he created `knights1`:

```
# groupadd -f knights1<Enter>
```

Example 3.2 useradd or adduser with Group Names

After Don Quixote hired Sancho Panza for Rueful Figures Inc., Sancho was given the username `panzasan`. As we saw, Sancho was to be added to the

Table 3.1 Rueful Figures, Inc., User Groups

GROUP NAME	DESCRIPTION	EXAMPLES
knights1	The company's "knights errant"—staff members who will conduct noble deeds	Don Quixote: username quixoted; Sancho Panza: username panzasan
knights2	Staff who conduct fruit growing, harvesting, and marketing functions; they also provide support to the knights1 group	Master Nicholas the Barber: username nicholas; Perez the Curate: username perez; others
knights3	Executive and management staff	Lady Dulcinea: username dulcinea; Sancho's wife Juana Gutierrez: username gutiejua; Lady Molinera: username molinera; Lady Toloso: username toloso
knights4	Special group for horses, burros, and other support animals	Don Quixote's horse Rozinante: username horseroz; Sancho's burro Dapple: username burrodap

group called knights1. Here is how Freston created Sancho's user account and simultaneously added his user account to the knights1 group account:

```
[root@hostname /root]# adduser panzasan -g knights1<Enter>
```

As we mentioned previously, Freston could also have used:

```
[root@hostname /root]# useradd panzasan -g knights1<Enter>
```

Passwords

Passwords are the primary mechanism for ensuring system security. For users to log in the first time, the system administrator must have already set the user's initial password.

Here is how a system administrator sets the initial password for a user. The syntax is

```
# passwd username<Enter>
```

After their first login, users can set their own passwords but they generally cannot set their own usernames. After a user changes his or her password, it is *encrypted,* which means that users (including the root user) cannot decode the password. Organizational policy generally requires that users change their passwords periodically (for example, every day, week, or month). Example 3.3 shows how users can set new passwords for themselves.

Example 3.3 passwd

According to Rueful Figures, Inc.'s policy, Freston set Sancho's initial password to be identical to his username panzasan, like so:

```
# passwd panzasan<Enter>
Changing password for user panzasan
New UNIX password: panzasan<Enter>
Retype UNIX password: panzasan<Enter>
passwd: all authentication tokens updated successfully
```

Impressed with Rueful Figures, Inc.'s security policy, Sancho later changes his password from panzasan to 1m@5qu1r3. Here is how he does it:

```
[username@hostname 'home dir']$ passwd<Enter>
Changing password for panzasan
```

```
(current)UNIX password: panzasan<Enter>
New UNIX password: 1m@5qu1r3<Enter>
Retype new UNIX password: 1m@5qu1r3<Enter>
passwd: all authentication tokens updated successfully
```

Remember that in reality, Linux does not display the old and new passwords on the terminal screen. We showed the passwords just for illustrative purposes.

Please notice that to prevent a user from being inadvertently locked out of the system through a simple typing error, the new password has to be entered twice. The system accepts the new password only if the new password meets Linux's basic rules and if the two typed versions match. The old password immediately becomes invalid.

Although Linux does not set restrictions on usernames, passwords are a different story. We have already mentioned that the root user must establish the first password for every user. We have also seen how new passwords must be entered twice. In addition, passwords are case sensitive.

The following are some guidelines for the root user when establishing passwords and for ordinary users who are changing their password:

- Make your password at least six characters long.
- Do not make the new password similar to the last one.
- Do not use any word that Linux would term to be a dictionary word.
- Try to make a password that is fairly easy to remember but not easy for others to guess. Some bad examples include the word `password`, the sequence `qwerty`, the user's first or last name, the sequence `123456`, a name or phrase that might be attributed to the user, such as `cat_lover`, and the user's phone number, birth date, or favorite sports team.
- Try to mix numbers with letters.
- Do not make your password identical or even similar to your username (in fact, Linux will compare the two and reject the proposed password if it *is* too similar).
- Although the authors for the `passwd man` page disagree, we suggest you write your password down and keep it secure (but also available) at all times.

If, when specifying passwords, the root user violates these guidelines, Linux will likely warn them that the chosen password is a `BAD PASSWORD` and will provide a reason why. Having warned the root user, however,

Linux will not prevent them from establishing the specified password anyway.

On the other hand, if an ordinary user violates the guidelines, Linux will warn them in the same manner but will also *prevent* them from establishing the password.

Command Syntax

Because we have already started to enter commands, it is a good idea to discuss Linux commands in general. Linux commands generally follow the syntax and format of UNIX commands. The order and correct separation of elements are important. The name of the *command* (or process, as some might refer to it) always comes first. The command name can then be followed by one or more *options* (some call them flags) that can, in turn, be followed by one or more *arguments* (these might also be called flags by some; it's best to be as precise as possible in your references, though). You must separate options from the command name and from other options by single spaces. Also, options must be preceded by a hyphen (-). In the following, `-f` and `-l` are options to their respective commands:

```
[username@hostname "home dir"]$ mail -f newmail<Enter>
[username@hostname "home dir"]$ wc -l filename<Enter>
```

We will be discussing both of those commands in more detail soon. For now, however, suffice it to say that the first command line reads, "Bring me the contents of my mailbox for processing (reading, replying, deleting, and the like). Then, when I am finished, return the undeleted messages to an alternate mailbox called `newmail`." Notice that the filename *newmail* is the argument to the command `mail` and its option, `-f`.

The second command line reads, "Count the number of lines in the file called *filename*."

You can group multiple options together and precede them by a single hyphen. For example,

```
$ ls -lf<Enter>
```

This command says, "List the files found in the directory I'm in now, but only the files (not the directories). Also, provide detailed information

about those files." The `-f` means "files only," and the `-l` means "detailed description."

If you do not precede an option with a hyphen, the system might try to treat it as an argument instead, which could result in an error message.

An argument is a further refinement of the command, usually indicating an object to be retrieved and worked on or an object to be created as a result of the requested process (like the file *newmail*, which we mentioned earlier). If you use more than one argument, then each argument must be separated from the option(s) and from other argument(s) by a single space. Unlike options, however, arguments cannot be bunched together.

Online Information for Linux and UNIX Commands

Linux and UNIX have a time-honored and effective tradition of providing various forms of online help. In this section, we introduce the most popular and helpful sources for help with Linux/UNIX commands: `man` pages, `info` pages, and the `usage` utility.

man Command

Almost every Linux command, system call, or special file has an online manual page (`man` page), which is an authoritative online document that is located in your Linux system directories and that you can access instantly. The `man` page is used in most if not all Linux and UNIX environments. Many users and system administrators use `man` almost exclusively.

Syntax

To consult the `man` pages for a command, subroutine, special file, or other element, the syntax is simple:

```
$ man commandname<Enter>
```

Notice that this time, `commandname` is an argument and not a command. The command here is `man`. If at any time you want to suspend or stop `man`

operation, press Ctrl-Z. Later, you can re-enter the `man` page where you left it by simply entering `fg`. These commands are discussed in Chapter 11.

Types of Information Available

You usually access the `man` pages from the command line. Once you have done so, several of the following categories of information will appear immediately on the screen in ASCII text format. Just *which* categories appear will depend on the information's origin and author as well as the format the author chose to use. This point is where the history of the selected command or function might have some influence:

NAME: The name of the command, subroutine, file, and so on and a brief description of what it does

SYNOPSIS: Synopsis is the syntax to use when invoking the command, subroutine, or file. (Synopsis is a term commonly used in the Linux and UNIX world to mean syntax.) Typically, the syntax follows the format we introduced here in this chapter, but you might also see a complete listing of all the options and arguments for the command for the particular version of Linux you are using. (In Example 3.3, we make further comments pertinent to the Synopsis section.)

INTRODUCTION or DESCRIPTION: A more in-depth definition or description of what the command does. This section can also list or discuss the options and arguments, or it might list step-by-step instructions to follow before, during, or after the command is executed.

OPTIONS: The available options and arguments.

FILES: A list of files and their relation to the command. (For instance, you might need to consult a certain file to find the parameters of the command.)

ENVIRONMENT: A list of the configuration files that are checked for certain parameters before a command is executed.

EXAMPLES: Although examples are often found in the Introduction or Description section, an author might list a number of common examples for using the command, subroutine, or file in this Examples section.

SEE ALSO: This part contains names of related commands or files or names of commands that might produce the same effect.

HISTORY: The version of Linux or UNIX in which this command first appeared, the original developers, and other historical information.

BUGS: Additional tips or cautions regarding how, when, where, and why you should use this command, file, or subroutine. You might also find directions regarding how and when *not* to execute the command.

SUMMARY: Another section for presenting options, descriptions, uses, and the like.

AUTHOR: The name of the individual or organization that developed the command, subroutine, or file or the name of the person or organization providing this information page. You might also find an address for reaching the author directly. Often, this section indicates whether the command originated with another version of UNIX or whether it was developed especially for Linux.

DERIVATION: At the bottom of the `man` page, you might find additional information regarding the origin of the command (sources such as AT&T or Berkeley or version numbers or dates).

DATE: The date of the `man` page's last update. This date does not indicate when the last update of the command's utility, options, or arguments occurred. The `man` pages themselves are not always updated when someone modifies an element of the command/program or file.

Navigating man Pages

After a `man` page is open, press the following keys to move around in it:

- The space bar navigates down the file one screen at a time.
- *b* navigates up through the file one screen at a time.
- *d* navigates down through the file one-half screen at a time.
- *u* navigates up the file one-half screen at a time.
- Enter or the down arrow key navigates down the file one line at a time.
- *h* displays a help screen with a number of other navigation commands.
- *q* quits the help screen and returns you to the `man` page.
- / followed by an alphanumeric string and Enter searches for the string.

- *n* finds the next occurrence of the previous search.
- *q* exits the `man` page.

After you exit the `man` page, these keys are no longer active for navigation until you invoke the `man` command again or until you enter `fg`, as we mentioned previously.

Printing man Information

If you want to print a hard copy of a `man` page for a particular command, type

```
$ man commandname | lpr<Enter>
```

Alternatively, if you are using a PostScript printer, type

```
$ man -t commandname | lpr<Enter>
```

Of course, using these options and pipelined commands requires that the printer be set up beforehand. If you intend to do so now, please flip to Chapter 5, "Using Files in Linux," where we discuss printing.

Example 3.4 man Command

Soon, we will see Sancho use the `who` command. But if he wants to know something about the `who` command beforehand (or at any time), he can access its `man` page as follows:

```
$ man who<Enter>
```

The first page of the system's response would resemble:

```
WHO(1                          FSF                          WHO(1)

NAME
          who - show who is logged on

SYNOPSIS

          who [ OPTION ] f [ FILE | ARG1 ARG2 ]

DESCRIPTION

          -H, --heading
              print line of column headings
```

```
            -i, -u, --idle
                 add user idle time as HOURS:MINUTES, . or old

            -l, --lookup
                 attempt to canonicalize hostnames via DNS

            -m, only hostname and user associated with stdin

            -q, --count
                 all login names and number of users logged on
(etc.)
```

The coding in the Synopsis section uses these typographical conventions.

Optional fields appear in square brackets, `[` and `]`. They mean that you need not enter an option or argument. Mandatory fields, if there were any, would appear in curly brackets: { and }. If there were mandatory fields, then you would have to enter a value for the respective option or argument after the command name. When you see a pipe character (|) separating two mandatory options or arguments, then you would have to enter one or the other with the command. In the case of `who`, though, the arguments—although separated by a pipe—are optional. So you would not have to supply anything unless you chose to do so.

You might have noticed the `FILE` option in the `who man` page. When users are logged in, a record of their login is kept in the file */var/run/utmp*. By default, the `who` command looks there for the information. If the system administrator has configured the system to use another file to record login information, then you would have to specify that filename after the `who` command.

info Pages

The `info` pages use the same database as `man` but present the information in a different format. Some say `info` is more powerful because it enables you to examine information in more bite-sized chunks and shows the relationships between the information you have searched for and other related information.

Another important feature of `info`, especially for new users, is that it presents its own instructions on the first screen and also enables you to enter a primer immediately after you invoke `info`. This hand-holding approach greatly enhances the likelihood of a successful search.

The syntax could not be simpler:

```
$ info<Enter>
```

After invoking the command, the first ASCII character-based `info` screen appears and declares that you are at the top of the `info` directory tree. Basic instructions follow as well as the opportunity to start the `info` primer.

If you do not require the primer, press the Tab key or the up and down arrow keys to move among the many available user commands, system calls, subroutines, devices, file formats, games, system administration functions, and more. Example 3.5 lists the basic categories but not all the functions you can access from this screen. When you have selected a category, press the Enter key. You are then presented with a screen containing information about your selection.

After a user exits from `info`, however, there is no command similar to `fg` (as we used with `man`) to return the user to the point where they left `info`. They have to re-enter at the beginning and go through the whole procedure again.

Users can also invoke `info` from `xterm` windows, and additional `info` or `man`-like programs can be invoked from the X Window system desktop. The names and types of these programs vary with each X window manager. From KDE, for example, you can invoke `KDE Help`. `KDE Help` enables you to navigate through hyperlinks with a mouse as opposed to the keyboard navigation we used to get through the ASCII-oriented `info` pages. See Chapter 14, "The Linux X Window System," for more information about the X Window System and X window managers.

NOTE **Check the `man` pages and other information sources for the options and arguments for all the commands mentioned in this chapter.**

Example 3.5 Obtaining Command Information with info

Let's say that Sancho wants to compare the `who` information provided by `man` to that provided by `info`. To invoke the `info` command, he types:

```
$ info<Enter>
```

Linux responds with the `info` directory tree, whose approximately 250 lines are structured like the following:

```
File:dir            Node:top       This is the top of the INFO tree

"Basic Instructions"

*Menu:

Texinfo documentation system
Miscellaneous (i.e., list of common commands)
Programming Languages
GNU Gettext Utilities
GNU Packages
Libraries
Individual Utilities
Net Utilities
World Wide Web
Printing Tools
C library code
Programming
GNU Emacs
GNU Admin
Programming & development tools
GNU libraries
Indent Code Formatter
GNU programming tools
```

We know that Sancho wants to know something about the `who` command. With the Tab key or the up or down arrow keys, he moves the cursor to `*who` under Miscellaneous and selects `who` by pressing Enter. Linux then jumps, or hyperlinks, to the `who` information:

```
'who': Print who is currently logged in
= = = = = = = = = = = = = = = = = = = = = = =
     'who' prints information about users who are currently logged on.
"Synopsis"
"Explanation"
"Options"
(etc.)
```

Sancho can navigate up and down through the information. He can press the following:

- The space bar to navigate down the file one screen at a time
- *h* to display help
- / followed by a string and Enter to search for the string in the text
- *n* to move to the next related topic or function (if any)

- *p* to move to a previous related topic or function
- *d* to return to the first screen (that is, back to the top of the `info` directory)
- *q* to exit info

The usage Utility

The `usage` utility can prove invaluable. It is based on the same database used by `man` and `info`. You invoke this facility in two ways. First, let's invoke it by requesting command usage help. The syntax is as follows:

```
$ commandname -help<Enter>
```

The second way to invoke `usage` is basically accidental. If you enter a command incorrectly, Linux responds with information on what is wrong with what you did, and then (by using `usage`) it tells you how to correct the command entry.

Example 3.6 illustrates the two ways `usage` gets invoked.

Example 3.6 Invoking the usage Utility

Sancho has always been something of a *sneakernet* proponent—that is, he still occasionally uses floppy disks. Naturally, he wants to know how to use the floppy disk drive on his Rueful Figures, Inc. system. Freston, the system administrator, has told him that he has to become familiar with the `mount` command. Sancho seems to recall a `-d` option that he could use with `mount`. Here, he will check by invoking `usage`. First, he deliberately requests information for the `mount` command:

```
$ mount - help<Enter>
Usage:     mount -V               : print version
     mount  -h               : print this help
     mount                   : list mounted filesystems
     mount -l                : idem, including volume labels
So far the informational part. Next the mounting.
The command is 'mount [ -t fstype] something somewhere'.
Details found in /etc/fstab may be omitted.
(etc.)
```

Sancho does not see a `-d` option, but he thinks there might be an oversight. So, he will take a chance and deliberately enter the `mount` command with a `-d` option to see how Linux responds:

```
$ mount -d /dev/fd0<Enter>
```

Here is Linux's response:

```
mount:invalid option -d
Usage:    mount -V               : print version
     mount  -h              : print this help
     mount                    list mounted filesystems
     mount -l               : idem, including volume labels
So far the informational part. Next the mounting.
The command is 'mount [ -t fstype] something somewhere'.
Details found in /etc/fstab may be omitted.
(etc.)
```

Convinced that there is no -d option after all, Sancho decides to pursue other `mount` options. Meanwhile, we will discuss the `mount` command in Chapter 4, "Files and Directories in Linux."

Viewing the Date: date and cal Commands

Now that we know 1) something about command syntax and 2) where to go for help with our commands; let's look at some common commands we use in everyday administration.

Freston's responsibilities as system administrator include setting the date globally for the Rueful Figures, Inc. network. Once he has done that, the users can then display the date on their own systems by entering the `date` command.

When you use the `date` command to display the date, you can customize it in several ways—but by default, it requires no options. (For more information on setting the date, consult the `man` pages for the `date` command.) The syntax is simply

```
$ date<Enter>
```

Linux responds with a time based on a 24-hour clock (for example, 1 P.M. is indicated as 13:00:00), followed by the date and the time zone (for example, CST for Central Standard Time).

The syntax for the `cal` (calendar) command is as follows:

```
$ cal [month] year<Enter>
```

When entering the month, use the number of the month (a value between 1 and 12; for example, January is 1), but *not* the name of the month or any type of abbreviation. If you do not specify a month, the system responds with a calendar for the current month. When entering the year, use the full four digits (for example, 2001) and not the two-digit abbreviation. Although you will not get an error message when you use two digits, you will likely receive erroneous results.

In Example 3.7, we show several ways to use the `date` and `cal` commands.

Example 3.7 date and cal

Sancho wants to check his appointment calendar. First, he checks today's date:

```
$ date<Enter>
Mon Jul 2 1:07:34 CST 2001
```

Now, he wants to have a look ahead at the whole month of July. If the current month is July, then he only needs to enter

```
$ cal<Enter>
```

If it is any other month, Sancho must enter

```
$ cal 07 2001<Enter>
```

The response will be:

```
      July 2001
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
```

Now, he decides to view the entire year:

```
$ cal 2001<Enter>
```

Linux responds with a three-month-wide by four-month-long listing of the monthly calendars in the specified year. But the text on Sancho's screen scrolled by so rapidly that he was unable to see the first three months, and now he cannot even see the names of months four through six. All he sees clearly now are the calendars for the last six months of the year. He calls Freston to ask for help. Freston gives him some keyboard tips and advises him to practice temporarily stopping, restarting, and terminating the scrolling of command output. Freston tells him to try the command a few more times but to use the following key combinations while the output is sent to the terminal screen:

- Ctrl-S will stop the scrolling until you deliberately restart it or terminate the output completely.
- Ctrl-Q will resume scrolling after you have stopped it.
- Ctrl-C terminates the current command.

Freston also tells Sancho that if he wanted to view the whole year, screenful by screenful, he should enter the following command at the beginning:

```
$ cal 2001 | more<Enter>
```

What does Freston mean by "screenful by screenful"? This last `cal` command contains the | character, which is called a pipe, followed by the `more` command. By using the `pipe` (which is a command unto itself and actually dates back to the first days of Unix) and adding `more`, we are telling the system to fill the screen with output from the `cal` command starting from January 1 to whatever the screen can hold and then to stop and wait for further instruction. He can then press Enter to advance one line at a time through the rest of the output or press the space bar to move a whole screen at a time. He can also use the Ctrl-Q and Ctrl-C combinations listed previously. When he sees a new command prompt at the bottom, he will know that he has reached the end of the output and that the system is ready for a new command.

The pipelining, or piping, of commands is discussed again in Chapter 7, "Shell Basics." Meanwhile, the `more` command—and a somewhat newer and more versatile variation called the `less` command—are discussed in Chapter 5.

Requesting Data on Logged-In Users

Occasionally, administrators, and even ordinary users, need to know who is logged in to the system. Linux/UNIX enables them to find out by providing several handy commands.

The basic purpose of the `who`, `who am i`, `whoami`, and `finger` commands is to obtain information about those logged in users. As you'll see, though, each provides its own "twist" on that information.

who, who am i, and whoami Commands

The output from the `who` command is simply a list of users who are currently logged in along with their stations and the time they most recently logged in.

The `-u` and `-m` options are often added to the `who` command. The response to `who -u` displays all usernames, their respective real names, workstation names, login times, and the identities of the processes they are running.

The response to `who -m` is identical to that of `who am i`, which appears in Example 3.9. For both commands, the system responds with the username of the person who entered the command, the name of the user's workstation, and the user's login time.

Example 3.8 who

A week after he was hired, Sancho wants to contact Don Quixote. He knows enough about the `who` command now to check to see whether the good Don is on the network this morning. He enters

```
$ who<Enter>
root       console     Jul      9     07:30
quixoted   tty2        Jul      9     08:10
perez      tty3        Jul      9     08:15
nicholas   tty4        Jul      9     08:18
dulcinea   tty5        Jul      9     08:10
panzasan   tty6        Jul      9     08:07
```

Example 3.8 shows that the root user (Freston) has been logged into the system console since 7:30 A.M. on July 9 and that several other users have logged into their respective terminals since then.

Sancho finds it interesting that Freston is logged in at the console and not at his customary terminal 1 (`tty1`; tty is actually a "hangover" term inherited from the old teletype communication days). Meanwhile, the console is generally defined as the device (typically a terminal as opposed to a full-fledged workstation) directly attached to the computer on which Linux is running. By default, the console is the device that receives system messages. Every computer on which a copy of Linux has been installed must have a console defined at all times, and there must be only one console for that computer.

Meanwhile, Sancho also sees that Don Quixote is apparently in his office and is connected to the network.

Example 3.9 who am i and whoami

Freston's job is not always as straightforward as other users'. At times, he has to switch identities to perform other functions. It can be confusing sometimes. Here is a way that he can check to see just who he is logged in as. He types the following:

```
$ who am i<Enter>
hostname!quixoted tty1 Jul 10 11:10
```

He could also have typed:

```
$ whoami<Enter>
quixoted
```

Either way, the system thinks that he is Don Quixote! Oh, that's right—he remembers that he had logged into this session earlier as `quixoted` to check something for the Don and had not changed back to his own identity. No wonder he has not had the functionality to which he thought he was entitled. Now, he has to decide whether to change back to his own identity or not. For the sake of convenience, he probably will.

finger Command

The `finger` command displays information about the users who are currently logged into the system, as shown in Example 3.10. The default response format is the full username, login time, user's home directory, and user's login shell. You can use the `finger` command to look up infor-

mation about users who are logged into a remote system, as well. You must know the correct name of the remote system, however.

Example 3.10 finger

One morning, Don Quixote intends to send a message to Sancho but needs to know whether Sancho is on the system. Here is how he uses `finger` to find out:

```
[username@hostname username]$ finger<Enter>
Login     Name     Tty     Idle     Login Time     Office     Office Phone
root     root     tty1     2        Jul 10 07:30
quixoted          tty      2        Jul 10 11:16
panzasan          tty6              Jul 10 08:30
```

Using `finger` without options gave him a list of all the currentlylogged-in users. Luckily, there were not too many. If he had wanted to just check for Sancho, he could have entered

```
$ finger panzasan<Enter>
Login: panzasan Name: (null)
Directory: /home/panzasan Shell: /bin/bash
On since Tue Jul 17 8:32 (CST) on tty6
No mail.
No Plan.
```

So, Don Quixote knows that Sancho is on the system. But what was that other information all about?

A user can create files called *.project* and *.plan* (please notice that the filenames must be preceded by a period, which we will call *dot* to indicate that they are normally hidden) to enhance the responses received when someone inquires about them with the `finger` command. The detailed process is beyond the scope of this book, but we will present a quick outline of it. Suppose that Nicholas the Barber (username nicholas) created hidden files called *.plan* and *.project* in his home directory by using a text editor such as `vi` (for details on `vi`, see Chapter 10, "The vi Editor"). When creating these files, he would have to remember to provide the `r` permission for others on those files as well as the `x` permission for others on the *nicholas* home directory. (File and directory permissions are discussed in detail in Chapter 6, "Linux File Permissions.") When users invoke the `finger` command with the `nicholas` username argument, they are provided with all the information Nicholas has written to those files.

Sending and Receiving Mail: mail Command

The `mail` command is interactive and is used to both send and receive mail messages.

To Send Mail

To send a message to another user on the same system, the format is

```
$ mail username<Enter>
```

To send a message to more than one user on the same system, enter all user names after the `mail` command, separating each by one space:

```
$ mail username1 username2<Enter>
```

To send a message to a user on another system, type the username followed by `@` and the name of the system:

```
$ mail username@hostname<Enter>
```

To send a message to more than one user on another system, separate each `username@hostname` entry by one space.

Example 3.11 shows how one user can send mail to another user on the same system (that is, on the same host) and to one on a different system. Notice that whenever the `mail` command is invoked, the system responds with a `Subject:` prompt. Type a description line that will appear in the receiver's list of incoming mail, and then press Enter again. Now you can type whatever you want to communicate to the receiver.

When you have finished your message, press Enter and then press the combination Ctrl-D. The system responds with a `Cc:` prompt. If you want to send copies of the message to others, type their usernames. When you are finished, press Ctrl-D again. The system sends the message and responds with a shell prompt (usually `$` and a cursor).

Example 3.11 Sending Mail with the mail Command

Knowing that Sancho is on the system today, Don Quixote sends a message to him:

```
$ mail panzasan<Enter>
Subject: Meeting at El Toboso<Enter>
Sancho! Don't forget the meeting tonight at Lady Dulcinea's!
signed, Quixote de La Mancha
<Ctrl>-d
Cc: <Enter>
```

What if our noble knight Quixote wishes to send a message to a user on another system? Knowing that Dulcinea is working at her home office in El Toboso, here's what he might enter:

```
$ mail dulcinea@eltoboso<Enter>
Subject: Meeting Still OK, My Lady?<Enter>
Is it still OK for your me (your devoted Knight-Errant), my squire, and
our comrades to meet with you tonight at your casa in El Toboso? We wish
to discuss our plans for future noble exploits.
I have already taken the liberty to inform Sancho, Nicholas the Barber
and Perez the Curate. I will inform the others after I receive your
blessing.
signed, Quixote de La Mancha
<Ctrl>-d
Cc: <Enter>
```

To Receive Mail

Whenever you log in, the system lets you know whether you have mail messages waiting in the *username* file (where `username` refers to your username) in the */var/spool/mail/* directory.

The `You have mail` notification and the mail messages themselves are not displayed simultaneously when a new message arrives. System administrators can customize the shell (in other words, the system) to check on all mailbox files periodically; for example, once every 600 seconds. If the shell detects a new message in a user's */var/spool/mail/username* file, it then displays the `You have mail` notification. System administrators can customize other aspects of this mail notification, as well. That said, to view new messages, the format is simply

```
$ mail<Enter>
```

The `mail` program responds with a listing of the new messages in the file. Note that once you invoke the `mail` program, it has its own specific prompt (`&`), which tells you that it is now running. Let's use an example to explore `mail` a little more.

Example 3.12 Receiving mail with mail

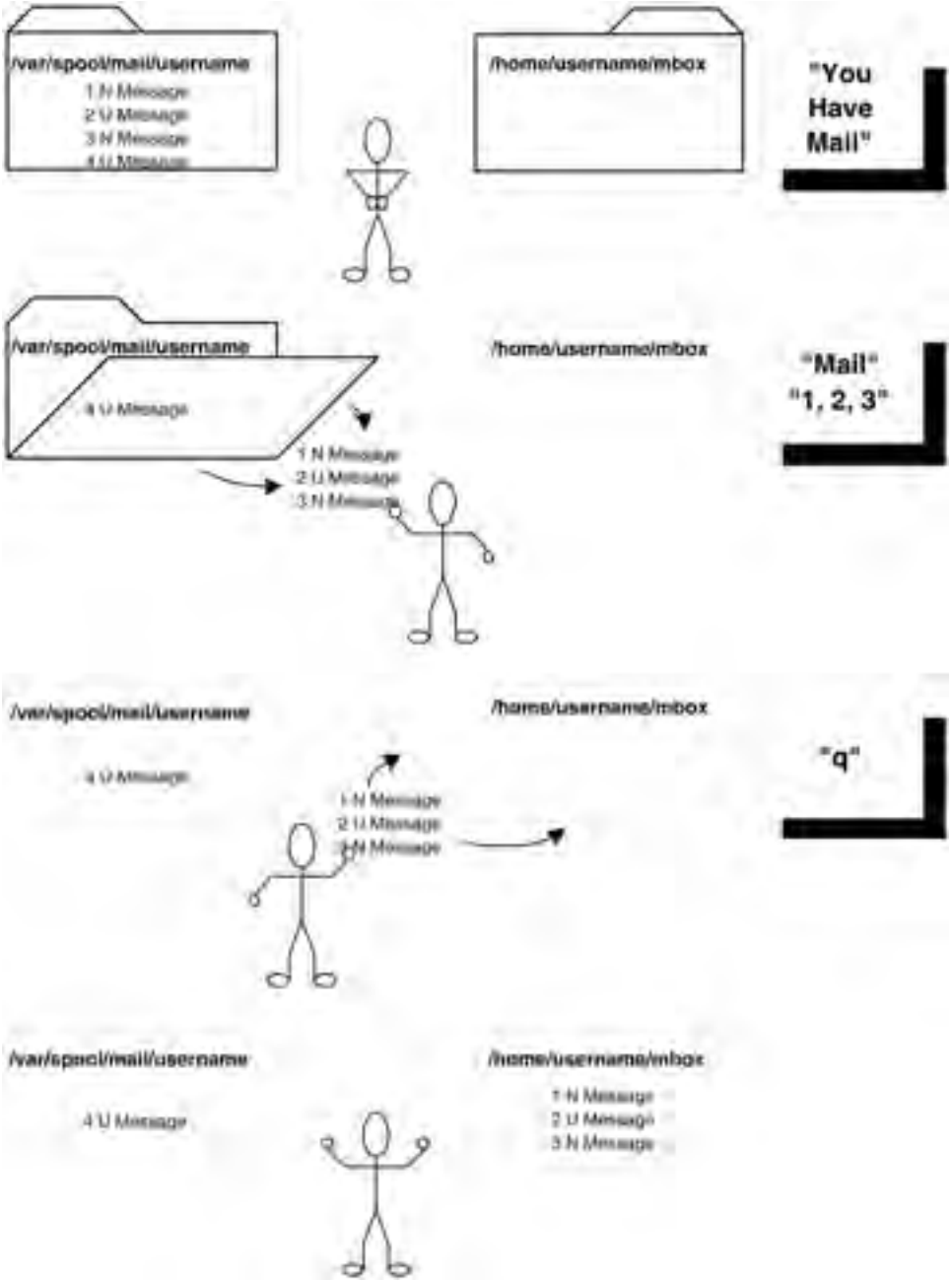


Figure 3.1 Receiving, reading and storing mail.

Sancho is working on `tty6` when he receives the following system message:

```
You have mail in /var/spool/mail/panzasan.
```

Note that here, the username file is *panzasan*.

Sancho's system is configured to check for mail every 10 minutes. It has apparently done so and has given him the new mail notification. To check the mail, Sancho enters

```
$ mail<Enter>
Mail version 8.1.6/6/93 Type ? for help
"/var/spool/mail/panzasan": 2 messages 1 new
U 1 quixote@hostname.local Tue Jul 17 10:50 10/267 "Meeting at El
Toboso"
N 2 nicholas@hostname.local Tue Jul 17 11:05 16/311 "Meeting Postponed?"
&
```

Note the `U` and `N` designations at the beginning of each listed message in Example 3.10. `U` means unread (a holdover message from a previous time when the user viewed his or her mail but did not read that particular message). The `N` designation means new.

When presented with the `&` prompt, Sancho knows that he can immediately check his unread messages simply by pressing Enter. He does so to read the first message, which he had not read the last time he checked his mail.

```
&<Enter>
Message 1:
From quixoted Tue Jul 17 10:15:47 2001
Date: Tue, Jul 17 2001 10:15:47 -600
To: panzasan@hostname.localdomain
Subject: Meeting at El Toboso<Enter>
Sancho! Don't forget the meeting tonight at Lady Dulcinea's!
signed, Quixote de La Mancha
```

Sancho then reads the new message 2 by typing:

```
& 2<Enter>
```

(Because he had already finished reading message 1, he could have just pressed Enter here, too.)

```
Message 2:
From nicholas Tue Jul 17 10:50:32 2001
```

```
Date: Tue, Jul 17 2001 10:50:32 -600
From: nicholas@hostname.localdomain
To: panzasan@hostname.localdomain
Subject: Meeting Postponed?
Sancho, Perez just dropped by my office. Please call us regarding the
status of the meeting Don Quixote scheduled for tonight at Lady
Dulcinea's. The Don told us yesterday that he thought the meeting was a
'go' but that he had not received confirmation from the Lady yet. Is it
still on? Meanwhile, are you keeping a close watch over him? We don't
need another 'windmill' episode.

Nicholas and Perez
```

Here are few subcommand options—also called subcommands because they invoke their own functionality—for reading mail:

- To read the first message in the listing, press Enter or type `1` and then press Enter.
- To skip the first message and read the second message, type `2` and press Enter.
- To read the first new message in any listing, type `t` and press Enter.
- To read the messages in sequence, simply press Enter after starting at your chosen first message. Then, press Enter after reading each message in turn.

You can also type the following letters (among others) at the `&` prompt while reviewing your messages:

`f`	lists the headlines of messages in the mailbox
`d`	deletes messages
`m`	forwards messages
`r`	sends a reply
`q`	exits `mail` and leaves unread messages in the queue
`s`	appends a message to a file

To obtain a full list of available subcommands to use while in `mail`, type `?` at the `&` prompt.

Now, what if Sancho wanted to check messages that he has received and read in the past? The `mail -f` command displays a list of messages in his personal mailbox (that is, in the */home/panzasan/mbox* file). Normally, when he quits the `mail` program, undeleted (but already read) messages are written back to that *mbox* file and not back to the (incoming) */var/spool/mail/panzasan* file from which all the new messages came. By

starting the `mail -f` aspect of `mail`, he can review and deal with these messages in the same way he dealt with the new or unread messages in */var/spool/mail/panzasan* by using `mail`. Figure 3.1 illustrates a typical mail reception and storage process.

Sending Messages to the Screen: write and wall Commands

Both of the commands discussed in this section provide the capability to contact other users on the system:

- `write` enables you to contact one or more specific users
- `wall` enables you to contact all users on the system

Let's discuss each in turn.

The write Command

The mail command showed us how to send messages to others' mailboxes and how to receive and manipulate messages in our mailbox. By contrast, the `write` command displays a message immediately on the specified user's screen. You can send messages to a user on the local system or on another system on the network (simply replace `username` with `username@hostname`). If, by using `who` or `finger`, you determine that your target user is logged in on more than one terminal, you can display your message on all the terminals unless you just want to specify the terminal(s) to which you want your message to be sent, as follows:

```
$ write username ttynumber<Enter>
```

Otherwise, to reach the target user on any and all terminals, the syntax is just the following:

```
$ write username<Enter>
```

For a specific user to receive the message, however, he or she must be logged in at the time and must not have refused permission for the message to appear on his or her terminal. By default, Linux does not turn off the `write` permission; that task is accomplished by the user with the `mesg` command (covered later in this chapter).

NOTE **You cannot turn off messages sent to you from the root user.**

In a `write` session, each user alternately sends and receives messages. Long messages can be placed in a file and directed or redirected to the other user (or users). The syntax is

```
$ write username < filename.ext<Enter>
```

Example 3.13 illustrates `write` being used by two users. The receiving user uses the same syntax to reply to the originating user.

Example 3.13 write

Don Quixote needs to inform Sancho that Lady Dulcinea (real name: Aldonza Lorenzo, but she has long ago given up trying to convince the noble old fellow that he is mistaken. Besides, the human resources coordinator has her correct name) has confirmed the meeting, so he takes a chance and interrupts Sancho by sending the following:

```
$ write panzasan tty6<Enter>
Sancho, Lady Dulcinea has confirmed that we can meet at El Toboso at 6!
BYO nachos y salsa!<Enter>
signed, Quixote de La Mancha<Enter>
<Ctrl>-d
```

Sancho receives the following on his screen immediately:

```
Message from quixoted@hostname on tty2 at 13:13
Sancho, Lady Dulcinea has confirmed that we can meet at El Toboso at 6!
BYO nachos y salsa!
signed, Quixote de La Mancha
EOF
```

To get back to their own prompts, the receivers of the messages must press Enter.

The wall Command

The `wall` command sends a common message to all users who are logged into the system. The syntax is

```
$ wall textofmessage<Enter>
```

By default, all users can use the `wall` command. Example 3.14 shows the use of the `wall` command by the root user.

Example 3.14 wall

Early Tuesday afternoon, Freston has to send the following emergency message (remember, the root user has the # prompt instead of $):

```
# wall Warning!! System Going Down in 15 Seconds! Please Log Off Now!!
Sorry, colleagues!<Enter>
```

And every logged-in user immediately receives the following message:

```
Broadcast message from root (tty1) Tue Jul 17
13:24:48 2000 . . .
Warning!! System Going Down in 15 Seconds! Please Log Off Now!! Sorry,
colleagues!
```

Again, to get back to their own prompts, the receivers of the message must press Enter.

Conversing Online: The talk Command

The `talk` command enables two users to hold a conversation. As shown in Example 3.14, one user invites the other to talk as follows:

```
$ talk username<Enter>
```

You can use the `talk` command locally on one system or across a network. To talk across a network, the syntax for both the invitation and the response is

```
$ talk username@hostname<Enter>
```

If the invitation is accepted, the screen on each terminal splits in two horizontally. The messages typed by the other user appear in the top window; replies are typed in the lower window. To close the connection, press Ctrl-C.

Example 3.15 talk

We already know that Don Quixote and Sancho Panza are both logged in. To talk to Quixote, Sancho enters

```
$ talk quixoted<Enter>
```

Don Quixote de La Mancha receives the following message immediately:

```
Message from TalkDaemon@hostname at 14:14 . . .
talk: connection requested by panzasan
talk: respond with: talk panzasan
```

To accept the invitation, Quixote follows the instruction and enters

```
$ talk panzasan<Enter>
```

What about those times when the Don (or better, *you*) do not want to be interrupted by messages or invitations to converse? Or what about when you want to resume receiving messages and conversations after having denied them for a while? The next section explains how to block and unblock messages.

Blocking Messages and Conversations: The mesg Command

You have seen how you can use the `write`, `wall`, and `talk` commands for communicating with users. But sometimes users should not be interrupted. For those times, they can use the `mesg` command, as illustrated in Example 3.16. The syntax is

```
$ mesg [y/n]<Enter>
```

Note that the system does not acknowledge the command. With `mesg` turned off (with `n`), others will not receive feedback from the system (with one exception: messages from the root user cannot be turned off). The `write` and `wall` command messages from the root users always reach all users on the system.

Example 3.16 mesg

If Don Quixote had not wanted to be interrupted, he could have entered

```
$ mesg n<Enter>
```

After he enters that command, whenever anyone enters the `write quixoted` command, they will immediately get a `write: quixoted has messages disabled` message.

If later our noble knight wants to make himself available to receive messages and conversations again, then he could type

```
$ mesg y<Enter>
```

By default, the system's shell startup process permits messaging. If a user has just entered `mesg n` during a login session and then logs out and logs in again, the system resets the username to `mesg y`. To prevent messages from coming through after exiting and logging in again, the user must reset `mesg n`.

Users can override the default `mesg y` behavior by including `mesg n` in their *$HOME/.bash_profile* file. In other words, they can set `mesg n` in the script that runs automatically when they log in. We will discuss the *$HOME/.bash_profile* file and other similar files in Chapter 11, "Shell Variables and the User Environment."

Additional Tools: clear, echo, banner, and wc Commands

From time to time, users and administrators have a need for additional tools to help them with routine duties. Here, we present four tools that are not related, are not revolutionary in scope or power, and are not needed every day. But they do help prevent confusion. Have a look at them, and for each one ask yourself: Has there ever been a time when I could have used this tool? Almost every time, you will likely answer "Yes."

The clear Command

If your screen is full of confusing commands and responses, incoming messages, and the like, you might want to use the `clear` command. This command appears to execute simply and easily. What could be simpler than erasing a bunch of now unnecessary characters and leaving you with only a prompt and a clear terminal screen?

There is more to `clear` than meets the eye, however. To determine how to clear the screen, the `clear` process first checks the `TERM` environment

specifications in RAM and then the */usr/share/lib/terminfo* directory, which contains the terminal definition files. If the `TERM` variable is not set or is set incorrectly, the command results in no action.

The echo Command

The `echo` command makes the terminal reiterate what you have just typed, as shown in Example 3.16. This procedure seems trivial when simply entered interactively on the screen, but the command can be valuable when included in shell scripts or similar files (for example, batch files). For instance, the `echo` command is helpful when you are writing a script file and you want to be notified when certain instructions are executing.

Example 3.17 echo

Freston spends the first couple of hours a day fighting user fires. After that, he tries to dedicate some time to automating some of his administration functions. Here, he practices with `echo` prior to using it in a configuration script:

```
$ echo Installing modem drivers now. . .<Enter>
Installing modem drivers now. . .
```

A few of the argument-like conventions that you can use with the `echo` command are `\b` to display a backspace character, `\n` to display a `newline` (where the Enter key has been pressed) character, and `\t` to display a Tab character. See the online `man` pages or other information sources for an exhaustive list.

The banner Command

The `banner` command displays ASCII character strings in large format on the screen or printed as hard copy. It constructs the characters out of # symbols and displays them from the top down (not from left to right). Like `echo`, this command might seem trivial when you play with it at your terminal, but it can prove invaluable in a large office or network environment when, for example, you want to identify individual print jobs from a shared printer. The syntax is as follows:

```
$ banner [-wn] ascii_text<Enter>
```

Some versions of Linux will not have the proper $PATH specified to easily use banner. In that case, they might get a message to the effect of command not found. In such cases, specify the whole path to the banner command as such:

```
$ /usr/games/banner [-wn] ascii_text<Enter>
```

The -w option adjusts the width of the output. The -n option is the specified character width of the output you want. (Your screen width is normally 80.) You cannot specify a width without also using the w option. If you want to display more than one word, you must put quotation marks ("") around the phrase. Otherwise, banner prints only the last word it was given, if anything at all. All these options are shown in Example 3.18.

Example 3.18 banner

Without options, banner prints extremely large. So, to display "Hello!" large enough to see:

```
$ banner -w40 Hello!<Enter>
```

To print a phrase and give yourself the ability to scroll up and down through the output:

```
$ banner -w40 "Hello Friends!" | less<Enter>
```

To send the output to a file:

```
$ banner -w40 "Print Job 1" > pjob1<Enter>
```

If the file does not exist yet, it is created during this process.

Perez the Curate wants to print a hard copy of an inspirational message that he will deliver at the El Toboso meeting. So that he can watch for it to be printed on the printer across the room, he uses banner to append a Print Job 1 label to the existing message file that he wants to print:

```
$ banner -w40 "Print Job 1" >> peptalk1<Enter>
```

Please note, though, that if the report file already exists, banner appends its output to the end of the file. If Perez had wanted the banner output at

the front of the file, then the output must be inserted before the rest of the material.

Now, you might ask, "But how do I print these documents so that I can see what `banner` did?" Again, document printing is discussed in Chapter 5, "Using Files in Linux."

The wc Command

When you need to know certain attributes of a certain file, one command you can use is the `wc` (word count) command. The basic syntax and a few command options are as follows:

```
$ wc [-l] [-w] [-c] filename.ext<Enter>
```

The `-l` option counts the number of lines; `-w` counts the number of words; and the `-c` option counts the number of characters (that is, the number of bytes). If you type:

```
$ wc filename<Enter>
```

you might get something resembling the following:

```
   17     126       1085        filename
(lines) (words) (characters) (name of file)
```

If you do not specify any options, you always get lines, words, characters, and the filename. You can shorten or lengthen the output by specifying options.

Example 3.19 wc

`wc` might be a helpful command for Perez to check the length of his inspirational message. Let's say that he feels he is capable of effectively inspiring Don Quixote and Sancho if he speaks at three words per second. But he knows that if his message goes beyond seven minutes, the Don and Sancho will become distracted and bored. He can check the length of *peptalk1* by entering the following:

```
$ wc -w peptalk1<Enter>
```

If the result is greater than approximately 1200 words, he might have to do some editing. As it is, he gets the following response:

```
1215 peptalk1
```

Hmm, what should he do? If he had not specified the -w option and had only typed

```
$ wc peptalk1<Enter>
```

he might have gotten

```
165 1215 9407 peptalk1
```

NOTE **Just a reminder: for further information on various commands, consult their respective online** `man`**,** `info`**, or other similar pages. Other information sources are discussed in Chapter 15, "Linux Documentation and Support."**

Exercises

1. Log in as the root user, using `password` as the root user's password.

```
hostname login: root<Enter>
Password: ********<Enter>
Last login: Day Mon No. hh:min:sec on ttyx
[ root@hostname /root ]# _
```

2. As the root user, create a user account called `teamxx` where the `xx` should be replaced by a two-digit number.

```
[ root@hostname /root ]# useradd teamxx<Enter>
```

3. Now, give `teamxx` the password `abc123` (later, as `teamxx`, you will change this password).

```
[ root@hostname /root ]# passwd teamxx<Enter>
Changing password for user teamxx
New UNIX password: abc123<Enter>
```

(You will get a message saying `BAD PASSWORD: it is based on a dictionary word`. Ignore it.)

```
Retype new UNIX password: abc123<Enter>
Passwd: all authentication tokens updated successfully.
```

4. Log out as the root user and log in again as `teamxx`.

```
[ root@hostname /root ]# logout<Enter>

hostname login: teamxx
Password: abc123<Enter>
[ teamxx@hostname teamxx ]$
```

5. Log into the system with the user name `teamxx`.

```
$ Login: teamxx<Enter>
Password: abc123<Enter>
```

6. Change the `teamxx` password.

```
$ passwd<Enter>
Changing password for "teamxx"
(current) UNIX password: abc123
New UNIX password:(specify a password)
Retype new UNIX password: (re-type the new password)
```

   If you are successful, you will get the following:

```
passwd:all authentication tokens updated successfully.
```

7. Verify that the new password has been set. Log out and then log back in with the new password.

```
$ exit<Enter>
login: teamxx
Password:
```

8. Access the `man` pages for the `man` command itself.

```
$ man man<Enter>
```

Read the text so that you can better understand the functionality of the `man` command. Which navigation key sequences would you use to do the following?

- Move down one screen at a time: space bar
- Move down one half screen at a time: *d*
- Move up one half screen at a time: *u*
- Move down one line at a time: down arrow key or Enter
- Move up one line at a time: up arrow key
- Search for occurrences of the text string `ormat`: `/ormat`
- Search for the next occurrences of the same text string: *n*
- Quit from the `man` page: *q*

9. Invoke `info` from the command line.

```
$ info<Enter>
```

Navigate downward in the first screen until you find the `date` command and then select it. What is the name of the file from which the `date` information has been extracted? What is the name of this node? What is the name of the next node?

10. Using `info` navigation keys, go to the next node. Press the *n* key. Is this node's name the same as what you anticipated from reading the `date` page in Exercise 9?
11. Press the *p* key to go back as far as you can to the most previous node in this file. What is the name of the most previous node?
12. Exit from the `info` pages by using the *q* key.

NOTE Exercise 13 is specific to the X Window System. If you are running KDE, go to Exercise 14. If you want to skip exercises on X Window and KDE variants of Linux, continue with Exercise 15.

13. Exercises for the X Window System:
    - At the command line, enter `startx` and then press Enter to invoke the X Window System. When X Window System starts,

you will probably be placed in a window titled `xterm`. If necessary, click to activate the window (that is, to turn the title bar to a bright color and to ready the window for input).

- Invoke the X Window version of `man` by typing the following in the window's command line:

```
$ xman &<Enter>
```

- A small window titled Manual Browser appears with three buttons: Help, Quit, and Manual Page.
- Click the Manual Page button. The Manual Page window appears, containing `xman` information.
- Click the Sections menu and while holding down the mouse button, move the cursor to the System Commands entry. Release the mouse button. Another manual page appears with an alphabetical listing of system commands.
- Scroll down, if necessary, and click the `date` command. Scroll up and down the new window to read about `date`. When you are finished, click Sections or Options and continue your investigation of `xman`.
- When you are ready to return to the original `xterm` window, exit `xman` by clicking the X button (the close button) in the upper-right corner of the window frame.

14. Exercises for KDE (the K Desktop Environment):
    - Invoke KDE, and using its start button or equivalent, find and invoke `KDE Help`, which is the KDE online help browser. A Welcome window with three major categories appears.
    - Under the KDE Help Contents category, click the System `man` page contents link. The Online Manuals window appears.
    - Click Section 1—User Commands. The Online Manuals—Section 1 window appears.
    - Scroll up and down by clicking the mouse in the right scroll bar until you see the `date` command. Click to select it. The `man` page for `date` appears.
    - Practice navigating by clicking the various blue titles with underlines or by clicking the menus on the gray menu bar. For example, to go back to the original KDE Welcome window, click GoTo and

then select Contents. When you are finished, click the File menu and select Close or Quit.

- To leave KDE altogether, use the KDE Start button and select Logout. You are returned to the Linux command-line prompt.

15. At the command line, invoke the help option for `mount`. How does Linux respond? Pay special attention to the options that Linux says can be used with `mount`.
16. Try to invoke `mount` but deliberately use an incorrect option. How does Linux respond?
17. Display the system's date.

```
$ date<Enter>
```

18. Display a count of the number of lines in the */etc/passwd* file.

```
$ wc -l /etc/passwd<Enter>
```

19. Display the entire calendar for the year 2002.

```
$ cal 2002<Enter>
```

Now, try displaying it with the addition of `| more`.

```
$ cal 2002 | more<Enter>
```

20. Display the month of September for the year 1752.

```
$ cal 9 1752<Enter>
```

Do you notice anything peculiar about September?

21. Display the month of August for the year 1999 and then for the year 99.

```
$ cal 8 1999<Enter>
$ cal 8 99<Enter>
```

Are 1999 and 99 the same?

22. Display a list of users who are currently logged into your system. Check to see when they logged in.

```
$ who<Enter>
```

or

```
$ finger<Enter>
```

NOTE **If you use the `finger` command and see `???` in the Name field, optional user information was not added to the user profile (that is, in the *`/etc/passwd`* file) when it was created.**

23. Display only your login name.

```
$ whoami<Enter>
```

24. Use `banner` to display `Out to Lunch`.

```
$ banner -w40 "Out to Lunch"<Enter>
```

(Did `banner` work by itself? If not, try `/usr/games/banner`. If that does not work, use the `find banner` command to determine where `banner` is for your version/distribution of Linux.)

25. Remember to use Ctrl-C or Ctrl-Q to get back to your user prompt after displaying the `banner` message.

```
$ q<Enter>
```

26. Use the `echo` command to write the character string `Out to Lunch` to your display.

```
$ echo Out to Lunch<Enter>
```

27. Use the `clear` command to clear your screen.

```
$ clear<Enter>
```

NOTE **If you are using an ASCII terminal and the `clear` command does not work, check to see that the `TERM` variable is correctly set.**

28. Send a note to yourself by using the `mail` command. Provide a subject but ignore the `Cc:` (carbon copy) prompt.

```
$ mail teamxx<Enter>
Subject: A Reminder to Myself<Enter>
The meeting starts at 6:00 p.m. at El Toboso.<Enter>
<Ctrl>-d
Cc: <Enter>
```

29. Start the `mail` process and list the messages in your mailbox. Read your message, save it, and quit the mail program (your message might not arrive right away because the daemon has to check for it and then deliver it to you; just wait a few minutes).

```
$ mail<Enter>
& h<Enter>
& 1<Enter>
& q<Enter>
```

30. Access your mail, list your messages, and delete the message you saved in your personal mailbox (here, we presume that it is message number 1). Use the `h` subcommand again to ensure that the message has been deleted. Exit the mail program.

```
$ mail -f<Enter>
& h<Enter>
& d1<Enter>
& h<Enter>
& q<Enter>
```

31. If you have time and there are other users, then practice sending mail to someone who is logged into your system. Otherwise, log out from the system.

```
$ <Ctrl>-d
```

See Appendix B for answers to questions presented during these exercises.

Quiz

1. In how many ways can you invoke the `usage` facility? What are they?
2. Which of the following commands or options uses a different database than the other three?

```
locate
info
man
Help/usage
```

3. What combination of commands and options do you use from the command line when you want to print `man` information but you do not have a PostScript printer?
4. Match the environments with the appropriate information source command:

K Desktop Environment	`xman`
ASCII/command-line info	`info`
Fvwm95"Help"	`"Help"`

5. Which of the following illustrates the correct Linux syntax for the `mail` command?

```
$ mail newmail -f
$ mail f newmail
$ -f mail
$ mail -f newmail
```

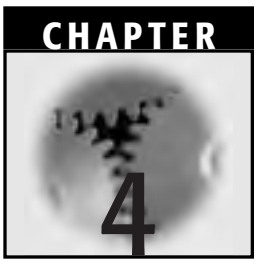
6. What command would you use to send a mail message to `username`?
7. List three commands that you can use to communicate with logged-in users.
8. What output would you expect from the following command:

```
$ cal 8
```

9. Which of the following commands would you use to determine when a particular user logged in?

```
$ who am I
$ who
$ finger everyone
$ finger username
```

See Appendix C for answers.



Files and Directories in Linux

In Chapter 3, "Getting Started Using the Linux System," we discussed a few basic Linux user commands. In this chapter, we begin to look at some basic Linux administrative functions—those dealing primarily with the directories on your system. We will start by defining files (but only to contrast them with the definition of a directory). We address files and file manipulation in more detail in Chapter 5, "Using Files in Linux." Meanwhile, other topics in this chapter include the structure and hierarchy of Linux file and directory systems, navigating around the directory structure, managing directories, and using floppy disks (in ext2 and DOS formats). If you are a UNIX veteran, you might consider these topics elementary or almost trivial. Try to take time to review these concepts and tasks, however, because they are fundamental to creating and maintaining a proper Linux file and directory system.

File System Structure and Hierarchy

The Linux operating system does not impose any internal structure on the contents of a file, nor are any specific attributes required. Only the application or tool is concerned with a file's structure and contents.

File Types

In Linux, everything—including the devices attached to the system—is represented as a file. First, though, what is a file? The answer is as follows: a file is 1) a collection of data or 2) a stream of characters (that is, a byte stream). A typical file can contain either text or code data. Text files are readable by a user and can be displayed or printed. Code data, also known as a *binary file*, is readable by the computer and might even be executable.

Linux does not impose any internal structure on the content of a file (we will allude to this fact again when we discuss the creation and use of floppy disks). The user is free to structure and interpret the contents of a file in whatever way he or she believes is appropriate for his or her needs. You might notice that this philosophy and technology is unlike "that other major operating system" that is so prevalent in our homes and workplaces. Thus, in Linux, only the application/utility/tool cares about the structure and contents of a file; the operating system does not require the file to have any specific attributes. Meanwhile, all information about a file, except for its contents, is stored in the file's *inode* (index node), which we will discuss a little further in the next section.

Linux can recognize the following file types:

Ordinary files. Contain either text or code data. Text files are readable by users and can be displayed or printed. Code data, also known as binary files, are readable by the computer. Binary files can be executable.

Directories. Contain information that the system needs in order to access all types of files but do not contain the actual data. Each directory entry represents either a file or a subdirectory. Directories and subdirectories constitute a method of storing files in some type of logical order (such as alphabetical or numerical).

Special files. Usually represent devices used by the system. An application of special files appears in the *Path Names* section later in this chapter.

Directory Contents

Although it is customary to refer to a directory as a type of envelope that contains entire subdirectories, files, and their contents, in truth a directory is a unique type of file that is used to organize other files into a hierarchical structure. Thus, it contains only the information that is needed to access the files or other directories that are affiliated with it according to (hopefully, generally) some sort of logical order. As a result, a directory occupies less space than other types of files.

A directory resembles a table of contents. It lists the names of files and subdirectories and their corresponding inode numbers. When users execute a command to access a file, they use the filename. The system consults the directory to match the filename with its corresponding inode number and then accesses the inode table, which holds information about the file's characteristics (including its location). Then, once the system knows the location of the file, the data can be located. See Figure 4.1 for a partial listing of contents of the *quixoted* directory, which is shown in its entirety in Figure 4.2, as well as an illustration of the corresponding inode table entries.

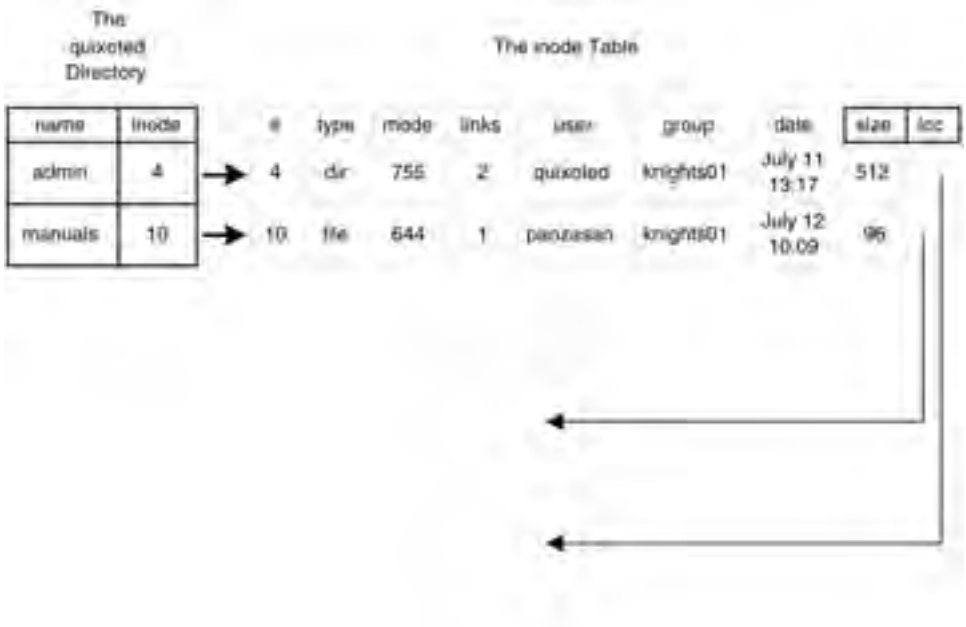


Figure 4.1 Directory contents.

The following categories of information are stored in the inode table:

- The file type (directory or file)
- The mode (directory or file permissions; see Chapter 6, "Linux File Permissions," for details)
- Links (which enable you to refer to a file by more than one name; for more information about linking, see Chapter 5)
- The userid of the file's owner
- Group permissions
- The date the file was last accessed and modified
- The file size
- The file's location

Remember that the filename is stored in a directory, *not* in the inode table. You might ask, "Why not store the inode information in the directory and dispense with the inode table?" Our answer is as follows: restricting information in the directory to filenames and inode numbers simplifies directory management and allows for the efficient use of disk space.

Hierarchical Structure

The file structure depicted in Figure 4.2, called a *directory tree*, represents only part of a typical Linux file system. In this depiction, directory names appear in boxes and filenames are unboxed words.

The top of the structure is the root (/) directory. The root contains many directories that are critical in system operations. Root subdirectories depicted in Figure 4.2 are described in Table 4.1.

You can also access files on other computers on the network. The details of that process are beyond the scope of this book. For now, just remember that from a user's perspective, the network is configured such that remote files appear to behave just like local files. The */home*, */usr*, */tmp*, and */var* directories are examples of directories whose files can be accessed by local and remote systems.

Path Names

The purpose of a path name is to tell you the location of a file. You write a path name as a string of names separated by forward slashes (/). The rightmost name is the filename and can represent any type of file; the other

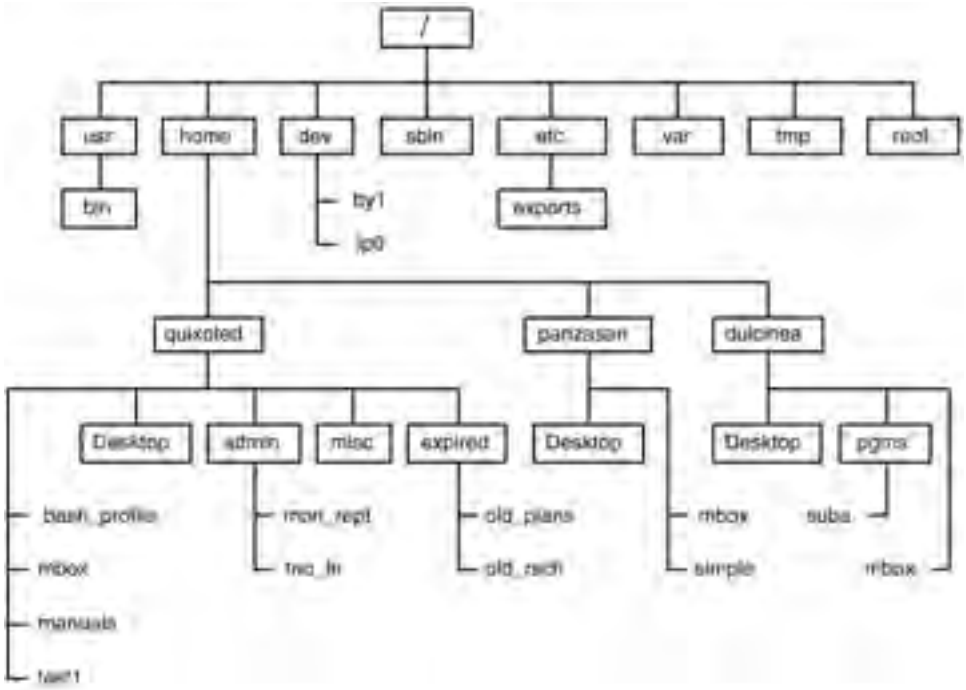


Figure 4.2 Linux directory tree.

Table 4.1 Root Subdirectories

SUBDIRECTORY	DESCRIPTION
`/usr`	System programs such as /usr/bin, which in turn contain user commands such as `ls`, `cat`, and `date`. Because the */usr* directory generally contains system-related rather than user-related commands and utilities (despite its name), users do not have write access to this directory. In addition, because the files in this directory are system related, the directory is not considered very dynamic.
`/dev`	Special files that represent devices
`/home`	User login directories and files. When a user is added to the system, he or she is allocated an individual subdirectory, the name of which is the same as the user's login name. When users log in, they are put in their own directory and they can do whatever they want with it, such as create subdirectories within it and create, delete, move, copy, and rename files. Because users generally change their files regularly, the */home* directory is considered quite dynamic.
`/sbin`	System utilities for system startup

continues

Table 4.1 continued

SUBDIRECTORY	DESCRIPTION
`/etc`	System configuration files used by system administrators
`/var`	An abbreviation of variable, the *var* directory contains files that change with system activity. For example, this directory is typically used by or for many user-oriented programs, such as mail or printing.
`/tmp`	Holds files that are temporarily needed or created by applications and programs. For example, the `vi` editor (discussed in detail in Chapter 10, "The vi Editor") uses the */tmp* directory as a buffer space until the file being worked on is written to disk. Compiler programs use */tmp* to hold files written during compilation. When the compiler is finished, the files created there are eliminated.
`/root`	The top of the directory structure belonging to the root user

names must be directories. A full path name, which is also referred to as an *absolute* path name, always begins with a forward slash (/) to indicate that it begins at the root directory. Path names that do not begin with a forward slash are termed *relative*.

The path names in Example 4.1 refer to the files depicted in Figure 4.2. In the full path name, the first forward slash represents the root directory. In the relative path names, the current directory is presumed.

Example 4.1 Absolute and Relative Path Names

Let's look at the full or absolute path name for Don Quixote's weekly report, typically submitted to Lady Dulcinea on Monday morning:

```
/home/quixoted/admin/mon_rept
```

Now, let's look at the relative path names of other files in the Rueful Figures, Inc. directory structure. Let's presume that our current directory is Don Quixote's home directory, */home/quixoted*:

```
admin/mon_rept
./test1
../dulcinea/pgms/suba
```

NOTE **Notice that this example contains two special files (one begins with a single dot; the other begins with a double dot). Both types were mentioned previously.**

In *./test1*, the dot (.) represents the current directory; in other words, it says "start in this directory and look for the file called *test1*." In the last example, *../dulcinea/pgms/suba*, the leading two dots represent the parent directory to the current directory—in effect, saying "go back up one level to the home directory, look in Lady Dulcinea's home directory (*dulcinea*) for a directory called *pgms*, and then look there for a file called *suba*."

When you are trying to find a file, particularly in a complex, multi-layered structure, the single dot (.) and the double dot (. .) special files can save time by eliminating the need to describe the location of the file in relation to the root directory. They can be helpful also when programming an application if you do not know the absolute tree structure where the application might eventually be installed, but you do have a good idea where files are with respect to one another or with respect to an executable.

Navigating the Directory Structure

Now that we know something about how directories are structured in Linux/UNIX, let's see how we can move around in the structure. Prior to knowing how to get around, though, it's best to know where you are to begin with. So the first command we'll show you is `pwd`. Then, to enable you to move around, we'll introduce `cd`.

Locate the Working Directory Path: The pwd Command

The `pwd` (print working directory) command is a Linux/UNIX command for finding out which directory you are in; that is, where you are in the directory tree. This command always returns the full or absolute path name of the current working directory, as shown in Example 4.2. Without `pwd`, you would have no way of knowing which directory you are in. (Such information can be added to the command prompt, which we will discuss later.)

Example 4.2 Determining the Current Directory with pwd

Don Quixote is looking for a file he created a while back. The file is called *trio_ltr*, and it is a memo to Sancho, Nicholas, and Perez. He logs in and checks to ensure that the directory he is in is indeed his home directory:

```
$ pwd<Enter>
/home/quixoted
```

Navigating Directories: The cd Command

The `cd` (change directory) command enables you to navigate the directory structure. As with any commands that operate on directories, you can specify the relative or full path name. With relative path names, however, you *must* be certain of the directory in which you are working. If you are not sure, use the `pwd` command. Here is the syntax:

```
$ cd [directoryname]<Enter>
```

Using the `cd` command with no arguments automatically returns you to your home directory—the directory you were automatically placed in when you logged into the system. This function can be very handy or very confusing.

Example 4.3 Navigating Directories with cd

The Don is still searching for *trio_ltr*. He thinks he might have left it in his *doc* directory. So, he decides to navigate from his (current) home directory, */home/quixoted*, to the */home/quixoted/doc* directory. Here is how he performs this task the easy way from where he is, using only the relative path name:

```
$ cd admin<Enter>
```

To do it the more complex way, using the full path name, he could have entered

```
$ cd /home/quixoted/doc<Enter>
```

Now, from his *doc* directory, to go back to his home directory all he needs to do is enter

```
$ cd<Enter>
```

In fact, no matter where he is, if he wants to go back to his home directory he only needs to enter that command.

Also, no matter where he is, if he wants to navigate up one level to the parent directory of his current working directory, he only needs to enter

```
$ cd ..<Enter>
```

Please note, though, that the only time the `cd ..` command would fail him is if he did not have permission to access that directory. We will discuss file and directory permissions later in Chapter 6.

Managing Directories

An understanding of directory structure and the ability to navigate that structure provide a foundation for the most important aspects of directory management: the creation and deletion of directories and the examination of their contents. These activities are among the most important and basic of an administrator's duties. Without these essential skills, users and administrators would be unable to organize their applications, data, and information. Perhaps even more important: Without these skills, users and administrators might never find the data and information they've already created.

Creating Directories: The mkdir Command

The `mkdir` (make directory) command creates new directories and names them according to the specified directory name. You can specify multiple directory names as long as you separate each by a space.

The syntax for the command is as follows:

```
$ mkdir [-m] directoryname(s)
```

Each new directory or subdirectory automatically contains the standard entries: dot (`.`) and dot-dot (`..`). Use `-m` as an option before the chosen directory name to specify which permissions to set for the new directory when it is created. Ordinary users can create directories where they have

write permission. (File and directory permissions are discussed in more detail in Chapter 6.)

NOTE **Unlike DOS, Linux does not enable you to abbreviate the `mkdir` command to `md`.**

Example 4.4 Creating Directories with mkdir

Don Quixote wants to create a subdirectory to contain proposals, plans, and reports for noble endeavors. Suppose that he has navigated to the root directory. He can use a one-step procedure, like the following, to create a directory called *noble* as a subdirectory of his */home/quixoted* directory:

```
$ mkdir /home/quixoted/noble<Enter>
```

Or, he can use a two-step process such as

```
$ cd /home/quixoted<Enter>
$ mkdir noble<Enter>
```

Deleting Directories: The rmdir Command

Removing a directory involves two steps. First, empty the directory of its contents. A directory is considered empty when it contains only the dot and double-dot entries. Second, remove the directory. The syntax is simple:

```
$ rmdir directoryname<Enter>
```

Although you can be in the target directory to remove its contents, you cannot be in the target directory when you intend to remove the directory itself. Two conditions must be met: The target directory must not be the working directory, and the target directory must be empty. Refer to Example 4.5.

Example 4.5 Deleting Directories with rmdir and rm

Refer to Figure 4.2. If the Don wants to remove a directory without going to the effort of emptying it first, he can enter the following:

```
$ cd /home/quixoted<Enter>
$ rm -r expired<Enter>
```

If he wants to remove an empty subdirectory while he is in his home directory (*/home/quixoted*):

```
$ rmdir misc<Enter>
```

(Actually, the Don will not remove these directories and their contents until later, after Example 4.11. But, when the time comes, these are the commands that he will use.)

NOTE Linux does not display a screen message when the `rmdir` command is successful, so it is a good idea to make sure that the command worked as expected. To do so, execute the `ls` command on the parent directory. (See the *Listing Directory Contents* section, coming soon in this chapter.)

Creating or Removing Multiple Directories Simultaneously

You can create two types of subdirectories. *Horizontal* subdirectories share the same parent directory. *Vertical* (or recursive) subdirectories are subdirectories of subdirectories of subdirectories of . . . well, you get the idea. Figure 4.3 shows both types of subdirectories.

To create a horizontal subdirectory, you issue the `mkdir` command and list the subdirectories that you wish to create with a space between each name. If you were creating some generic subdirectories named *dira*, *dirb*, and dirc, you would enter the following:

```
$ mkdir dira dirb dirc<Enter>
```

To create vertical sudirectories, add the `-p` (path or parent) option to `mkdir`. For example, once you have created *dira*, you could type the following to create some generic vertical subdirectories named *dird* and *dire* under *dira*:

```
$ mkdir -p dira/dird/dire<Enter>
```

Example 4.6 Creating Horizontal and Vertical Subdirectories with mkdir

Recruiting Lady Dulcinea (nee Aldonza Lorenzo) was probably the best thing Rueful Figures, Inc. ever did, although accepting RFI's offer was pretty much against her better judgment. At this stage, as her duties expand, the

Lady needs to create some extra directories within and beneath her home directory (*/home/dulcinea*). To see where she is starting from, please refer to Figure 4.2.

First, she wants to create horizontal directories for:

- The reports submitted to her periodically
- Company mission statements, objectives, and other policies
- Special correspondences and other documents (such as the odes, songs, poems, drawings, and plans that Don Quixote seems to generate endlessly)

So, she navigates to her home directory and then enters the `mkdir` command:

```
$ cd /home/dulcinea<Enter>
$ mkdir repts objs spec<Enter>
```

Dulcinea has decided that the Don and Sancho will report to her separately every week, but she acknowledges that Sancho was recruited by the Don and is still under the Don's supervision (although everyone suspects that it might unofficially be the other way around). So, she wants to create subdirectories for each of their reports. Thus, within (or beneath, if you prefer) the new *repts* directory, she will create vertical subdirectories for them, too. Here is how she performs this task:

```
$ mkdir -p repts/don/sancho<Enter>
```

Have a look at Figure 4.3 to see the results of Lady Dulcinea's work so far. For the sake of simplicity, we are only showing Lady Dulcinea's portion of the directory structure.

Example 4.7 rmdir to Delete Vertical Directories

Lady Dulcinea knows that if she ever has to remove the *repts* subdirectory and the subdirectories beneath it, she would enter

```
$ rmdir -p repts/don/sancho<Enter>
```

In this example, the last specified directory, *sancho*, would be removed first, followed by *don* and then *repts*. Freston reminds her, however, that if for some reason now or in the future she does not have write permission to one or more of the specified directories, or if one or more of the directories are not empty, the `rmdir` command would terminate (prematurely) at that

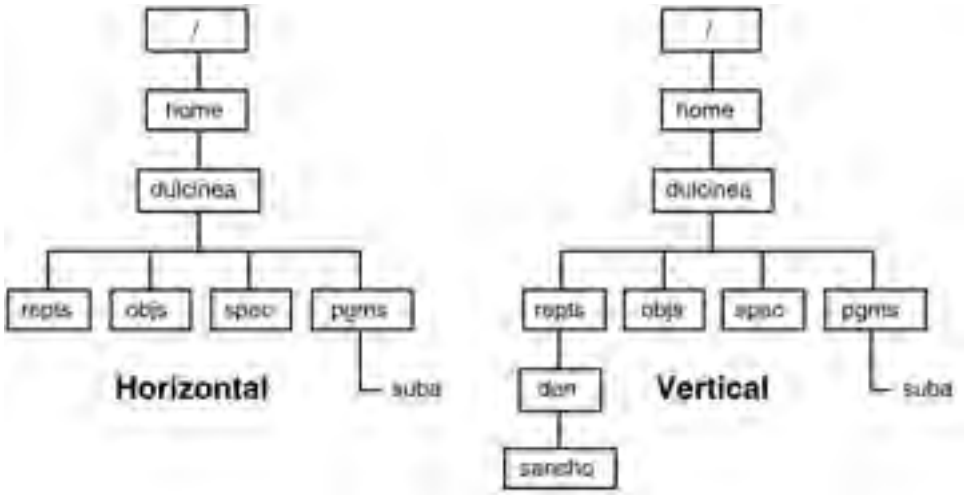


Figure 4.3 Creating horizontal and vertical subdirectories simultaneously.

point. For that reason, it is a good idea for her to check the status of her directories (with various combinations of the `cd` and `ls` commands) after the `rmdir` command has finished executing.

Listing Directory Contents: The ls Command

The `ls` (list contents) command displays the contents of one or more directories. This command has several useful options, some of which will be illustrated in action in Examples 4.8, 4.9, 4.10, and 4.11. First, though, here is the basic syntax:

```
$ ls [-a][-R][-l][directoryname]<Enter>
```

If you do not specify a directory, the contents of the current working directory will be displayed. The `-a` option displays all hidden files. (For more information about hidden files, see the section *Linux File Name Guidelines* in Chapter 5.) The `-R` option displays all subdirectories and their respective contents, except the hidden files, right down to the bottom of the directory tree. Combining the `-a` and `-R` options results in a listing of subdirectories, files, and hidden files to the bottom of the directory tree.

By default, the information that the `ls` command returns is sorted in alphabetical order, and no distinction is made between files and directories.

Example 4.8 Displaying Directory Contents with ls, ls -a, and ls -R

To list the contents of his home directory, Don Quixote enters the following commands. To see where he is starting from, please refer to Figure 4.2 and remember that the Don created the subdirectory *noble* (which contains the file named *windmills*) in Exercise 4.5, but he has not yet deleted the expired subdirectory and its contents.

The first `pwd` command, to determine where he is to begin with, is a sort of best practice:

```
$ pwd<Enter>
/home/quixoted
```

Comfortable that he is in his home directory, the Don then enters the second command:

```
$ ls<Enter>
admin Desktop expired manuals mbox misc noble test1
```

Color-coding on his terminal screen tells the Don which entries are files (white printing) and which are directories (blue). He knows, however, that there is at least one hidden document in his *quixoted* subdirectory. To relist the contents of the same directory (but this time, to include all hidden files), he re-enters

```
$ ls -a<Enter>
. .. .bash_profile admin Desktop expired manuals mbox misc noble test1
```

To list all contents to the end of his portion of the directory tree (that is, to list contents recursively), he enters

```
$ ls -R<Enter>
admin Desktop expired manuals mbox misc noble test1
admin:
mon_rept trio_ltr
Desktop:
expired:
old_plans
old_rsch
```

```
misc:
noble:
windmills
```

The `ls` command with the `-l` (long listing; also called a detailed listing) option displays file and directory information from the inode table, as we will show in Example 4.9. It also displays the names of the files or directories.

Example 4.9 Displaying Detailed Directory Contents with ls -l (long listings)

In Example 4.8, Don Quixote listed the contents of his home directory and its subdirectories. Although the lists were apparently adequate, there is still some information he needs about each directory and file. He relists the contents again, but this time he asks for a detailed description of each entry by using the `ls` command with its `-l` option:

```
$ ls -l<Enter>
total 8
drwxrwxr-x 2 quixoted knights1 4096 May 26 10:18 admin
drwxr-xr-x 2 quixoted knights1 4096 Mar 07 08:11 Desktop
drwxrwxr-x 2 quixoted knights1 4096 May 17 09:21 expired
-rw-rw-r-- 1 panzasan knights1  144 Jul 11 11:42 manuals
-rw------- 1 quixoted knights1 1116 May 26 10:18 mbox
drwxrwxr-x 2 quixoted knights1 4096 Apr 05 14:30 misc
drwxrwxr-x 2 quixoted knights1 4096 Jul 11 15:21 noble
-rw-rw-r-- 2 quixoted knights1  257 Mar 17 13:29 test1
```

The `total` amount refers to the number of 4096-byte blocks allocated to the files in the directory.

Table 4.2 presents the detailed information for two of the entities listed in Example 4.9: the *misc* directory and the *test1* file. Let's look at each field in a little more detail:

Field 1 represents the file, directory, and permission bits. File and directory permissions are covered in Chapter 6. For now, we will just

Table 4.2 Output from the `ls -l` Command

FIELD 1	FIELD 2	FIELD 3	FIELD 4	FIELD 5	FIELD 6	FIELD 7	FIELD 8
drwxrwxr-x	2	quixoted	knights1	4096	Apr 05	14:30	misc
-rw-rw-r--	2	quixoted	knights1	257	Mar 17	13:29	test1

point out that a `d` in the first position indicates that the entity is a directory. If there is a hyphen (`-`) in the first position, that indicates a file.

Field 2 is the link count. For directories, the link count indicates the number of subdirectories in that directory. Directories always have a link count of at least two (for `.` and `..`). The link count for files indicates the number of names given to a single file. When you enter the command `rm` *filename*, for instance, you reduce the link count for a file by 1. A file is removed only when its link count reaches 0. For a discussion of linking and removing files, see Chapter 5.

Field 3 is the username of the person who owns the entry (the file or directory). The owner is generally the person who created the entry, but ownership can be transferred.

Field 4 is the name of the group for which group protection privileges are in effect. Groups are generally created to organize users based on job description, organizational structure, project assignment, and so on, and their privileges determine the files and directories to which the group has access. Every user belongs to at least one group. The user in Example 4.9 is assigned to the group knights01. We touch briefly upon group concepts in Chapter 6 when we discuss file and directory permissions.

Field 5 is the character count of the entry. Note that directory space is allocated in 4096-byte increments. Thus, just looking at the output here will not necessarily give you a good idea of the number of files or subdirectories in the directory in question. Note that you do not make the directory allocation size smaller when you delete files. To decrease the size of a directory, you must move files to newer, smaller directories.

Fields 6 and 7 indicate the date and time that the contents of the file or directory were last modified. Other `ls` command options display other time attributes (for example, `-u` displays the last access time to the file and `-c` displays the time the inode was modified).

Field 8 is the name of the entry, whether a file or a directory.

Example 4.10 Long Listings, Including inode Numbers, with ls -li

The Don will use the `-li` options with the `ls` command to display file information, including the inode number. But first, please notice that he is

confining his inquiry to the *test1* file *only* by specifying its filename as an argument to the `ls` command:

```
$ ls -li test1<Enter>
187018 -rw-rw-r-- 1 quixoted knights01 257 Mar 17 13:29 test1
```

Although typically, ordinary users are not usually interested in inode numbers, the functions of various file manipulation commands (such as `mv` and `cp`) are better understood if inode tables and the information in them are understood first.

Displaying Directory Information: The ls and stat Commands

The `ls` (list) and `stat` (statistics) commands can be used to display information about files and directories, because Linux treats directories as files for these purposes. In Example 4.11, the `ls` command displays inode table information. As shown in the example, the options `-l`, `-d`, and `-i` result in a long listing with inode information and the inode number in the first displayed field.

Example 4.11 Listing Directory Contents with ls

To get a long listing with inode information and the inode number in the first displayed field, Don Quixote enters

```
$ ls -ldi expired<Enter>
109096 drwxr-xr-x 5 quixoted knights01 4096 May 18 15:22 expired
```

Linux systems maintain three timestamps for files and directories. The following are additional options you can use with `ls` to display each of these timestamps:

- `ls -lu` displays the access time.
- `ls -l` displays the modification time.
- `ls -lc` displays the updated time.

The `stat` command displays inode information for a specified file or directory. The syntax is as follows:

```
$ stat directoryname<Enter>
```

To explain the results of the command, let's provide an example.

Example 4.12 Inode and Timestamp Information with stat

To get inode information for a specific (namely *expired*) subdirectory in his home directory, Don Quixote enters

```
$ stat expired<Enter>
File: "expired"
Size: 4096          Blocks: 8              Directory
Access: (0775/drwxrwxr-x)   Uid: ( 501/ quixoted)  Gid: ( 607/ knights01)
Device: 306  Inode: 109096  Links: 5
Access: Wed Jul 12 18:13:21 2001
Modify: Mon Jul 10 13:22:17 2001
Change: Mon Jul 10 13:22:17 2001
```

In the `stat` results shown in Example 4.12, let's look at the `Access`, `Modify`, and `Change` lines:

- `Access` means that the file has been read or written to and tells us the time and date when that last happened. When a file is read but no changes are made to it, the access time is changed but not the modification or change times.
- `Modify` indicates that the contents of the file or directory have been changed and indicates the time and date when that last happened.
- `Change` tells us that the inode information has been changed and indicates the time and date when that last happened.

At this point, as we mentioned previously in Example 4.5, Don Quixote deletes the *expired* and *misc* directories and their contents. Figure 4.4 illustrates what Don Quixote's portion of the RFI directory now looks like.

Formatting and Accessing Floppy Disks

Many users, especially new or ex-DOS and ex-Windows users, do not know that they can use floppy disks with Linux/UNIX. Although many of us work with UNIX-type systems on large networks and have abandoned `sneaker netting` to varying degrees, most if not all users still require it from time to time. In this section, we will create a Linux-like `ext2` file system on one floppy and a DOS file system on another floppy.

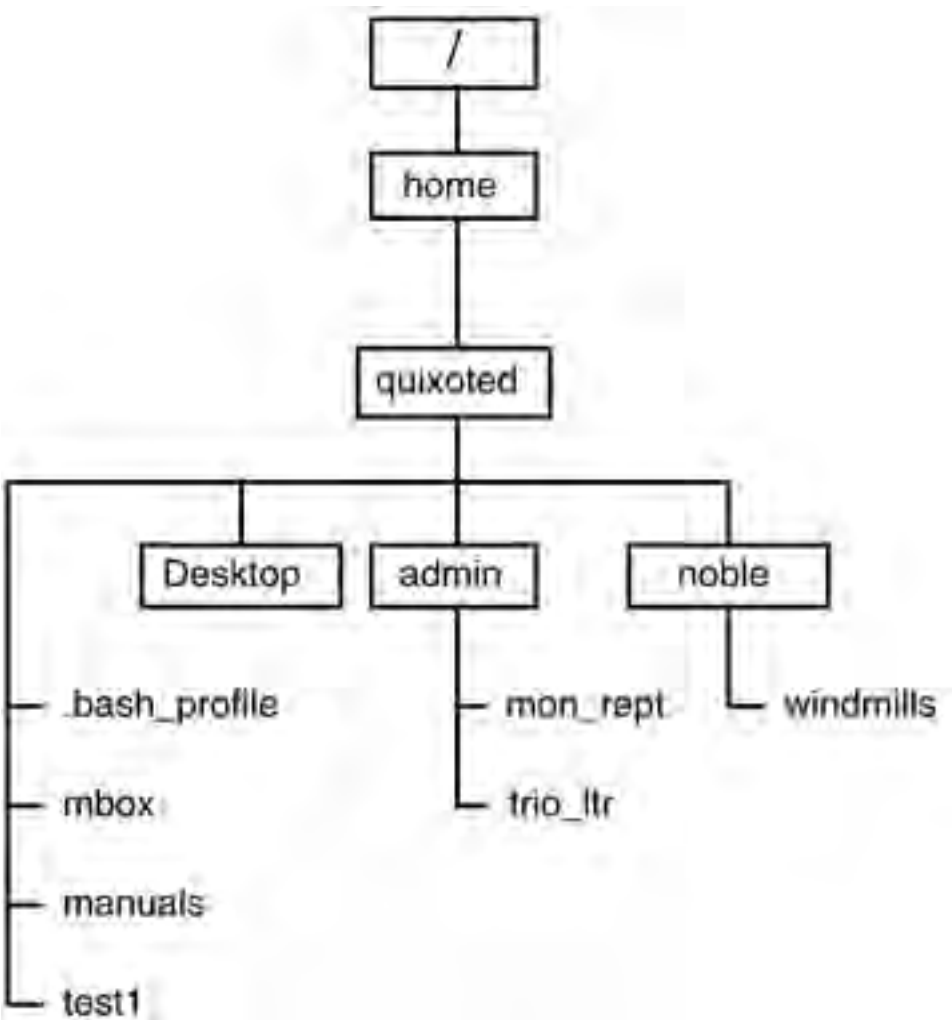


Figure 4.4 Don Quixote's portion of the RFI directory structure.

Floppies with ext2 File Systems

You can create and manipulate files on floppies in as many ways as the half-dozen types of file systems that Linux supports. The `ext2` file system procedure described in this section should be used on floppies moving from one Linux-like system to another.

We will start at the beginning and format a floppy disk. Insert the floppy disk into your floppy disk drive. The following procedure could be done

as a root user or as an ordinary user. Let's do it as an ordinary user. So, with the floppy disk in the floppy drive, type

```
$ fdformat /dev/fd0H1440<Enter>
```

You had to specify that the floppy disk was in device `/dev/fd0`; that is, in the first floppy disk drive. (If it had been in a second floppy drive, you would have had to specify `/dev/fd1`.) You enter the `H1440` (with no space between this specification and the `/dev/fd0` device name) to tell the system that `/dev/fd0` contains a 3.5-inch high-density floppy disk with a capacity of 1.44MB. If your floppy disk is of another type, refer to the `fdformat man` page for the appropriate description.

As the system executes the `fdformat` command, it responds with

```
Double-sided, 80 tracks, 18 sec/track. Total capacity 1440 kB.
Formatting ... (counts to 80) done
Verifying ... (also counts to 80) done
```

Now that the floppy is formatted, the next step is to create a file system on it. Linux supports a half a dozen or so types of file systems that you can create on a floppy. In this first procedure, we will create an `ext2` file system to be compatible with the `ext2` file systems created on the hard disk during the Linux installation. Continuing as an ordinary user, type

```
$ /sbin/mkfs -t ext2 /dev/fd0<Enter>
```

Ordinary users should use `/sbin/mkfs` as the command because the directory */sbin* is probably not in their `$PATH`. If you had performed this task as the root user, you would only have had to enter `mkfs` as the command because */sbin* is already in root's `$PATH` environment variable.

The `-t` option indicates that the user is about to tell the system what kind of file system to install on the floppy disk. The option is followed by the argument `ext2`, which is the default Linux file system type. After you enter the command, the system executes it and responds with

```
mke2fs 1.19, 13-Jul-2000 for EXT2 FS 0.5b, 95/08/09
Linux ext2 filesystem format
Filesystem label=
OS Type: Linux
Block size = 1024 (log=8)
Fragment size = 1024 (log=8)
```

```
184 inodes, 1440 blocks
72 blocks (5.00%) reserved for the super user
First data block=1
1 block group
8192 blocks per group, 8192 fragments per group
184 inodes per group
Writing inode tables: done
Writing superblocks and filesystem accounting information: done
```

Simply put, the system has told you that it has finished formatting the floppy disk and is now awaiting further instruction. (For an explanation of the output, which is beyond the scope of this book, refer to one of the sources of information mentioned in Chapter 15, "Linux Documentation and Support.")

The next step is to `mount` the file system, which means that the system must be told to make the newly defined file system (the newly formatted floppy disk, in this case) part of its tree-like file hierarchy. Enter the next command as the root user, because ordinary users, when they try it, will be disappointed. All they will get is the `mount: only root can do that` error message. Here, then, is the command:

```
# mount -t ext2 /dev/fd0 /mnt/floppy<Enter>
```

NOTE **It is best and easiest to perform this task (mounting file systems) as a root or superuser. For security reasons, an ordinary user is not allowed to mount or unmount a file system. An ordinary user can get around this restriction, however, by using `mtools`, which we will discuss in Chapter 5.**

The `mount` command tells the system to make the `ext2` file system, found on `/dev/fd0`, part of the total file system and to refer to the newly added file system as */mnt/floppy* (that is, a subdirectory called */floppy* in a subdirectory called */mnt*, all part of the root directory). You could call it other names, but this name is the Linux convention. If you have entered everything correctly, the system will do as you command and respond with another root prompt.

To view what has occurred thus far, enter the following line:

```
# ls -la /mnt/floppy<Enter>
```

The system responds with

```
total 14
drwxr-xr-x 3 username   username   1024      Apr 28 16:36 .
drwxr-xr-x 4 root       root       1024      Jan 31 11:31 ..
drwxr-xr-x 2 root       root       12288     Apr 28 16:36 lost+found
```

The system has already named the largest subdirectory on the floppy *lost+found*. You can change its name if you want. Also, notice that ownership of the single-dot special file has been given to username because it was as an ordinary user that we began this process. If we had done the whole procedure as the root user, then ownership of the single-dot file would have been root root.

You are now ready to use the floppy disk as part of your file system, using Linux commands.

When you are through using the floppy disk and wish to remove it, please remember that you cannot just "pop" the floppy disk out right away. First, you have to unmount the floppy disk's file system (again, as the root user) from the total file system. To do that, type

```
# umount /mnt/floppy<Enter>
```

or

```
# umount /dev/fd0<Enter>
```

Note that the command is `umount`, *not* "unmount." Again, you have to be a root user to unmount a file system just as you have to be a root user to mount it in the first place. In addition, you cannot unmount from the floppy diskette's own file directories; you have to `cd` out of them. Otherwise, you will get a `device busy` error message.

Now, you can remove the floppy. When you want to use it again, you must re-enter the `mount` statement to get Linux to recognize it again. But you do not have to edit the */etc/fstab* file, because those specifications remain.

DOS File Systems

In the previous section, we formatted a floppy disk to be compatible with a Linux-like `ext2` file system. To sneaker net between a Linux system and a DOS file system, however, you use commands that are similar but different than those discussed in the preceding section. We will start again with formatting a floppy disk.

Again, as an ordinary user, type the following:

```
$ fdformat /dev/fd0H1440<Enter>
```

Again, you had to specify that the floppy disk is in the `/dev/fd0` device (that is, your first floppy disk drive). You told the system `H1440` (with no space between this specification and the `/dev/fd0` device name), which means that `/dev/fd0` contains a 3.5-inch high-density diskette with a capacity of 1.44MB. The system executes the command, and as it does so, it responds with the following:

```
Double-sided, 80 tracks, 18 sec/track. Total capacity 1440 kB.
Formatting ... (counts to 80) done
Verifying ... (also counts to 80) done
```

Once the floppy disk is formatted, the next step is to create an MS-DOS file system on it and specify that the floppy disk is in `/dev/fd0`:

```
$ /sbin/mkdosfs /dev/fd0<Enter>
```

Again, we have shown you how to perform this task as an ordinary user. Thus, the command was `/sbin/mkdosfs`. If you had been the root user, the command would simply have been `mkdosfs`. The system responds with

```
/sbin/mkfsdos 2.2 (06 Jul 1999)
```

The next step is to mount the file system; that is, to make the newly defined file system (the just-formatted floppy disk, in this case) part of the overall system file hierarchy. Now, you can use the `mount` command by using one of several choices. For example, you can use

```
# mount -t msdos /dev/fd0 /mnt/floppy<Enter>
```

to tell the system to mount the `fd0` device as part of the file system and refer to it as the */mnt/floppy* directory (that is, a subdirectory called */floppy* under a subdirectory called */mnt*, which itself is under the root directory).

NOTE **Once again, you must mount file systems as the root user. For security reasons, an ordinary user is not allowed to mount or unmount a file system. An ordinary user can get around this restriction, however, by using `mtools`, which we will discuss in Chapter 5.**

You also told the system that the format of the files in the directory will be DOS but not the type of DOS that supports long filenames. If that is not suitable—for example, if you want to mount the type of DOS files developed under Microsoft Windows *9x*, operating systems that do support long filenames—use the following:

```
# mount -t vfat /dev/fd0 /mnt/floppy<Enter>
```

If you have entered everything correctly, the system will do as you command and simply respond with another root prompt.

To check on what has occurred thus far, type the following:

```
# ls -la /mnt/floppy<Enter>
```

Notice that you could enter the command as the root or as an ordinary user.

If the floppy disk does not contain any files yet, the system responds with

```
total 8
drwxr-xr-x 2 root root 7168 Dec 31 1969 .
drwxr-xr-x 4 root root 1024 Jan 31 11:31 ..
```

If the floppy contains files, you get the same response along with a listing of those files. Whenever you mount a DOS diskette in this manner, we recommend that you include either the `-t msdos` option or the `-t vfat` option because it is more likely that `ext2` might already be specified in */etc/fstab* if a floppy diskette has ever been used with that system. Specifying `msdos` or `vfat` on the command line here overrides any specification that is already in */etc/fstab*.

Now, you are ready to use the floppy disk as part of your file system by using Linux commands. As with the previous procedure, when you are finished with the floppy and wish to remove it, you cannot simply pop it out. First, you must, as the root user, unmount the floppy disk's file system from the total file system. Enter the following:

```
# umount /mnt/floppy<Enter>
```

or

```
# umount /dev/fd0<Enter>
```

Again, you must be the root user to unmount a file system just as you have to be a root user to mount it in the first place. In addition, you cannot be in

the floppy diskette's own file system when you try to `umount`; if you are, you will get that frustrating `device busy` error message. So, before you `umount`, `cd` to a directory level above that of the floppy disk.

Finally, now you can remove the floppy. But remember that when you want to use it again, you must re-enter the `mount` statement to get Linux to recognize its file system again.

Exercises

1. Using the `pwd` command, verify that you are in your home directory (the directory you are placed in when you first log into the system):

```
$ pwd<Enter>
```

2. Change your current directory to the root directory:

```
$ cd /<Enter>
```

3. Verify that you are in the root directory:

```
$ pwd<Enter>
```

4. Request a simple listing of the files and subdirectories in that directory:

```
$ ls<Enter>
```

5. Then, request a long listing of the files and subdirectories in that directory:

```
$ ls -l<Enter>
```

6. Issue the `ls` command with the `-a` and `-R` options:

```
$ ls -a<Enter>
$ ls -R<Enter>
```

What is the effect of each option?

NOTE **The `-R` option results in extensive output. After you have seen enough, press Ctrl-C to end command execution. Or, you can practice with Ctrl-S and Ctrl-Q to interrupt and restart execution.**

7. Return to your home directory and list its contents, including hidden files:

```
$ cd<Enter>
$ ls -a<Enter>
```

8. Create a directory in your home directory called *mydir*. Then, display a long listing of both *home/directoryname/mydir* and the parent directory of *mydir*. What are the sizes of each directory?
9. Change to the */home/mydirectory/mydir* directory. Use the `touch` command to create two zero-length files called *myfile1* and *myfile2* in your *mydir* directory.

```
$ cd mydir<Enter>
$ touch myfile1<Enter>
$ touch myfile2<Enter>
```

10. Display a long listing of the contents of your `mydir` directory. What are the sizes of `myfile1` and `myfile2`?

```
$ ls -l<Enter>
```

11. View the long listing again, but this time display the inode numbers, too. What are the inode numbers of each file?

```
$ ls -li<Enter>
```

12. Change back to your home directory and issue the `ls -R` command to view your directory tree.

```
$ cd<Enter>
$ ls -R<Enter>
```

13. Use the `stat` command to view the inode information in your *mydir* directory. Why might the "Last Accessed" date be more current than the other two dates?

```
$ stat mydir<Enter>
```

14. Use the `rmdir` command to remove the *mydir* directory. Does it work?

```
$ rmdir mydir<Enter>
```

NOTE **To remove a nonempty directory, you must use the `rm -r` command, not the `rmdir` command. For more on `rm -r`, refer to Chapter 5.**

```
$ rm -r mydir<Enter>
```

See Appendix B for answers.

Quiz

1. Using the directory tree structure shown in Figure 4.2 and using */home* as your current directory, how would you refer to the *suba* file in Lady Dulcinea's */pgms* directory using both full and relative path names?
2. When specifying a path name, what is the difference between using double dots (`..`) and a single dot (`.`)?
3. What will the following command do?

```
$ cd ../..<Enter>
```

4. What conditions must be satisfied for the `rmdir` command to complete successfully?
5. Match the following `ls` command options with their functions.

COMMAND OPTION	FUNCTION
`-a`	Provides a long listing of files
`-i`	Lists hidden files
`-d`	Lists subdirectories and their contents
`-l`	Displays the inode number
`-R`	Displays information about a directory

6. Referring to Figure 4.2, assume that your current working directory is */home/quixoted* and that you entered the following command:

```
$ mkdir test<Enter>
```

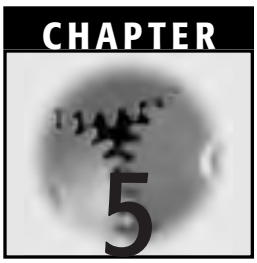
What happens? Why?

7. Referring again to Figure 4.2, assume that you are in the directory called */home/panzasan*. What is the difference in the results after issuing the following commands?

```
$ mkdir dir1/dir2/dir3
$ mkdir dir1 dir2 dir3
```

See Appendix C for answers.

CHAPTER



5

Using Files in Linux

In Chapter 4, "Files and Directories in Linux," we defined files and directories and then navigated and manipulated directories. In this chapter, we will describe several more aspects of files and file manipulation, including naming files, copying files, moving files, and referencing a file by more than one name. We will also show you how to look at the contents of a file page by page, rather than as a fast-scrolling display on your screen. The chapter closes with that all-important file manipulation function: printing files.

Files and Directories: A Quick Review

As stated in Chapter 4, a file is a collection of data or a stream of characters. A typical file can contain either text (which can displayed or printed) or code data (more commonly called a binary file, which can also be executable). In Linux, everything is represented as a file—even system devices. Linux does not impose any internal structure on the contents of a file, nor does it require the file to have any specific attributes. Only the application or tool is concerned with a file's structure and contents.

Linux can recognize three major file types: ordinary files, directories, and special files. All information about a file, except its contents, is stored in the file's *inode* (index node).

Directories and subdirectories are designed so that we can store files in some type of logical order. Directories contain information that the system needs to access all types of files, but directories do not contain the actual data. A directory resembles a table of contents, listing filenames and subdirectories and their corresponding inode numbers. When users execute a command to access a file, they use the filename. The system consults the directory to match the filename with its corresponding inode number. It then accesses the inode table, which holds information about the file's characteristics (including its location). Once it learns the file's location, it can locate the data. Meanwhile, special files usually represent devices used by the system.

Linux Filename Guidelines

The following are some guidelines and rules for creating filenames in Linux:

Describe the file's content. A filename should indicate what the file contains. Make sure you do not create confusion when putting similar but not identical files in different directories, however.

Use only alphanumeric characters and selected symbols. You can use all letters in the alphabet as well as numerals in filenames. Letters can be uppercase, lowercase, or a combination. In addition, the following symbols can be used: number sign (#), at sign (@), underscore (_), plus sign (+), and hyphen (-).

Do not use embedded spaces. Spaces should not be used within a filename because they might interfere with subsequent command execution, resulting in syntax errors. You *are* allowed to create a file with spaces if you enclose its name in single or double quotation marks (that is, `"filename"` or `'filename'`). We strongly suggest that you not do this, however; think of the syntax you would have to use to execute commands on such filenames.

Do not use shell metacharacters. The following characters are not allowed because of their use in command execution, system calls, and so forth: asterisk (*), question mark (?), greater-than and less-than signs (> or <), forward slash and backslash (/ and \ ), semicolon (;), ampersand (&), exclamation point (!), open and closed brackets ([and]), vertical pipe (|), single and double

quotation marks (‘ and “), and open or closed parentheses, (and). If you inadvertently include one of these characters, the shell’s interpretation will be unreliable and inconsistent.

Do not begin with a plus sign or a hyphen. You can use the plus sign (+) and hyphen (-) within a filename, but do not use them to begin the filename.

Do not use command names. You should not name a file using a command name unless you are creating an executable program file. The inadvertent use of a command might wreak havoc in a file system.

Other filename characteristics you should keep in mind are as follows:

Filenames are case sensitive. Although potentially confusing, this requirement can result in increased flexibility.

Filenames have a maximum length of 255 characters. Unlike some UNIX-based systems, Linux does not restrict filenames to 14 characters. We recommend, however, that you keep filenames to a reasonable length, such as 16 characters. Bear in mind that some applications work with only 8-dot-3 filenames (that is, xxxxxxxx.xxx, or eight-character filenames and three-character extensions), such as DOS filenames. To ensure compatibility with other environments—if that is a concern—consider restricting filename length and format.

Some applications append their own extensions or suffixes (such as `.tmp` or `.sam`) to denote a specific file type, so your filename conventions might have to accommodate this practice.

If the filename begins with a dot (.), the filename will be hidden from the standard `ls` commands. Linux/UNIX allows dots as legitimate filename characters as long as they are wholly contained within the filename. The system does not presume anything about the file based on the location of the dot.

Some applications that run on Linux/UNIX, however, do not recognize the dot as part of the filename. Therefore, they might react unpredictably when faced with such names.

Creating and Updating Files: The touch Command

We generally know how to create files. We can use all sorts of applications (such as text processors, database applications, graphics applications, and

so on). But UNIX and Linux have another interesting tool to create files: the `touch` command. We will see Lady Dulcinea use `touch` in Example 5.1. Here is the syntax:

```
$ touch [-c]filename<Enter>
```

Basically the command has two purposes:

- If the filename specified with `touch` does not yet exist, `touch` will create a *zero-length* (that is, empty) file. If you do not want the creation to take place, use the `-c` option.
- You can also specify a directory name with `touch`. If the file or directory specified does exist, the last modification date and time (as displayed with the `ls -l` command) is updated to reflect the current date and time unless you specify a preferred time variable.

The `touch` command can be helpful when you are about to invoke an application against one or more files, and that application checks the files' modification times before taking some action (such as backup or restore) with the files. By using `touch` alone or with its options, you can alter the dates on certain files so that they will or will not be affected by the application.

Example 5.1 Creating and Altering Files with touch

Lady Dulcinea is operating in her home directory. She wants to create a zero-length file called *nobledeeds* in her *objs* subdirectory. She also wants to change the access date on the existing *suba* file in the *pgms* subdirectory. Please refer to Figure 5.1, which depicts her portion of the RFI directory structure as it is now and as she proposes to change it.
First, she checks the existing situation:

```
$ pwd<Enter>
/home/dulcinea
$ cd pgms
$ ls -l<Enter>
-rwxrwxr-x 1 dulcinea       knights3 320 Jun 14 07:30 suba
```

Now, as a reference, she checks the time and date:

```
$ date<Enter>
Wed July 18 16:53:46 CST 2001
```

She proceeds to make her change to *suba* and then checks her results:

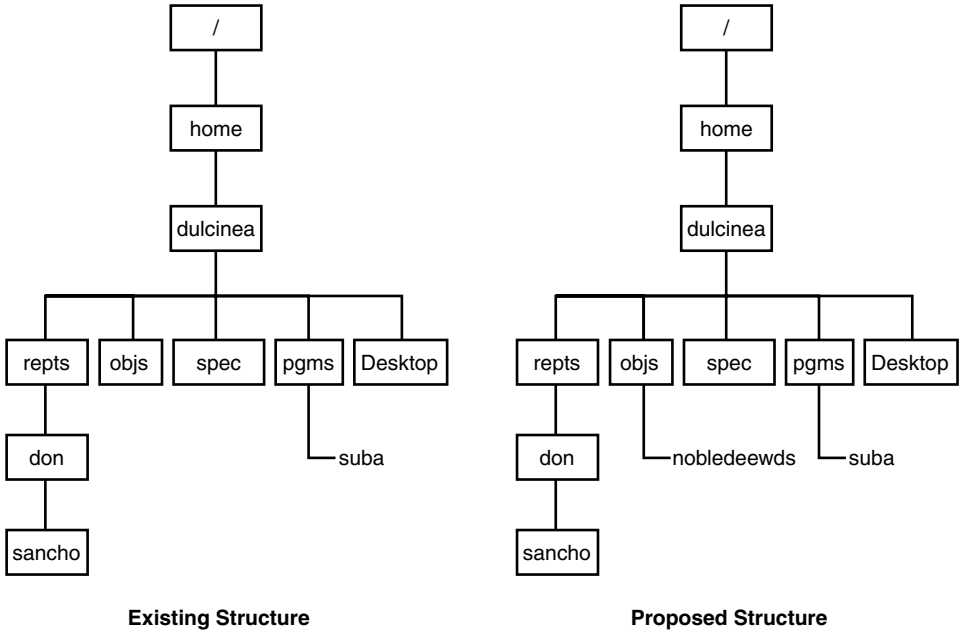


Figure 5.1 Lady Dulcinea's portion of the RFI directory structure.

```
$ touch suba<Enter>
$ ls -l<Enter>
-rwxrwxr-x 1 dulcinea       knights3 320 Jul 18 16:54 suba
```

She then proceeds to create *nobledeeds* under the *objs* subdirectory. Again, she checks her results:

```
$ touch /home/dulcinea/objs/nobledeeds<Enter>
$ cd ../objs
$ pwd
$ ls -l<Enter>
-rw-r--r-- 1 dulcinea       knights3 0 Jul 18 16:55 nobledeeds
```

Linking Files: The ln Command

We briefly mentioned file linking early in Chapter 4, "Files and Directories in Linux," and later in Example 4.12. We create such links with the `ln` command. The `ln` (link) command, in its simplest form, enables one file to have at least two different names in the directory structure. In other words,

one copy of a file is referenced by multiple names. This type of link is called a *hard link*. The owner of the file, the file permissions, and the inode number remain the same for both copies. The syntax is as follows:

```
$ ln sourcefile targetfile<Enter>
```

It is important to remember that in order to create links, the proper permissions must be set on the respective directories: where the source file resides and where the target link will be created. In older versions of Linux/UNIX, it was much easier to create links because it was likely that the default permissions allowed it without too much bother. Because of security concerns, however, the latest versions of Linux do not set default permissions so that links can easily be created without proper permissions. Please refer to Chapter 6, "Linux File Permissions," for a discussion of permission bits.

Once we have gone through the following example, you might want to refer to Chapter 4 to see how we dealt with links when managing directories.

Example 5.2 Linking Files with ln

Sancho, at Don Quixote's request, created a file called *manuals* (which includes a list of mandatory reading materials for those who wish to become chivalrous knights) and filed it in Don Quixote's home directory, */home/quixoted*. Now, he would like to refer to it as *knightdata* in his home directory, */home/panzasan*. Please refer to Figure 5.2, which depicts Sancho's portion of the RFI directory structure as it is now and as he proposes to change it.

Here is how he creates the hard link in his home directory:

```
$ pwd<Enter>
/home/panzasan
$ cd /home/quixoted<Enter>
$ ln /home/quixoted/manuals knightdata<Enter>
$ ls -l knightdata<Enter>
-rw-r--r-- 1 panzasan       knights1 144 Jul 11 11:42 knightdata
```

You can create another type of link, the *symbolic link*, by using the `-s` option with the `ln` command. Symbolic links are often used to allow two or more different directories in two or more different file systems to point to the same files. (Symbolic links are not discussed in this book.)

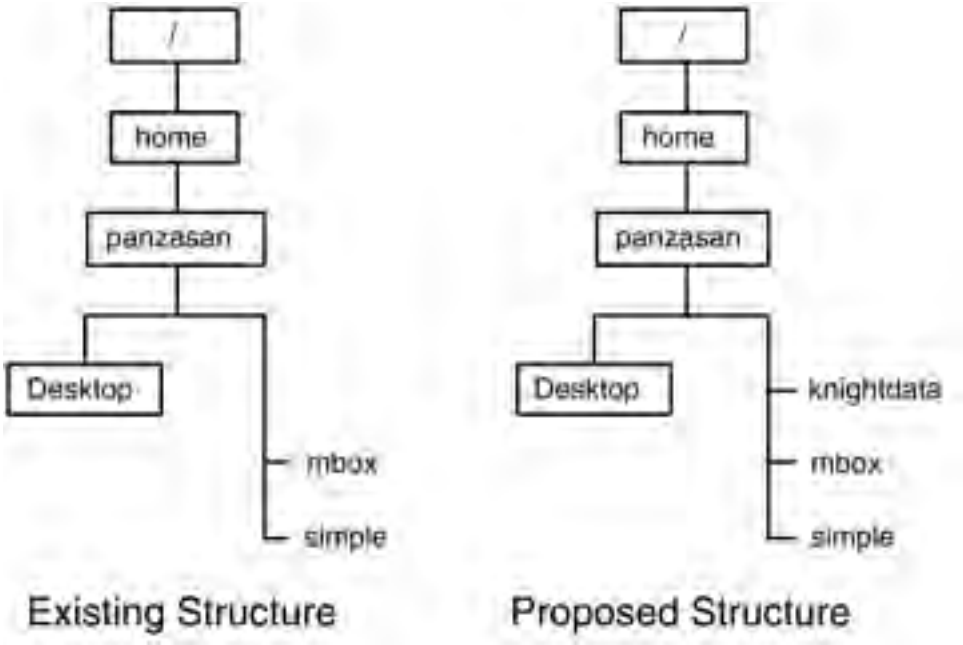


Figure 5.2 Linking files; Sancho's directory structure.

Viewing File Contents

File contents can be viewed in several ways. In this section, we cover the most popular methods: using the `cat` command and using the `more` and `less` commands.

Listing File Contents: cat Command

The `cat` (concatenate) command displays the contents of all specified files:

```
$ cat filename1 filename2 . . .<Enter>
```

If you want `cat` to number all the lines in a file, use the `-n` option as such:

```
$ cat filename -n<Enter>
```

The most obvious problem with `cat` is that when viewing files that are longer than a single screen, the file scrolls until the bottom of the file is reached. Thus, you can easily miss what was at the top of the file. You can mitigate this situation in a couple of ways. First, while the file is actually being displayed, you could press Ctrl-S to freeze the scrolling of the screen during output. To resume scrolling, you could then press Ctrl-Q. This action might not be appropriate, however, if the scrolling happens quickly.

The second and recommended method is to use the `| more` ("pipe-more") and `| less` ("pipe-less") commands in conjunction with `cat`, as such:

```
$ cat filename | more<Enter>
```

"Pipe-more"and "pipe-less" cause `cat` to display the first screenful of file contents and then stop and wait for to the user to go forward only (with pipe-more) or backward *and* forward (with pipe-less) through the file contents at the user's preferred pace. We will discuss the `more` and `less` commands in more detail in the next section.

In Chapter 10, "The vi Editor," we will also discuss how `cat` can read from a device called `standard in` and display to a device called `standard out`. You can also use the `cat` command to create or even print a file by combining it with the redirect output symbol (>). In fact, we tell you how to use `cat` for printing at the end of this chapter, in the section *`cat` Can Print, Too*. Otherwise, please check the `man` pages and other sources for additional options.

Example 5.3 Listing File Contents with cat

As mentioned earlier, Don Quixote had asked Sancho to start a file called *manuals* that will eventually list materials that must be studied by those Rueful Figures, Inc. staff members who aspire to be knights. Now, the Don would like to have a look at the file. Here is what he enters:

```
$ cd<Enter>
$ pwd<Enter>
/home/quixoted
$ cat manuals<Enter>
```

Displaying a File Page by Page: The more and less Commands

In the previous section, we mentioned the problem of file contents on the terminal screen scrolling by so rapidly that you cannot comprehend the

contents. We mentioned that you could use Ctrl-S and Ctrl-Q to stop and restart scrolling, but it is probably a better idea to use the `more` and `less` commands to display the contents of a file one screen at a time. Both commands pause after the first screenful of information is displayed and await instruction.

To invoke `more`, the syntax could not be simpler:

```
$ more filename<Enter>
```

When you use the `more` command, the `--more--` status message appears at the bottom of each screenful of file contents (with the exception of the last screenful of information, when the word `END` appears in parentheses). When `more` is reading from a file, a percentage appears alongside the `more` text indicating the proportion of the file already displayed.

To maneuver through the file contents, you use special keypresses that are listed in Table 5.1.

The `less` command is a more recent (and still not really as well known) improvement over `more`, and the syntax is very similar:

```
$ less filename<Enter>
```

You get more mobility when you examine files with the `less` command, as shown with the additional keypresses listed in Table 5.2.

When you use the `less` command, only a colon (:) appears at the bottom of each screenful of output, again with the exception of the last screenful.

The `more` and `less` commands have several more handy options in their `man` pages. (In fact, the `man` pages themselves use the `less` command to display their contents.)

Example 5.4 Listing File Contents Screen by Screen with cat | less

Because of the support requests he has recently received, Freston cannot help but notice that RFI staff members have been adding to and changing

Table 5.1 Moving through a File Displayed with the more Command

KEYPRESS	EXPLANATION
`<Spacebar>`	Moves down through the file one screen at a time
`<Enter>`	Moves down through the file one line at a time
`<Ctrl>-c` or `<Ctrl>-z`	Ends `more` and returns to the command line

Table 5.2 Moving through a File Displayed with the less Command

KEYPRESS	EXPLANATION
<Spacebar>	Moves down through the file one screen at a time
<Enter>	Moves down through the file one line at a time
d or u	Moves down or up through the file half a screen at a time
down or up arrow	Moves down or up through the file one line at a time
<Ctrl>-c or <Ctrl>-z	Ends less and returns to the command line

their individual portions of the corporate directory structure. To monitor the dimensions of the home directory portion of the RFI directory structure, Freston lists, recursively, the home directory structure. He redirects the information to a file called *rfi_home_trees_jul_21* in his home directory (later, in Chapter 10, "The vi Editor," we will discuss the commands that he might have used to do that). Then, he lists the contents of that file to his terminal screen with the following command:

```
# cat rfi_home_trees_jul_21 | less<Enter>
```

Copying, Moving, and Deleting Files

Users and system administrators alike consider printing to be the most important file manipulation function. That is certainly true from the standpoint of accomplishments in the workplace, but more goes into those accomplishments than simply printing. For example, you might print a draft document for review, finalize it, and then print the document again. But you probably performed a lot of file manipulation before that draft review or crowning moment. And you probably did more file manipulation afterward; perhaps you copied the document or moved the file in the directory for housecleaning, security, sharing, or collaboration.

Copying a Single File: The cp Command

At first glance, the cp (copy) command appears to be fairly straightforward:

```
$ cp source target<Enter>
```

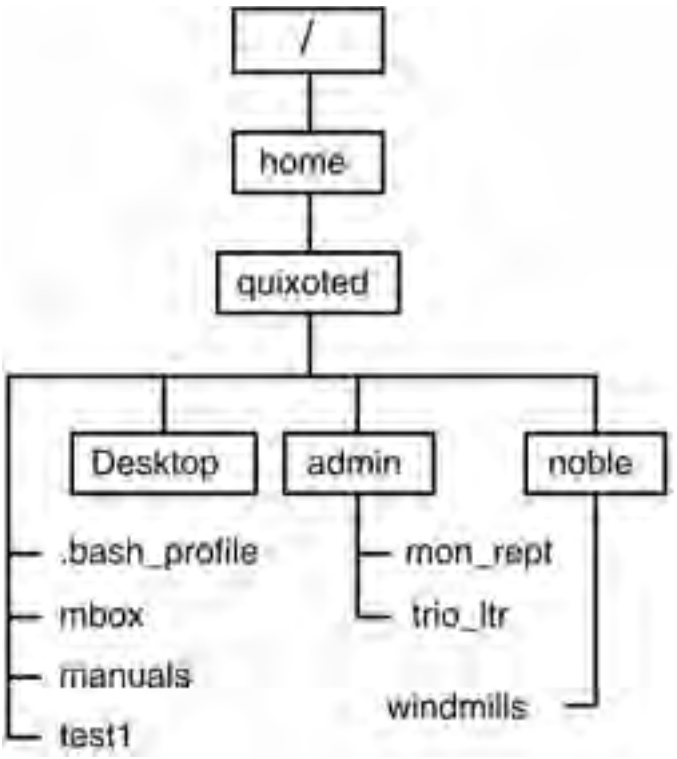


Figure 5.3 Copying files; Don Quixote's directory structure.

You follow the command with the source filename (if the file is not in the current working directory, you must specify either the relative or full path name) and the target (also called the destination). See Example 5.5.

The `cp` command has many options. We recommend that you check the `man` pages. You might want to look at other sources of information as well. For example, you could check out the `help` utility by typing the following:

```
$ cp-h | more<Enter>
```

What effect does `cp` have on inode information? All information on the source file remains unchanged except that the last access time and date are updated. New inode numbers, permissions, and other relevant entries are created for the new files.

Example 5.5 Copying Single Files with cp

One part of Freston's responsibilities is to report on system activity to Lady Dulcinea. One aspect of this reporting includes creating a copy of his

latest *rfi_home_trees_mmmdd* file and sending it to Lady Dulcinea's */home/dulcinea/repts* directory. Here is how he does that:

```
$ pwd<Enter>
/root
$ cp rfi_home_trees_jul21 /home/dulcinea/repts/<Enter>
```

Example 5.5 might be a little misleading for new users. You can change the name of the source file in flight, as seen in the example, only if the target directory is also the current working directory. Otherwise, you must execute a separate renaming process (if such is desirable) after the copy of the file has arrived. Renaming files is discussed later in this chapter.

Copying Multiple or Special Files: The cp Command

When copying multiple files, the target must be a directory:

```
$ cp -i file1 file2 . . . target_dir<Enter>
```

WARNING **We used the `-i` option with the `cp` command just in case the `cp` target is the name of a file that already exists. Otherwise, without the `-i`, if the target file does already exist it would be overwritten without any error or notification message appearing. To prevent that from happening and to ensure that the system prompts the user, the `-i` option should be included.**

Note that the copies will have the same name as the originals.

Example 5.4 Copying Multiple Files with cp

Don Quixote's existing structure is shown in Figure 5.4. He decides that copies of his manuals and *test1* files also belong within the *noble* subdirectory. Here is how he copies them:

```
$ cd<Enter>
$ pwd<Enter>
/home/quixoted
$ cp manuals test1 /noble<Enter>
$ cd noble<Enter>
$ ls
manual windmills test
```

As indicated in Example 5.4, this situation was also an opportunity to use relative path names.

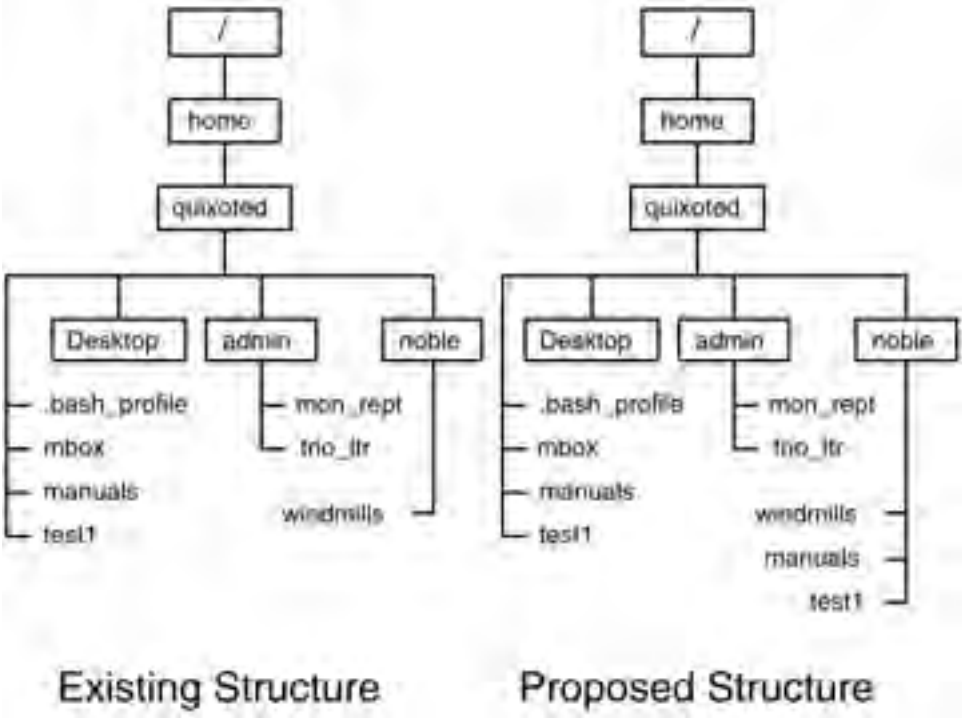


Figure 5.4 Copying multiple files.

This situation would also have been an opportunity to use the . and .. special files introduced in Chapter 4. How would the noble Don have done that? He could have started in the lower */home/quixoted/noble* subdirectory, not in the higher */home/quixoted* directory, and entered the following:

```
$ cd /home/quixoted/noble<Enter>
$ pwd<Enter>
/home/quixoted/noble
$ cp ../manuals ../test1 ./<Enter>
```

Copying Recursively: The cp Command

To recursively copy a directory and its files and subdirectories, including the files within the subdirectories, use the `cp -R` command. Here is the syntax:

```
$ cp -iR sourcedirectory targetdirectory<Enter>
```

Using `cp -R` recursively in this manner enables the replication of complete data trees. Meanwhile, if the target directory does not yet exist, the `cp` command will create it. Again, we added the `-i` option as a best practice to ensure that files are not overwritten unless an advisory message is presented.

Example 5.5 Copying Files Recursively with cp -R

At this point, Don Quixote wants to copy the files called *mon_rept* and *trio_ltr*, found in the *admin* subdirectory within his home directory, to a new subdirectory called *newdir* that does not yet exist but will be within his home directory, too. Please refer to Figure 5.5 for an illustration of his plan.

Here is how he does it:

```
$ cp -iR /home/quixoted/admin /home/quixoted/newdir<Enter>
```

Moving and Renaming Files: The mv Command

Like the `cp` command, the `mv` command appears straightforward:

```
$ mv source target<Enter>
```

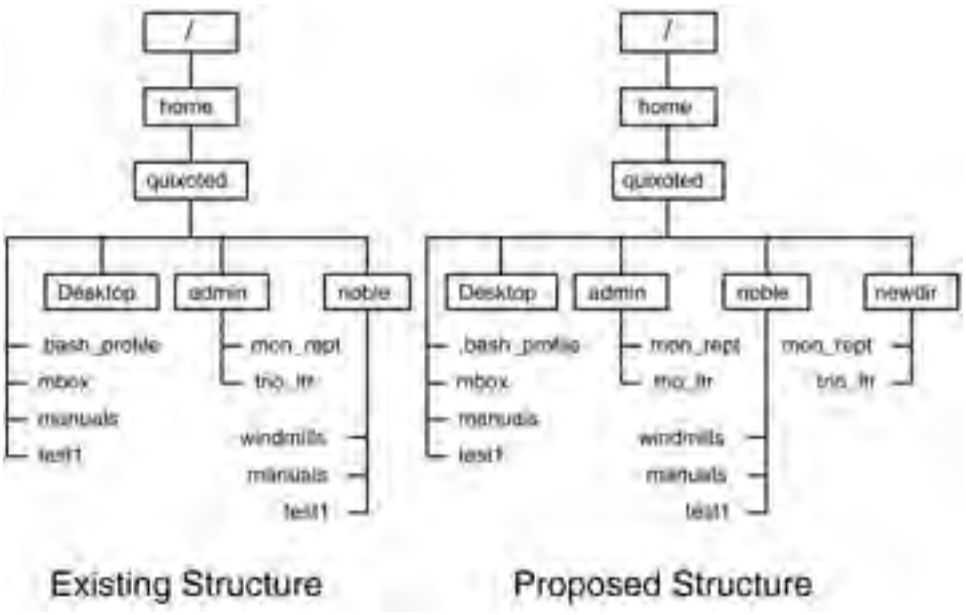


Figure 5.5 Copying files recursively.

The mv command is followed by the source filename (you must specify the relative or full path name if the file is not in the current working directory) and the target. The target can be a directory (if you are moving a file only) or a filename (if you are moving and renaming). See Example 5.6 for a file-moving example. If you are moving more than one file, the target must be a directory for the command to execute.

Generally, when files are moved and the destination is a directory name, the files retain their original names. If you specify a filename as the target, however, you will be moving and renaming the file at the same time. Example 5.7 will show you how to perform this task, too.

The mv command has the same effect as cp on inode information. All information on the source file remains unchanged except that the last access time and date are updated. New inode numbers and entries are also created for the new files.

WARNING Just like with the cp command, if the mv target is the name of an existing file and you have the correct permissions set for that file and the directory, the file will be overwritten. No error or notification message will appear. To prevent this situation and to cause the system to prompt the user, use the -i option with the mv command.

We recommend that you check the man pages for all the options of the mv command. You should check out other information sources as well, such as the help utility:

```
$ mv -h | more<Enter>
```

Example 5.6 Moving Files with mv

Don Quixote thinks that the file *test1*, which instructs potential knights-errant about bravery in the face of great odds, would be more appropriately placed under the *admin* subdirectory than under the *noble* subdirectory where it is now. Here is how he will move it:

```
$ cd<Enter>
$ pwd<Enter>
/home/quixoted
$ mv /noble/test1 /admin<Enter>
```

Example 5.7 Renaming Files with mv

Later, the Don recounts to Freston how he moved *test1* from *noble* to *admin* because the file dealt with bravery. Now, he tells Freston, he will actually

rename *test1* to the more appropriate name, *bravery*. Freston tells Don Quixote that had the Don realized beforehand that he would eventually rename *test1*, he could have done both at once (in other words, move the file *and* rename it) by using the same `mv` command. Here is how Freston tells the Don that it should have been done to begin with:

```
$ mv /noble/test1 /admin/bravery<Enter>
```

Deleting Files and Directories: The rm Command

The `rm` command removes the entries for the specified file(s) from a directory. But you must have the required permissions to remove the files. Here is the syntax for the `rm` command:

```
$ rm [-i] filename1 filename2 ...<Enter>
```

The `-i` option is the interactive version of the `rm` command. You do not receive confirmation after the files are removed. Example 5.8 shows the `rm` command with and without the `-i` option.

You can also remove directories with the `rm` command. The syntax is similar:

```
$ rm [-i][-r] directoryname1 directoryname2 ...<Enter>
```

You can also use the `-r` option for recursive removal of directories and their respective files.

WARNING **Be careful when using the `rm` command with the `-r` option; that is, to remove directories and their contents recursively. Removal in this manner does not require the directories to be empty before execution.**

In Example 4.9 in Chapter 4, we briefly mention the effects of the `rm` command on files with links. You might want to refer back to that example now in light of the discussion here.

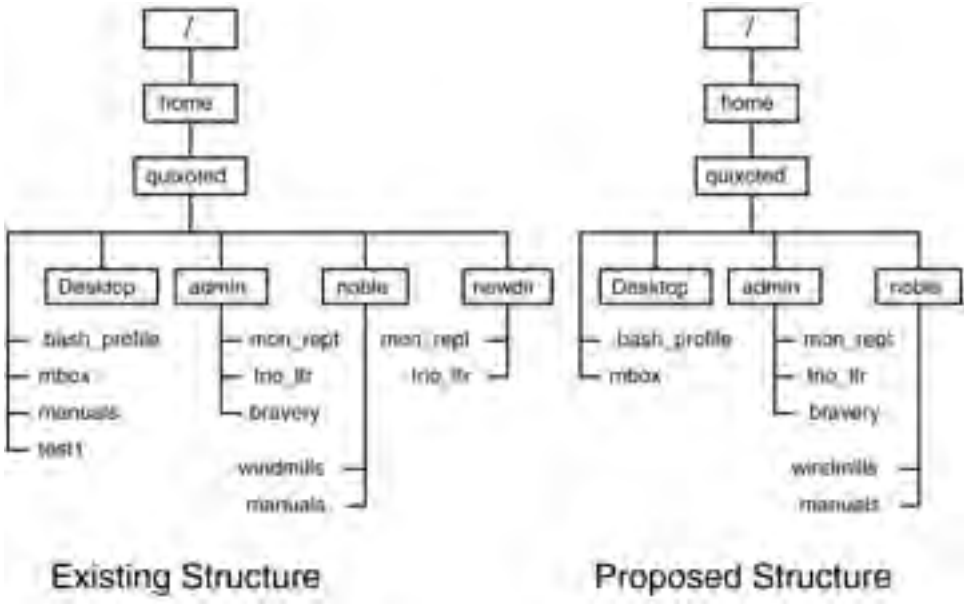


Figure 5.6 Removing files and directories.

Example 5.8 Deleting Files with rm

Don Quixote looks at his portion of the RFI directory structure (please look at the "Existing Structure" side of Figure 5.6) and decides to clean it up a little.

First, Don Quixote will remove one copy of *manuals*—the one that is *not* in the *noble* subdirectory:

```
$ cd /home/quixoted<Enter>
$ rm manuals<Enter>
```

He wants to remove the original copy of *test1* now, too. But, instead of just removing it outright, he will remove it interactively:

```
$ rm -i test1<Enter>
rm: remove 'test1'? y<Enter>
```

Example 5.9 Deleting Directories with rm

All Don Quixote wants to remove now, in order to complete his cleanup, is the *newdir* subdirectory and its files named *trio-ltr* and *mon_rept*. The

copies never did come in handy as he originally planned. Here is how he removes *newdir* recursively:

```
$ pwd<Enter>
/home/quixoted
$ rm -r ./newdir<Enter>
```

The mtools Utilities

Now that we have discussed several Linux/UNIX commands and their effects on Linux/UNIX files and directories, let's visit the `mtools`.

The `mtools` utilities are a public domain collection of UNIX utilities for creating, accessing, and manipulating DOS disks from Linux/UNIX without having to mount or unmount the DOS file systems (in our case, that means not having to mount and unmount DOS floppy disk file systems). The `mtools` are maintained by Alain Knaff and David Niemi. Some UNIX veterans refer to `mtools` commands simply as `m` commands.

Obtaining and Loading mtools

A copy of the `mtools` utilities might be packaged with your Linux distribution. If so, they probably work fine. But two or three years back, the `mtools` enclosed with distributions were unreliable, so it was best to get the latest version of `mtools` from one of the `mtools` Web sites. If you would like to obtain and use these utilities, or if you find that your existing version is buggy, check out one of the following sites to download the latest version of `mtools` or the latest patches to your version:

- www.tux.org/pub/knaff/mtools/ (U.S. Web site)
- http://mtools.linux.lu/ (European Web site)
- ftp://www.tux.org/knaff/mtools/
- ftp://ibiblio.unc.edu/pub/Linux/utils/disk-management/

Please note that more versions are available at the FTP sites than at the HTTP sites. We recommend that you download the text files (especially the `mtools` manuals from one of the first two Web sites) as well, because they include the most up-to-date instructions for loading `mtools` on your system and also have an in-depth discussion of `mtools` in general.

Suppose that you download the *mtools-3.9.8-2.i386.rpm* file from among the several formats and versions available. The latest full version of `mtools`, version 3.9.8, was released in late May 2001. Once you have downloaded that version, you can then install it with those versions of Linux that use `rpms`, for example, by entering

```
# rpm -U mtools-3.9.8-2.i386.rpm<Enter>
```

If you use another version, simply follow the instructions for obtaining and loading alternate versions (found by following the FAQ links at the HTTP Web sites).

Basic mtools Features

The `mtools` commands are usually identical to DOS commands with the following difference: exception of the addition of an `m` prefix. The `mtools` commands follow the DOS convention of referring to DOS file systems as drives. For example, the first floppy disk drive is called the A: drive, and the first hard disk drive is customarily called the C: drive. DOS can even refer to a drive by more than one (letter) designation, or in the case of multiple floppy drives, using one drive letter to refer to both drives (in a predetermined search order). Furthermore, DOS filenames are generally preceded by the drive letter and a colon. UNIX file names do not have such prefixes.

If you are accessing only the floppy drives, `mtools` enables you to do so without mounting and unmounting their respective file systems, and you usually do not have to modify the */usr/local/etc/mtools.conf* file. You must ensure that this file is properly configured, however, if you will be accessing DOS hard disk partitions and DOS emulation image files.

If you find it necessary to use pattern matching, remember that the `mtools` utilities use Linux/UNIX syntax and conventions (for example, use * as a wildcard instead of DOS's *.*). Also, to add options to commands, use a hyphen (-) instead of DOS's slash (/).

mtools Commands

The `mtools` utilities have an extensive `man` page, and a great deal of information is also available on the Internet sites listed previously.

We present only a sampling of the commands available in `mtools`. These commands resemble those used frequently by DOS users and DOS

file system administrators. The `mtools` commands are not case sensitive. Table 5.3 lists the syntax and purpose of about half of the existing `mtools` commands.

Refer to the `man` pages and other sources for information regarding the options. Remember that all pattern matching follows UNIX syntax and conventions, not DOS's. With commands such as `mcopy` and `mmove`, you can determine whether the files are going from a DOS file system to a UNIX file system by determining whether the *sourcefile* or *targetfile* has the DOS letter-drive designation.

MToolsFM

MToolsFM is a program designed to give people easy GUI-based file management of floppy disks under Linux and UNIX-like operating systems. MtoolsFM was formerly called mfm, but that was occasionally confused with MFM, the Motif File Manager. The latest version is MToolsFM-1.8.

It uses `mtools` functionality. But the advantage to MToolsFM is that because there are fewer and fewer people who know DOS, the GUI interface gives them the functionality but does not require them to learn a different set of command line commands.

MtoolsFM can be downloaded from www.core-coutainville.org/MToolsFM/, where it is maintained by Christian Ospelkaus (christian@core-coutainville.org).

Printing Files: The lpr, lpq, and lprm Commands

Printing with Linux could take up a book on its own. Here, we will try to condense these introductory instructions and examples as much as possible. If you will be setting up printing with Linux, however, we recommend that you read the latest Linux HOWTO on printing at www.linuxprinting.org. The HOWTO has been maintained (and copyrighted, too) by Grant Taylor since 1992.

Connecting Your Printer and Creating a Print Queue

Prior to discussing, however briefly, some basic printing commands, let's follow "first things first." First, connect a printer to your system. We used a Hewlett-Packard LaserJet 6L printer. We connected it to the parallel port.

Table 5.3 mtools Commands

SYNTAX	PURPOSE
`mattrib +/-[flags] dosfilename(s)`	Changes file attributes
`mcat [-w] drive:`	Copies an entire disk image from or to the floppy device; same as the Linux/UNIX cat command
`mcd dosdirectory`	Changes the `mtools` working directory on the DOS drive
`mcopy -[options] dossourcefile dostargetfile`	Copies DOS files to and from UNIX
`mdel -[options] dossourcefile dostargetfile`	Deletes a DOS file
`mdeltree -[options] dosdirectory (ies)`	Removes a DOS directory and all its subdirectories and their contents
`mdir -[options] dosfilename(s)`	Displays the contents of a DOS directory
`mformat -[options] dosdrive:`	Adds an MS-DOS filesystem to a diskette that has already been formatted by a UNIX low-level `fdformat` command
`mlabel -[options] drive:[label]`	Adds a volume label to a disk
`mmd -[options] dosdirectory(ies)`	Creates a DOS subdirectory
`mmount dosdrive [args]`	Mounts a DOS disk; available only on Linux
`mmove -[options] dossourcefile dostargetfile`	Moves or renames an existing DOS file or subdirectory
`mrd -[options] dosdirectory(ies)`	Removes a DOS subdirectory
`mren -[options] dossourcefile dostargetfile`	Moves or renames an existing DOS file or subdirectory
`mtype -[options] dosfile(s)`	Displays a DOS file

Next, let's create a print queue for the printer. A print queue mechanism enables more than one user to use the same printer without having to wait for the printer to become available. But to create a queue, we have to leave the command line world for a moment and enter the world of the X Window system, which itself is covered in more detail in Chapter 14, "The Linux X Window System."

To invoke an X Window System manager (in other words, a GUI desktop), go to your command line and, as the root user, enter

```
# su - root
# Password:
# startx<Enter>
```

One or another of the Linux Desktop Manager programs will be invoked. The following instructions will presume that you have KDE installed and functioning.

On the KDE desktop, click the `shell` terminal icon on the toolbar (on the toolbar, the icon we are looking for is one that consists of a miniature terminal window with some sort of abalone or similar shell overlaying it). Or, if there is no such icon, press the K button and search within its menus for the `Terminal` bar and click it. At some point, you will be successful and a terminal window will appear on the desktop. Activate the terminal window by clicking it.

Once you are in the terminal window, ensure that you are logged in as a root user. Ordinary users cannot set up printing functions. Then, as the root user, type

```
# printtool<Enter>
```

A separate `printconf-gui` dialog box/window will appear. There will likely be no print queues listed yet. On the toolbar, click `New`.

A separate `Edit Queue` dialog box will appear (see Figure 5.7). On the *left-hand side* (LHS), there will be the following list of variables for which you will have to specifiy values:

```
Name and Aliases
Queue Type
Printer Driver
```

Highlight `Name and Aliases`. Then, on the *right-hand side* (RHS), in the `Queue Name` field, type the name you will use to identify your printing queue. Because we were installing a Hewlett-Packard LaserJet 6L

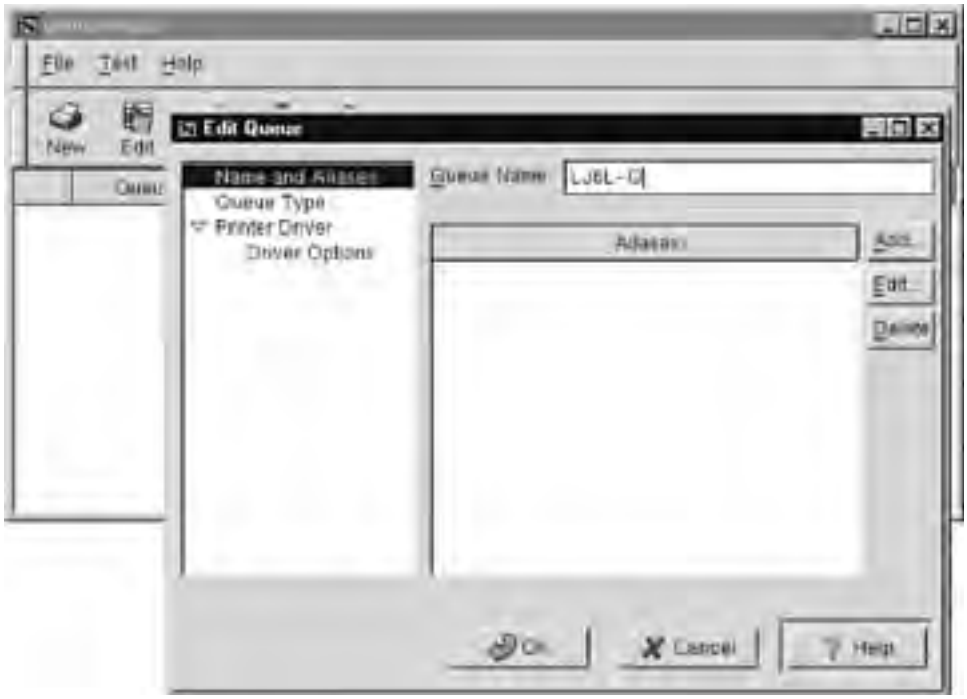


Figure 5.7 The Edit Queue dialog box; Name and Aliases.

printer, we specified `LJ6L-Q` as our queue name. If you want to use one or more aliases for the queue, then add your preferred alias names to the `Aliases` field. Aliases are not mandatory, but the queue name is.

Back on the LHS, highlight `Queue Type` (see Figure 5.8). On the RHS now, leave the `Queue Type` value as the default `Local Printer` (that is, if you are attaching the printer to the system you are connected to and not on a server elsewhere on the network), and leave the `Printer Device` value as the default `/dev/lp0` (UNIX-speak for "the first parallel port").

Now, highlight `Printer Driver` on the LHS (see Figure 5.9). On the RHS, scroll down to the name of your printer's manufacturer (in our case, it was `HP`) and double-click it. Another listing appears consisting of the names of the printer models made by the manufacturer you specified. Scroll down to the actual model name of your printer (in our case, it was `LaserJet 6L`) and highlight the name. We left the `Printer Driver` value as the default `lj5gray` and then clicked the OK button at the bottom of the `Edit Queue` dialog box to indicate that we were finished.

Leave the `Driver Options` at their respective defaults (see Figure 5.10). We recommend that you do not change these until you have proven

Figure 5.8 The Edit Queue dialog box; Queue Type.

that the printing scenario works. You can revisit this screen at any time to make further adjustments.

Back in the `printconf-gui`window, we see that the queue we created is now listed (see Figure 5.11). Highlight the name of the queue and click the Default button on the toolbar. A check mark should appear at the LHS of the queue name. Starting and or refreshing the `lpd` daemon is absolutely mandatory at this point. Essentially, these menus have updated the */etc/printcap* file. The daemon now has to reread this file to activate the new printer. What is the easiest way to perform this task? You must click the `File` drop-down menu and select `Restart lpd`.

Once the `lpd` daemon has been restarted, you should test the printer. Click the `Test` drop-down menu and select `Print ASCII Test Page`. See Figure 5.12.

At this point, you should have a functional printing environment. We recommend that you do not edit any of the print files such as the */etc/printcap* or any of the queue directories manually. Always use the `printtool` utility or the printing commands to manage this environment. The reason is

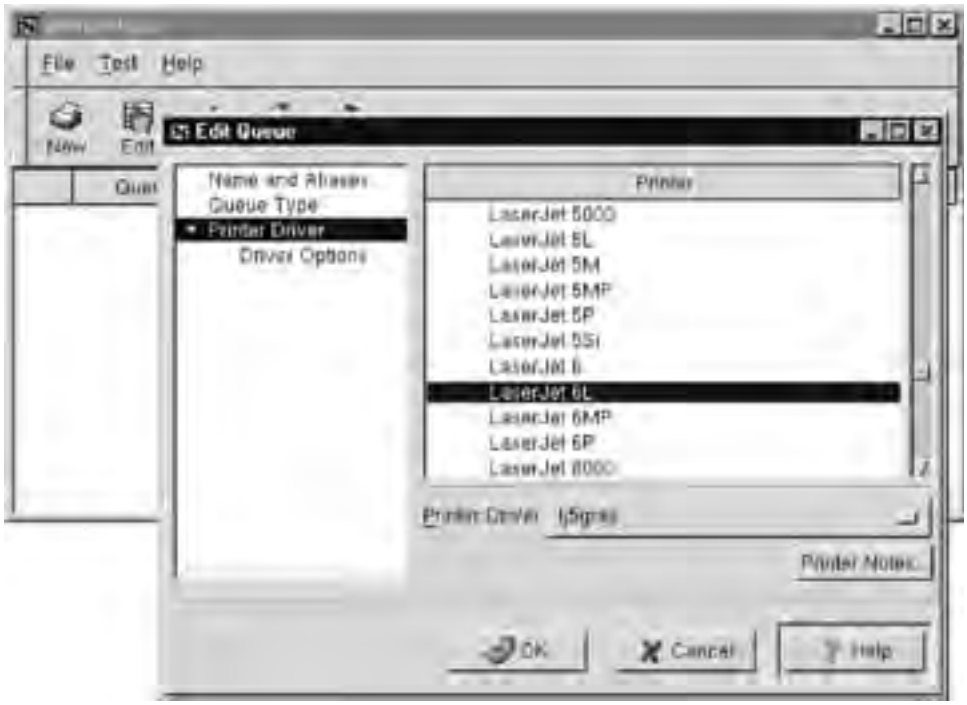


Figure 5.9 The Edit Queue dialog box; Printer Driver.

because many of these entries for print files and parameters are maintained across several files. The commands and utilities take care of the consistency over all these files and will prevent your printing environment from corruption.

```
# exit<Enter>
```

On the GUI desktop, click the K^ button and log out. After verifying that you are indeed logging out, you will be returned to your normal command prompt.

Commands for Printing

Now that we have connected a printer and created a print queue, we can get to the actual printing commands. Or, to paraphrase the great comedian Bill Cosby, "(We) told you all *that* to tell you *this*."

Figure 5.10 The Edit Queue dialog box; Driver Options.

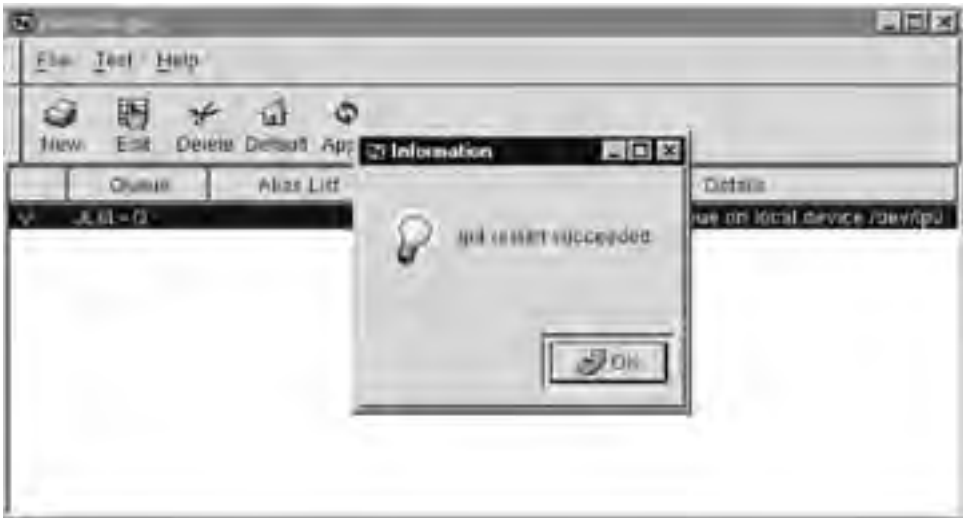


Figure 5.11 The printconf-gui dialog box; restarting the lpd daemon.

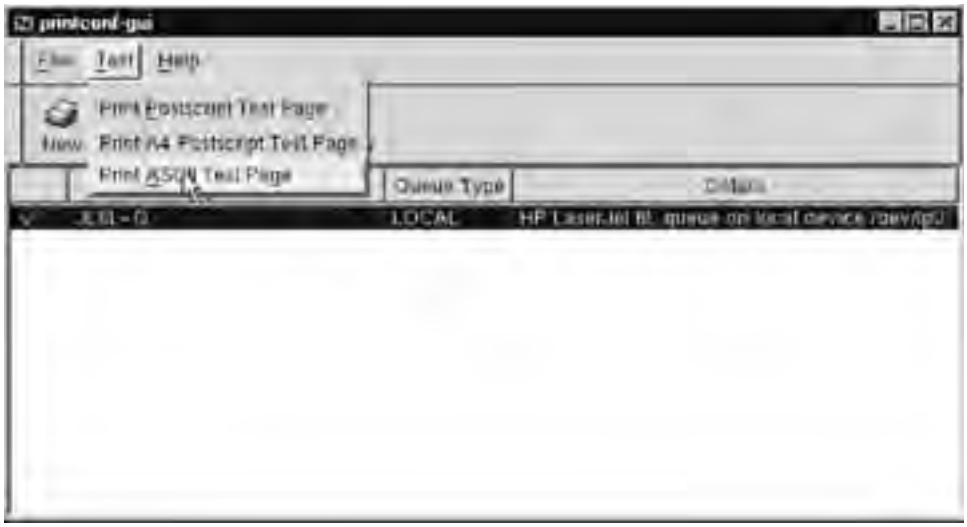


Figure 5.12 The printconf-gui dialog box; printing a test page.

Printing with lpr

To queue a file for printing, ordinary users use the `lpr` command. The root user and superuser have additional printing commands at their disposal.

To send a print job to the default printer, here is the syntax:

```
$ lpr filename1 filename2 ...<Enter>
```

To specify a printer other than the default printer, where `lp1` is the name of the alternate printer, you enter the following:

```
$ lpr -P lp1 filename<Enter>
```

The `lpr` process sends a copy of the file to the */var/spool* directory, where the copy sits for however much time it takes for a free line-printing daemon, called `lpd`, to discover and print it. For a short queue, the timing is almost instantaneous; the print job seems to go straight to the printer.

For large files, where copying them to the spool directory might cause memory congestion, you could create a just-in-time delivery-oriented, symbolic link between the spool directory and the file. When the print job is to be executed for the specified file name, the `lpd` daemon processes the link, which causes the file to be directly transferred in time for printing:

```
$ lpr -s filename<Enter>
```

The `lpr` command has many options and arguments. We recommend that you check the `man` pages and other sources for more information. Two other important print commands, `lpq` and `lprm`, are shown in Example 5.10.

Example 5.10 Printing Files with lpr

Don Quixote wants to read and review a poem he wrote about the windmills that so soundly defeated him on a previous Noble Deeds Division misadventure. He also wants to review his weekly report (the file called *mon_rept*) on hard copy before sending it to Lady Dulcinea. Here is how he sends the two print jobs to his default printer:

```
$ lpr windmills mon_rept<Enter>
```

Listing and Canceling Print Jobs

To list print jobs in the default print queue, display its current status:

```
$ lpq<Enter>
```

To check the status of any queue that is not the default queue (assuming again that `lp1` is the alternate printer):

```
$ lpq -P lp1 filename<Enter>
```

To cancel a print job:

```
$ lprm jobnumber<Enter>
```

To find the job number, issue the `lpq` command beforehand.

Example 5.11 Listing Print Jobs with lpq

Don Quixote, after sending his print jobs to the queue, does a quick check of the print queue to see whether he has to wait long for the printing to be done and what the current status of his two print requests is:

```
$ lpq<Enter>
Rank Owner Job Files Total Size
active quixoted 301 windmills 255 bytes
active quixoted 302 mon_rept 2537 bytes
```

The Don sees that his two print jobs are first in line. Both are small, so they should be done soon.

Example 5.12 Canceling Print Jobs with lprm

At the last second, Don Quixote decides to add to his weekly report. So, he cancels the printing of *mon_rept* for the time being:

```
$ lprm 302<Enter>
```

cat Can Print, Too

Before we leave this topic and this chapter, it is worth pointing out that the `cat` command can also be used to print text and files. To use `cat` in this manner also requires redirection, which we have mentioned previously under *Viewing File Contents*.

Exercises

1. Change to the */home/mydirectory/mydir* directory. Use the `touch` command to create two zero-length files called *myfile1* and *myfile2* in your *mydir* directory:

   ```
   $ cd mydir<Enter>
   $ touch myfile1<Enter>
   $ touch myfile2<Enter>
   ```

2. Display a long listing of the contents of your *mydir* directory. What are the sizes of *myfile1* and *myfile2*?

   ```
   $ ls -l<Enter>
   ```

3. View the long listing again, but this time display the inode numbers, too. What are the inode numbers of each file?

   ```
   $ ls -li<Enter>
   ```

4. Using the `pwd` command, verify that you are in the home directory, */home/directoryname*.

```
$ pwd<Enter>
```

5. List the contents of your home directory, including its hidden files.

```
$ ls -a<Enter>
```

6. View the contents of the */etc/motd* and */etc/passwd* files. Use the `cat`, `more`, and `less` commands to see how each command handles the output. The */etc/motd* file contains the message of the day (what you see after you first log in). The */etc/passwd* file contains a list of all the users who are authorized to use the system.

```
$ cat /etc/motd<Enter>
$ cat /etc/passwd<Enter>
$ more /etc/motd<Enter>
$ less /etc/motd<Enter>
$ more /etc/passwd<Enter>
$ less /etc/passwd<Enter>
```

7. Turn off the printer attached to your system. Print the */etc/motd* file on the system printer.

```
$ lpr /etc/motd<Enter>
```

8. Check the status of your print job.

```
$ lpq<Enter>
```

9. Turn the printer back on.
10. Copy the */bin/cat* file into your current (that is, home) directory:

```
$ cp /bin/cat /home/teamxx<Enter>
```

or

```
$ cp /bin/cat . <Enter>
```

11. Copy the */usr/bin/cal* file into your current (home) directory:

```
$ cp /usr/bin/cal /home/teamxx<Enter>
```

or

```
$ cp /usr/bin/cal . <Enter>
```

12. List the files in your current directory. You should see the two files you just copied.

```
$ ls<Enter>
```

13. In your home directory, create a subdirectory called *bin*:

```
$ mkdir bin<Enter>
```

14. Move and rename the two files that you copied in Exercises 5 and 6 into your new subdirectory. Name them *mycat* and *mycal*, respectively:

```
$ mv cat bin/mycat<Enter>
$ mv cal bin/mycal<Enter>
```

15. Make the new subdirectory (*bin*) your current directory:

```
$ cd bin<Enter>
```

16. List the contents of the directory to verify that the files were copied.

```
$ ls<Enter>
```

17. Use the `mycat` command to list the *.bash_profile* file in your home directory:

```
$ mycat ../.bash_profile<Enter>
```

or

```
$ mycat /home/teamxx/.bash_profile<Enter>
```

18. Make your home directory the current directory:

```
$ cd<Enter>
```

19. Create another subdirectory called *goodstuff* in your home directory:

```
$ mkdir goodstuff<Enter>
```

20. Copy a file called */etc/profile* into the new directory and name the new file *newprofile*:

```
$ cp /etc/profile goodstuff/newprofile<Enter>
```

21. Use the `cat` command to look at the file:

```
$ cat goodstuff/newprofile<Enter>
```

22. Is it hard to read? Try the `more` and `less` commands:

```
$ more goodstuff/newprofile<Enter>
```

or

```
$ less goodstuff/newprofile<Enter>
```

23. The *newprofile* filename is too long to input time after time. Change its name to *np*. List the contents of the *goodstuff* directory to make sure that you have accomplished the task:

```
$ mv goodstuff/newprofile goodstuff/np<Enter>
$ ls goodstuff<Enter>
```

24. This point is a good place to check everything out. Starting from your home directory and working down, display a hierarchical tree of your files and subdirectories.
25. Ensure that you are in your home directory. Remove the *goodstuff* directory:

```
$ pwd<Enter>
$ rmdir goodstuff<Enter>
```

    Could you do it? Why or why not?

26. Change to the *goodstuff* directory. Request a listing of the contents of the *goodstuff* directory (including any hidden files). Remove the files. Do another listing of the *goodstuff* directory, including the hidden files. Note that the `.` and `..` files are still there. The directory is considered empty if these are the only two entries left.

```
$ cd goodstuff<Enter>
$ ls -a<Enter>
$ rm np<Enter>
$ ls -a<Enter>
```

27. Now, remove the directory:

```
$ cd .. <Enter>
$ rmdir goodstuff<Enter>
```

28. Assume that you want to find the program that generates today's date. Use the `slocate` command to search on the keyword `date`:

```
$ slocate date<Enter>
```

    From the list produced, find the command that displays the date. What is the name of the command, and what directory or directories is it in?

29. Having found the `date` command in Exercise 2, use `man` without options to obtain the correct syntax of the command:

```
$ man date<Enter>
```

When you have finished with the `man` page, type `q` to exit from it.

See Appendix B for answers.

Quiz

1. What is the effect of executing the following commands in succession?

```
$ cd /home/quixoted<Enter>
$ cp file1 file2<Enter>
```

2. What is the effect of executing the following commands in succession?

```
$ cd /home/quixoted<Enter>
$ mv file1 newfile<Enter>
```

3. What is the effect of executing the following commands in succession?

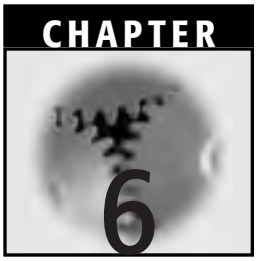
```
$ cd /home/quixoted<Enter>
$ ln newfile myfile<Enter>
```

4. List the commands that can be used to view the contents of a file.
5. Which of the commands listed in Question 4 is used automatically when you invoke `man` pages? How do you know?
6. Which of the following are valid filenames?
   - `!`
   - `aBcDe`
   - `-myfile`
   - `myfile`
   - `my.file`
   - `my file`
   - `.myfile`

7. Once you have configured printing queues and installed printer drivers, what must you do to enable the printing environment? How would you do it?
8. What file has been updated by configuring the queues and installing the printer drivers?

See Appendix C for answers.

CHAPTER



6

Linux File Permissions

Along with passwords and authentication systems, file and directory access—known as *permissions*—forms the basis for system security. Although similar in nature, directory permissions and file permissions are not the same—and occasionally, that difference leads to confusion and even security breaches.

We begin the chapter with a review of the `ls -l` command, which you use to view existing file and directory permissions. Then, we focus on information found in the first column of the `ls -l` response information—the permission bits—which provide a summary of the file mode of these entries. We introduce several terms, commands, and issues involved in setting and changing permissions.

Review of the ls -l Command

We described the output of the `ls -l` (long listing) command in detail in Chapter 4, "Files and Directories in Linux." Example 6.1 and Table 6.1 illustrate our quick review of this frequently used command.

Table 6.1 Response to the ls -l Command

FIELD 1	FIELD 2	FIELD 3	FIELD 4	FIELD 5	FIELD 6	FIELD 7
total 6						
drwxrwxr-x	5	quixoted	knights1	4096	May 26 10:18	admin
drwxr-xr-x	5	quixoted	knights1	4096	Mar 07 08:11	Desktop
-rw-r--r--	1	panzasan	knights1	144	Jul 11 11:42	manuals
-rw-------	1	quixoted	knights1	1116	May 26 10:18	mbox
drwxrwxr-x	5	quixoted	knights1	4096	Apr 05 14:30	misc
drwxrwxr-x	5	quixoted	knights1	4096	Jul 11 15:21	noble

Example 6.1 The ls -l Command

```
$ pwd<Enter>
/home/quixoted
$ ls -l<Enter>
```

The following is a description of the different fields of output:

Field 1 consists of the object identifier and file or directory permission bits.

Field 2 is the link count. Ordinary files have a link count of 1. Directories have a link count of 2. The `ln` command can increase the link count by 1; removing a file reduces its link count by 1. When the link count reaches 0, the file is removed.

Field 3 is the username of the person who owns the entry.

Field 4 is the name of the group for which group protection privileges are in effect.

Field 5 is the character count of the entry.

Field 6 is the date and time the entry was last modified.

Field 7 is the name of the file or directory.

To display information about only a particular directory, use the `-d` option with the `ls -l` command. Directories are treated like ordinary files.

Permissions

In this chapter, we will introduce the term `entry` when we discuss directory and subdirectory contents. Files, directories, linked files, block special files, character special files, symbolic links, named pipes, and sockets are all examples of entries. When a user creates an entry, he or she becomes the owner of that entry. Some people also use the term `object` to indicate an entry.

The owner of an entry generally wants to control who will have access to or use of the entry. Each entry, once created, has attributes called permission bits that are used for access control. Because it is possible to change the nature of these permission bits (or permissions, a term that we will use interchangeably) on an entry, the owner has the power to make the entry either more secure or more widely available.

Table 6.2 Permission Bits

ENTRY	OWNER	GROUP	OTHERS
d	rwx	rwx	rwx

Table 6.2 illustrates a breakdown of permission bits available to each entry.

The single character on the far left of the permission bit string identifies the type of entry, or object. The leftmost group of three permission bits—in other words, the three located next to the object identifier—refers to the permissions the owner of the entry has with respect to the entry. The middle three are the permissions that the group has for the entry (that is, the group to which the entry will belong). The rightmost three bits are the permissions held by other users who are not owners of the entry and who are also not in the group to which the entry belongs.

For both files and directories, `r` is the permission to read; `w` is the permission to write; and `x` is the permission to execute. Although the names of these permissions are identical for files and directories, the permissions themselves mean different things. You might ask, "What do we mean by that?"

When viewing or establishing the permissions for an ordinary file,

- `r` permits the viewing of file contents.
- `w` permits the changing and storing of file contents.
- `x` permits the execution of the file (`r` is also needed).

But for a directory,

- `r` permits the viewing of files in the directory.
- `w` permits the creation and removal of files in the directory (`x` is also needed).
- `x` permits a user to be in the directory (that is, to `cd` to a directory and to access files from the directory).

The `x` permission is required to access any files or subdirectories within a directory. In other words, the `x` permission is required on all directories above the specific directory, as well.

To remove a file from a directory, you need only `w` and `x` permissions for the directory. You do not necessarily need permissions on the file itself.

Changing Permissions: The chmod Command

You use the `chmod` (change permission mode) command along with symbolic notation or numeric notation to specify changes to the existing permissions for a file or directory. How do you perform this task? The answer is, by adding or deleting permissions.

Symbolic Notation

Symbolic notation requires you to know the existing permissions prior to specifying new permissions. Then, when you specify the new permissions, the specification is made with respect to the existing permissions. The syntax for changing permissions using symbolic notation with the `chmod` command is as follows:

```
$ chmod symbolmode filename<Enter>
```

Table 6.3 summarizes the `symbolmode` parameters. You can specify multiple symbolic modes by separating them with commas. The opera-

Table 6.3 Symbolic Parameters

SYMBOLMODE	DEFINITION
u	Owner of the file
g	Owner's group
o	Other users on the system
a	Owner, group, and others
+	Add the permission
-	Remove the permission
=	Clear the existing permission mode, then set to the mode specified
r	Read permission
w	Write permission
x	Execute permission

tions are performed in the order in which they appear (in other words, from left to right). Example 6.2, which deals with the use of symbolic parameters to change permission modes, will illustrate what we mean.

When you use the symbolic mode to specify permissions, the first set of parameters you specify set the permission field, as follows:

- `u` file owner (when discussing permissions, owner permissions are often referred to as user)
- `g` group
- `o` all others
- `a` owner, group, and all others

Using `a` is the same as specifying the `ugo` option. The `a` option is the default permission field. That is, if you omit the permission field, Linux defaults to the `a` option.

The second set of flags determines whether permissions are to be added, taken away, or set as specified:

- `+` adds the specified permissions.
- `-` removes the specified permissions.
- `=` clears the selected permission field and sets it to whatever follows on the command line. If you do not specify a permission mode following the =, then Linux will remove all permissions from the selected field.

The third set of parameters determines the permission as shown here:

- `r` read permission
- `w` write permission
- `x` execute permission for files; search permission for directories

To change permissions, you should first issue the `ls -l` command to check the current permission settings (that is, the current file mode). Then, you use the `chmod` command to specify the symbolic modes. Finally, it is a good idea to check these new permissions by issuing the `ls -l` command again. This procedure is followed in Example 6.2.

NOTE With symbolic and numeric notation, you do not separate entries with spaces. This situation is contrary to your experience with entering commands thus far.

Example 6.2 chmod and Symbolic Notation

Back sometime prior to Chapter 4, "Titles and Directories in Linux,", Sancho, at Don Quixote's request, had created a file called *manuals* (which includes a list of mandatory reading materials for those who wish to become chivalrous knights) and filed it in Don Quixote's home directory */home/quixoted*. Later, in Example 5.2, Sancho linked to the same file but referred to it as *knightdata* in his home directory, */home/panzasan*.

But we had not yet explained how Sancho was able to create a file and store it in Don Quixote's home directory at the beginning. Sancho would not ordinarily have had access to Don Quixote's home directory.

To allow Sancho to save a file in Don Quixote's home directory, */home/quixoted*, Don Quixote would have had to change the permissions on that directory. Here is how he did it:

```
$ pwd<Enter>
/home/quixoted
$ cd ..<Enter>
$ pwd<Enter>
/home
$ ls -l quixoted<Enter>
drwx - - - - - -   5     quixoted   knights1   4096  Mar 07    08:11
     quixoted
$ chmod g+rwx quixoted<Enter>
$ ls -l quixoted<Enter>
drwxrwx - - -      5     quixoted   knights1   4096  Jul 10    08:11
     quixoted
```

We can see that Don Quixote first checked the directory he was in and found that it was his home directory. He knows that he cannot change the status of the directory if he is in it, so he changed to the directory */home* and verified that he was there. Then, he checked the original permissions on the */home/quixoted* directory. At that point, he changed the group permissions to allow anyone in knights1 to have read, write, and execute permissions on */home/quixoted* (because he was already in */home*, he only needed to specify the relative pathname *quixoted*). Then, he checked the status of */home/quixoted* again and found that he was successful. Now, anyone in the group knights1 could save a file in his home directory.

Numeric Notation

The syntax for using the `chmod` command with numeric notation is similar to that used with symbolic notation:

```
$ chmod numeric filename<Enter>
```

With numeric (also called octal) notation, you specify absolute file or directory permissions. You do not need to know the existing values beforehand. This situation contrasts with symbolic notation, which requires you to know the existing permissions before adding, subtracting, or redefining new permissions with respect to the existing permissions.

A scenario might help to explain the numeric notation concept. As we go through it, please refer to Figure 6.1.

Let's say that you have created a file called *newfile* and you want the owner and the group to be able to read and write to it, but you only want it to be readable by all other system users.

First, to enable the owner to read and write to the file, you add the `r` and `w` permissions to the three leftmost permission bits (in other words, the owner's permission bits).

Each permission you set within the nine permission bits is represented by a binary value 1. Each lack of a permission is represented by a 0. For example, `rw-r--r--` translates to 110100100 in binary.

Use Figure 6.1 to translate the binary notation to numeric (octal). In Figure 6.1, we see that the owner's `r` and `w` permission bits would have values of 400 and 200, respectively. Second, to set the same permissions for the group, you add 40 (`r`) and 20 (`w`). Finally, to set the read permission for all other system users, you add 4 (`r`). The numeric result comes from simple addition:

400 + 200 + 40 + 20 + 4 = 664

So, to translate the permission mode from a symbolic (alphabetic) form to a numeric form, you add the numbers that correspond to the individual permissions required and then specify that number as an argument to the `chmod` command.

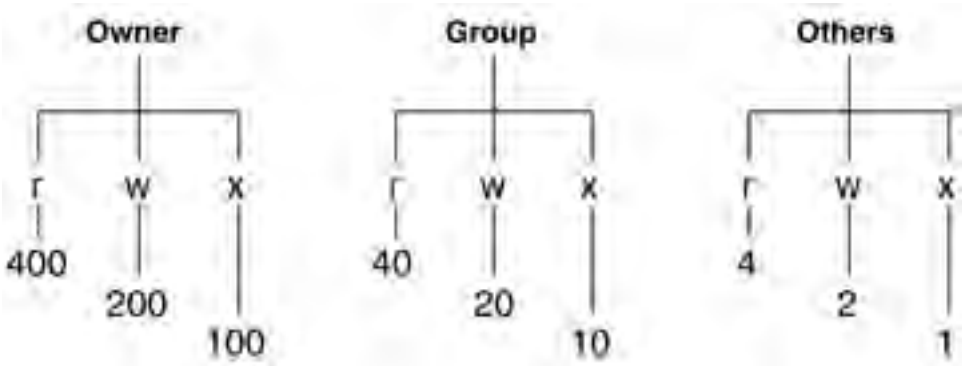


Figure 6.1 Translating binary to numeric notation.

The chmod command is then

```
$ chmod 664 newfile<Enter>
```

At first glance, the numeric format seems easier than the symbolic format. There are, however, some additional permissions that cannot be unset by using numeric notation. For example, you cannot remove the SETGID attribute on a directory by using numeric notation. You would have to explicitly remove the permission using symbolic notation. If you want to know more about SETGID and similar permissions, please check other information sources. Those topics are unfortunately beyond the scope of this introductory-level book.

Meanwhile, specifying permissions with numeric notation can occasionally result in safety messages. For example, suppose that you are the owner of a file and have no permissions on the file, but you try to remove it. The Linux system will then ask whether you want to override the protection setting on the file you want to remove. If you respond yes, Linux removes the file. The same would happen if you were a member of the group.

Table 6.4 summarizes the relationships between the symbolic notation and its binary notation and numeric notation counterparts. Use that table as you follow along with Example 6.3.

Example 6.3 chmod and Numeric Notation

In Example 6.2, Don Quixote changed the permissions on his home directory by using chmod with symbolic notation. How would he have done it with numeric notation instead? Here is how:

```
$ pwd<Enter>
/home/quixoted
$ cd ..<Enter>
$ pwd<Enter>
/home
$ ls -l quixoted<Enter>
```

Table 6.4 Numeric Parameters versus Other Notations

NOTATION	OWNER	GROUP	OTHERS
symbolic	rwx	rw -	r - -
binary	111	110	100
numeric/octal	7 (4 + 2 + 1)	6 (4 + 2 + 0)	4 (4 + 0 + 0)

```
drwx - - - - - -    5     quixoted   knights1   4096  Mar 07     08:11
     quixoted
$ chmod 770 quixoted<Enter>
$ ls -l quixoted<Enter>
drwxrwx - - -       5     quixoted   knights1   4096  Jul 10     08:11
     quixoted
```

Once again, we see that Don Quixote first checked the directory he was in and found that it was his home directory. Knowing he cannot change the status of that directory if he is in it, he changed to the directory */home* and verified that he was there. Then, he checked the original permissions on the */home/quixoted* directory and found them to be equivalent to 700 (that is, 400 + 200 + 100 + 0 + 0). He then changed the group permissions to allow anyone in knights1 to have read, write, and execute permissions (in other words, he added 40 + 20 + 10 = 70) on */home/quixoted*. Again, because he was already in */home*, he only needed to specify the relative pathname *quixoted*. Then, he checked the status of */home/quixoted* again and found that he had been successful. Thereafter, anyone in the group knights1 could save a file in Don Quixote's home directory.

Setting Permissions: The umask Command

The real default permission values for a file in Linux, using numeric and symbolic notation, are 666 and `-rw-rw-rw-`, respectively. Those for a directory are 777 and `drwxrwxrwx`, respectively. So, why have we listed different figures in Table 6.5? The answer lies in a masking subroutine called `umask`.

Notice how the differences in the default file modes all hinge on the application of the `x` permission for directories? As we mentioned earlier, if there were no `x` permissions, no one would be able to `cd` into the directories to see, let alone manipulate, other subdirectories or files found therein.

Table 6.5 File and Directory Values (Default umask Applied)

ENTRY	ID	SYMBOLIC NOTATION	NUMERIC NOTATION
File	Root	`-rw-r--r--`	644
	User	`-rw-r--r--`	644
Directory	Root	`drwxr-xr-x`	755
	User	`drwxr-xr-x`	755

Checking Permission Modes

The umask value is subtracted from 666 from a file permission standpoint and from 777 from a directory standpoint to determine the permissions that will be attributed to files and directories as that user creates them. The resulting 644 (666 minus 022) for file permissions means that the owner and the owner's group by default have read and write access to the files created, but other system users have only read access. The resulting 755 for directory permissions means that the owner and the owner's group have read, write, and execute permissions and that others have only read and execute permissions to any directories created.

The umask command is commonly called a file system option, which restricts access to certain files or directories. The command is actually a masking subroutine that creates and applies file modes, or permissions, on files or directories as they are created, however. If you want to use umask to check permission modes, the syntax is simply

```
$ umask<Enter>
```

Example 6.4 shows you how to check the permission mode with umask.

When a user logs in, his or her username, user number, group name, and group number are inputted to the /etc/profile program, which determines whether the user qualifies as a root user or as an ordinary user as far as umask is concerned.

If the user is an ordinary user or a root user, Linux by default assigns a umask of 022 (in earlier versions, the default umask for users was 002, and for root users the umask was 022). So, the file permissions are 644 and the directory permissions are 755. The owner has read and write permissions on created files, but others have only read permissions. On created directories, the owner has read, write, and execute permissions, and the owner's group and others have read and execute permissions.

Those who advocate higher security, especially on a network, recommend a default umask of 022 as the absolute minimum with alternatives such as 027 or even 077. Generally, the higher the umask number, the tighter the security.

Example 6.4 Checking Permission Modes with umask

After a discussion of system security with Freston, Lady Dulcinea decides that she wants to know what modes her files and directories are and will be. To do so, she must find out what her umask is:

```
$ umask<Enter>
022
```

To ensure that this later version of Linux is consistent whether you are a user or a root user, Freston shows Lady D. his default (root user) `umask`:

```
# umask<Enter>
022
$ ls -ld /home/dulcinea<Enter>
```

Changing Permission Modes

You can also use the `umask` command to specify the permission bits to be set on any new file or directory created in the same session only. Arguments are numeric:

```
$ umask numeric<Enter>
```

Subtracting the new `umask` parameter from the default values of 666 and 777 enables you to determine the new permission bits you want for files and directories, respectively.

Keep in mind the following items when changing permission modes:

- The `umask` value, and thus the permission bits, are applicable to only the session in progress and are not in effect after the owner/user logs out and logs back in. The default values apply upon a new login.
- The new `umask` value affects only files and directories created after the new `umask` value is specified. Thus, the command has no effect on existing files and directories. To alter permissions on existing files and directories, you must use the `chmod` command.

Remember that in general, the `chmod` command applies permissions and `umask` subtracts permissions. By manipulating these commands, users and system administrators can balance proper security with accessibility.

Creating Personal Directories

That said, we will demonstrate how to create a directory (a subdirectory, to be absolutely correct) with the appropriate permissions for storing personal documents and data. Such a directory would enable only the owner

to access it and to store information in it. Later, only the owner would access it and review and change subdirectory structure(s) and/or files within the personal directory. The `ls -l` listing of such a directory would resemble the following:

```
drwx-----  5     me    mygroup     4096 date time  mydirectoryname
```

The numeric mode of such a directory would be 700.

NOTE The permission modes proposed here are now the default modes for home directories created with the latest versions of Linux (kernel versions of approximately 2.4.5 and later). Thus, if you have the latest version of Linux, this information might be a little redundant. But if you are working with an older version, then this material might still be beneficial to you.

Be careful of personal directories, though—your home directory or any you might create. They can still be accessed and read by the root user (in other words, system administrators).

As we have learned before, the command for creating a directory is `mkdir`:

```
$ mkdir directoryname<Enter>
```

But there are two different ways to create such a directory:

- Altering your `umask` temporarily, creating the directory, and then changing your `umask` back to another value (likely, the original value). This method is the slower way.
- Creating the directory and then using `chmod` to change its permission bits. By a step or two, this method is the easier, faster way.

Consider the following points when creating personal directories:

- Users should maintain their own execute (x) values on their personal directories to ensure their own access.
- Users should be aware that although the personal directory is invisible to all other users, the root user can still access the personal directory and read the files contained in it.

Let's look at examples of both methods now.

Example 6.5 Creating a Personal Directory with umask and mkdir

Don Quixote wants to create a private directory called *personal* in his home directory. He remembers that it can be done with `umask`. Here is how he does it:

```
$ cd<Enter>
$ pwd<Enter>
/home/quixoted
$ umask 077<Enter>]
$ mkdir personal<Enter>
$ ls -l<Enter>
total 8
drwxrwxr-x5    quixoted   knights1   4096 May 26 10:18    admin
drwxr-xr-x5    quixoted   knights1   4096 Mar 07 08:11    Desktop
-rw-rw--r--1   panzasan   knights1    144 Jul 11 11:42    manuals
-rw-------1    quixoted   knights1   1116 May 26 10:18    mbox
drwxrwxr-x5    quixoted   knights1   4096 Apr 05 14:30    misc
drwxrwxr-x5    quixoted   knights1   4096 Jul 11 15:21    noble
drwx------5    quixoted   knights1   4096 Jul 24 10:05    personal
$ umask 022<Enter>
$ umask<Enter>
022
```

Example 6.6 Creating a Personal Directory with mkdir and chmod

You change the directory permissions with `chmod`:

```
$ chmod numeric directoryname<Enter>
```

This way, the owner/user can move to his or her home directory, create a subdirectory within it, change the mode of the new subdirectory, and then check the permissions on the new subdirectory. With this strategy, the user does not need to alter the `umask` and then return the `umask` to the previous value.

Let's see how Don Quixote could have used the `chmod` strategy to create the same private directory called *personal* within his home directory:

```
$ cd /home/quixoted<Enter>
$ mkdir personal<Enter>
$ chmod 700 personal<Enter>
$ ls -ld personal<Enter>
drwx-----/5     quixoted   knights1   4096 Jul 23 13:45    personal
```

Samples of Commands and Their Required Permissions

We provide Table 6.6 as a reference to help you ensure that you set the proper permissions on files and directories to accomplish what you want. Remember that to remove a file, you need write permission on the directory that contains the file. You do not need write permission on the file itself.

Exercises

1. Change to your *myscripts* directory. Display a long listing of the files in that directory. Note the owner and permissions for the files that you copied in the Chapter 5 exercises. Record the permissions for *mycal* and *mycat*:

Table 6.6 Required Permissions for Selected Files and Directories

COMMAND	SOURCE DIRECTORY	SOURCE FILE	TARGET DIRECTORY
`cd`	`x`	N/A	N/A
`ls`	`r`	N/A	N/A
`ls -l`	`r, x`	N/A	N/A
`mkdir`	`x, w` (parent)	N/A	N/A
`rmdir`	`x, w` (parent)	N/A	N/A
`cat, more`	`x`	`r`	N/A
`mv`	`x, w`	None	`x, w`
`cp`	`x`	`r`	`x, w`
`touch`	`x, w*`	N/A	None
`rm`	`x, w`	None	N/A

* The write permission is required in the source directory when using the `touch` command to create a zero-length file but not when using `touch` on an existing file to update the modification date.

```
$ cd myscripts<Enter>
$ ls -l<Enter>
```

2. Execute a long listing on the original *cal* and *cat* files in the */usr/bin* directory and compare the permissions to those in the *myscripts* directory:

```
$ ls -l /usr/bin/cat /usr/bin/cal<Enter>
```

3. Change the modification time of *mycal* and *mycat* in the *myscripts* directory. Check to see that the time changed:

```
$ touch mycal mycat<Enter>
$ ls -l<Enter>
```

   Describe another use for the `touch` command.

4. Execute the necessary commands so that you can reference the *mycal* file in the *myscripts* directory by the name *home_mycal* in your home directory.

```
$ ls mycal /home/teamxx/home_mycal<Enter>
```

   or

```
$ ln mycal ../home_mycal<Enter>
$ ls -l mycal<Enter>
$ ls -l /home/teamxx/home_mycal<Enter>
```

   or

```
$ ls -l ../home_mycal<Enter>
```

   Compare the detailed information for both files. Is there a difference? What is the link count?

5. Change the directory to your home directory. Execute *home_mycal*:

```
$ cd<Enter>
$ ./home_mycal<Enter>
```

NOTE **Because of default path limitations in Linux, simply typing** `home_mycal` **will not work.**

What does the output look like? Now, change permissions on the *home_mycal* file so that you, the owner of the file, have read-only permission. Try running the `mycal` command:

```
$ chmod 455 home_mycal<Enter>
$ ls -l home_mycal<Enter>
$ myscripts/mycal<Enter>
```

Can you do it? Why or why not?

6. Remove *home_mycal*:

```
$ rm home_mycal<Enter>
```

Did that remove myscripts/mycal? Why or why not?

```
$ ls -l myscripts/mycal<Enter>
```

7. Change the directory to the *myscripts* directory. Using symbolic notation with the `chmod` command, remove the read permission on the other permission bits from the *mycat* file. Check the new permissions:

```
$ cd myscripts<Enter>
$ chmod o-r mycat<Enter>
$ ls -l mycat<Enter>
```

8. Using octal notation, change the permissions on *mycat* so that the owner permission bits are set to read-only with no permission for anyone else. Check the new permissions:

```
$ chmod 400 mycat<Enter>
$ ls -l mycat<Enter>
```

9. Execute the `mycat` command against the *.bash_profile* file:

```
$ mycat ../.bash_profile<Enter>
```

or

```
$ mycat /home/teamxx/.bash_profile<Enter>
```

Did it work? What happened?

10. Make your home directory the current directory. Check to see whether you are in your home directory:

```
$ cd<Enter>
$ pwd<Enter>
```

11. Alter the permissions on the *myscripts* directory so that you have read-only access to it:

```
$ chmod u-wx myscripts<Enter>
```

or

```
$ chmod u=r myscripts<Enter>
```

or

```
$ chmod 455 myscripts<Enter>
```

12. Use a long list to check that you have set the permissions correctly:

```
$ ls -l /home/teamxx<Enter>
```

or

```
$ ls -ld myscripts<Enter>
```

13. Try getting a simple list of the contents of the directory. Try a long list:

```
$ ls myscripts<Enter>
$ ls -l myscripts<Enter>
```

    Did they work? Why or why not?

14. Try to execute *mycal*:

```
$ myscripts/mycal<Enter>
```

    Did it work? Why or why not?

15. Try to remove *mycal*:

```
$ rm myscripts/mycal<Enter>
```

    Did it work? Why or why not?

16. Return the permissions of *myscripts* back to its original form (`rwxr-xr-x`) and then remove *mycal*:

```
$ chmod 755 myscripts<Enter>
$ rm myscripts/mycal<Enter>
```

17. Experiment with other permission combinations. When you are through, make sure to change the permissions back to `rwx` for the owner.

See Appendix B for answers.

Quiz

NOTE **Questions 1, 2, and 3 are based on a file called *reporta*, which has the following set of permissions:** `-rwxr-xr-x`.

1. What is the file mode expressed in numeric (octal) notation?

2. Change the file mode to `-rwxr--r--` using the symbolic format.
3. Repeat the operation in Question 2 using the numeric (octal) format.
4. Assume that the *jobs* directory contains the *joblog* file. Using `ls -lR` should verify it:

```
$ ls -lR<Enter>
total 8
drwxr-xr-x5    perez knights2   4096  Jun 17 08:09  jobs
./jobs:
total 8
-rw-rw-r--1    perez knights2    100  Jun 18 13:22  joblog
```

   Can Nicholas, who is a member of the knights2 group, modify the *joblog* file?

5. This question is based on the following listing. Assume that the *jobs* directory contains the *work* directory, which in turn contains the *joblog* file:

```
$ ls -lR<Enter>
total 8
drwxrwxr-x5    perez knights2   4096  Jun 17 08:09  jobs
./jobs:
total 8
drwxrw-r-x2    perez knights2   4096  Jul 21 09:41  work
./jobs/work:
total 8
-rw-rw-r--1    perez knights2    100  Jun 18 13:22  joblog
```

   Can Nicholas, who is a member of the knights2 group, modify the file *joblog*?

6. This question is based on the following listing. Assume that the *jobs* directory contains the *work* directory (note that the permissions on *work* have changed), which in turn contains the *joblog* file:

```
$ ls -lR<Enter>
total 8
drwxr-xr-x     5    perez knights2  40961 Jun 17 08:09  jobs
./jobs:
total 8
```

```
drwxrwxrwx     2    perez knights2  40961 Jul 21 09:41  work
./jobs/work:
total 8
-rw-rw-r--     1    perez knights2   1001 Jun 18 13:22  joblog
```

Can Nicholas, who is a member of the knights2 group, copy the *joblog* file to his home directory?

See Appendix C for answers.

CHAPTER

7

Shell Basics

As soon as you log into a Linux system, you are exposed to a shell-generally, the Bash variant, which we discuss in detail in this chapter. The `shell` program is an interface to the operating system, translating your typed input (or input from other sources) into instructions for the operating system. DOS's `command.com` program does just about the same thing.

This chapter describes the ins and outs of the basic Linux shells. Although the focus is on Bash, we introduce elements of the `tsch` shell and include certain twists on the commands relevant to still other shells.

Knowing what the shell is doing with your input will make you a more efficient and effective Linux user and system administrator. When you see how you can customize your input with wildcards, metacharacters, and pipelining, you will begin to understand their power.

The Linux/UNIX Shells

The shell is but one of the interfaces to Linux/UNIX. There are others, such as the X Window System, which we will discuss in Chapter 14, "The Linux X Window System." Remember that any shell is just an interface program and is not the operating system itself. After you log in, typing your username and password, the operating system starts the shell program. On one hand, if you are using a command line-based shell, then the shell program responds by giving you a command-line prompt, such as

```
[username@hostname /directory]$
```

Or, if you are the root user, you will see

```
[root@hostname /root directory]#
```

As you have noticed, in this book we have shortened these prompts to $ and #, respectively.

On the other hand, the X Window System, as a shell, would present you with a GUI desktop image.

What does a shell do? It has three basic functions: a command interpreter, a job controller, and a comprehensive programming language.

First, let's look at the shell as a command interpreter. When you present a command to the shell, it looks at the command name to see whether it matches an internal shell command that it can execute itself. It checks also to see whether the command is an alias for another command. If the command is not an internal command, nor an alias for an internal command, the shell searches the hard disk for the program corresponding to the command name. If it finds one, the shell executes the program and feeds it the arguments (if any) that accompanied the command name entry.

What happens when the shell cannot locate a program? It responds with an error message, such as

```
shellname: commandname: command not found
```

As a job controller, the shell:

- Enables multiple task execution. On occasion (perhaps more often than we realize), it can enable some programs (also referred to as processes or jobs) to run in the background where they likely require

no user interaction, and it also enables another program to run in the foreground (which likely interacts with the user).

- Switches between the foreground processes and background processes as well as between other multiple tasks as necessary
- Suspends jobs without losing track of where the processes stopped so that they can begin again at the same point
- Enables programs to be piped to achieve complex results with single but perhaps complex commands. Remember, with pipelining (or *piping*, as it is more commonly called), we have seen the output of one program become the input to another, which starts automatically unless the user specifies that interaction is required.
- Enables you to write scripts (we will introduce you to shell script writing later in Chapter 15, "Linux Documentation and Support"). A shell script can invoke other shell scripts as long as their locations are in the search path.

Thus, the shell is also a comprehensive programming language that does not need a compiler because it interprets the logic itself. Keep in mind, however, that different shells use different syntaxes to execute shell scripts.

Types of Linux/UNIX Shells

The two major UNIX shells are the Bourne shell and the C shell. The Bourne shell is similar to the first UNIX shell developed by Bell Labs. The C shell was developed at the University of California at Berkeley and has a format similar to the C programming language. Another prominent shell is the Korn shell, developed at AT&T Bell Labs, which was based on the Bourne shell but also incorporated some C shell functionality.

Your copy of Linux likely contains several shells. Check your documentation for a complete listing as well as for an indication of which shells are automatically installed as part of the basic installation. Table 7.1 lists some typical shells.

You can check to see whether any of the preceding shells (or others) are on your installation medium, as well as which ones were installed with your Linux operating system, by viewing the listing in the */etc/shells* file.

To determine the default shell you log into, check the listing after your username in the */etc/passwd* file or check your environment by typing `env` followed by the Enter key to see which directory is listed as the `SHELL=`

Table 7.1 Typical Linux/UNIX Shell Programs

NAME	DESCRIPTION
ash	A Bourne shell clone that supports all Bourne shell commands but is smaller than Bash
ash.static	A version of ash that is not dependent on software libraries
bash	Stands for Bourne Again Shell, the default Linux shell. An sh-compatible interpreter, it reads standard input or input from a file; incorporates features from both Korn and C shells; and is intended to conform with IEEE POSIX Shell and Tools specifications.
mc	Midnight Commander, a visual shell. It looks like a file manager but has many more features.
pdksh	A reimplementation of the Korn shell intended for interactive and shell script use
rsh	The restricted shell used for network operations
sash	A statically linked shell that contains simplified versions of some basic commands. It is useful for system recovery.
tcsh	An enhanced version of the C shell with additional features and fancier prompts. In Linux, this shell is the usual choice as the alternative to Bash.
zsh	An enhanced version of the Bourne shell with most Korn shell, Bash, and `tcsh` features (and more)

`variable` value. Although Bash is the usual default shell, you can specify which shell you want to use—either by default or for a specific process or task. On some systems, you can change your shell with commands such as `chsh` or `passwd` with the `-s` option. For further help, check your information sources.

Example 7.1 Which Is the Login Shell?

Freston, an experienced shell programmer, is contemplating changing his login shell. But first, he would like to see what shell he logs into by default. He types

```
# cat /etc/password<Enter>
```

Freston's one-line password profile,

```
root:x:0:0:root:/root:/bin/bash
```

appears near the top of */etc/passwd* while the other users appear near the bottom. The ordinary users' profiles are similar to Freston's. For example, here is the one belonging to Nicholas the barber:

```
nicholas:x:606:604:/home/nicholas:/bin/bash
```

Example 7.2 Listing the Available Shells on Your Linux System

Freston continues his shell investigation by checking to see what shells are available on his Linux system. Now, he enters

```
# cat /etc/shells<Enter>
/bin/bash2
/bin/bash
/bin/sh
/bin/ash
/bin/bsh
/bin/tcsh
/bin/csh
/bin/ksh
```

Example 7.3 Changing Shells

Freston decides that he would prefer using the `tcsh` (enhanced C shell) as his login shell. Here is how he changes to it:

```
# chsh root<Enter>
Changing shell for root.
Password: xxxxxxxx<Enter>
New shell [/bin/bash]: /bin/tcsh<Enter>
Shell changed
```

Freston knew from experience to enter the full path name */bin/tcsh* for his chosen login shell. He knows that if he were only to enter `tcsh`, the existing shell would reply `chsh: shell must be a full path name` and then dump him back to his prompt to start over again.

To check that the shell has indeed been changed, Freston again enters

```
# cat /etc/password<Enter>
```

This time, Freston's one-line password profile is

```
root:x:0:0:root:/root:/bin/tcsh
```

Command-Line Parsing

Earlier, we mentioned Linux/UNIX shells as command interpreters. Here is what the shell does as a command interpreter.

When a command is presented to the shell, it looks at the command name to see whether it matches an internal shell command that it can execute by itself. It also checks to see whether the command is an alias for another command. If it is not an internal command and not an alias for an internal command, the shell searches for the program according to the command name submitted on that part of the hard disk defined in its `PATH` environment variable. (See Example 11.1 for an example of the `PATH` variable.) If it can find the program somewhere in its specified `PATH`, the shell executes it and feeds it the arguments, if any, provided at the command line.

What if the program cannot be found in the specified `PATH`? The shell responds with an error message:

```
shellname: commandname: command not found
```

Please remember that the program might be on the hard disk but not within the `PATH`. If you suspect that this situation is the case, try using a command such as the following to find it:

```
$ whereis commandname<Enter>
```

(The `whereis` command and other related commands are discussed in Chapter 13, "Shell Programming.") If the shell tells you where the program is, re-enter the command name at the command line but use the absolute path name as well as the name of the program. For example, in later distributions of Linux, the `banner` command might have to be invoked by using

```
$ /usr/games/banner -w40 "hello friends"<Enter>
```

(This information is handy to remember when doing Exercise 14 at the end of this chapter.) Before the shell begins searching for the executable

(command) program in its various locations, it parses the command line. In other words, the shell effectively examines the command line in the following order:

1. Redirection (`<`, `>`, `2>`, `|`, `>>`, `2>>`)
2. Variable and command substitution
3. Wildcard expansion (`*`, `?`, `[]`)

After the shell has examined the command in the preceding order, it submits the command to the operating system to be executed.

Metacharacters and Wildcards

Metacharacters are characters that have a special meaning to the shell, although that meaning can vary from one shell to another. For that reason, you should never use them as part of a filename. An exception is the hyphen (-), which you can use as long as it is not the first character in the filename.

The following is a list of metacharacters:

```
~ ! # $ % ^ & * ( ) { } [ ] |  \ ; " ' < > ? /
```

Wildcards are a subset of metacharacters. They are used for a process called pattern-matching notation for files, or pattern matching for short. That means that Linux shells, like those of all UNIX operating systems, can reference more than one filename by using these symbols. Here are some examples of wildcards:

```
* ? ! [ ]
```

The Asterisk (*) as a Wildcard

For certain commands, you can include one or more asterisks (*) with a character or string of characters in a filename specification. The inclusion of the asterisk causes the shell—not the operating system—to initiate a type of pattern-matching process called wildcard expansion. The wildcard expansion process substitutes all possible filenames that meet the parameters specified by the characters included before or after the asterisk in the command sequence. The shell substitutes filenames containing zero to any number of additional characters as long as the specified character or string also appears.

Table 7.2 Examples of Asterisk Wildcard Expansions

COMMAND	EXPLANATION
`echo *a*<Enter>`	"list on the screen all filenames that contain the letter *a*"
`echo a*<Enter>`	"list on the screen all filenames that begin with *a*"
`echo *a<Enter>`	"list on the screen all filenames that end with *a*"
`echo a*a<Enter>`	"list on the screen all filenames that begin *and* end with *a*"

Table 7.2 lists some examples with explanations of common types of asterisk wildcard expansion.

Example 7.4 provides a couple of illustrations of normal asterisk wildcard expansion.

The exception to wildcard expansion is that there will *not* be a match to any files that begin with a single dot; that is, to any hidden files. Why not? The answer is, mostly for security reasons. Those files are intentionally hidden from prying eyes and fingers. They should not be exposed or even jeopardized by being included (perhaps innocently and inadvertently) to processes invoked by wildcard expansion, which introduces a certain lapse of user or administrator control. Also, if single-dot matches were allowed, the directories called "." and ".." (that is, the current and parent directories, respectively) would also be fair game to whatever processes would be invoked. That could be very dangerous with certain commands (for example, `rm`).

Example 7.5 provides an example of the exception.

Now, wouldn't you know it—there are exceptions to the exceptions here. A match will be made to a single dot string if the single dot is included as part of the search string.

Example 7.6 provides an example to the "exception to the exception."

These principles are easier to grasp through examples. Let's look at Examples 7.4, 7.5, and 7.6. But first, those examples will be more meaningful if you take the time beforehand to create the following files with a text editor such as `vi` (be careful if you are not familiar with European-style phone numbers):

- Filename: */home/gutiejua/admin/phone#s/RFI_Tel1*

```
Madrid Head Office - +34 91 555 12 34
Travel Coordinator - Madrid - +34 91 555 00 01
Curate's Office - +34 91 555 23 45
Barber/Surgery/MASH - +34 91 555 45 67
```

```
La Mancha Field Office (Noble Deeds) - +34 958 55 34 56
El Toboso Field Office (Executive Suites) - +34 958 55 56 78
Valencia Field Office (Citrus Orchards) - +34 96 555 78 90
Valencia Warehouse - +34 96 555 89 01

Sevilla Field Office (Citrus & Other Products) - +34 95 555 90 12

Stable/Spa - Toledo (Trainers, Rozinante, Dapple et al) - +34 95 555
67 89
```

- Filename: */home/gutiejua/admin/phone#s/RFI_Tel2*

```
Group: Knights3

A. Lorenzo (aka Lady Dulcinea) - Tel: +34 91 555 01 23
Cell: +34 61 666 12 34
Casa: +34 958 55 19 01
J. Gutierrez - Tel: +34 91 555 01 24
Cell: +34 61 666 46 81
Casa: +34 958 55 13 15

Group: Knights1

D. Quixote de La Mancha - Tel: +34 91 555 13 57
Cell: +34 61 666 24 68
Casa: +34 958 55 01 03
S. Panza - Tel: +34 91 555 35 79
  Cell: +34 61 666 46 80
  Casa: +34 958 55 13 15

Group: Knights2

Perez - Tel: +34 91 555 23 45 or +34 91 555 23 46
Casa: +34 958 55 15 17
Nicholas - Tel: +34 91 555 45 67 or +34 91 555 45 68
Cell: +34 61 666 68 02
Casa: +34 958 17 19

Others:

Freston - Tel: +34 91 555 91 35
Pager: +34 91 666 8024
Cell: +34 61 666 0246
Casa: N/A
```

- Filename: */home/gutiejua/admin/phone#s/RFI_Tel3*

```
Emergencies - +34 91 800 09 87
Human Resources - +34 91 900 55 87
```

```
Fax/Mail Room - +34 91 555 32 10
Web Site Design/Management - +34 958 55 19 21
```

Example 7.4 The Asterisk (*) Wildcard

Intending to refine the telephone list files, Juana creates a new */tempwork* subdirectory under her home directory. Then, she copies the *RFI_Tel* files from her */admin/phone#s* subdirectory to the new */tempwork* subdirectory, where she will work on them:

```
$ cd<Enter>
$ mkdir tempwork<Enter>
$ cd admin/phone#s<Enter>
$ cp R* ../../tempwork<Enter>
```

To be sure they are copied to the right location, she goes to her new */tempwork* subdirectory and echoes all files whose filenames begin with `RFI` to the screen:

```
$ cd /home/gutiejua/tempwork<Enter>
$ echo RFI*<Enter>
RFI_Tel1 RFI_Tel2 RFI_Tel3
```

The files are all copied correctly, and now Juana's portion of the RFI directory structure looks like Figure 7.1.

Lady Dulcinea is passing by at that moment. She suggests that Juana should try another asterisk command, such as this one:

```
$ echo *_*<Enter>
RFI_Tel1 RFI_Tel2 RFI_Tel3
```

Example 7.5 An Exception to the Asterisk (*) Wildcard Expansion

Perez wants to check his operating environment. To list his hidden *.bash* files, located in his home directory, he mistakenly enters

```
$ echo *bash* <Enter>
*bash*
```

It surprises him that the shell just echoes his input back to the screen but does not list any files. He will call Freston for help.

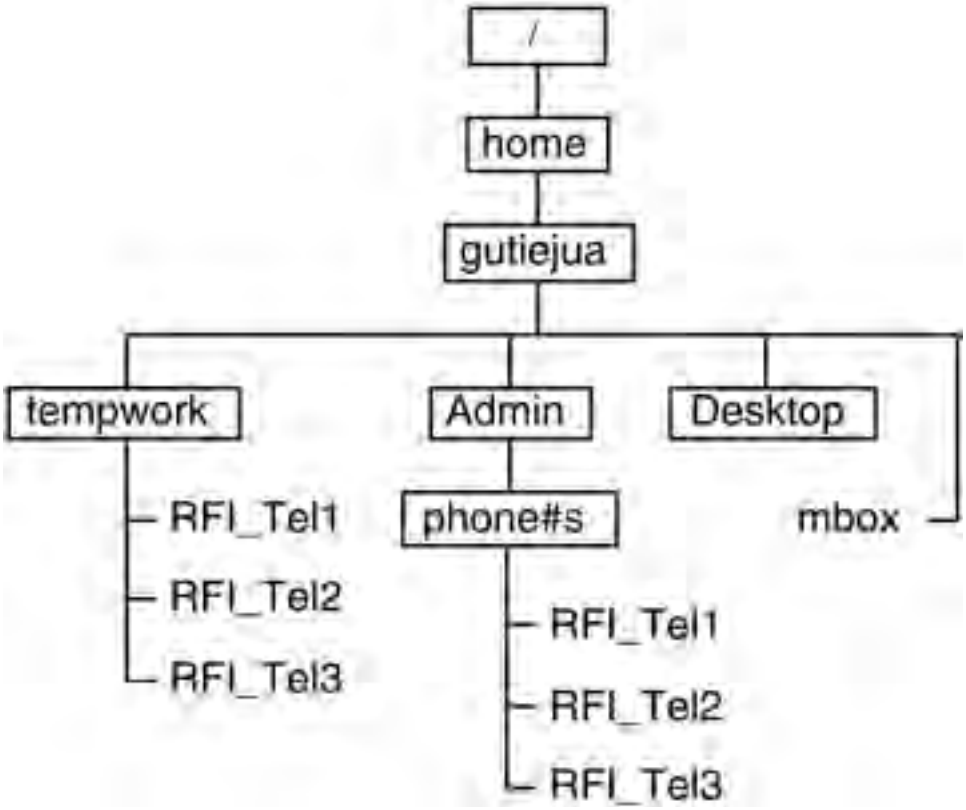


Figure 7.1 Juana's portion of the RFI directory structure.

Example 7.6 An Exception to the Exception to Asterisk (*) Wildcard Expansion

Upon receiving the call for help (see Example 7.2), Freston advises Perez to enter the following in his home directory:

```
$ echo .bash* <Enter>
.bash_history .bash_logout .bash_profile .bashrc
```

Perez is rewarded by a listing of the names of all his hidden *.bash* files.

The Question Mark as a Wildcard

A question mark (?) is another wildcard that you can use in a filename specification. The shell substitutes filenames that have only a single character in the same position as the question mark, as shown in Example 7.4.

Thus, the shell still does wildcard expansion, but the expansion goes to only one character instead of from zero to any number of characters.

As with the asterisk wildcard, the exception is that matches will not occur to files beginning with a dot (.). Example 7.8 illustrates the exception.

From our experience with the asterisk wildcard, we could predict quite naturally that there would be an "exceptions to the exception" here, too. A match will be made to a single dot string if the single dot is included as part of the search string. Example 7.9 illustrates this situation.

Example 7.7 The Question Mark (?) Wildcard

Juana has found another way to list her telephone list files. She goes to her */home/gutiejua/admin/phone#s* subdirectory and enters

```
$ ls RFI_Tel?<Enter>
RFI_Tel1 RFI_Tel2 RFI_Tel3
```

Still looking for a technique that would yield faster typing, she tries

```
$ ls ???_????<Enter>
RFI_Tel1 RFI_Tel2 RFI_Tel3
```

Example 7.8 An Exception to the Question Mark (*) Wildcard Expansion

Perez wants to check his operating environment. To list his hidden *.bash* files, located in his home directory, he mistakenly enters

```
$ echo ?bash* <Enter>
?bash*
```

This time, it does not really surprise him that the shell just echoes his input back to the screen and does not list any files. But instead of calling Freston, he recalls what happened with the asterisk exceptions and perseveres on his own. See Example 7.9.

Example 7.9 An Exception to the Exception to the Question Mark (*) Wildcard Expansion

Perez, remembering what happened when he was working with the asterisk wildcard, enters the following in his home directory:

```
$ echo .bash_???????<Enter>
.bash_history .bash_profile
```

Well, at least he sees two of the four possible files listed. But, he decides that this method is really not an efficient way to obtain a comprehensive listing of hidden *.bash* files.

Square Brackets for Lists

When you want to match only one of several characters, you use the square brackets. Similar to the ? wildcard, the position defined by the brackets is expanded by the shell but to only one character. Unlike the ? wildcard, however, only filenames with the same characters, as specified between the brackets, are used as input to the command. Again, this process is easier to see in action, so refer to Example 7.10.

Example 7.10 Using [ ] for Lists

Juana has found yet another way to list her telephone list files. She goes to her */home/admin/phone#s* subdirectory and enters

```
$ ls RFI_Tel[123]<Enter>
RFI_Tel1 RFI_Tel2 RFI_Tel3
```

Example 7.11 An Exception to the Square Bracket [ ] Wildcard Expansion

Perez is still listing his hidden *.bash* files, which are located in his home directory. He mistakenly enters

```
$ echo [.]bash* <Enter>
[.]bash*
```

Again, it is no surprise to him that the shell just echoes his input back to the screen and does not list any files. Again, he perseveres on his own. See Example 7.12.

Example 7.12 An Exception to the Exception to the Square Bracket [ ] Wildcard Expansion

Perez, remembering what happened with his previous asterisk and question wildcards, enters the following in his home directory:

```
$ echo .bash[-_]*<Enter>
.bash_history .bash_logout .bash_profile
```

Now, he sees three of the four possible files listed. Still, he observes, this way is not an efficient method for obtaining a comprehensive listing of hidden *.bash* files, either.

The Exclamation Point as a Wildcard

You use the exclamation point (`!`) within square brackets as a wildcard to exclude possibilities. Again, the position defined by the location of the brackets is expanded by the shell but to only one character, as shown in Example 7.13. Unlike the square list brackets, however, only filenames whose characters in that position are *not* members of the set of characters specified within the brackets are used as input to the command.

It might help to think of `!` as meaning *not*.

WARNING **Using wildcards with certain commands, especially the `rm` (remove) command, can be risky. You might get unexpected and unwelcome results. We recommend that you use the interactive form (that is, `rm -i`) when you want to include wildcards. This recommendation is especially helpful when you want to introduce recursive options (`-R`) that might alter the contents of directories all the way down the directory tree.**

Example 7.13 The Exclamation Point (!) and Bracket [ ] Wildcards

Continuing from Example 7.12, Perez makes a copy of his *.bash_history* file but renames the copy *.bash-history* (with a hyphen in the middle instead of an underscore). Then, he enters the same command he used in Example 7.12 to find all possible *.bash* files with either a hyphen or an underscore in the middle:

```
$ cp .bash_history .bash-history<Enter>
$ echo .bash[-_]*<Enter>
.bash_history .bash-history .bash_logout .bash_profile
```

Now, he decides to use the exclamation point wildcard to exclude the new *.bash-history* file, as such:

```
$ echo .bash[!-]*<Enter>
.bash_history .bash_logout .bash_profile
```

His exclamation point tells the shell to ignore the *.bash* files, which have hyphens in the middle. The result: He sees the same three files as he saw in Example 7.12. The shell, predictably, has ignored the *.bash-history* file.

Square Brackets for Ranges

You can use square brackets not only for lists but also for ranges. Here, too, the position defined by the location of the square brackets is expanded by the shell, but to only one character. But this time, only filenames whose characters in that position are members of the *range* of characters specified in the brackets are used as input to the command. This feature enables a much larger list of characters while reducing the amount of typing required to define the list. See Example 7.14.

Example 7.14 Square Range Brackets [-]

Juana, still experimenting with listing her phone-list files, finds that she can use the range brackets, too, as follows:

```
$ ls *[1-3]<Enter>
RFI_Tel1 RFI_Tel2 RFI_Tel3
```

Quoting Metacharacters to Disable Shell Interpretation

Quoting is the disabling, overriding, negating, or escaping (you will encounter *all of* these terms in reference to this concept) of the special meaning of other metacharacters, variables, or command names. It is called quoting because two of the three methods use single or double quotes.

Basically, these quoting techniques, equipped with their own sets of metacharacters, are used to override or disable the shell's normal interpretation of other metacharacters. Table 7.3 summarizes the functions of the quoting metacharacters and presents simple examples.

The quoting metacharacters cause the enclosed metacharacter to be interpreted literally (for example, the single quotes cause `$HOME` to be interpreted as just that—`$HOME`—because the dollar sign becomes just a dollar sign and not a single character command) or to not be expanded in the manner that the shell would have otherwise interpreted and acted upon. For example, `echo "*.*"` returns `*.*`, not a listing of all files whose filenames contain a single dot in any but the first position.

WARNING **Remember that you must use the single and double quotes in pairs. Otherwise, your secondary command-line prompt appears. (In most**

cases, an angle bracket [>] appears on the next lines, and the shell expects instructions. To return to the primary prompt, press Ctrl-C.)

Table 7.3 Quoting Metacharacters and Their Functions

METACHARACTER	INSTRUCTION TO SHELL	EXAMPLES
Single quotes (' ')	Ignore all metacharacters between quotes	`$ echo '$HOME'<Enter>` `$HOME`

Double quotes (" ")	Ignore all metacharacters between quotes except the dollar sign ($), backquote (`), and backslash (\)	`$ echo "$HOME"<Enter>`
		`/home/quixoted`
		`$ echo "*.*"<Enter>`
		`*.*`

Backslash (\)	Ignore the special meaning of the next metacharacter	`$ echo \$HOME<Enter>` `$HOME`

The backslash character deserves special scrutiny. If you are using a backslash to continue a command on another line, that backslash disables the special meaning of the pressed Enter key that immediately follows, which normally submits whatever is on the line to the shell for interpretation. By using the backslash there, the shell is being told, "The command isn't finished yet; ignore the next Enter and let me continue entering instructions." Then, the next Enter key submits the command to the shell.

Check out how we use the backslash in Example 7.15. Instead of the shell seeing three double quotes and presenting a secondary prompt (which was not our intention), it presents the statement followed by the double quotation mark (which was our intention).

Example 7.15 Quoting Metacharacters

```
$ echo "This is a double quotation mark \""<Enter>
This is a double quotation mark "
```

Standard Files: Redirection and Piping

For each process invoked within the Linux system, three files are automatically opened:

Standard input (`stdin`). The location where a command expects to find its input. Usually, that location is the keyboard.

Standard output (`stdout`). The location where the command expects to send its results. Usually, that location is the screen, but it can also be a file.

Standard error (`stderr`). The location where the command expects to send its error messages. Usually, that location is the user's terminal screen, but it can also be the system console.

The description of the default location is stored in a command's file descriptor table. You can change the locations, though (for example, screen, file, or console) by using redirection, which we will describe shortly.

File Descriptors

Each command or utility opens its own file descriptor to keep track of data files, inputs, outputs, and error messages. File descriptors differ depending on the command or utility that is running. File descriptor entries might refer to special device pointer files that in turn point to system devices, such as a terminal or a disk drive. See the file descriptors in Table 7.4. Note that descriptors and operators from 0 to 2 have default values, but a user can define others from 3 to 9. Meanwhile, please remember that in Linux/UNIX, not all filenames refer to data files. Also, for now, please do not worry too much about the device pointer numbers. We will be using them specifically later when we discuss combined redirection.

Consider the `cat` command (which we were introduced to in Chapter 5, "Using Files in Linux"), which is used to list the contents of files whose names we provided on the command line. If you do not specify a filename, `cat` simply echoes back to the terminal screen whatever you type on the command line before you press Enter. To `cat`, the default `stdin` is the keyboard (not a file), and `stdout` is the terminal screen.

Table 7.4 File Descriptor Parameters

DESCRIPTOR NAME	OPERATOR	DEVICE POINTER	DEFAULT DEVICE
Standard in (`stdin`)	<	0	Keyboard
Standard out (`stdout`)	>	1	Screen
Standard error (`stderr`)	2>	2	System console
User Defined	3< or 3>	3	No default device. Can be defined by user.
User Defined	4< or 4>	4	No default device. Can be defined by user.
.			
.			
User Defined	9< or 9>	9	No default device. Can be defined by user.

Example 7.16 Defining a File Descriptor

If Sancho wanted to define descriptor 3, say, to output to a specific file in his home directory, then he might use a syntax like the following:

```
$ exec 3> /home/username/Sancho.output.file<Enter>
```

Then, he could follow it up with a command such as:

```
$ date >&3<Enter>
```

or

```
$ ls -li >&3<Enter>
```

Input Redirection

Redirection enables you to specify where you want a program to expect its input to come from. In other words, if you do not want it to come from the default location, which for most commands is usually the keyboard, then

you use redirection to indicate where the program should go to find its input data. The syntax for redirected input is

```
$ command < filename<Enter>
```

NOTE **Some commands, such as `ls` and `rm`, do not accept redirected input. Other commands pause and request information from the user if you do not give them a file or data which which to work.**

Example 7.17 Using < to Redirect Input

Here, Sancho illustrates how the default standard input for the `mail` program is the keyboard:

```
$ mail quixoted<Enter>
Subject: Upcoming Journey - August
Good morning Senor! Dapple and I will be ready to leave anytime.
Signed, Sancho
<Ctr>l-d
Cc:<Enter>
```

At this point, let's assume that the *rozinante* file, which is a quick summary profile of Don Quixote's faithful steed, has already been created. Sancho will use redirected input to tell the `mail` program to send the *rozinante* file as the letter to *quixoted*.

```
$ mail quixoted < rozinante<Enter>
```

Sancho used the left angle bracket (<) to tell `mail` to take *rozinante* as the standard input, supplanting `mail`'s expected default standard input, which would be characters typed at the keyboard. This way, it is easier to create and format the *rozinante* file, correct typing errors, or add other embellishments to it before sending it to the Don.

With input redirection in place, though, `mail` does not return the `Subject:` and `Cc:` prompts that we saw before unless the `-sc` argument is added to the `mail` command.

Output Redirection

Redirection also enables you to specify where you want standard output to go; that is, if you do not want it to go to the default location (which for

most commands is usually the terminal screen). There are several types of redirection depending on what you want to do.

NOTE **Some commands produce output that cannot be redirected in this way. For example, print commands write to the printer, period.**

Destructive Redirection

If your intention is to add data to an existing file or to create a file for the data "on the fly," then you can use destructive redirection. It is called "destructive" because, if the target file already exists *and* it already has data in it, that data will be overwritten.

The syntax is

```
$ command > filename<Enter>
```

Example 7.18 illustrates the default standard output of the `ls` command followed by the instruction to `ls` to send its output to the *phone_no.out* file. If the file does not yet exist, it is created. If the *phone_no.out* file already exists, its contents are overwritten (called destructive redirection).

Example 7.18 Destructive Output Redirection Using >

Juana, when she lists the names of her telephone list files, illustrates the default behavior (especially the standard output, which is what we will concentrate on) for the `ls` command. Typically, the standard output (`stdout`) is her terminal screen. First, she changes to her */gutiejua/admin/phone#s* subdirectory. Then, she lists the contents with `ls`:

```
$ cd /gutiejua/admin/phone#s<Enter>
$ ls<Enter>
RFI_Tel1 RFI_Tel2 RFI_Tel3
```

Table 7.5 lists the default descriptors and behavior for the `ls` command as illustrated by what Juana has just done. Because these are default attributes, the Status, at this point, is "unchanged" for all descriptors.

Juana will now instruct the `ls` command to redirect its output to the *phone_no.out* file. If the *phone_no.out* file exists, it will be overwritten. If it does not exist yet, it will be created:

```
$ ls > phone_no.out<Enter>
```

Table 7.5 Typical Descriptor Table for the `ls` Command

DEVICE POINTER	OPERATOR	DESCRIPTOR NAME	DEFAULT DEVICE	STATUS
0	<	stdin	keyboard	unchanged
1	>	stdout	terminal screen	unchanged
2	2>	stderr	terminal screen	unchanged

Table 7.6 The `ls` Command's Modified Descriptor Table

DEVICE POINTER	OPERATOR	DESCRIPTOR NAME	DEFAULT DEVICE	STATUS
0	<	stdin	keyboard	unchanged
1	>	stdout	phone_no.out file	changed
2	2>	stderr	terminal screen	unchanged

Because of Juana's instruction, the shell recognizes that the `ls` command's descriptor table is effectively changed as shown in Table 7.6.

Nondestructive Redirection

If your intention is to add data to a file or simply avoid overwriting existing data, you can append output to an existing file (called *nondestructive redirection*). The syntax is as follows:

```
$ command >> filename
```

This type of nondestructive redirection is shown in Example 7.19.

Example 7.19 Nondestructive Output Redirection Using >>

Freston regularly and periodically uses the `who` command to check to see who is on the system. When he does so, he redirects `who`'s output to an existing text file called *who_survey.out* for future reference and for statistical purposes. To redirect the output and append it to that existing file, he tells the `who` command to send its output to the *who_survey.out* file, appending the output to any data that is already in the file:

```
$ who >> who_survey.out<Enter>
```

Table 7.7 The who Command's Modified Descriptor Table

DEVICE POINTER	OPERATOR	DESCRIPTOR NAME	DEFAULT DEVICE	STATUS
0	<	stdin	keyboard	unchanged
1	>	stdout	who_survey. out file	changed
2	2>	stderr	terminal screen	unchanged

Because of Freston's instruction, the shell recognizes that the `who` command's descriptor table is effectively changed as shown in Table 7.7.

cat and Redirected Output

Here is a handy, alternate use for output redirection. We know already that the `cat` command is normally used to list the contents of files. But if we combine `cat` and redirected output, we can create text files quickly without invoking a text editor. In fact, for small files, this procedure might be considered as good as, or even superior to, using a text editor. Here is the syntax, and we have included some lines of text:

```
$ cat > newfilename<Enter>
Line 1 of newfilename ...<Enter>
Line 2 of newfilename ...<Enter>
After this line 3, we want to end the file ...<Enter>
<Ctrl>-d
```

Note that we use Ctrl-D at the end of the file. Using Ctrl-C, on the other hand, would have canceled the command.

Here are two more examples of this type of redirection:

- This first example creates a zero-length file called *newfile*. If *newfile* already exists, this process removes the original contents of the file.

```
$ cat > newfile<Enter>
<Ctrl>-d
```

- This second example creates a file called *file3*, which contains the combined contents of *file1* and *file2*.

```
$ cat file1 file2 > file3<Enter>
<Ctrl>-d
```

Example 7.20 cat and Redirection for File Creation

The first part of Example 7.20 shows the typical use for `cat`: listing file contents. The second part uses `cat` with redirection to create a new file.

Freston uses the `cat` command in its normal capacity to list the contents of Juana's *RFI_Tel3* file:

```
$ cat /home/gutiejua/admin/phone#s/RFI_Tel3<Enter>
Emergencies - +34 91 800 09 87
Human Resources - +34 91 900 55 87
Fax/Mail Room - +34 91 555 32 10
Web Site Design/Management - +34 958 55 19 21
```

Now, he wants to create the beginning of his own specialized telephone list called *RFI_et_al_Tel#s*. He uses `cat` with redirection to create the new file:

```
$ cat > /root/RFI_et_al_Tel#s<Enter>
RFI Web Site Design/Management - +34 958 55 19 21
Fernando's Web Creation, Inc. - +34 91 555 66 22
Rodrigues' Internet Services, Inc. (Sevilla) - +34 95 555 22 66
<Ctrl>-d
```

Table 7.8 lists the file descriptors for the `cat > /root/RFI_et_al_Tel#s` example.

Error Redirection

Error messages are normally sent immediately to the screen or to the system console. Sometimes, however, you want to redirect error messages for various reasons. There are two types of error redirection:

- Redirecting error messages to a file
- Redirecting error messages to */dev/null*

We will discuss each one in some detail.

Table 7.8 The cat Command's Modified Descriptor Table

DEVICE POINTER	OPERATOR	DESCRIPTOR NAME	DEFAULT DEVICE	STATUS
0	<	stdin	keyboard	unchanged
1	>	stdout	RFI_et_al_Tel#s file	changed
2	2>	stderr	terminal screen	unchanged

Redirecting Error Messages to a File

Occasionally, you will want to collect error messages from scripts or shell programs for future reference or for troubleshooting. One application for this type of redirection is system error logging or network analysis.

Here is the syntax for destructive redirection of error output to a file:

```
$ command 2> filename<Enter>
```

The difference between error redirection and general redirection is the inclusion of the `2` before the right angle bracket symbol (>).

The following is the syntax for redirecting and appending error output (that is, for nondestructive redirection) to a file, which is a similar format to redirecting output in general:

```
$ command 2>> filename<Enter>
```

NOTE **It is important to remember that the operator symbols for standard error redirection are `2>` and `2>>`, with no spaces between the characters.**

Example 7.21 Redirecting Error Output to a File

Juana has provided Nicholas with a copy of her *RFI_Tel3* telephone listing file. He wants to compare it with Freston's new *RFI_et_al_Tel#s* file, the one Freston mentioned to him at lunch. See how the standard error output from Nicholas' next command, to `cat` *RFI_Tel3* and */root/RFI_et_al_Tel#s*, goes directly to the screen or system console:

```
$ cat RFI_Tel3 /root/RFI_et_al_Tel#s<Enter>
Emergencies - +34 91 800 09 87
Human Resources - +34 91 900 55 87
Fax/Mail Room - +34 91 555 32 10
Web Site Design/Management - +34 958 55 19 21
cat: /root/RFI_et_al_Tel#s: Permission denied
```

Nicholas now tries a variation of the previous procedure. He redirects the */root/RFI_et_al_Tel#s* error message to the *errfile* file (new or existing) and then uses `cat` to read the error message:

```
$ cat RFI_Tel3 /root/RFI_et_al_Tel#s 2> errfile<Enter>
Emergencies - +34 91 800 09 87
Human Resources - +34 91 900 55 87
Fax/Mail Room - +34 91 555 32 10
```

```
Web Site Design/Management - +34 958 55 19 21

$ cat errfile<Enter>
cat: /root/RFI_et_al_Tel#s: Permission denied
```

In Example 7.20, the */root/RFI_et_al_Tel#s* error message was sent directly to the *errfile* file. If *errfile* did not exist beforehand, it was created. If *errfile* already existed, its contents were overwritten. (As with output redirection, this overwriting capability is called destructive redirection.) The *RFI_Tel3* results were sent to the screen, but Nicholas had to invoke the `cat` command with *errfile* to see what happened to */root/RFI_et_al_Tel#s*' results. The file descriptor is as follows in Table 7.9.

Redirecting Error Messages to /dev/null

Here, we will learn how to redirect error messages to */dev/null*, commonly called the *bit bucket*—a special file that remains empty regardless of what you dump into it. It is used for all sorts of purposes, such as creating processes to check monitoring capability, creating network traffic for analysis, and disposing of error messages you might not want to be bothered with.

Here is the basic syntax for redirecting a generic `fileb` error message to */dev/null*:

```
$ cat fileb 2> /dev/null<Enter>
```

Example 7.22 Redirecting Error Output to /dev/null (the Bit Bucket)

Nicholas tries another variation of the procedure he has tried in Examples 7.20 and 7.21. He is very interested in `cat`'ing *RFI_Tel3* but is not as interested in succeeding with `cat`'ing */root/RFI_et_al_Tel#s*. So, he decides to send any error messages to */dev/null*:

Table 7.9 The Example 7.20 cat Command's Modified Descriptor Table

DEVICE POINTER	OPERATOR	DESCRIPTOR NAME	DEFAULT DEVICE	STATUS
0	<	stdin	keyboard	(unchanged)
1	>	stdout	screen	(unchanged)
2	2>	errfile	errfile file	(changed)

```
$ cat RFI_Tel3 /root/RFI_et_al_Tel#s 2> /dev/null<Enter>
Emergencies - +34 91 800 09 87
Human Resources - +34 91 900 55 87
Fax/Mail Room - +34 91 555 32 10
Web Site Design/Management - +34 958 55 19 21
```

Table 7.10 illustrates the file descriptor table for this second error message redirection.

Combined Redirection

Combined redirection is basically a combination of two or more redirections of input, output, and/or errors. There are two basic types:

- General combined redirection, where the ordering of commands does not matter
- Association, which is a type of combined redirection where the ordering of commands *does* matter

When we discuss association, we will actually be using the device pointer numbers that we have seen in the descriptor tables.

Typical Combined Redirection

The following is the syntax for a basic (but similar to what we have seen before), potentially destructive combined redirection:

```
$ command < infile > outfile 2> errfile<Enter>
```

Here is the syntax for nondestructive redirection using the same input file:

```
$ command >> appendfile 2>> errfile < infile<Enter>
```

Table 7.10 Example 7.1 cat Command's Modified Descriptor Table

DEVICE POINTER	OPERATOR	DESCRIPTOR NAME	DEFAULT DEVICE	STATUS
0	<	stdin	keyboard	(unchanged)
1	>	stdout	screen	(unchanged)
2	2>	/dev/null	/dev/null file	(changed)

In these two syntax examples, the order in which the redirections appear does not matter. As we have stated previously—to clarify why one redirection is called potentially destructive and the other is nondestructive—notice that, in the first example, we direct the command output to `outfile` and any error information to *errfile*. If *outfile* and *errfile* already contain information, that information is overwritten (and destroyed) with the new information from this new command. In the second example, the results from command execution are appended to *appendfile*, preserving whatever information might already be in the file. Similarly, error information, if any, is appended to *errfile*, preserving whatever information might be there as well.

Association

Association is another type of combined redirection. The best way to discuss it is to demonstrate it. Please have a look at Example 7.23. As we mentioned previously, our discussion of association will include the use of the device pointer numbers that we have seen in the descriptor tables.

Example 7.23 Redirection Using Association: A Simple Task

Let's revisit Nicholas' situation, which we were introduced to in Examples 7.21 and 7.22. He still wants to `cat` the two files (his own copy of *RFI_Tel3* and Freston's *RFI_et_al_Tel#s*). He wants to redirect the `cat` output to the *combo* file. He also wants any error messages to be sent to &1, which means that they should also be sent to the standard output location (because 1 is the device pointer for standard output, as we have seen in the descriptor tables) as well. Within this very command, however, standard output has just been redefined to mean the *combo* file. So, the error messages will be sent to *combo*, too. Here is the command, then:

```
$ cat RFI_Tel3 /root/RFI_et_al_Tel#s > combo 2>&1<Enter>
```

In the second half of this example, we will change the syntax and the order in which the redirections and associations appear to show that the order does matter. But in this part of the example, the error messages will be redirected to the standard output location—which is the terminal screen—and *then* standard output will be redefined as *combo*:

```
$ cat RFI_Tel3 /root/RFI_et_al_Tel#s 2>&1 > combo<Enter>
cat: /root/RFI_et_al_Tel#s: Permission denied
```

```
$ cat combo<Enter>
Emergencies - +34 91 800 09 87
Human Resources - +34 91 900 55 87
Fax/Mail Room - +34 91 555 32 10
Web Site Design/Management - +34 958 55 19 21
```

Error messages ultimately went to the screen. Then, Nicholas had to `cat` *combo* to see his original `cat` process results because that is where they went.

Example 7.24 Redirection Using Association: An Analysis Task with Proper Syntax

In this example, Perez wants to get a list of all the files in all the directories on the system. But once he has executed the command, he wants to be able to go back and analyze the results to see how effective his syntax was and to see which directories he has and does not have permission to access. He wants his successful long file listing *and* his error messages to be sent to the *list.file* file so that he can call that file up later, at his convenience, to check the results. Here is what he enters:

```
$ ls -l /*/* > list.file 2>&1<Enter>
```

After the command executes, all Perez sees is another shell prompt, because *all* results will be in *list.file*. Meanwhile, the file descriptor table for this example is shown in Table 7.11.

Example 7.25 Redirection Using Association: An Unsuccessful Analysis Task because of Improper Syntax

The next command presents an example of how Perez might *not* want to execute the same listing:

```
$ ls -l /*/* 2>&1 >list.file<Enter>
```

Table 7.11 The `ls` Command's Descriptor Table: Proper Syntax

DEVICE POINTER	OPERATOR	DESCRIPTOR NAME	DEFAULT DEVICE	STATUS
0	<	`stdin`	keyboard	(unchanged)
1	>	`./list.file`	`list.file` file	(changed)
2	2>	`./list.file`	`list.file` file	(changed)

This time, the association is out of proper order. The `ls` command is told to send error messages to &1, which, in this case, means to the same location as `stdout`. Later in the command, `stdout` is changed to the *list.file* file, but at that point, it is too late for the error messages. They will go to the original `stdout`, which by default for `ls` is the terminal screen. But what if there are lots of errors? Perez can go back later and look at his successful output because it will be in the *list.file* file. But how will he be able to analyze his errors? Most, if not all, of his errors would likely be lost; only a few of the last ones might still be displayed on the screen.

As predicted, Perez' terminal screen is bombarded with errors. Table 7.12 illustrates the descriptor tabulation.

In Table 7.12, please notice that the errors are sent to `stdout` and not to `stderr`. By default, error messages from `ls` are sent to `stderr`, which for `ls` is the terminal screen. And, by default, the `stdout` for `ls` is also the terminal screen. So, by telling `ls` to send error messages to `stdout` and not `stderr`, it becomes just a technical coincidence that the error messages still end up on the terminal screen. Of course, if the command had been properly configured as it was in Example 7.22, then the error messages would have been sent to *list.file*.

Connecting Commands with Pipes

In Linux, as in UNIX, you can connect two or more commands on a single command line by using the pipe symbol (|). This concept is called *pipelining* or *piping* and is one of the earliest and most revolutionary aspects of UNIX. Here are the only requirements:

- Any command to the left of a pipe must send its output to standard output (`stdout`).
- Any command to the right of a pipe must take its input from its standard input (`stdin`).

Table 7.12 The `ls` Command's Descriptor Table: Improper Syntax

DEVICE POINTER	OPERATOR	DESCRIPTOR NAME	DEFAULT DEVICE	STATUS
0	<	`stdin`	keyboard	(unchanged)
1	>	`./list.file`	`list.file` file	(changed)
2	2>	`stdout`	terminal screen	(changed)

We will discuss three types of command piping:

- Typical command piping
- Filter commands
- Tee commands

NOTE **If one of the commands in a pipe fails, the entire pipe fails. So, if you fail to get any input, you cannot necessarily conclude that the last command, or any particular command, is the faulty one.**

Typical Command Piping

The output from the command to the left of the pipe is not displayed because it is given straight over to the input of the command to the right of the pipe. You see the results of the final process only, as shown in Example 7.26. The syntax is simply

```
$ command1 | command2<Enter>
```

Example 7.26 Typical Command Piping

Juana will count the number of entries (in other words, files and directories) in her home directory by using command piping. Please refer to Figure 7.2, which depicts her portion of the RFI directory structure.

```
$ ls | wc -w<Enter>
4
```

Freston tells Juana that the piped command line she used in this example is the same as the following sequence of commands. Skeptical, Juana enters them:

```
$ ls > tempfile<Enter>
$ wc -w tempfile<Enter>
5 tempfile
```

Juana asks Freston, "If the piped command was equivalent to the sequence of commands, why did the word count differ from one to the other?"

Freston replies, "Simple, really. In the command sequence, you created a new file called *tempfile*, and its name got caught up in the word count. Check to see."

So Juana `cat`'s *tempfile* to check. When she is through checking, she deletes *tempfile*:

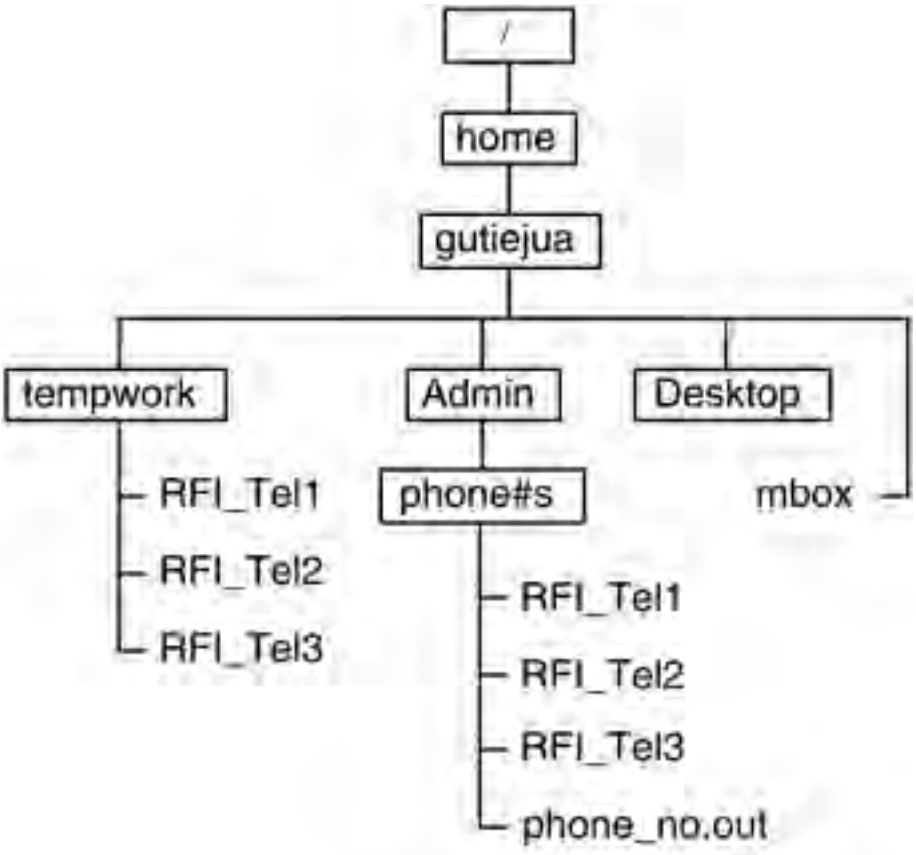


Figure 7.2 Juana's portion of the RFI directory structure.

```
$ cat tempfile<Enter>
admin
Desktop
mbox
tempfile
tempwork
$ rm tempfile<Enter>
```

Filter Commands

Here, we will show you how the various elements of a pipe execute in parallel.

A command is referred to as a *filter* if it can read its input from standard input, alter it in some way, and write the output to standard output. A filter can thus be used as an intermediate command between pipes. When

you use filters as intermediate steps in piped commands, you save processing steps and time. The syntax is

```
$ command1 | filtercommand<Enter>
```

Example 7.27 Using Filter Commands

Here, Juana wants to know quickly how many files in her */home/gutiejua/admin/phone#s* subdirectory begin with the letter `r`. She will use the `ls` command to list all the files in that subdirectory and then pipe the information to the `grep` command. The `grep` command (covered in more detail in Chapter 8, "Basic Linux Utilities") will find all filenames beginning with the letter `r`. The output of the `grep` command is then piped to the `wc -l` command. The `wc` command counts the number of lines of input and thus the number of files whose names begin with `r`:

```
$ cd /home/gutiejua/admin/phone#s<Enter>
$ ls | grep ^r | wc -l<Enter>
3
```

Referring to Figure 7.2, you can see that the three telephone list filenames—*RFI_Tel1*, *RFI_Tel2*, and *RFI_Tel3*—begin with `r`, so the preceding result, 3, is correct.

Thus, the `grep` process acted as a filter because it found the three filenames and passed them to `wc` while discarding the fourth filename.

Just for your information and practice, here is another way we could have expressed the piped command:

```
$ ls -R /home/gutiejua/admin | grep ^r | wc -l<Enter>
```

The difference, of course, is that in this version, we started in a different directory and gave the command the absolute path name.

Split Outputs: The tee Command

You can `tap` information from a command pipeline. The `tee` command acts as a filter for capturing a snapshot of information at a specific point in a pipe. The command puts a copy of the data at that point into a file and passes the original data to standard output, which is used through the standard input of the next downstream command:

```
$ command1 | tee filename | command2<Enter>
```

Thus, the `tee` command takes its input and routes it to two destinations:

1. By default to the terminal, unless you pipe its output to another command (such as in Example 7.25)
2. To the file of your choice

Meanwhile, the `tee` command does not alter the data. For this example, please refer to Figure 7.3.

Example 7.28 Tapping Information with the tee Command

Juana wants to combine the filenames of the three RFI telephone lists (*RFI_Tel1*, *RFI_Tel2*, and *RFI_Tel3*) into one file called *RFI_TEL*. She goes to her */home/gutiejua/tempwork* subdirectory before beginning the pipe command. She checks where she is, then she will check to make sure the proper files are there to begin with. Then, she will issue the `pipe`'d command, whose ultimate output will return a count of the number of filenames entered in her newly created file *RFI_TEL*:

```
# cd /home/gutiejua/tempwork<Enter>
#pwd<Enter>
/home/gutiejua/tempwork
# ls<Enter>
RFI_Tel1, RFI_Tel2, RFI_Tel3
$ ls | tee RFI_TEL | wc -l<Enter>
3
```

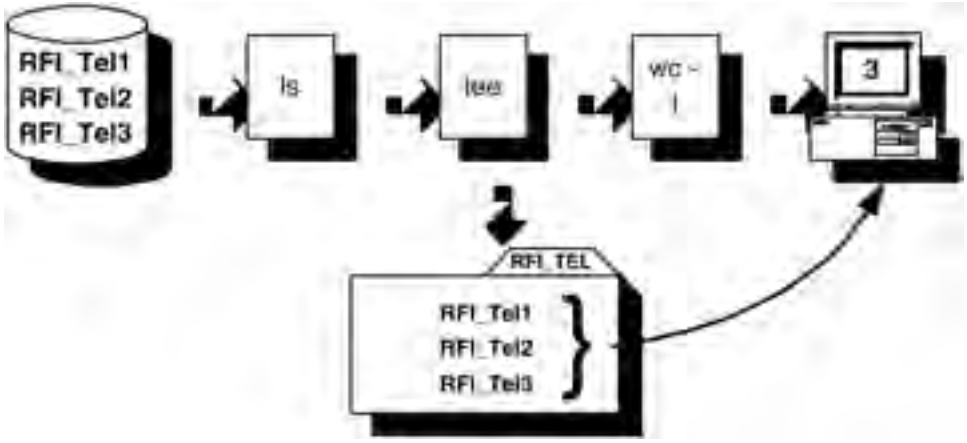


Figure 7.3 The tee command.

By the end of the `pipe`'d command's execution, there were actually four files (*RFI_Tel1*, *RFI_Tel2*, *RFI_Tel3*, and *RFI_TEL*), yet the `wc` command reported only three. If the pipes are truly parallel, shouldn't the `wc` command have reported four? Perhaps, but it reported three because it took longer for a file handle to be given to *tee* than it took for the entire series of piped commands to run.

We saw how the `tee` command took its input from `ls` and routed it to two destinations: to another command and to the *RFI_TEL* file.

Command Grouping with Semicolons

Contrary, perhaps, to the impression we have created so far, Linux/UNIX shells can cope with multiple commands on the same line. The semicolon (`;`) is the metacharacter that facilitates the process. The syntax is as follows:

```
$ command1 ; command2<Enter>
```

Placing several commands on the same line, each separated from the next by a semicolon, produces the same result as entering each command on a separate line, as shown in Example 7.28. Relationships between the output of one process and the input to the next are not necessary.

One reason you might consider using the semicolon in this way is because some commands (such as the `ls -R` command, which we will see in Example 7.29) might take a while to execute. Separating them with a semicolon allows the next command to execute without delay after the previous one has finished executing.

Example 7.29 Using ; for Multiple Commands

Freston has asked all the users to regularly and periodically (at the end of every month, actually) list the files and directories in their respective portions of the RFI directory structure. On July 31, Sancho performs this duty just before he logs out at the end of the day. Here is what he enters:

```
$ cd<Enter>
$ pwd<Enter>
/home/panzasan
$ ls -R > outfile<Enter>
$ exit<Enter>
```

Don Quixote performs the same procedure but issues all his commands on the same single command line as such:

```
$ cd ; ls -R > outfile ; exit<Enter>
```

Line Continuation with the Backslash

Line continuation is a shell feature that is useful when the options and arguments appended to a single command cause you to type past the length of a single command line. Simply type as far as you can and finish with a single backslash (\ ) followed by pressing Enter immediately. No other character can follow the backslash. The shell takes you to the next line and automatically presents you with an angle bracket (>), called the secondary prompt, which indicates that the shell is expecting more input from you:

```
$ command(s) continued_command\<Enter>
> continued_command(s)<Enter>
```

The secondary prompt does not interfere with the interpretation of the command. Do not confuse this prompt with the redirection angle bracket, which is a key you are actually required to type. See Example 7.30. You can change the secondary prompt from the angle bracket to another symbol or even to a phrase of your choice by changing the value in the PS2 variable (we will discuss environment variables in a later chapter).

Example 7.30 Using the Backslash (\) to Continue (or, Split) Commands

To compare his own telephone list file to Juana's *RFI_Tel2* file, Freston types the following:

```
# cat /home/gutiejua/admin/phone#s/RFI_Tel2 \<Enter>
> /root/RFI_et_al_Tel#s<Enter>
```

Shell History Commands

One of the shortcomings of working from the command line is the occasional (because of inadvertent typographical errors) or even frequent (for

example, if you are testing a procedure or performing several repetitive operations) need to re-enter commands that are identical or similar to those you have previously entered. In this section, we'll introduce three commands which, over time, will save you a lot of time and countless keystrokes: `history`, `fc`, and `!` (which is called "bang"). All three can reduce your frustration and speed your progress, especially on those busier days.

The history Command

The `history` command reads, numbers, and displays the text of the previous commands from the buffer and also from the *.bash_history* file in the user's home directory or from whatever file is named as a value for the `HISTFILE` variable (we discuss environment variables such as `HISTFILE` later in Chapter 11, "Shell Variables and the User Environment"). The default value is `HISTFILE=$HOME/.bash_history`, but you can change the default by adding or modifying `HISTFILE=filename` to your *$HOME/.bash_profile* file.

The maximum number of commands that `history` can access is the number of commands already in the buffer from the current session plus the number of commands stored in the *$HOME/.bash_history* file. The maximum size of that file is specified in the `HISTFILESIZE` variable. The maximum number of previous commands that `history` can display, however, is the number specified in the `HISTSIZE` variable. If no number is specified, the default is 17 (although many reference materials say 16).

The `history` command has several options and arguments. For example, `history -a` appends the current session's commands to the *.bash_history* file immediately and not upon your logout. Another example is `history n`, which displays only the previous `n` commands as long as `n` is less than or equal to the value of the `HISTSIZE` variable. For further information, check your information sources.

The fc Command

You can also use the `fc` command to display your command history as well as to edit those commands. Used with its options and arguments, `fc` can be handier than `history`. For example, `fc -l 5 120` lists all previous commands in your history list sequentially from number 5 to number 120. To list the same commands but in reverse order, use `fc -l -r 5 120`. The limitations on this maneuverability are that `HISTSIZE`, `HISTFILESIZE`, and other variable values must accommodate what you want to do.

You can also use `fc` to edit any of your previous commands by invoking the editor of your choice. An example is `fc -e vi 68`, which means "edit previous command number 68 with the `vi` editor." The shell responds by opening `vi` and displaying the command corresponding to number 68 on the top line. You do whatever editing you want and then exit `vi`, after which the shell executes the newly modified command.

Here is another example. Assume that your previous command number 68 is `set | more`. You want to change it quickly to `set | cat` and then execute it immediately without going into or exiting from a text editor. Just type the following:

```
fc -s more=cat 68<Enter>
```

The default text editor is automatically invoked and changes command number 68 to what you want, and then the shell executes it.

You might ask how you can control which text editor the shell uses. Set the `FCEDIT` variable to the name of your preferred text editor. If nothing is specified for `FCEDIT`, `vi` is used by default. One thing you will probably notice after using `fc` is that when you display your command history, you never see the `fc` command. All you see is the proper or revised command. The only time the `fc` command and its options are displayed is when command execution did *not* occur, which means that you made a mistake.

For other `fc` options and arguments, refer to your sources for additional information.

Re-executing Commands with the Bang Command

Regardless of the form of `fc` or `history` command you use to display your previous commands, you can re-execute any displayed commands by using the bang command, represented by the exclamation mark symbol (`!`) as follows:

```
$ !commandnumber<Enter>
```

or

```
$ !textstring<Enter>
```

The command number is obvious after the listing of the previous commands. The text string option can be handy, however, if you do not want to do a history listing and you remember the first few unique characters in the text of the desired command.

Recall that command number 68 was `set | more` in the preceding example. Assume that you want to set a variable, `FCEDIT=vi`, and you want to verify that the variable has indeed been set to `vi`:

```
$ FCEDIT=vi<Enter>
$ !68<Enter>
```

Your environment variables are then displayed. You could also have input the following and obtained the same result:

```
$ FCEDIT=vi<Enter>
$ !se<Enter>
```

You do not even need the whole word `set`, but only `se`. A word of caution, though: If another `set` command was in your history list, the shell would be confused, and you would likely end up with unanticipated results.

The bang command can save you a lot of typing. But when you examine a command history, you never find a `!` command per se, just the command that you invoked by using `!` (just like with the `fc` command). Again, the only exception is when you make a mistake.

The `!` command has other handy arguments and options, such as the inclusion of extra arguments with previous commands or the mixing of arguments from one previous command to another before re-execution. Again, consult your sources for more information.

NOTE **Some information sources indicate that a space is required between the exclamation point and the command number or text string, and some indicate that there should be no space between them. The correct syntax is no space between `!` and the command number or text string.**

Accessing Your Command History with the Up and Down Keys

The Up and Down keys can be powerful, too, for the convenience that they provide. You can use them to tap into your command history, starting with the RAM buffer where commands from the current session are saved, and

then proceeding into the hidden file called *.bash_history* in your *$HOME* directory (or whatever file has been specified for the `HISTFILE` variable to which we have referred previously). So, by using the Up or Down keys, you can recall to the command line commands that you have executed previously.

Naturally, you have to move at least one command up to be able to start utilizing the Down key, which moves you forward again in the history of your executed commands.

When you back up enough to enter the hidden *.bash_history* file, your initial position in that file is at the bottom, or the most recent command before your last logout. By specifying values for the environmental variables we described earlier in this chapter, you can control the size of the *.bash_history* file so that you can travel back a fair distance in your command history. Pressing the Up key once you reach the top of *.bash_history* however, results in a shell error "beep."

The benefit, then, of using the Up and Down arrow keys at the command line is that you can repeat procedures by recalling their commands and then pressing Enter to re-execute them. This feature can save you a lot of typing, especially when complicated syntax is involved. You can also do some sensitivity analyses by re-executing commands while changing options or arguments with each execution to compare results.

Exercises

1. Go to your home directory. Execute a simple `ls` command to list the non-hidden files in your home directory. Now, use the `ls` command with a wildcard character to list these files:

```
$ ls<Enter>
$ ls *<Enter>
```

   What is the difference in output of these two commands? Why?

2. Change to the */usr/bin* directory. List the files whose filenames begin with the letter `a`:

```
$ cd /usr/bin<Enter>
$ ls a*<Enter>
```

3. List all two-character filenames:

```
$ ls ??<Enter>
```

4. List all filenames that begin with the letters `a`, `b`, `c`, or `d`:

```
$ ls [abcd]*<Enter>
```

   or

```
$ ls [a-d]*<Enter>
```

5. List all filenames except those beginning with `c` through `t`. This list will be long, so you might want to pipe the output to `more` or to `less`:

```
$ ls [!c-t]* | more<Enter>
```

   Did you get any filenames that you did not expect? If so, do you know why?

6. Using all three methods of quoting, `banner` the literal symbol *:

```
$ banner '*'<Enter>
$ banner "*"<Enter>
$ banner \*<Enter>
```

   Why do all three work?

7. Make sure that you are in your home directory. Create a directory named *quoting* in your home directory:

```
$ cd<Enter>
$ pwd<Enter>
$ mkdir quoting<Enter>
```

8. Change to the *quoting* directory. Create a zero-length file named *filea* in that directory. Create a variable named `n` and set it to the value `hello`. Test what you did by displaying the contents of *quoting* and the value of `n`:

```
$ cd quoting<Enter>
$ touch filea<Enter>
$ n=hello<Enter>
$ ls<Enter>
$ echo $n<Enter>
```

9. From the *quoting* directory, execute the following five commands. Record the output:

```
$ echo '* $n `ls`'
$ echo "* $n `ls`"
$ echo \* \$n \`ls\`
$ echo * $n `ls`
$ echo * $n "ls"
```

10. Return to your home directory:

```
$ cd<Enter>
```

11. Using the `cat` command and redirection, create a file called *junk* containing a few lines of text. Use Ctrl-D at the beginning of a new line when you have finished entering text and want to return to the shell prompt, `$`:

```
$ cat > junk<Enter>

Type several lines of junk for this file.<Enter>
When you're finished, press the Enter key<Enter>
to go to a new line and then press Ctrl-d<Enter>
to return to the shell prompt.<Enter>
<Ctrl>-d
```

12. Append more lines of text to the file that you have created by using the `cat` command and redirection:

```
$ cat >> junk<Enter>
Type some more lines and append them to <Enter>
junk. When you are finished, go to a new line
and press the <Enter> key and then the Ctrl-d keys. <Enter>
<Ctrl>-d.
```

NOTE **Remember that there are no spaces between the (file append) angle brackets in the command line.**

13. Mail the *junk* file to yourself. Wait several seconds, and then open your mail, delete *junk*, and quit the program:

```
$ mail username << junk<Enter>
$ mail<Enter>
? t<Enter>
? d<Enter>
? q<Enter>
```

14. Using the `ls` command, list the files in your current directory:

```
$ ls<Enter>
```

    Make a note of the number of files.

15. List the files in your current directory, but this time redirect the output to the *temp* file:

```
$ ls > temp<Enter>
```

16. Use the appropriate command to count the number of words in the *temp* file:

```
$ wc -w temp<Enter>
```

    Is this count the same as in Exercise 14? If it is not the same, why?

    Display the contents of *temp*:

```
$ cat temp<Enter>
```

    Now remove the temp file:

```
$ rm temp<Enter>
```

17. This time, use a pipe to count the number of files in your current directory:

```
$ ls | wc -w<Enter>
```

    Was the result what you expected this time? Is it the same as in Exercise 14?

18. Use the command you created in Exercise 17, but this time insert a `tee` in the middle, trapping the result of the list in a file called *junk2*:

```
$ ls | tee junk2 | wc -w<Enter>
```

    Was the number displayed on the screen? Check the contents of *junk2* to make sure that it contains what you expected:

```
$ cat junk2<Enter>
```

19. Again, using piped commands:
    - List, in reverse order, the contents of your current directory.
    - Send the results of the reverse listing to a file named *junk3* and to a program to count the number of words in the reverse listing.
    - Append the final count to *junk3*.

Remember to use the append version of redirection. In this particular case, you might get unexpected results if you do not. It might not be a straight overwrite because the file is being used twice in the same command. Experiment if you are curious.

```
$ ls -r | tee junk3 | wc -w >> junk3<Enter>
```

20. A special file in the */dev* directory represents your terminal. Display the filename associated with your terminal. The output will be something like *tty0*, *lft0*, or *pts/x*:

```
$ who am i<Enter>
```

21. Repeat the command from Exercise 19 with two exceptions:
    - Rather than using *junk3*, `tee` the output to the special file that represents your terminal.
    - Do not append the results of the `wc` command to *junk3*. Display the count at your terminal:

```
$ ls -r | tee /dev/lft0 | wc -w<Enter>
```

22. On the same command line, display the date, who is logged in, the name of your current directory, and the names of the files in your current directory:

```
$ date ; who ; pwd ; ls<Enter>
```

    Do these commands have any relation to each other?

23. The primary purpose of this exercise is to use line continuation with a command that is too long to fit on one command line. The secondary purpose is to test what you have learned so far by letting you create a very long command string. You can choose to break the line anywhere you like. When completed, test your output by displaying the contents of the files that were created. This output should be one long command connected by pipes and redirection:

    - Do a long listing of the files in your home directory, including hidden files.
    - Capture the output to a file named *reverse.listing* and send the same output to a program that will count only the number of words.
    - Capture the number of words and place the number in four files named *file1* through *file4*.
    - Finally, send the output to a program to count the number of lines in *file4*, and redirect that number to a file named *file5*:

```
$ ls -al | tee reverse.listing | wc -w | tee file1 \<Enter>
> | tee file2 | tee file3 | tee file4 | wc -l > file5<Enter>
```

See Appendix B for answers.

Quiz

1. Put the following command-parsing steps in the proper logical order for the shell to interpret them correctly:

    - Command/variable substitution
    - Command execution

- Redirection
- Wildcard expansion

2. What will the following command match?

```
$ ls ???[!a-z]*[0-9]t
```

For Questions 3 through 8, assume the following:

- The home directory is */home/quixoted*.
- The current directory is */home/quixoted/docs*.
- The current directory contains three files: *aa*, *bb*, and *cc*.

For Questions 3 through 8, what are the results of the following commands?

3. `$ echo "Home directory is $HOME"`
4. `$ echo 'Home directory is $HOME'`
5. `$ echo "Current directory is 'pwd'"`
6. `$ echo "Files in this directory are *"`
7. `$ echo * $HOME`
8. `$ echo \*`
9. Identify the devices for this command:

```
$ cat file1<Enter>
```

standard input (0) _____

standard output (1) _____

standard error (2) _____

10. Identify the devices for this command:

```
$ mail tim < letter<Enter>
```

standard input (0) _____

standard output (1) _____

standard error (2) _____

11. Identify the devices for this command:

```
$ cat .profile > newprofile 2> l<Enter>
```

standard input (0) _____

standard output (1) _____

standard error (2) _____

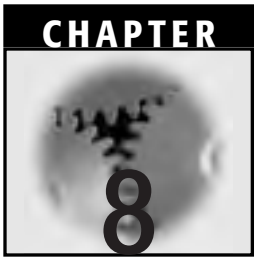
NOTE For Questions 12, 13, and 14, first create command lines to display the contents of a file called *filea* by using `cat`.

12. Place the output of the command in *fileb* and the errors in *filec*.
13. Place the output of the command in *fileb* and associate any errors with the output in *fileb*.
14. Place the output in *fileb* and discard any error messages (that is, do not display or store error messages).
15. What will the following command do?

```
$ banner hello > /dev/tty1<Enter>
```

See Appendix C for answers.

CHAPTER



8

Basic Linux Utilities

The commands covered in this chapter are useful for various practical administrative tasks, such as finding and manipulating files and the text within files. Specifically, we will discuss the `find` command, several `find`-like utilities, and the `grep`, `head`, `tail`, and `sort` commands. This chapter will not be our only exposure to `find`, however; it is only an introduction. We will use `find` again in Chapter 9, "Advanced Linux Utilities."

These commands are powerful and handy but also a bit more complicated than the commands we have discussed so far. If you keep their respective purposes, principles, and syntaxes in mind, however, with experience you will find that these utilities will enhance your efficiency and overall effectiveness.

Searching Directories for Files: The find Command

In this section, we will introduce the basic uses and syntax of the `find` command. The `find` command is a powerful utility because it can find files and automatically perform actions on those files. We will use Sancho Panza's portion of the RFI directory structure, as depicted in Figure 8.1, for the examples in this section and in the rest of the chapter.

The `find` program searches recursively downward through the file structure, starting in the directories you specify. The syntax is particular

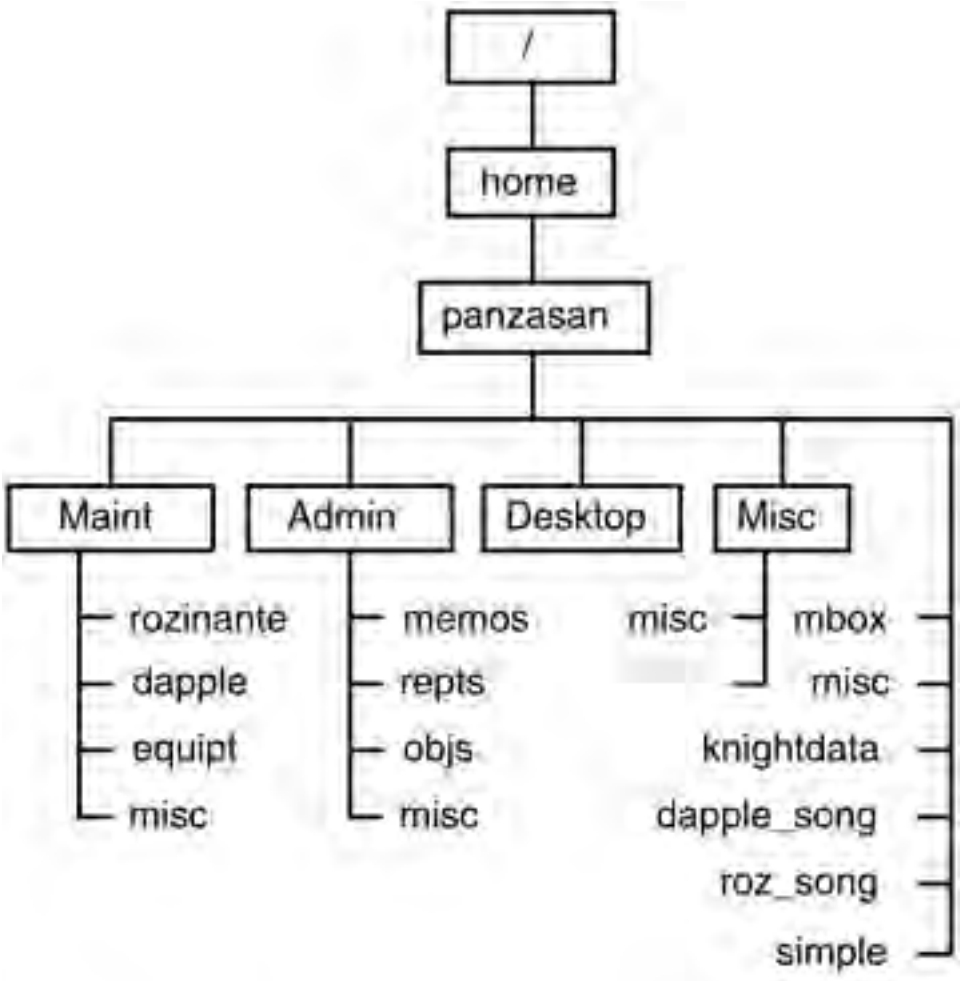


Figure 8.1 The find command; Sancho's directory structure.

(some believe that it is more like *peculiar*) and can generally be expressed as follows:

```
$ find [where to start looking?] [what to look for?] [what to do with it
when it's found?]<Enter>
```

The real syntax, though, is

```
$ find directory(ies) conditions<Enter>
```

The *conditions* could include or be called options, match criteria, actions, arguments, and all sorts of other designations. Nearly all information sources seem to have a different interpretation of what to call the required parameters, but their examples all look similar. Luckily, although no agreement exists on how to express the parameters for the `find` command, overall agreement does exist on how to use it.

Example 8.1 Using find with Conditions

Since we last looked at Sancho's directory structure, it has changed significantly. Recently, after some confusion while looking for maps and other data, Sancho has concluded that he relies on the filename *misc* maybe a little too much. To begin straightening things out, he wants to know how many times he has used it recently. Here is how he checks:

```
$ cd<Enter>
$ pwd<Enter>
/home/panzasan
$ find -name misc -print<Enter>
./misc
./Misc/misc
./Maint/misc
./Admin/misc
```

Sancho first changed directories to return to his home directory. Then, he invoked the `find` utility, which began searching at the top of his home directory. He also specified `-print` at the end of the command line, which is interpreted as printing to `stdout` (standard output; that is, printing to the terminal screen). If he had not specified anything, the output would still have printed to his terminal by default. In earlier versions of UNIX and UNIX-like operating systems, though, this situation was not the case. You had to explicitly tell the shell what to do with the output.

The `find` utility returned a list of four files called *misc*. But it also indicated Sancho's *Misc* subdirectory.

Two additional considerations are worthy of mention:

In the example, the `find` utility found the four *misc* files. Had there also been one or more `misc` subdirectories somewhere in Sancho's file structure, `find` would have returned a reference to them, as well. `find` would have returned references to only the subdirectories, however, not to files within them that are not called *misc*.

File and directory permissions apply to `find` as they do to other commands. If you have no business in certain directories or no permissions to certain files, `find` will not find them. You cannot use the command to circumvent permission requirements. If you lack appropriate permissions, you will receive an error message similar to the following on the terminal screen: `find: /directoryname: Permission denied`.

The find Command with a Noninteractive Single Action

Here, we can combine find with the execution of one more command (or, if you prefer, one more action) in tandem. The syntax is

```
$ find directory(ies) conditions -exec commandname options { } \;<Enter>
```

Note that because of the way we have coded the `find/exec` command combination, no interaction will occur between the processes and the terminal. This feature might or might not be desirable, as we will see in the next couple of examples.

Note also that the `\;` sequence must be part of the command line when `find` is combined with the `-exec` action. See Example 8.2. It must also appear when `find` is combined with the `-ok` action, too (we will discuss the `-ok` action in the next section).

Example 8.2 find with a Non-interactive Action

Sancho thinks that at least some of his various *misc* files should be deleted. But first, he wants to get some information about them. Instead of conducting `ls -l` several times (in other words, once for each directory that a

misc appears in), he will apply `find` with one non-interactive action as follows:

```
$ find . -name 'mi*' -exec ls -l { } \;<Enter>
-rw-rw-r--1   panzasan     knights1    21   Jul 15 08:48   ./misc
-rw-rw-r--1   panzasan     knights1    27   Jul 15 14:54   ./Misc/misc
-rw-rw-r--1   panzasan     knights1    32   Jul 20 11:37   ./Admin/misc
-rw-rw-r--1   panzasan     knights1    75   Jul 20 15:03   ./Maint/misc
```

Loosely speaking, the `find` command in Example 7.2 states, "Start in the current directory, and working downward, find all entries that begin with *mi*. As you find each entry, present its data on the screen in long list format."

Here is a breakdown of each element in the command:

- `.` (the dot) means to start in the current directory, which in this example is `/home/panzasan`, and move recursively through the file structure
- `-name` means that the names of the entries you want to find will follow
- `'m*'` indicates that the first character in each entry is `m`. The asterisk `( *)` indicates that subsequent characters do not matter. This part of the command is in single quotes because the asterisk is a metacharacter, and if it were not surrounded in single quotes, it would have a different effect on the command.
- `-exec` means that after finding each entry, execute the following command on it in turn
- `ls -l {}` requests a long listing of the entry. The `{}` argument means that as each entry is found, its name should be substituted here so that `ls -l` can be executed on it.
- `\;` is called an *escaped semicolon.* It indicates that we have reached the end of the `find` and `exec` command sequences.
- `Enter` submits the command sequence for parsing and execution

Again, because of the way we have coded the `find/exec` command combination, no interaction occurs between the process and the terminal. In this case, this situation is desirable because the second action (`ls -l`) is not really a drastic action.

In Linux and other more recent versions of UNIX, you can use `ls` as an option with `find`. Thus, you can use an abbreviated syntax to get the

same results as you did with the longer command from Example 8.2. Try the following:

```
$ find . -name 'm*' -ls<Enter>
```

The find Command with an Interactive Single Action

In the previous example, the second action was not drastic. When it executed, it did not impact the directory structure, contents or the file contents. But what if that second action was a little more drastic? What if, for example, Sancho had wanted to remove the entries that `find` found? What if he had not entered the appropriate options or arguments? What if he had not entered `m*` instead of `mi*`? He might have accidentally deleted his *mbox* and *memos* files.

So, in some cases, we see that interaction between the second process and the terminal might be useful. This section focuses on how to add such interactivity to the `find` command by adding the `-ok` option. This time, instead of just going ahead and performing the second command, Linux prompts the user for verification before proceeding, as shown in Example 8.3. The prompt asks a basic yes or no question; to respond, the user types `y`, `yes`, `n`, or `no` and then presses Enter.

The advantage to this interaction option is that the user has the ability to monitor the action and prevent inadvertent problems that might develop as a result of execution of the action. The syntax is as follows:

```
$ find directory(ies) conditions -ok commandname options { } \;
```

Consider using `-ok` with `find` in the following situations:

When your search pattern might not be or cannot be absolutely accurate (or absolutely surgical, if you are only trying to affect some entries and not all of them) and you do not want to affect any entries unnecessarily.

When your second command is fairly final. In Example 8.3, we are removing the successful `find` candidates, and that action is pretty final. If we were doing this action noninteractively and something were to go wrong, however, we would have to find a way to restore the entries we did not want to remove. This process could prove costly in some situations.

When the number of files to be located by `find` is relatively small. No one really wants to sit at a terminal for a long time typing `y` and Enter or `n` and Enter repeatedly. This activity is tedious, and tedious activities can lead to mistakes.

Table 8.1 lists some other common options that you can specify with the `find` command. For even more options, consult your information sources.

Example 8.3 find with an Interactive Action

Sancho is convinced now that his *misc* files should be deleted. He will use `find` in combination with a single action, but he will do it interactively.

Doing it interactively is a good thing—he is about to make a typographical error in his command sequence. But the interactivity will enable him to prevent the wrong files from being deleted:

```
$ find . -name 'm*' -ok rm { } \;<Enter>
< rm ... ./mbox > ? n<Enter>
< rm ... ./misc > ? y<Enter>
< rm ... ./Misc/misc > ? y<Enter>
< rm ... ./Admin/memos > ? n<Enter>
< rm ... ./Admin/misc > y<Enter>
< rm ... ./Maint/misc > ? y<Enter>
```

Table 8.1 Selected find Options

OPTION	DESCRIPTION
`-type c`	Find files of type `c` (character-special file). The type can also be `b` (block special file), `d` (directory), `f` (plain file), `l` (symbolic link), `p` (fifo or a named pipe), or `s` (socket).
`-size n[c]`	Find files containing `n` blocks, or if `c` is specified, `n` characters long (1 block = 4,096 bytes)
`-mtime +n|n|-n`	Find files that were last modified more than `(+n)`, less than `(-n)`, or exactly `(n)` days ago
`-perm nnn`	Find files whose permissions are set to octal number `nnn`
`-user username`	Find files belonging to the specified user
`-o`	Find files that match specified conditions
`-newer filename`	Find files that have been modified more recently than `filename`. (Note the similarity to `mtime`.)

If Sancho had not made the command interactive, then his *mbox* and */admin/memos* files would also have been deleted.

Using Additional Options with the find Command

This section includes four examples of `find` command options. Most of these options were described in the preceding section.

Example 8.4 find with a Line Continuation (Split Command)

Sancho remembers working on a profile of each of the two "beasts of burden" (Rozinante and Dapple) recently but has forgotten where he filed them. He knows they are both medium to small documents and that they are in his directory structure somewhere, so he enters

```
$ cd<Enter>
$ pwd<Enter>
/home/panzasan
$ find . -mtime -1 -type f -size +500c\<Enter>
>  -exec ls -l {} \;<Enter>
-rw-------1   panzasan   knights1   682  Jul 02 14:49  mbox
-rw-------1   panzasan   knights1  3930  Jul 02 08:32  ./.bash_history
-rw-r--r--1   panzasan   knights1   667  Jul 21 14:49  ./maint/rozinante
-rw-r--r--1   panzasan   knights1   993  Jul 22 09:44  ./Maint/dapple
-rw-r--r--1   panzasan   knights1   642  Jul 22 11:13  ./fleas
```

Here, he split the command so that one part is written opposite the primary prompt and the rest is written opposite the secondary prompt. We had Sancho perform this action to show how this command form works (and to show that it *will* work).

What is the command requesting? It says, "Starting from the current directory, find all files that are larger than 500 characters in length and that were modified within the last day. Then, as each successful file is found, create a long listing of it and present the long listing on the terminal screen." And, after a quick perusal of the response, Sancho sees that his two files are in his */home/panzasan/maint* subdirectory.

Example 8.5 find with Complex Options

Freston is randomly checking the permissions on files to see whether any staff members have altered their `umask` to 002 for file creation in their home directories during the past calendar week:

```
# cd /<Enter>
# pwd<Enter>
/
# find ./home -perm 664 -mtime -7 -print<Enter>
./home/perez/Quixote
```

In other words, in Example 8.5, Freston requested that the system find and display—again, from the current directory downward—the names of all files created in the past seven days with permissions set to `664` (that is, `-rw-rw-r--`).

Freston makes a note to call Perez and ask him why he has created the Quixote file with the permissions it has. Meanwhile, if you have any concerns regarding permission bits, please review the discussion in Chapter 6, "Linux File Permissions."

Example 8.6 find with the -o (OR) option

Sancho was working on profiles of both Rozinante and Dapple recently. Now, he has lost track of where he filed them. Luckily, he remembers their names. Here is how he finds them:

```
$ cd<Enter>
$ pwd<Enter>
/home/panzasan
$ find . -name rozinante -o -name dapple<Enter>
./Admin/rozinante
./Admin/dapple
```

In other words, Sancho requested that the system find and display the names of all files named *rozinante* or *dapple* from his current directory downward.

Example 8.7 find with Error Redirection

Freston was explaining to Sancho how he (Freston) was searching the whole directory structure for files pertaining to security that have the word *security* as part of their path name. He told Sancho how he (Freston) would do it:

```
# cd /
# find / -name "security" -print 2> errfile<Enter>
/etc/security
/usr/include/security
/lib/security
```

Interpreting his command, we see that Freston, in Example 8.7, requested that the system find, from the root directory downward (that is, through the entire system file structure), filenames containing the `security` text string as part of their path name. Then, the system was to display the successful path names on the terminal screen. Errors are not to be displayed on the screen but instead are to be listed in the *errfile* file in the current directory. If such a file does not exist, the command creates it.

Sancho, after a moment, tells Freston that he did the same thing and got the same result:

```
$ cd /
$ find / -name "security" -print 2> errfile<Enter>
/etc/security
/usr/include/security
/lib/security
```

Freston says, "Well, Sancho, although you (fortunately, even luckily) got the same output to the screen that I did, you didn't really get the same result. When I `cat`'d my *errfile*, I found it to be empty. So I'm confident that I was able to examine all directories. Is your *errfile* empty?"

Sancho checks and tells him, "No way! My *errfile* is full of errors! Many directories didn't allow me entry at all!"

The lesson here is to be careful with commands like this one. If the command does not appear to provide the expected output, or even if it does, it might still have created errors. But you will not know about the errors unless you check the *errfile* file. An ordinary user (as opposed to a superuser or a root user) is likely to encounter more errors in these situations due to a lack of permissions on directories or files.

Locating Commands: whatis, whereis, and which Commands

What if you are trying to execute a program and the shell tells you it cannot find the file or directory you refer to when you use the name of the command? Or, what if you are writing a program that will invoke a command

and you need the full path name of the command, but you do not know where it resides? Here are three quick `find`-like utilities that can help.

whatis

When you are not sure of a command name, when you are not sure of exactly what the command will do, or even when you want to resolve arguments among colleagues, the `whatis` is command can be helpful. The `whatis` command searches the `whatis` database, a file found in the */usr/bin* directory, for the command name that you want to investigate and then prints a brief summary of what that command does when invoked. The syntax for `whatis` is straightforward:

```
$ whatis commandname<Enter>
```

Be sure to use a complete word. No partial text strings are allowed. See Example 8.8.

Example 8.8 whatis

Sancho and the others have been advised to use the `grep` command to search for information within their files without having to `cat` them individually. Before he actually uses `grep`, Sancho decides to investigate it. He starts with `whatis` as follows:

```
$ whatis grep<Enter>
grep (1) - print lines matching a pattern
```

Thanks to `whatis`, Sancho at least has verified the correct command name to continue his search (if decides to do so). If he had the wrong name to begin with (such as "gresp" or something), he might have had to enter `whatis` several times to nail down the correct name. If more information is required, Sancho can search `grep`'s `man` pages or other information sources.

which

The `which` command is different from both `whatis` and `whereis`. It lists the path names of the files that will be executed if you run the specified command. Unlike `whereis` (which we will look at next), `which` searches your `PATH` variable. The major drawback to `which` is that it stops searching after it finds and reports on the first occurrence of the command name

that you specify. But an advantage to `which` is that if you are working in a C shell environment, it also checks the *.cshrc* file (if one exists) for aliases. Its syntax is as follows (see Example 8.9):

```
$ which commandname<Enter>
```

Example 8.9 which

Our intrepid Sancho continues his preliminary investigation of the `grep` command by trying `which`. It will show him at least one location of the executable `grep` file. He enters the following:

```
$ which find<Enter>
/bin/grep
```

whereis

The `whereis` command is a little different from `whatis`. You can use `whereis` to find commands, command sources, and manual pages. This command searches a standard built-in list of directories (*/bin*, */etc*, */usr/bin*, and */usr/local/bin*, among others) to find and print all matches of the specified command name. `whereis` does not search your own search path, however (that is, as spelled out in your `PATH` variable), so it might not find your individual shell scripts if they are in your local system directories or in your own *$HOME/bin* directory (assuming that you have created one). The syntax is

```
$ whereis commandname<Enter>
```

The `whereis` command, the way we used it in Example 8.9, is simplified compared with the syntax you will find in your information sources. They will also tell you that you can specify the type of file to look for, where to look for it, and so on. So, you can see that `whereis` is not quite as simple as `whatis`. As always, consult your information sources for further details. See Example 8.10.

Example 8.10 whereis

Sancho completes his investigation of `grep` by using `whereis` to look for `grep`'s executable file and its `man` page(s):

```
$ whereis grep<Enter>
find: /bin/grep /usr/share/man/man1/grep.1.gz
```

whereis has also substantiated the results of the which search results. Sancho correctly perceives that there is only one executable grep, which is common to ordinary users and root users.

Locating Data within a File: grep Command

You can use the grep (global regular expression parsing) command alone or with other commands. We mentioned this command in Chapter 7, "Shell Basics," in the discussion of commands that are used as filters to other commands. The syntax for the basic use of grep looks a lot like the syntax for any command:

```
$ grep [options] regular expression [filename1 filename2 ... filenameN]
<Enter>
```

There is another syntax for grep for those occasions when grep is used in tandem with other commands. That syntax is as follows:

```
$ commandname [options] | grep [options] | commandname [options]<Enter>
```

or

```
$ commandname [options] | grep [options]\<Enter>
> | commandname [options]<Enter>
```

We showed the tandem commands earlier as they would typically be typed: on two separate lines. By the time you enter three or more commands, you have probably decided to split them (to type part of the command structure at the primary prompt and the rest at the secondary prompt).

Generally, the grep command searches for a regular expression, a specified pattern of text, logical constructs, and/or metacharacters (patterns or wildcard symbols that stand for something special) within specified files. If and when such expressions are found, grep takes one or more slices (each line in the text file that contains the regular expression is a slice) from the files and writes the slices to stdout.

The metacharacter symbols used by `grep` might be identical to some used by the shell, but they might mean different things to `grep`. Consequently, sometimes certain symbols should be surrounded by single quotation marks (preferably) or double quotation marks. Meanwhile, the `grep` command's metacharacters are discussed in the next section.

You usually specify a filename with `grep`. If you do not, `grep` searches `stdin` for its input. It is acceptable to alter `stdin` to accommodate that (that is, you can use the technique in Chapter 7 to designate files or directories for `grep` to search).

If you have not altered `stdin`, the shell takes you to the next line on the screen and leaves you with a flashing cursor (no prompt, even) and waits for you to enter input. Remember, to `grep`, the terminal is the default `stdin`. If you execute `grep` and want to do that, fine. If and when you enter a text pattern that matches the specified search text pattern, `grep` will echo that text pattern to the screen. To stop the echoing, press Ctrl-C. The process terminates and returns a prompt.

Assume that you have entered a search text pattern and a single filename to search through, and `grep` finds your text pattern one or more times in that file. It returns copies of those lines to your screen but does not display any lines that do not match the text pattern. If you have entered more than one filename to search and `grep` has found your text pattern in more than one file, however, it returns each line it found and also precedes each line with the name of the file in which it found the text pattern. (Example 8.11 illustrates this type of response.)

Example 8.11 grep

Examples 8.11 and 8.12 will be more meaningful if you revisit the following files, which were created for Examples 7.1, 7.2, and 7.3:

- Filename: */home/gutiejua/admin/phone#s/RFI_Tel1*
- Filename: */home/gutiejua/admin/phone#s/RFI_Tel2*
- Filename: */home/gutiejua/admin/phone#s/RFI_Tel3*

Furthermore, these examples will be more meaningful if you take the time beforehand to create the following additional files with a text editor such as `vi`:

- Filename: */home/gutiejua/admin/Feed*

```
Rozinante, Dapple -

hay (alfalfa, timothy, etc) .8
oats .3
```

```
chop .2
vegetables (fresh) .65
vegetables (old, rotten) .6
fruits .4
supplements .1
nachos/salsa .15

Dogs, Cats -

dog food .5
dog treats .55
cat food .7
catnip .75
```

- Filename: */home/gutiejua/admin/Innfile*

```
file<Spacebar>Inn No. 4
file<Spacebar>Inn No. 7
file<Tab>Inn No. 8
file<Tab>Inn No. 1
file<Spacebar>Inn No. 8
```

NOTE **In the *Innfile* file, `<Spacebar>` indicates that you should press the space bar between the end of the word *file*, for example, and the beginning of the phrase *Inn No. x*. Similarly, `<Tab>` indicates that you should press the Tab key between the end of the word *file*, for example, and the beginning of the phrase *Inn No. x*. Also, there are really two *Inn No. 8* entries; there is no typographical error.**

Lady Dulcinea is curious about how many staff members make their homes near La Mancha and El Toboso. She asks Juana to check her *RFI_Tel** telephone number files. Juana knows that only two files (namely, *RFI_Tel1* and *RFI_Tel2*) contain such information. She uses `grep` with the regional area code 958 as follows:

```
$ grep 958 RFI_Tel1 RFI_Tel2<Enter>
RFI_Tel1: La Mancha Field Office (Noble Deeds) - +34 958 55 34 56
RFI_Tel1: El Toboso Field Office (Executive Suites) - +34 958 55 56 78
RFI_Tel2:    Casa: +34 958 55 19 01
RFI_Tel2:    Casa: +34 958 55 01 03
RFI_Tel2:    Casa: +34 958 55 13 15
RFI_Tel2:    Casa: +34 958 55 13 15
RFI_Tel2:    Casa: +34 958 55 15 17
RFI_Tel2:    Casa: +34 958 55 17 19
```

Example 8.12 grep for Extracting Data

Juana remembers compiling three telephone lists for Rueful Figures, Inc. and storing them in her *phone#s* subdirectory. Here is how she checks that number:

```
$ cd<Enter>
$ cd admin/phone#s<Enter>
$ ls | grep ^R | wc -l<Enter>
3
```

In the command in Example 8.12, Juana moved to her home directory first, then changed directories to her */home/admin/phone#s* subdirectory. Then, she listed all the files in that subdirectory and `grep`'d all files whose filenames began with an R and then counted the number of lines in the output to that `grep` command. Because `grep` found three files, *RFI_Tel1*, *RFI_Tel2*, and *RFI_Tel3* (for reference, see the file listings in Example 8.11), its output would have three lines. The `wc` command counts those `grep` output lines and reports a `3` to the terminal screen, which is the default `stdout` (standard output device). The `grep` command is used most often in this capacity to extract certain specified information from `stdin` for further processing.

In the next section, we will examine `grep`'s metacharacters as used in its regular expressions.

Admittedly, Juana could have just done an `ls` listing on the same directory and counted the files whose filenames began with an R. If she had had to check through a long list of files through several subdirectories, however, then a simple listing would have been insufficient.

grep: Regular Expressions with Metacharacters

Take another look at Example 8.11. You could get the same results by entering the following:

```
$ grep 958 RFI*<Enter>
```

For Juana's file structure, the asterisk wildcard would point to the three files. What if she had more than three files that began with RFI but she only wanted to check those three files, however? How could she restrict the search? Besides listing the three files explicitly, can you think of any

other syntax that would work? The next section should help answer these questions.

Meanwhile, we know that `grep` can search in specified files for specified regular expressions, which can include text patterns and metacharacters. In Example 8.12, we saw a metacharacter used with `grep` to search for every file that begins with the character R (`grep ^R`).

We also mentioned how the metacharacters used with `grep` might or might not mean the same as the metacharacters used by the shell. Take a look at Table 8.2. Can you see how `grep` metacharacters might cause confusion as well as unreliable and undesirable results? For this reason, we recommend that you surround `grep` expressions containing metacharacters with single quotes. Double quotes might be acceptable in some situations, but you will get the most reliable performance by using single quotes.

The following are three examples of `grep` metacharacter usage:

- `[a-f]` means any *one* character from the range `a` through `f`
- `^a` means any lines beginning with `a`
- `z$` means any lines ending with `z`

Table 8.2 Metacharacters in grep and the Shell

METACHARACTER	MEANING IN GREP	MEANING IN THE SHELL
.	Match any character	If followed by *filename*, execute *filename*
*	Match zero or more preceding	Match zero or more
^	Match beginning of line	Bourne shell pipe symbol
$	Match end of line	Variable (generally, the user prompt)
\	Escape the character following	Escape the character following
[ ]	Match one from this set or range	Match from this set or range
{ }	Match this range of instances	
+	Match one or more preceding	
?	Match zero or one preceding	Match zero or one

Table 8.3 Selected grep Options

OPTION	DESCRIPTION
`-v`	Print the lines that do not match the specified pattern.
`-c`	Print only a count of the matching lines.
`-l`	Print only the names of the files with matching lines.
`-n`	Print matching lines and their line numbers in the respective files.
`-i`	Ignore the case of the letters when making comparisons.
`-w`	Perform a whole word search.

grep Options

Table 8.3 presents some of the more common `grep` options. All of these options are used in Example 8.7, where we present the poem titled "Fleas."

Example 8.13 Applying grep to "Fleas"

Sancho, inspired by Don Quixote's poetic prowess (questionable though these examples might be), has written this poem and has saved it in his home directory. After you read it, you will probably wonder why:

```
$ cat fleas<Enter>
Fleas

You bite those bugs and give a yelp
You scratch anon, sometimes it helps
You chase your tail, you drag your butt
Oh, but you're suffering, my dear old mutt

I rub your belly and your back
To stem that burning itch attack
I scratch your ears, and scratch your head,
But those cursed fleas fill me with dread!

They are entrenched, they won't go 'way
Do what you will, they're here to stay
You long for the "powders", you yearn for the "collars"
Perhaps a mud pit in which to waller?

But, you know what? You see, old friend?
To rout those fleas, reach your happy end?
```

```
There is just one way to escape their wrath,
Yes, amigo, you must have a bath!
```

It might not be good, but it was fast. And besides, Sancho can now use "Fleas" to practice his `grep` skills. For example:

- To search for all lines containing the word *you,* in both upper-case and lower-case variants of *y*:

```
$ grep -i you fleas<Enter>
```

- To perform the search by using only metacharacters:

```
$ grep '[Yy]ou' fleas<Enter>
```

- To perform the search using the `egrep` command (discussed later):

```
$ egrep 'you|You' fleas<Enter>
```

- To search for all lines containing at least one character but to return only a line count of those lines:

```
$ grep -c '.' fleas<Enter>
```

- To search for all files that contain the word *you,* whether in lowercase or uppercase:

```
$ grep -li you *<Enter>
```

This example illustrates how you can combine options. Note that when subdirectories are encountered during the search, the following message appears: `grep: directoryname: Is a directory.`

- To search for all lines that do not contain the word *you*, lowercase or not:

```
$ grep -vi you fleas<Enter>
```

This example illustrates the combination of options, but the displayed output includes all blank lines, as well.

- To search for all lines in the *fleas* file, number them and send the output to the *fleas.num* file:

```
$ grep -n '.*' fleas > fleas.num<Enter>
```

Note how this example uses `grep` options and metacharacters and then uses redirection to send the output to the *fleas.num* file. To view the results, type `$ cat fleas.num<Enter>`.

- To search for all occurrences of the whole word *you* in the poem:

```
$ grep -w 'you' fleas<Enter>
```

Note that the words *your* and *You're* are also in the poem. If `whole word` had not been specified by the `-w` option, the occurrence of those words would also have been displayed.

More Examples of grep with Metacharacters

In this section, we provide more examples of `grep` in action to help ensure that you understand how to use this powerful command. These examples are representative of the tasks that administrators typically encounter.

- Select all lines (blank lines and non-blank lines) from the *RFI_Tel1* file:

```
$ grep '.*' RFI_Tel1
```

When you ask for this type of output, you are probably assuming that the file has blank lines. Single quotes surround the regular expression, because without them, the shell will interfere with `grep`'s execution by imposing the shell's interpretation on the asterisk.

- Select only the blank lines in *RFI_Tel1*:

```
$ grep '^$' RFI_Tel1<Enter>
```

- Select all the lines in the *RFI_Tel2* file that begin with the letter `M` and end with the number 7, with any number of characters in between:

```
$ grep '^C.*1$' RFI_Tel2
```

This request asks for lines that have a particular pattern within a file.

- Select all the lines in the *RFI_Tel2* file that contain a plus sign (+):

```
$ grep '\+' RFI_Tel2<Enter>
```

Other grep Commands: egrep and fgrep

We have already discussed how `grep` can be extremely useful when you have to extract text from a data stream, such as a text file. (Yes, `grep` can be used for other types of data streams, but this usage is not covered here.) The command's functionality or performance for extracting text in particular situations is further enhanced by the `egrep` and `fgrep` commands.

egrep: Searching for Alternates

The `grep` command itself cannot be used for the type of search request where you want to find all strings containing one item or another. The extended `grep` command, `egrep`, enables OR searches by letting you put a `pipe` symbol (|) between your specified alternate search expressions, as shown in Example 8.14. The `egrep` command also works with `grep`'s metacharacters and options. The `egrep` command is a little slower than a normal `grep`, mostly because it executes more than one `grep` process.

Example 8.14 egrep

Lady Dulcinea wants to know how many 800 numbers the RFI has to pay for. She also wants to know whether RFI has any 900 numbers:

```
$ egrep '800|900' RFI_Tel3<Enter>
Emergencies - +34 91 800 09 87
Human Resources - +34 91 900 55 87
```

Although she is not sure why the human resources department has a 900 number, she is satisfied that the emergencies number is an 800 number.

Faster Searching for Fixed Strings with fgrep

The `fgrep` command is similar to `grep` but does not provide the regular expression capability. Instead, you are limited to searching for fixed text strings, as shown in Example 8.15. This command enables you to use the

same options as `grep` with the exception of any that deal with metacharacters or other aspects of regular expressions.

Because `fgrep` does not require the extra translation, it requires fewer processor resources.

Example 8.15 fgrep

At one point, Perez needs to call Lady Dulcinea. He has saved his copy of the *RFI_Tel2* file in the *admin* subdirectory of his home directory. Here is how he can quickly access her phone number:

```
$ fgrep 'Dulcinea' ./admin/RFI_Tel2<Enter>
A. Lorenzo (Lady Dulcinea) - Tel: +34 91 555 01 23
```

Sorting Output: The sort Command

You use the `sort` command to sort the output of a file or the `stdout` of a process before it is sent to the screen or wherever you want it to go. Thus, `sort` helps to ensure that the output is in an acceptable order or that it is presented how you want. It works by reading `stdin` (which might be the `stdout` of another process), processing the data, and sending it to `stdout` or wherever you designate. The `sort` command's syntax is as follows:

```
$ sort [-t delimiter] [+field[.column]] [options] [file(s)]<Enter>
```

The processes of the `sort` command use dictionary or ASCII ordering by default, which you can override with options—several of which appear in Table 8.4. For information on `sort`'s many other options, consult your information sources.

Example 8.16 sort with Spaces and Tabs

In Example 8.16, Juana first uses the `cat` command to display the contents of the *Innfile* file. Then, by adding `+1` in the `+field` position (see the syntax), she can sort the contents of the file according to the second field of each line. Look at how tabs have priority over spaces:

```
$ cat Innfile<Enter>
Filename: /home/gutiejua/admin/Innfile
file<Spacebar>Inn No. 4
file<Spacebar>Inn No. 7
file<Tab>Inn No. 8
file<Tab>Inn No. 1
file<Spacebar>Inn No. 8
```

Table 8.4 Selected sort Options

OPTION	DESCRIPTION
`-b`	Ignore leading spaces and tabs.
`-c`	Check whether data in the file is already sorted; if so, produce no output.
`-d`	Sort in dictionary order.
`-f`	Ignore differences in uppercase and lowercase.
`-n`	Sort in arithmetic order.
`-ofile`	Place output in the file.
`-r`	Reverse the sort order.

```
$ sort Innfile +1<Enter>
Filename: /home/gutiejua/admin/Innfile
file<Tab>Inn No. 1
file<Spacebar>Inn No. 4
file<Spacebar>Inn No. 7
file<Tab>Inn No. 8
file<Spacebar>Inn No. 8
I
```

NOTE **The `sort` command uses `<Tab>` or `<Spacebar>` as the default delimiter between fields. If the input to `sort` consists of a combination of spaces and tabs throughout the data being sorted, the tabs and spaces are subject to the sorting process. This situation can result in what appears to be incorrect processing (as shown in Example 8.16).**

Example 8.17 Several sort Examples

In Example 8.17, Juana shows us several instances of the `sort` command in action. To do so, she will use the *Feed* file, which is in her *admin* subdirectory.

To display the *Feed* file as written, she enters

```
$ cat Feed<Enter>
Filename: /home/gutiejua/admin/Feed

Rozinante, Dapple -

hay (alfalfa, timothy, etc) .8
oats .3
chop .2
vegetables (fresh) .65
vegetables (old, rotten) .6
```

```
fruits .4
supplements .1
nachos/salsa .15

Dogs, Cats -

dog food .5
dog treats .55
cat food .7
catnip .75
```

Notice that there are four blank lines in `Feed`. Those blank lines will show up in each of the next three examples. They are not forgotten by the `sort` command.

- Now, Juana sorts all the lines in `Feed` into dictionary (in other words, alphabetical) order:

```
$ sort Feed<Enter>



cat food .7
catnip .75
chop .2
dog food .5
Dogs, Cats -
dog treats .55
Filename: /home/gutiejua/admin/Feed
fruits .4
hay (alfalfa, timothy, etc) .8
nachos/salsa .15
oats .3
Rozinante, Dapple -
supplements .1
vegetables (fresh) .65
vegetables (old, rotten) .6
```

- Now, she sorts alphabetically by the *second* character in the first word of each line:

```
$ cat Feed | sort +0.1<Enter>
```

```
nachos/salsa .15
cat food .7
catnip .75
oats .3
hay (alfalfa, timothy, etc) .8
vegetables (fresh) .65
vegetables (old, rotten) .6
chop .2
Filename: /home/gutiejua/admin/Feed
dog food .5
Dogs, Cats -
dog treats .55
Rozinante, Dapple -
fruits .4
supplements .1
```

Finally, Juana uses the `t` option to tell `sort` which character in the file to recognize as the field separator. The most common separators are colons, tabs, or `\n` (newline character). Juana will illustrate how to sort on the second field, numerically, using the dot (`.`) as the field delimiter:

```
$ cat Feed | sort -t. -n +1<Enter>
```

```
Dogs, Cats -
Filename: /home/gutiejua/admin/Feed
hay (incl. alfalfa, timothy, etc) .8
Rozinante, Dapple -
supplements .1
chop .2
oats .3
fruits .4
dog food .5
vegetables (old, rotten) .6
cat food .7
nachos/salsa .15
dog treats .55
vegetables (fresh) .65
catnip .75
```

Please notice that the blank lines, followed by the lines with no numbers or where there are no numbers in the second field, were listed first. The `hay` line shows a dot delimiter in the abbreviated word `incl.`, but there is no number following it. The number `8` appears in what `sort` would consider to be the third field.

Then, the `sort` command sorted the other lines, which met the search criteria more correctly, in numerical order. Notice also that the single-digit numbers were sorted before the double-digit numbers.

NOTE **`sort`'s options are unique to it. If you were using the `cut` command, by comparison, the delimiter would be `d`, not `t`.**

Displaying Parts of Files: The head and tail Commands

You use the `head` and `tail` commands when you want to view only parts of a file.

The head Command

In this section, we discuss only the simplest applications of the `head` command. Refer to your sources for further information regarding this command's additional options.

The `head` command is used to display the first `n` number of lines in a file, as illustrated in Example 8.18. The syntax is

```
$ head [-number of lines] [file(s)]<Enter>
```

If you do not specify a number, the default value of 10 is used. If no files are specified, `head` reads from the standard input device (`stdin`). If `head` is combined after another command, `head` still reads from `stdin` although it appears that `head` is reading from the `stdout` of the previous command. If more than one file is read, a header of the type `==>` *`filename`* `<==` is displayed before the respective lines of text.

The second part of Example 8.18 shows how you can `pipe` to `head` the output of one process to display only a specified number of lines.

Example 8.18 head

Juana cannot remember offhand whether she put the filename at the top or the bottom of the *Feed* file. From her home directory, she simply types

```
$ head -5 ./admin/Feed<Enter>
Filename: /home/gutiejua/admin/Feed

Rozinante, Dapple -

hay (incl. alfalfa, timothy, etc) .8
```

Now, again from her home directory, Juana will do a recursive listing, but starting from her *admin* subdirectory (in other words, seeming to jump over her */home/gutiejua* directory), and will send the output to `head`:

```
$ ls -R ./admin | head -7<Enter>
./admin:
Feed
Innfile
phone#s

./admin/phone#s:
RFI_Tel1
```

The tail Command

The `tail` command performs a bit differently than `head`. Its syntax is

```
$ tail [-number of lines | +number of lines] [file(s)]<Enter>
```

Note that you can specify a positive or negative number for the number of lines. (For other options, see your information sources.) The negative number tells `tail` to display text beginning at the `n`th line from the *end* of the file, so you get `n` lines of text. The first part of Example 8.19 illustrates this type of output.

The positive number specification tells `tail` to display text beginning at the `n`th line from the *beginning* of the file and to continue from there. The second part of Example 8.19 illustrates this situation.

One interesting option with the `tail` command is the `-f` option. With this option, `tail` continues reading additional lines from the input file as they become available. For example, suppose that Juana has an accounts-receivable file called *Repts_rec* that she knows will keep growing. She could monitor its growth by entering the following:

```
$ tail -f Repts_rec<Enter>
```

The result is the (default) last 10 lines of the file at the moment the command is processed. Then, additional lines are displayed as they are added to the file. Juana could stop it by pressing Ctrl-C.

Example 8.19 tail

Juana wants to see where she left off when compiling the *Feed* file. Specifically, she wants to know whether she listed feed for the support animals (the dogs and cats). Here is how she checks:

```
$ tail -6 Feed<Enter>
Dogs, Cats -

dog food .5
dog treats .55
cat food .7
catnip .75
```

Remember, the `-6` told `tail` to display text beginning at the sixth line from the *end* of the file, so Juana received six lines of text.

Earlier, Juana listed the first five lines of the *Feed* file with `head`. Now, she wants to continue listing—picking up from where she left off. She cannot remember just how long *Feed* is, so she uses the `tail` command to begin at line 6:

```
$ tail +6 animals<Enter>
oats .3
chop .2
vegetables (fresh) .65
vegetables (old, rotten) .6
fruits .4
supplements .1
nachos/salsa .15

Dogs, Cats -

dog food .5
dog treats .55
cat food .7
catnip .75
```

The `+6` told `tail` to display text beginning at the sixth line from the *beginning* of the file and to continue from there. That is why Juana got 14 lines of output. The *Feed* file is 19 lines long. If *Feed* had only been, say, nine lines long, then she would only have gotten four lines of output.

Exercises

1. Log into the system and ensure that you are in your home directory. Then, find and display all files in the */tmp* directory:

```
$ cd<Enter>
$ pwd<Enter>
$ find /tmp<Enter>
```

2. Find all files in your home directory that begin with the letter `s` and then have `ls -l` automatically execute on each filename found as a result of the search operation:

```
$ find . -name 's*' -exec ls -l { } \;<Enter>
```

3. Repeat the search in the preceding step but interactively prompt the user to display a long list on each file:

```
$ find . -name 's*' -ok ls -l { } \;<Enter>
```

4. Find all files starting from the */usr* directory owned by the `uucp` username. Modify the command line to count the number of files owned by `uucp`. You probably do not have read permission for some directories, which would result in a `permission denied` message on your terminal screen. Because you anticipate this situation, redirect all error messages to a file called *errfile*:

```
$ find /usr -user uucp 2> errfile | wc -l<Enter>
```

5. Display the *errfile* file from the preceding instruction to see whether any error messages were written:

```
$ pg errfile<Enter>
```

6. To demonstrate that `find` recursively searches all directories and subdirectories from the search path down, do the following. First, ensure that you are in your home directory:

```
$ cd<Enter>
$ mkdir level1<Enter>
```

Create a zero-length file named *letter1* in the *level1* subdirectory:

```
$ touch level1/letter1<Enter>
```

Change to the *level1* subdirectory:

```
$ cd level1<Enter>
```

Make a subdirectory under *level1* called *level2*:

```
$ mkdir level2<Enter>
```

Create a zero-length file named *letter2* in the *level2* subdirectory:

```
$ touch level2/letter2<Enter>
```

Change to your home directory:

```
$ cd<Enter>
```

From your home directory, issue the command to list all files starting with the letter `l`. Record the names displayed:

```
$ ls l*<Enter>
```

From your home directory, issue the command to find only files starting with the letter `l`:

```
$ find . -name 'l*' -type f<Enter>
```

Finally, record the names displayed.

7. Find all lines in the */etc/passwd* file for usernames that start with `team:`

```
$ grep team /etc/passwd<Enter>
```

8. Find all lines in the */etc/passwd* file that begin with the letter `t:`

```
$ grep '^t' /etc/passwd<Enter>
```

9. Find all lines in */etc/passwd* that contain a digit from 0 through 9:

```
$ grep [0-9] /etc/passwd<Enter>
```

10. Repeat the search in the preceding step, but this time display only the number of lines that contain the pattern:

```
$ grep -c [0-9] /etc/passwd<Enter>
```

11. Use the `ps` and `grep` commands to display the processes that have been initiated by users other than yourself:

```
$ ps ua | grep -v username<Enter>
```

12. Display the contents of the */etc/passwd* file in alphabetical order:

```
$ sort /etc/passwd<Enter>
```

13. Display the contents of the same file but in reverse order:

```
$ sort -r /etc/passwd<Enter>
```

14. Display the first ten lines of */etc/passwd*:

```
$ head /etc/passwd<Enter>
```

15. Display the first five lines of */etc/passwd*:

```
$ head -5 /etc/passwd<Enter>
```

16. Display the last ten lines of */etc/passwd*:

```
$ tail /etc/passwd<Enter>
```

17. The `tail` command is handy also for stripping header information from the output of a command. First, list all processes that are currently running on your system. Note the headings:

```
$ ps ua | less<Enter>
```

18. Next, display all processes running on your system excluding the header information:

```
$ ps ua | tail +2 | less<Enter>
```

See Appendix B for answers.

Quiz

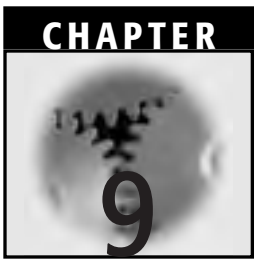
1. Which command would you use to locate all the files in your system that begin with the text string `mis`?
2. Explain the following command:

```
$ ps ua | grep -w root | grep -w /sbin*<Enter>
```

3. Explain the following command:

```
$ ls -l /home/teamxx | egrep 'um$|isc$|ync' | sort -r +8 | tail +2 |
head -7<Enter>
```

See Appendix C for answers.



Advanced Linux Utilities

In Chapter 8, "Basic Linux Utilities," we discussed some basic Linux utilities. This chapter continues that discussion and broadens your ability to perform file system administration. In several of the sections and examples in this chapter, we refer once again to Sancho's file structure.

Maximizing Work per Command: xargs

For efficient execution, the `xargs` command is one of the best commands that Linux/UNIX offers. `xargs` has two very valuable functions that we will discuss here.

xargs: Optimal Execution for "Smart" Commands

Some Linux/UNIX commands are smart enough to take input from a parameter line and execute with it. Moreover, if the parameter line is empty, the command is again smart enough to take input from `stdin`,

which is often a pipe. xargs is a very efficient command that can read several arguments one line at a time from its stdin and then assemble as many of them as possible into a single command line until it determines that it has reached its capacity. By "reached capacity," we mean that xargs would not be capable of executing if any more arguments are added. Also, the arguments to xargs are generally commands or programs themselves, as you will see in Example 9.1.

Thus, once xargs has filled its input to capacity this way, all the commands will be executed. If there are still more arguments in the source, xargs will assemble more of them into another single command line until it too is filled to capacity, and then those command(s) will be executed. The xargs command continues to perform this task until it exhausts the given supply of arguments/commands.

The syntax for xargs is as follows:

```
$ command [options] | xargs [options][arguments]<Enter>
```

We recommend that when xargs is executed, the -t option (which invokes verbose mode) should be included—at least until you are familiar with xargs and it is doing exactly what you want. After you get comfortable with xargs and can trust it, you will probably no longer choose to use the -t option.

Example 9.1 Using xargs with "Smart" Commands

In Example 9.1, we will illustrate two fairly simple applications of xargs. But first, let's take a look at the latest version of Don Quixote's portion of the RFI directory structure in Figure 9.1. He has been busy preparing for the next Noble Deeds Division journey. He has expanded his file structure so that he can dedicate a file to each provision.

Don Quixote has two objectives at this point:

- To print the large number of filenames within his new */home/quixoted/provisions* subdirectory
- To change the names of the existing files so that he can add new files with similar names later

Here is how he will use xargs and other commands to print the filenames:

```
$ cd /home/quixoted/provisions<Enter>
$ ls > prov_list<Enter>
```

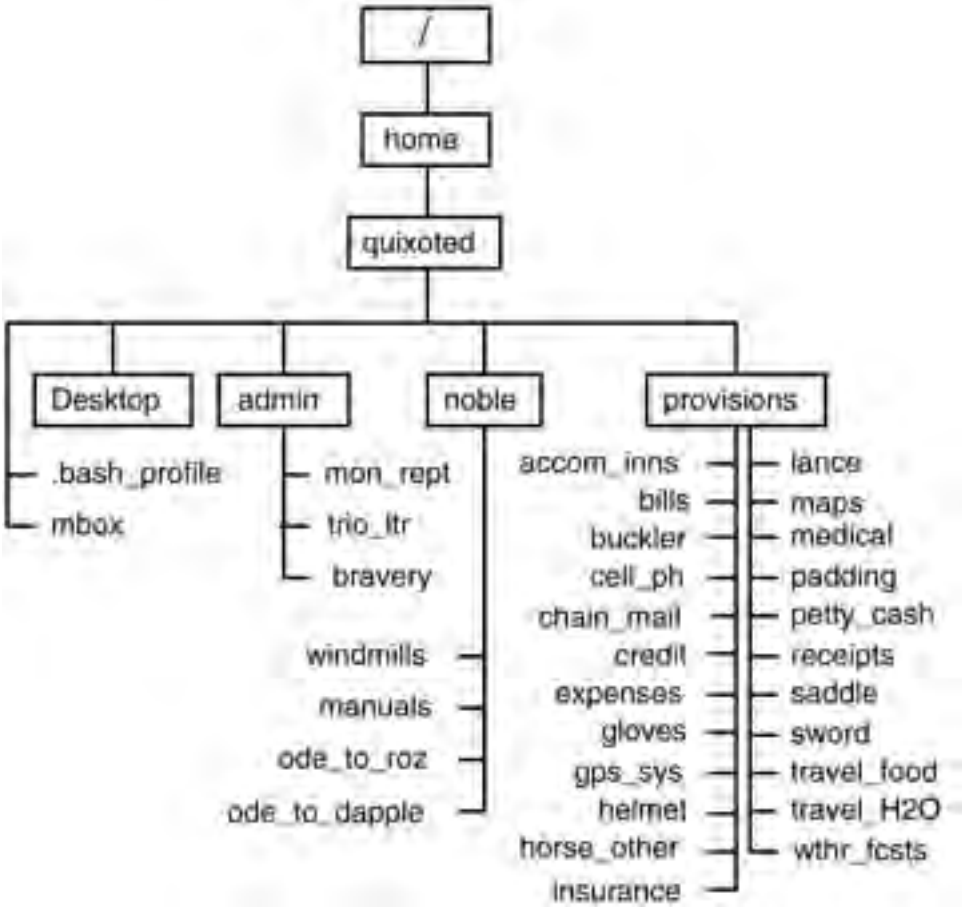


Figure 9.1 Don Quixote's portion of the RFI directory structure.

```
$ cat prov_list<Enter>
accom_inns
bills
buckler
cell_ph
.
.
.
travel_H2O
wthr_fcsts
$ xargs -t lpr < prov_list
lpr accom_inns bills buckler ... insurance
lpr lance maps medical ... wthr_fcsts
```

In this first step, Don Quixote illustrated the printing of a number of filenames at once. He moved to the */provisions* subdirectory, then created a file there named *prov_list*, which consists of the names of all the files in the subdirectory. Those were the names he wanted to print. He used `cat` to examine the filenames in the new file list.

In this step, the Don used the combined `xargs` and `lpr` commands and the results. The feedback from the shell indicated that `xargs` was capable of filling the command line with arguments from `accom_inns` to `insurance` before `lpr` was executed the first time. Then, `xargs` had the capability to pass the rest of the files, from *lance* to *wthr_fcsts*, to `lpr` for the second execution. Note that the Don was using the `-t` option to ensure that `xargs` had passed the input to `lpr` and that `lpr` did indeed execute.

Because the existing files pertained mostly to his last journey in March, he chose to change their filenames to reflect that date. Here is how he changed the names of all the files at once:

```
$ ls | xargs -t -i mv {} {}.mar01<Enter>
mv accom_inns accom_inns.mar01
mv bills bills.mar01
mv buckler buckler.mar01
mv cell_ph cell_ph.mar01
.
.
.
mv travel_H2O travel_H2O.mar01
mv wthr_fcsts wthr_fcsts.mar01
```

The second part of Example 9.1 shows how the Don inserted his filenames into the middle of a command line. His objective was to rename all the files in the *provisions* subdirectory by adding the `.mar01` suffix to each. He invoked a short listing (in other words, `ls`) of the files in that directory and then piped the filenames from `stdout` to `xargs`. `xargs`, then instructed that each filename, in turn, be inserted (the `-i` option inserts each line of standard input) between the curly bracket placeholders following the `mv` command. This form of `mv` is commonly used to rename files. Because he also specified the `-t` option, the Don was also able to monitor the execution of `mv` as each filename was passed to it. That is because the `-t` option causes the individual commands to be printed to `stderr` just before executing the `mv` command. Later, after the last filename, *wthr_fcsts*, was changed to *wthr_fcsts.mar01*, the combination command finished and the primary shell prompt returned.

xargs: Tandem Execution for "Not-So-Smart" Commands

We mentioned in the introduction to this section that there are two uses for `xargs`. Our first discussion and example illustrated how `xargs` can work with files that are smart enough to take input from a pipe. But unfortunately, not all commands are that smart. For example, `rm` (the delete command) is one of those not-so-smart commands. Faced with this situation, what would you do if you had a lot of repetitions of a command (such as `rm`) to perform? Your options are as follows:

- Invoke the command several times manually.
- Use some type of automated loop to invoke the command for every file, which spawns a subprocess every time the command is invoked (taxing on time and processing capability).
- In the case of `rm`, use `find` to identify the input filenames and then execute `cat`, `xargs`, and `rm` in tandem.

The third option is the best. This example illustrates `xarg`'s second and very valuable use: taking `cat`'s output and passing it to `rm`. We will see how this process works in Example 9.2.

Example 9.2 xargs

Juana's portion of the RFI directory structure is illustrated in Figure 9.2.

Juana now wants to remove the copies of *RFI_Tel1*, *RFI_Tel2*, and *RFI_Tel3* that she has kept in her */home/gutiejua/tempwork* subdirectory. For the first part of the process, she will collect their filenames into a single file called *tempphonelists*. Then, she will `cat` that file to be sure that it is correct:

```
$ cd <Enter>
$ cd ./tempwork<Enter>
$ find . -name '*Tel*' -print > ./tempphonelists<Enter>
$ cat tempphonelists<Enter>
./RFI_Tel1
./RFI_Tel2
./RFI_Tel3
```

Now, in the second and final stage, Juana will use `xargs` and `rm` against *tempphonelists* to remove the copies of the phone list files:

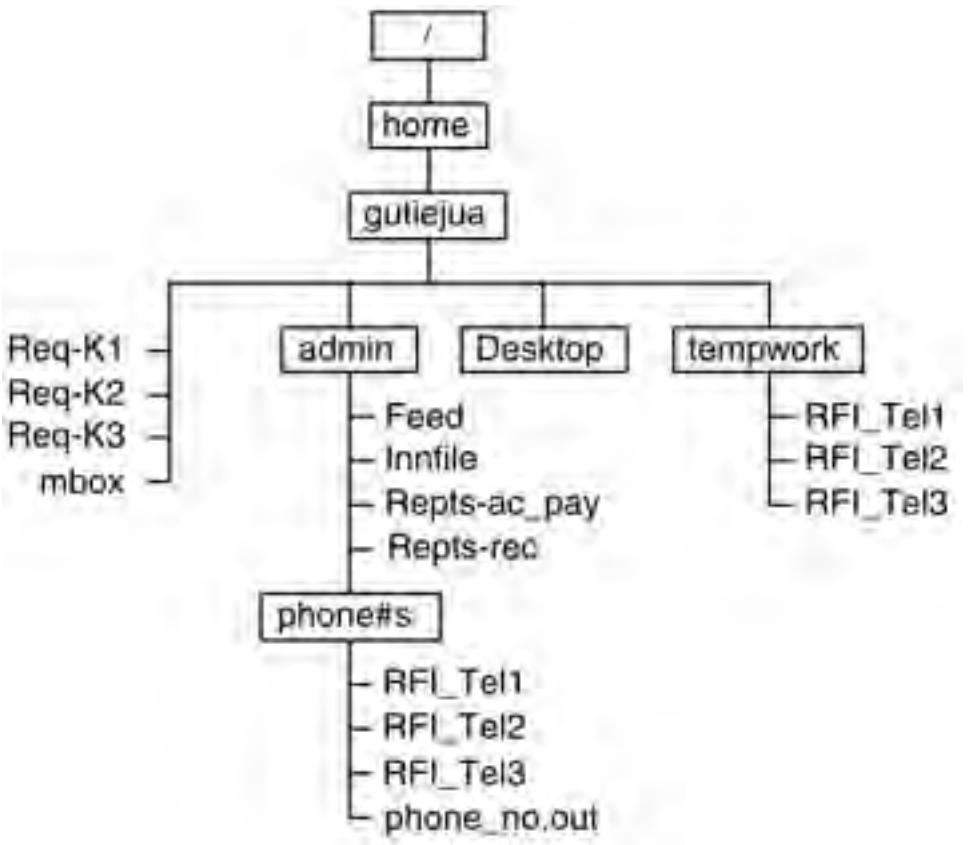


Figure 9.2 Juana's portion of the RFI directory structure.

```
$ cat tempphonelists | xargs -t rm<Enter>
rm ./RFI_Tel1 ./RFI_Tel2 ./RFI_Tel3
```

The first command was relatively simple: `find` identified all three phone list files and put their names into the *tempphonelists* file. With the second command, `cat` passed the list of filenames to `xargs` through `stdout/stdin`. `xargs` then translated the information from that `stdin` pipe and passed each parameter (in this case, each filename) to the parameter line for the subsequent `rm` command. The `xargs` command was smart enough to know the length of the parameter line it could produce before `rm` was to be invoked. Thus, `rm` was invoked the optimum number of times, which illustrates why this method is the most efficient way of handling this procedure.

You might have noticed that Juana also used the `-t` option with `xargs`. The `-t` option again caused the individual command to be printed to

stderr just before the command executed. Juana, like the Don in the previous example, was allowed to monitor (in real time as it happens) the number of times that rm is invoked. That is why the line rm ./RFI_Tel1 /RFI_Tel2 ./RFI_Tel3 appears while the tandem cat, xargs, and rm commands are executed. In this case, rm itself was executed only once.

Combining the xargs, find, and grep Utilities

Combining the find, grep, and xargs utilities has its benefits. All are powerful and efficient in their own right, but combining them multiplies their usefulness and can make you an even more efficient system user. Let's illustrate their combined power by moving to Example 9.3 immediately.

Example 9.3 Combining find, xargs, and rm

RFI staff members cooperate with Freston when he searches for files with modification dates older than 30 days so that he can remove them from the system. Of course, some old files are important, and Freston also cooperates with them to ensure that backup copies are available for future reference. But the object is not to clutter the storage capabilities of the system.

Normally, if he did not use xargs and other commands in tandem, Freston would have to enter a command like the following:

```
# find . -type f -mtime +30 -exec rm {} \;
```

That might not appear inefficient or objectionable at first glance, but it means that rm would be invoked every time a file was found that matched the find criteria. How many times would it have to be invoked if Freston was searching through a very large file system?

So this coding is what Freston enters to find and remove the old files. He uses find, xargs, and rm in tandem as follows:

```
$ find . -type f -mtime +30 | xargs -t rm <Enter>
rm ./oldfilex ./oldfiley ./oldfilez ./oldfileaa etc. . . .
```

Using xargs allows Freston to pass multiple parameters to rm, so rm is invoked far fewer times to remove all of the old files. In addition, Freston finds the xargs syntax easier to remember than the combined find, exec, and rm syntax. Note that in Example 9.3, we show Freston using the -t xargs option so that he can see how many times rm would actually be invoked.

Example 9.4 Combining find, xargs, and grep

Sancho wants to call and talk to Dapple's trainer, but because he is still new to RFI and he has only been to the stable/spa in Toledo once, he cannot remember who Dapple's trainer is. But he knows that each animal's profile has the trainer's name in it. He was just reading the profiles and adjusting them recently, but their filenames have slipped his mind. But he can quickly examine his current directories and find all the files containing the word "trainer" (regardless of whether it is in upper or lower case). This is what he enters:

```
$ find . -type f | xargs -t grep -lwi 'trainer' lpr<Enter>
grep -lwi trainer lpr ... recursive listing of files ...
./Maint/rozinante
./Maint/dapple
```

First, `find` passes a list of the files in the current directory to `xargs`. The `xargs` command invokes `grep` to look into all the files for the entire word `trainer` (the `-w` option) regardless of case (the `-i` option) and list the names of the files containing `trainer` (the `-l` option). The output list is handed to `lpr`, which prints a recursive list of the files examined to the screen. After that, the names of the files that contain `trainer` are printed: *rozinante* and *dapple*, both of which are in Sancho's *Maint* subdirectory.

We can see that `grep` was invoked only once because the example file structure contains a small number of files. If there had been many more files to examine, invoking `grep` more often might have been necessary.

Linux/UNIX Shortcut: The alias Command

You would probably appreciate being able to execute frequently used commands with fewer keystrokes. Or, there might be long-winded commands, frequently used or not, for which you would *really* appreciate using fewer keystrokes.

Enter the `alias` command, which you can use to:

- Create an abbreviated command name—or perhaps some kind of mnemonic sequence—that becomes a new internal shell command that will accomplish the same effects as the longer command it replaces.
- Investigate to find any aliases that have already been created.

We have already seen examples of the use of aliases in Examples 11.2 (the *etc/bashrc* file) and 11.4 (the *.bashrc* file).

An alternative to creating command names using `alias` is to write a shell script to do the same thing. Because a new `alias`-created command is an internal shell command, however, the shell gives it precedence over any script you might have created to do the same thing. In addition, creating shell scripts involves using a text editor such as `vi`, and testing and using shell scripts requires you to give the script files the proper permissions. Then, you have to make sure that the `PATH` variable includes a reference to the location of the shell script so that it can be found and executed. Of course, sometimes a shell script is just what you want or need, especially when you want to do something more complex (for example, executing a sequence of commands automatically when you log in). Still, from the standpoint of creating, testing, and executing, the use of the `alias` command is a more simple and straightforward technique.

Examining Existing Aliases and Creating New Ones

You can use the new alias by itself as a command, or you can add arguments to it. But the arguments must pertain to the command (if there is more than one command within the alias, the arguments must pertain to the last command).

For investigating what aliases might already exist, here is the syntax:

```
$ alias<Enter>
```

And for creating new aliases, the syntax is

```
$ alias (-options)newaliasname='commandname (-options)
[arguments]'<Enter>
```

For instance, assume that you have a new alias, `l`, which is equivalent to `ls -l`. If you want to see hidden files too, you should type `l -a` at the prompt.

By adding aliases to global function and alias files, the system administrator can make those aliases available to all or specific groups of other users.

In Example 9.5, we first use the `alias` command alone to view the aliases that have already been defined in our profile, function, and alias files.

Example 9.5 Using the alias Command to Check for Existing Aliases and to Create New Ones

To find existing aliases, use

```
$ alias<Enter>
alias dir-p='ls -l | more'
alias which='type -path'
```

He found two: `dir-p`, which he created in a previous section, and `which`, the default alias that helps us identify commands.

To create three new aliases for common functions to reduce lengthy keyboarding, use

```
$ alias l='ls -l'<Enter>
$ alias p='ps f'<Enter>
$ alias r='fc -e -'<Enter>
```

To check that the new aliases are listed with the original two, use

```
$ alias<Enter>
alias dir-p='ls -l | more'
alias which='type -path'
alias l='ls -l'
alias p='ps f'
alias r='fc -e -'
```

Next, he used the `alias` command with arguments to create three new aliases. Two of them perform listing functions: `alias l='ls -l'` creates a long listing of the non-hidden files and subdirectories in your current directory, and `alias p='ps f'` creates a list of processes and indicates which child processes were spawned by which parent processes. The last alias (`alias r='fc -e -'`) is a shorthand form for "repeat the last or immediately previous command." Finally, we use the `alias` command by itself again to verify that the three new aliases were created.

Using and Removing Aliases

Suppose that you no longer need an alias you have been using and want to delete it. To do so, you use the `unalias` command. Its syntax is as follows:

```
$ unalias aliasname<Enter>
```

In Example 9.6, we check for existing aliases, verify that the alias we created in Example 9.5 works, remove that alias, and then verify that it no longer works.

NOTE **Keep in mind that these aliases will work only in the shells in which they were created. If you want them to work elsewhere, you have to enter them in your *$HOME/.bashrc* file (or whatever file you use for the `BASH_ENV` variable) or into function or alias files for your other shells. We will discuss those files and variables in Chapter 11, "Shell Variables and the User Environment."**

Example 9.6 Identifying and Removing an Alias

To identify existing aliases, use

```
$ alias<Enter>
alias dir-p='ls -l | more'
alias l='ls -l'
alias p='ps f'
alias r='fc -e -'
alias which='type -path'
```

To verify that the `l` alias works, use

```
$ l<Enter>
-rw-r--r- 1 flintsfr staff 524 Jun 13 12:45 xfile1
-rw-r--r- 1 flintsfr staff 1455 Jul 15 14:13 xfile2
```

To remove the `l` alias and then check to see whether it works, use

```
$ unalias l<Enter>
$ l<Enter>
bash: l: command not found
```

You can remove more than one alias at a time by entering their names in a list after the `unalias` command. Be careful to separate each name with a space.

NOTE **Remember that `unalias` removes the alias from the current shell's alias list, but if the alias appears somewhere in the definition of the `BASH_ENV` variable file, it will be back the next time you log in and will also appear in any**

subshells that you create (even now). So you must take care to remove it, if that is your intention, from all definition files where it might be found.

Comparing find Functions and Shell Functions

In this section, we illustrate the difference between the functionality of certain shells and the `find` command. We discussed the capability of the `find` command to travel down through directories (in other words, to search recursively), which is one of its primary benefits.

During most routine operations, the shell interprets the command line and then provides the appropriate arguments to an executable command. Thus, the commands do not understand directory structures and rely on the shell to expand wildcards and then provide the full directory or path name of candidate files for execution. Juana's file structure has expanded rapidly since she started working with Lady Dulcinea (Figure 9.3 depicts her portion of the RFI directory structure).

Example 9.7 Shell versus find

Lady D. has asked for a summary of recent accounts-payable activity. Juana knows the information is in one of her *Repts** files, but to find it she needs to remember which part of her directory structure it is located. So, she will ask the Linux shell to find all filenames that begin with `R`:

```
$ cd /home/gutiejua<Enter>
$ ls R*<Enter>
Req-K1 Req-K2 Req-K3
```

She realizes that using `ls` alone, she did not get a thorough-enough search. She found all the request files (special requests sent to her from each Knights group) in her current directory, but she could not find any report files. They appear to be in another subdirectory somewhere.

Unfortunately, in these types of situations where you try to enter a fairly simple search command, the shell does not look any further than the current directory. To get the shell to look beyond the current directory and traverse the three-tiered directory structure in this example, Juana would have had to enter the following:

```
$ ls R* */R* */* /R*<Enter>
```

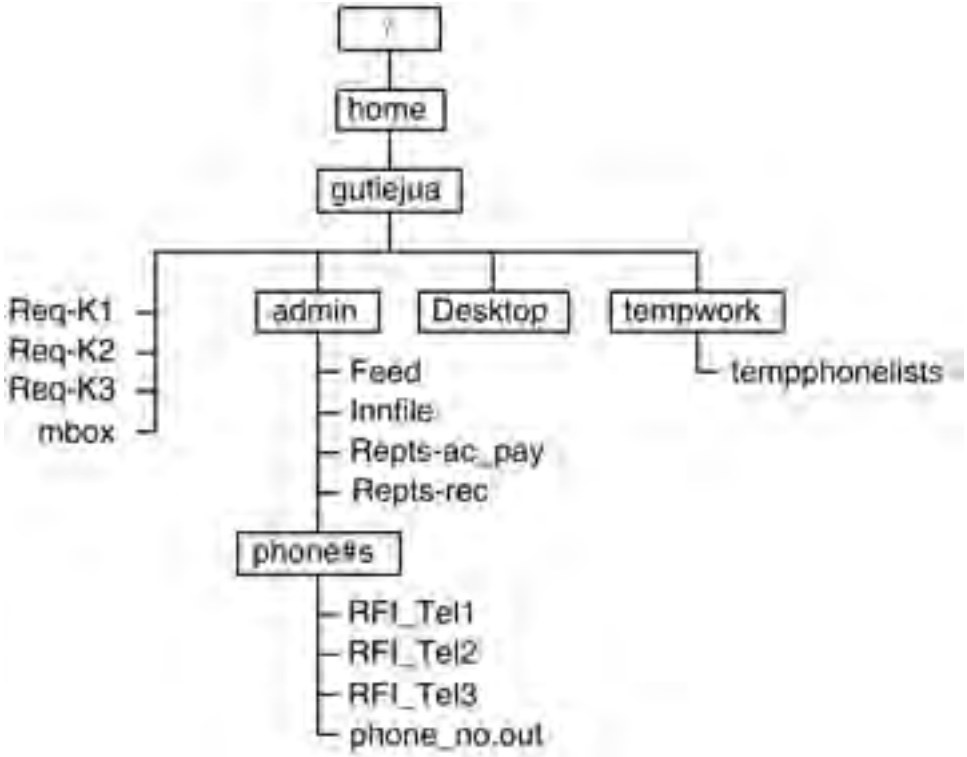


Figure 9.3 Juana's latest directory/file structure.

This search is tedious and confusing to type; moreover, she would have had to investigate the structure beforehand to know how many arguments to enter so that *all* the directory levels would be searched. Finally, the shell would not have allowed `ls` to check any hidden directories (those beginning with a dot), either.

For this reason, she should have used the `find` command. The syntax is easier, subdirectories are searched, hidden directories (if applicable) are searched, and she would have to know beforehand the depth of the file structure under the initially specified directory. For these reasons, many administrators and programmers consider `find`'s recursive search capability its best characteristic.

So now, she will ask `find` to find all the filenames that begin with `r`:

```
$ find . -name 'R*'<Enter>
./admin/phone#s/RFI_Tel1
./admin/phone#s/RFI_Tel2
```

```
./admin/phone#s/RFI_Tel3
./admin/Repts-ac_pay
./admin/Repts-rec
./Req-K1
./Req-K2
./Req-K3
```

She now sees that she saved her *Repts-ac_pay* file in her *admin* subdirectory, but she notices that a high proportion of her filenames seem to begin with `R`. "Huh?"she thinks. "If I ever get any spare time, I suppose I'll have to do something about that, too."

The find Command with the -links Option

In this section, we discuss the use of the `-links` option with `find`. There are two reasons to use this command combination:

- If you can have the same file referenced by two separate names, the only way you can tell whether they are indeed the same file is by examining the inode number. If the inode numbers are identical, they are the same file.
- More importantly, for security reasons you do not want outsiders linking to insider programs or files. You must have a way of seeing and confirming where all links are pointing to evaluate whether they are necessary and legitimate. Therefore, it is important to conduct a `find -links` check periodically to search for such potential infiltration.

The syntax is as follows (and here, we use "line continuation" from Chapter 7, "Shell Basics," to split and then continue the commands):

```
$ find directory(ies) -type [options] -links +[options]\<Enter>
> | xargs ls -li<Enter>
```

Example 9.6 is typical of a `find -links` check. Note that you should be certain to perform `-links` checks on only *files* because directories by their nature have at least two links (their own and their link with the parent directory). Note that the linked files' inode numbers, link counts, and file sizes are the same. The only difference among the three is their respective filenames. Rest assured that they all refer to the same file, regardless of their names.

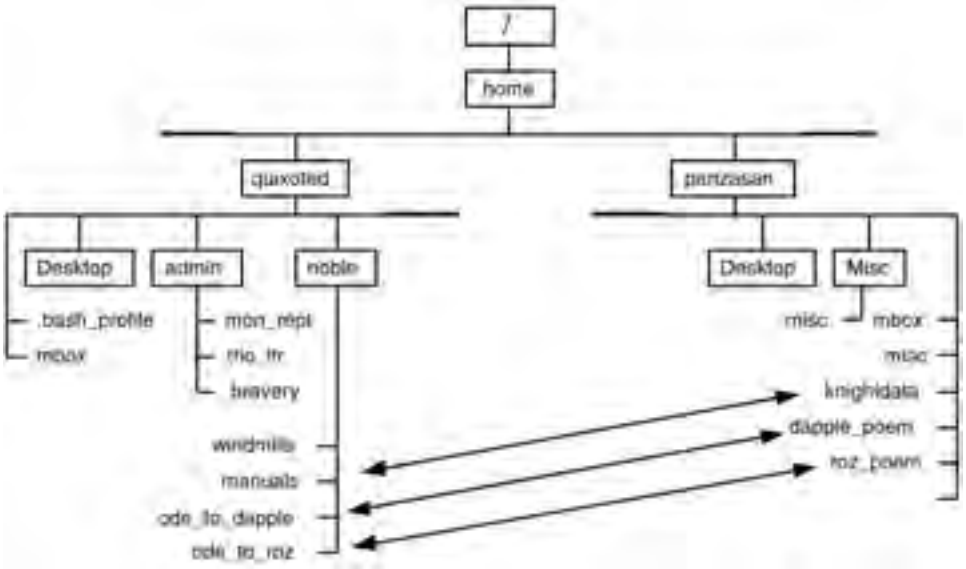


Figure 9.4 Linked files between Don Quixote's and Sancho's directories.

Example 9.8 Find -links

Figure 9.4 shows portions of both Don Quixote's and Sancho's directory structures. If we showed all of both, the figure would be a little too confusing. But you can see that they are adding new files all the time. Notice, though, that there are three files in each user's structure that are linked to files in the other's structure:

- *manuals* in */home/quixoted/noble* to *knightdata* in */home/panzasan*
- */home/quixoted/noble/ode_to_dapple* to */home/panzasan/dapple_poem*
- */home/quixoted/noble/ode_to_roz* to */home/panzasan/roz_poem*

Here is how Sancho might check for links in his file structure:

```
$ cd<Enter>
$ find . -type f -links 2 | xargs ls -li<Enter>
13444   -rw-r--r--2   panzasan   knights1   144  Jul 11 11:42
     ./knightdata
16698   -rw-r--r--2   quixoted   knights1   605  Jul 21 15:35
     ./dapple_poem
18138   -rw-r--r--2   quixoted   knights1   392  Jul 17 14:37
     ./roz_song
```

In his own directories, here is how Don Quixote might check and find the same links:

```
$ cd<Enter>
$ find . -type f -links 2 | xargs ls -li<Enter>
13444-rw-r--r--   2   panzasan   knights1   144  Jul 11 11:42
     ./manuals
16698-rw-r--r--2   quixoted   knights1   605  Jul 21 15:35
     ./ode_to_dapple
18138-rw-r--r--2   quixoted   knights1   392  Jul 17 14:37
     ./ode_to_roz
```

You can see how the inode numbers and various other parameters match from one end of each link to the other. The filenames are unique, however.

Example 9.8 used `xargs` in tandem with `find -links`, but Sancho or the Don could also have used a pure `find`, such as the following:

```
$ find . -type f -links 2 -exec ls -li {} \;
```

In this case, their choice could be attributed to a matter of syntax preference.

Reducing Keystrokes: Using find with alias

In this section, we show aliases being used to simplify and substitute for certain long commands that are used periodically for system monitoring and maintenance. The syntax for creating an alias is as follows:

```
$ alias shortcommand='commandname [options] [arguments]'<Enter>
```

Remember, when you set aliases from the command line, as we do in Examples 9.9 and 9.10, those aliases are effective only for the current login session. We will show you a technique in Chapter 12, "Linux Processes and Process Control," to ensure that `aliases` survive from session to session.

Meanwhile, the `aliases` illustrated here (and others) can be *undefined* by using the `unalias` command.

Example 9.9 Checking for Links with an alias Command

Over coffee, Sancho tells Juana how he checked for links in his file structure (please refer to Example 9.8). Freston overhears and says, "If you ever do that again, save yourself some time and effort by creating an

alias command." Then, he offers to show Sancho how to create an alias called "linkcheck," which is similar to the one he (Freston) uses to check for file links:

```
$ alias linkcheck='find . -type f -links 2 | xargs ls -li'<Enter>
```

After that, all Sancho would have to do is cd to his own home directory and then enter

```
$ linkcheck<Enter>
13444  -rw-r--r--   2   panzasan   knights1   144   Jul 11 11:42
./knightdata
16698-rw-r--r--2   quixoted   knights1   605   Jul 21 15:35
     ./dapple_poem
18138-rw-r--r--2   quixoted   knights1   392   Jul 17 14:37
     ./roz_song
```

Example 9.10 Removing Old Files with an alias Command

In Example 9.3, we saw how Freston combined find, xargs, and rm to search for files with modification dates older than 30 days so that he can remove them from the system.

He has created an alias command called oldrm to achieve the same objectives and to save time. Here is how he created that alias command:

```
$ alias oldrm='find . -type f -mtime +30 | xargs rm'<Enter>
```

Notice that he has left off the -t (verbose) option. Freston, a long-time Linux/UNIX user, does not really need to minutely monitor the rm process.

So now, Freston enters the following code to find and remove old files:

```
$ oldrm<Enter>
rm ./oldfilex ./oldfiley ./oldfilez ./oldfileaa etc. . . .
```

Determining File Types: The file Command

The file command is used to classify a file according to its content. This command uses up to three tests:

- A file system test
- A magic number test (`file` itself will use the */usr/share/magic* file to look for a specific number in the candidate file that would tell Linux/UNIX whether the file is executable)
- A language test (if the file is determined to be a text file)

Here is `file`'s syntax:

```
$ file [options] /path/filename(s)<Enter>
```

The output from `file` is usually a message saying that the candidate file is some variation of "text," "data," or "executable" unless it cannot find the candidate file. If it cannot find the file, `file` returns an error message. Here is an example of such a message when we tried to determine a file type for a nonexistent file that we named, appropriately, *nonexistentfile*:

```
$ file nonexistentfile<Enter>
Nonexistentfile: can't stat 'nonexistentfile' (No such file or
directory)
```

Why is `file` beneficial? It can give you a quick indication of whether a file is the type of file that you can more easily display on your terminal screen or send to a printer. Or, it can tell you that a file is an executable file. Without it, attempting to display an executable file on your screen could cause confusion or even hang up your terminal. For binary files, `file` can even provide an indication of the operating system and the version used to compile the file.

The `file` command has its own set of options. Check your information sources. In Example 9.9, we will see a simple directory search. In Example 9.10, we will see the `-f` option, which tells `file` to determine file types for a list of filenames found in another file specified immediately after the `-f`. In Example 9.10, that specified file—the one with the list of filenames inside it—is called *listoffiles*. Note that when specifying a filename as an argument to the `file` command, you should ensure that each filename appears separately on a single line in that argument file.

Example 9.11 Simple File Type Determination with the file Command

Juana, while exploring her home directory, sees a subdirectory called *Desktop*. Because she works solely from the command line (unless she deliberately

invokes a specific application), she wonders what such a subdirectory contains. Here is what she does:

```
$ cd<Enter>
$ ls<Enter>
admin   Desktop   mbox   Req-K1   Req-K2   Req-K3   tempwork
$ cd Desktop<Enter>
$ file ./*<Enter>
./Autostart:             symbolic link to ../.kde/Autostart
./Linux Documentation:   ASCII Text
./Printer:               ASCII Text
./kontrol-panel:               ASCII Text
```

Example 9.12 Determining File Types with the file Command and a filename Argument

With all the development, configuration, troubleshooting, Internet browsing, and reporting that Freston has to do to keep the RFI system functioning, it is no surprise that he occasionally loses track of the files over which he has jurisdiction—even those files within his own */root* directory structure. Here, we see him checking his file types. First, he `cd`s to his home directory then checks to be sure he is really there. Then, he creates the *listoffiles* file, whose contents are a listing of the filenames in his */root* directory only (for a start; he can do a recursive `-R` listing or examine individual subdirectories later):

```
# cd<Enter>
# pwd<Enter>
/root
# ls > listoffiles<Enter>
# file -f listoffiles<Enter>
4.1.0 tgz: gzip compressed data, deflated, last modified: Mon
Jun 4 10:52:13 2001, os: Unix
BugReport: ASCII English text
Desktop: directory
.
.
.
```

To repeat, the option we used here was `-f`. The option told `file` to check the listing of files found within the file specified immediately after the `-f` (that is, in the file called *listoffiles*). Note that when using such an argument file to list files for `file` to check, you should ensure that each filename appears alone on a single line in that argument file.

Example 9.13 More Simple file Examples

Sancho knows that he, like us, will soon be learning about the Linux/UNIX text editor called `vi`. He has used it before, but slowly and carefully. He only knows the bare, essential commands. He thinks `file` can tell him more about `vi` as an executable application. Here is how he finds out more:

```
$ file /bin/vi<Enter>
/bin/vi: ELF 32-bit LSB executable, Intel 30386,version 1,
dynamically-linked (uses shared libs), stripped
```

Now, he wants `file` to examine his `dapple` profile file called *dapple*:

```
$ file /home/panzasan/admin/dapple<Enter>
/home/panzasan/admin/dapple: English text
```

We suggest that you take the time to check the `file` `man` page and also scroll through the */usr/share/magic* file to understand more fully how `file` checks for "magic numbers" coded within executable files.

Comparing Text Files: The diff Command

Occasionally, you might want to determine the differences between two text files. For example, two users might be working on different sections of the same report at the office or on the same sections of the same report—or even on the same report at different times. These are times when the `diff` command comes in handy. This command can compare the lines of text in two files or even compare all similarly named pairs of text files in two directories (if the arguments are directory names instead of filenames). The syntax is as follows:

```
$ diff [-options] filename1 filename2<Enter>
```

Thus, the `diff` command enables a user to compare two text files line by line (we will discuss the `cmp` command later, which enables other types of files to be compared and on a byte-by-byte basis). `diff` has numerous options; check your information sources. In Example 9.12, we will use two

common options: the `-q` option, which gives you a quick report on whether two files actually differ, and the `-y` option, which prints the two files side by side on the screen so that you can compare them yourself. We will also see how the `sdiff` command is equivalent to the `diff` command with the `-y` option.

Some other valuable options are listed in Table 9.1.

In Examples 9.12 and 9.13, we will see how `diff` provides a sort of coded output as well as verbatim copies of some lines from the compared files. The codes `diff` provides are a mix of numbers and letters. The numbers refer to the line numbers in each file. Also, in Table 9.2, we list and explain the letter codes that `diff` provides. The `diff` command, its options, and its output codes might seem a bit complicated at first. But with practice, they are easy to grasp and make valuable analytical tools.

Example 9.12 Comparing Files with diff

Don Quixote has suggested that Sancho read and review a number of books if he (Sancho) is to accompany the Don on his noble and chivalrous

Table 9.1 Selected diff Options

OPTIONS	DESCRIPTION
`-b`	Ignore leading, repeating, end-of-line spaces, tab characters, and so on.
`-e`	Print a text editor script that you might use to modify the first file so that it would be identical to the second file.
`-w n`	For two-column output, the total number of characters used for both columns must have a maximum *n* characters. (The default is 130 characters, but most users prefer to set it to 80 or so.)

Table 9.2 diff Output Codes

CODE	DESCRIPTION
`a`	Add or append lines to the first file to obtain the result shown in the second file.
`c`	Lines that have been changed between the first and second file
`d`	Lines deleted from the second file. (Although Example 9.12 did not include a `d`, its meaning is reasonably straightforward.)

adventures. So, in the month or so since he was hired, Sancho has been doing a lot of reading.

Previously, we saw how Sancho had created a file called *manuals* in the Don's directory structure and then created a link to that file called *knightdata* in his own directory structure. Because he has read and reviewed a number of books since he originally created *knightdata*, he has expanded the reading list. What he did was copy *knightdata* to *knightdata.july01* and then added the new titles to *knightdata.july01*. The two files are depicted in Figure 9.5.

Now, Sancho will use `diff` to compare the two files, *knightdata* and *knightdata.july01*. First, he uses `diff` with the simple `-q` option to see whether the files are indeed different:

```
$ diff -q knightdata knightdata.july01<Enter>
Files knightdata and knightdata.july01 differ
```

Next, he will use `diff` without any options to see how the two files differ:

Figure 9.5 The *knightdata* and *knightdata.july01* files.

```
$ diff knightdata knightdata.july01<Enter>
3,5c3,5
< Required Reading
< for
< "Knights-Errant"
- - -
>   Required Reading
>         for
>   "Knights-Errant"
7c7
< 1. Amadia of Gaul
- - -
> 1. The Bible
10c10
< 3. The Bible
- - -
> 3. Amadis of Gaul
11a12,16
> 5. Chronicles of the Nine
      Worthies
> 6. The Garden of Flowers
> 7. The Knight Platir
> 8. Don Olivante de Laura
```

Now, he will use `diff` with the `-y` option to compare the two files side by side. The first file specified will appear on the left-hand side of the screen; the second file specified will appear on the right-hand side:

```
$ diff -y knightdata knightdata.july01<Enter>
```

When the response to this command appears, identical lines are printed side by side, verbatim, like so:

```
Rueful Figures, Inc.  Rueful Figures, Inc.
```

When the lines differ, a pipe symbol (|) appears between them as follows:

```
1. Amadia of Gaul     |     1. The Bible
```

When one file or another has a line that the other does not, then that line is printed with a greater-than sign (>) or a less-than sign (<) symbol printed next to it. In the following line, the second file has a line that does not appear in the first file:

```
            6. The Garden of Flowers
```

Depending on his preference, instead of entering

```
$ diff -y knightdata knightdata.july01<Enter>
```

Sancho could have entered the equivalent:

```
$ sdiff knightdata knightdata.july01<Enter>
```

Example 9.13 diff for Comparing Files

It is not uncommon to have several versions of the same, or essentially the same, document floating around your system. As you can see from Figure 9.6, in the period between March 2001 and August 2001, RFI's Web site development team underwent significant change:

- The team expanded from five to seven members.
- Perez rotated out to resume his other RFI duties. He was replaced by his colleague, Nicholas.

Figure 9.6 The *RFI_Web.mar01* and *RFI_Web.aug01* files.

- Don Quixote had other duties and had to leave the team. He insisted that his place be taken by his horse Rozinante, however. To avoid a confrontation, the other members shrugged, rolled their eyes, and then tolerated his request.
- Juan Haldudo was dismissed by RFI when Lady Dulcinea found he had betrayed RFI to competitors. No great loss—he had never been enthusiastic about the establishment of the Noble Deeds Division, for one, and had only reluctantly contributed to Web site development anyway.
- Lady Molinera, an enthusiastic public relations specialist, had been recruited by the Don and was placed on the Web site team.
- Long-time agricultural colleague and friend Pedro Alonso also joined RFI, albeit reluctantly. He believed in the Citrus Division but was not enthusiastic about "Noble Deeds."
- Ms. Urganda was also recruited. She was a clever, sort of self-taught medical person and also a friend of the Don. She agreed to be called "Ms." because *no one* was willing to call her "Dr."

Let's use this simple example to discuss what `diff` discovers and displays when it is invoked by Juana and when it compares the original Web site development team list (filename: */gutiejua/admin/RFI_Web.0301*) to the new one (filename: */gutiejua/admin/RFI_Web.0801*):

```
$ cd<Enter>
$ cd admin<Enter>
$ diff RFI_Web.0301 RFI_Web.0801<Enter>
3c3
< March 2001
- - -
> August 2001
6,9c6,11
< 2. Don Quixote
< 3. Perez
< 4. Juan Haldudo
< 5. Juana
- - -
> 2. Rozinante (!?)
> 3. Nicholas
> 4. Juana
> 5. Lady Molinera
> 6. Pedro Alonso
> 7. Ms. Urganda
```

```
11,13c13
<
<
< filename: /gutiejua/admin/RFI_Web.0301
- - -
> filename: /gutiejua/admin/RFI_Web.0801
(END)
```

In examining the `diff` output, we can see that three changes were made from */gutiejua/admin/RFI_Web.0301* to */gutiejua/admin/RFI_Web.0801*. There are three lines that include some sort of combination of number and letter codes: `3c3`; `6,9c6,11`; and `11,13c13`. Each of those lines denotes the beginning of a `diff` explanation of the respective change.

The first code, `3c3`, tells us that a change has been made to line 3 from the first file to that of the second file. Counting down from the top of each file, we see that line 3 indicates the date of the member list. Now, note that the line beginning with a less-than sign (`<`) shows us line 3 as it appears in the first file (that is, `March 2001`). Then, there is a line with three dashes that serves to separate line 3 of the first file from what is coming next. What comes next is a line that begins with a greater-than sign (`>`), which shows us line 3 as it appears in the second file (that is, `August 2001`). It is easy to see that `diff` is telling us that the date has been changed from the first file to the second, and if we wanted the files to be identical, we would have to make some sort of change to line 3 in */gutiejua/admin/RFI_Web.0301* or */gutiejua/admin/RFI_Web.0801*. You have probably noticed that the first `3` in `3c3` means line 3 in the first file and the second `3` means line 3 in the second file. The `c` code separates the first from the second.

Lines appearing in the first file but not in the second file begin with a less-than sign (also called a left angle bracket) (`<`). Conversely, lines that appear in the second file but not in the first file begin with a greater-than sign (also called a right angle bracket) (`>`). Any lines that have been modified between the first and second files are shown as `<` and `>` but with three dashes on a line interjected between the two. We have focused on the third line, where `diff` notified us of changes in the dates for the respective team lists.

The second code, `6,9c6,11`, is a little more complicated. The first `6,9` is on the side pertaining to the first file. Then, there is the `c`. Then, there is another `6,11` on the side pertaining to the second file.

What does it mean? Keep reading the output and you see four lines with `< 2. Don Quixote`, `< 3. Perez`, `< 4. Juan Haldudo`, and `< 5.`

Juana on them, respectively. The diff command is telling us that these are lines 6 through 9 in the first file. Then, we see the line with the three dashes, which indicates that references to the first file are complete for now. Also, because there is another 6,11 combination after the c, we can now anticipate seeing lines 6 through 11 from the second file. Sure enough, > 2. Rozinante (!?),> 3. Nicholas, > 4. Juana, > 5. Lady Molinera, > 6. Pedro Alonso, and > 7. Ms. Urganda appear on the next six lines. So, lines 6 through 9 from the first file were completely replaced by new lines 6 through 11 in the second file.

Similarly, the third code—11,13c13—tells us that lines 11 through 13 from the first file were replaced by line 13 in the second file. diff then lists the lines 11 through 13 from the first file: two blank lines indicated by two < symbols followed by > filename: /gutiejua/admin/RFI_Web .0301. Then, there are three dashes on a line followed by what is on line 13 in the second file: < filename:/gutiejua/admin/RFI_Web.0801.

Additional File Comparison Commands: comm and diff3

Two similar commands that might be of interest are comm and diff3. The comm command shows you which information is common to both files as well as the information that is unique to each file. It presents lines that are only in the first file in column one, lines that are only in the second file in column two, and lines that are found in both files in column three.

The diff3 command enables the user to compare three files. The output coding is a little different, but the principles are the same as in diff. For further information, consult your information sources.

Comparing All Types of Files: The cmp Command

In the preceding section, we discussed the diff command, which is used to compare text files line by line. The cmp (file compare) command differs from diff in several ways. First, cmp is used for text files and other types of files as well. Furthermore, cmp compares byte by byte rather than line by line. In text files, cmp makes a character-by-character comparison. Its syntax is as follows:

```
$ cmp [-options] file1 file2<Enter>
```

When used with no options, `cmp` prints a message and stops as soon as it encounters the first difference between the files. The `cmp` response states the names of the files it has compared and then indicates exactly which byte is the first to differ between the two files. In the first part of Example 9.14, you can see that the first byte to differ between the two files being compared (the same team lists we compared in Example 9.13) is character 41 on line 3 (the `3` in the date in */gutiejua/RFI_Web.0301* versus the `8` in the date in */gutiejua/RFI_Web.0801*).

Example 9.14 Are These Files Different?
Comparing Files with cmp and no Options

Juana places a copy of the new and old *RFI_Web.0x01* files (in other words, the Web site development team rosters for March and August 2001) into Lady Dulcinea's directory structure. Later, Lady D. does a quick check to see whether the new file is different from the old one. Here is how she does it, using `cmp` without options:

```
$ cmp /gutiejua/RFI_Web.0301 /gutiejua/RFI_Web.0801<Enter>
/gutiejua/RFI_Web.0301 /gutiejua/RFI_Web.0801 differ: char 42,
line 3
```

Example 9.15 How Different Are These Files?
Comparing Files with cmp and the -l Option

Later, Lady D. returns to check just how different the new Web site development team roster is from the new one. She will compare the two files using `cmp` and the `-l` option:

```
$ cmp -l /gutiejua/RFI_Web.0301 /gutiejua/RFI_Web.0801<Enter>
42   115     101
43   141     165
44   162     147
45   143     165
.
.
.
154  61      146
155  12      151
(END)
```

In Example 9.15, Lady D. asked for a more detailed comparison by typing the `-l` option after the `cmp` command. The command replies by listing *all* the bytes that differ between the two files (five screens full, as it turns out). The output is in three columns:

- The first column contains the decimal value of the byte number.
- The second column contains the corresponding octal value of the character found in that position in the first file examined.
- The third is the corresponding octal value of the character found in that position in the second file examined.

There are no differences until byte number 42 is encountered. The first file, *RFI_Web.0301*, has a character in that position whose octal value is 115. The second file, *RFI_Web.0801*, has a character in that position whose octal value is 101. For all text files, these octal values refer to the values of the characters in the ASCII character set.

The `cmp` command is not the best comparison tool for text files, as you have undoubtedly surmised by now. It is more suitable for comparing data or program object files.

Compressing Files: The gzip, gunzip, and zcat Commands

Data files, including text files, are generally compressed so that they can be stored in less space or transmitted in a shorter period of time. Compression (or compaction) programs generally look for redundancies in files and then use one representative token for all identical pieces of data. In this way, no data is lost.

Compressing Files with gzip

Linux uses a command called `gzip` to compress data in specified files using an algorithm called Lempel-Ziv (LZ77) coding. The syntax is as follows:

```
$ gzip [-options] filename(s)<Enter>
```

Other UNIX-like operating systems might use `gzip` or another program called `compress`, which uses a different version of the Lempel-Ziv coding and is generally used with the `tar` command for archiving data. The `gzip` command can compress (ASCII) files by as much as 80 percent.

By default, `gzip` names the compressed file with the original filename but appends `.gz` to the end of the filename to indicate that it is a compressed

file, as shown in Example 9.16. It then deletes the original file. Also by default, `gzip` transfers the original ownership and permission modes and access and modification times to the new *.gz* file.

You can use numerous options to enhance the use of `gzip` or to override the defaults. For a complete listing and description of options, consult your information sources. Three of the handiest options are described in Table 9.3.

Example 9.16 gzip with the Verbose (-v) Option

Don Quixote's niece and housekeeper in La Mancha knows that the Don and Sancho will be traveling in August. They never know how to reach the Don when he is traveling (they are reluctant to say "when he's on the road" because he has landed on gravel and asphalt so many times before). So they call Juana and ask for a copy of the telephone lists. Juana is glad to comply. She combines all three *RFI_Tel** files into one file called *RFI_Phone#s* and then compresses that one file in preparation for transmission by e-mail. Here is how she creates the single file with `cat` and then uses `gzip` with its `-v` (verbose) option so that she can see that the task is done:

```
$ cd /home/gutiejua/phone#s<Enter>
$ cat RFI_Tel* > RFI_Phone#s<Enter>
$ gzip -v RFI_Phone#s<Enter>
RFI_Phone#s:   58.0% -- replaced with RFI_Phone#s.gz
```

Table 9.3 Selected gzip Options

OPTION	DESCRIPTION
`-v`	Verbose. Print the original name and the new compressed file-name along with the percent compression to which the file has been subjected.
`-r`	Recursive. When used with a directory name, execute `gzip` on all files in the directory and from that directory down through the file system.
`n`	Speed of compression; *n* is a number from 1 through 9. 1 executes compression at the fastest speed with the least compression. 9 executes compression at the slowest speed, resulting in the highest compression. The default value is 6.

Examining and Manipulating Compressed Files with zcat

The next command, `zcat`, is used for two reasons:

- To examine the contents of a compressed file without going through the process of uncompressing it
- To use a compressed file as input to one or more piped commands

We use `zcat` for the first purpose in Example 9.17. The output from `zcat` is the same as the output of the uncompressed file, similar to what you would get with `cat`. You can see that `zcat` does the same thing for compressed files that its counterpart `cat` does for uncompressed files. Like `cat`, `zcat` cannot go back and forth through the file output. For this purpose, you might want to use the `zless` command, which behaves just like the `less` program (see Chapter 5, "Using Files in Linux," to review the `less` command) except that it requires compressed files as input arguments. The syntax for `zcat` is

```
$ zcat [-options] filename(s)<Enter>
```

Example 9.17 Examining Compressed File Contents with zcat

After Juana has combined the telephone list files into one and then compressed that one file (see Example 9.16), she wants to check that the compressed file is in good shape as far as content and format before she transmits it to La Mancha. She uses `zcat`, piped to `less`, to do so:

```
$ zcat RFI_Phone#s.gz | less<Enter>
```

`zcat` responds with an output similar to the one she would have received if she had used `cat` on the original uncompressed file.

Uncompressing Files with gunzip

The `gunzip` command uncompresses files that have been compressed with `gzip` or `compress`. Basically, `gunzip` reverses the `gzip` process, restoring the compressed file to its original uncompressed components. Its syntax is as follows:

```
$ gunzip [-options] filename(s)<Enter>
```

This command uses most of the same options as `gzip` (see your information sources). For example, we will use the `-v` option with `gunzip` in Example 9.18.

Example 9.18 gunzip

Once Don Quixote's niece receives the telephone list (filename: *RFI_Phone#s*; see Example 9.16) from Juana at RFI, she uncompresses it so that she can read and print it. Here is how she uses `gunzip` to do so:

```
$ gunzip -v RFI_Phone#s.gz<Enter>
RFI_Phone#s.gz:58.0% -- replaced with RFI_Phone#s
```

NOTE **The `zip` and `unzip` commands might also be available with your Linux distribution. You might prefer to use them instead to compress and uncompress files.**

Displaying Nonprintable Characters: The cat Command Options

Here, we discuss three options that you can use in conjunction with the now-familiar `cat` command (refer to Chapter 5 for our original discussion of `cat`) to determine the following:

- Whether invisible or otherwise unprintable characters exist in text files or filenames (that is, directories)
- The nature of the spaces in your text files

The second determination might be important because of unexpected `diff` command output or problems with output from the `sort` command.

When someone is composing a document or specifying a file or directory name, there are key sequences that can be entered accidentally that will cause unexpected results in documents or file or directory names. Take a look at Table 9.4, for instance, for a listing of some odd key sequences that are sometimes entered when you use the `vi` text editor. The table also shows what the key sequences will look like during the creation of the file in `vi`, when the same file is `cat`'d at the command line and also when it is `cat`'d with the `cat`'s `-etv` options.

The fact that no characters will show up with `cat` makes these what we call nonprintable. Thus, it might not be possible to resolve these issues or problems based on the typical viewing of directory or file contents. Using

Table 9.4 Some Odd vi Key Sequences That Can Lead to Confusion

KEY SEQUENCE	VI RESULTS	CAT RESULTS	CAT -ETV RESULTS
<Ctrl>-4	^\		^\
<Ctrl>-5	^]		^]
<Ctrl>-7	^_		^_
<Ctrl>-g	"beep"	deletion	deletion
<Ctrl>-i	<Tab>		^I
<Ctrl>-k	^K	move to lower line and continue	^K
<Ctrl>-l	^L	move to lower line and continue	^L
<Ctrl>-r	"	delete	delete
<Ctrl>-t	<Tab>		
<Ctrl>-p	^P		^P
<Alt>-2		@	
<Alt>-7	{	{	

`cat` with the options discussed here *might* help you determine how at least some of the file contents or names were created and therefore might help you figure out how to read or manipulate them later. The syntax is the same as for `cat` in other circumstances, of course:

```
$ cat [-options] [file(s)]<Enter>
```

The options used in Example 9.19 are defined in Table 9.5.

Table 9.5 Selected cat Options

OPTION	DESCRIPTION
-e	Display $ to indicate the end-of-line key; that is, Enter or Return.
-t	Display ^I to indicate a Tab used as a space.
-v	Display ^G to indicate the otherwise invisible Ctrl-G key sequence.

Example 9.19 cat for Displaying Nonprintable Characters

Juana checks her "Requests" files (*Req-K1*, *Req-K2*, and *Req-K3*) every morning. In late July, she reads this garbled message that originates from Don Quixote:

```
$ cat Req-K1<Enter>
Date: July @, 2001
     Subject: Request for Advance Funds for August Trip
Status: Urgent
Deadline: ASA
From: Quixote de
                 a Mancha

     need 100 Euros as an advance. For the August trip.
Sancho will requisition his own funds, etc.
zinante needs new equipment [shoes, blanket] and provisions.
Will likely leave   August     at }:3} HRS
lus, can someone make an appointment for me at Toledo spa?
     now the manager?
     hope this meets with your approval and
Very much appre¢iate your ¢ooperation. t to . ady Dulcinea's, too.
Signed, Q de
                   a M
```

She calls the Don and discusses the situation. Once everything is satisfactory, she calls Freston, who suggests that she do the following with the same message:

```
$ cat -etv Req-K1<Enter>
Date: July @^\, 2001$
^ISubject: Request for Advance Funds for August Trip$
Status: Urgent$
Deadline: ASA^P$
From: Quixote de ^La Mancha$
$
^I need 1^]00 Euros as an advance. For the August trip.$
Sancho will requisition his own funds, etc.$
zinante needs new equipment [shoes, blanket] and provisions.$
Will likely leave ^IAugust     ^_ at }^]:3} HRS$
^Plus, can someone make an appointment for me at Toledo spa?
^Know the manager?
^I hope this meets with your approval ^I and ^Lady DulM"-inea's,
too.$
^[[AVery much appreM"-iate your M"-ooperation. t to .$
Signed, Q de ^La M$
```

It helped Juana decipher *some* of the key sequences but not all. To determine all of them, Juana would have to see the original document in Don Quixote's files. Again, the options used in Example 9.19 were defined previously in Table 9.4.

Use cat in a Pipe Sequence for ? in Filenames

When executing listings of directory contents, you might sometimes encounter filenames containing a question mark (`?`). The creator of such a filename might have intentionally inserted the question mark, but this symbol can also indicate the presence of other characters that cannot be readily interpreted by the shell. When dealing with these odd filenames in directories, you can use `cat` as the second process in a piped set of commands:

For example, a typical `ls` command might reveal the following:

```
testfile1    te?stfile2
```

Now, we try `cat`:

```
$ ls | cat -vt<Enter>
```

The output might look something like the following:

```
testfile1
te^Gstfile2
```

This example shows a Ctrl-G (that is, `^G`) in the middle of what otherwise would have been called *testfile2*. How do you correct the filename? You could simply rename the file as follows:

```
$ mv te?stfile2 testfile2<Enter>
```

NOTE **You cannot enter `te^Gstfile2` at the command line. If you try, you will hear an error "beep" from Linux.**

If the correction strategy fails, you can try another technique. First, you need to obtain the inode number of the file with `ls -li`. (Remember that the filename displayed is probably *te?stfile2* because a simple `ls -li` does not display the hidden Ctrl-G characters.)

Assume that the inode number for the file is 1311. Now, enter the following:

```
$ find . -inum 1311 -exec mv {} testfile2 \;<Enter>
```

NOTE **Artificially creating this situation is a bit difficult. But things such as these do happen. We stuck to Ctrl-G (or ^G, if you prefer that style of notation) as our chosen invisible character because it is the only one that Linux enables us to use.**

Assigning Unique Filenames: Appending Information

Sometimes an application—or a user or a programmer, too—has to assign a unique name to a file. In this section, we discuss two methods of assigning unique names:

- Automatically appending the process number as a suffix to the end of a filename
- Automatically appending the date as a suffix to the end of a filename

Appending a Process Number to a Filename

By appending two dollar signs to the end of a filename, as shown here and in Example 9.20, the shell automatically appends a two- to five-digit process number to the end of that filename. This technique will likely produce a unique filename every time. The syntax is as follows:

```
$ touch filename$$<Enter>
```

Example 9.20 Appending a Process Number to a Filename

Every day, Pedro Alonso, in RFI's Citrus Division, orders new equipment, fertilizer, seed, cuttings, and so on. He needs a way to make the daily orders files unique. What he does is:

```
$ touch citdiv_order$$<Enter>
$ ls citdiv_order*<Enter>
citdiv_order342 citdiv_order1105 citdiv_order12038
```

NOTE **For both techniques, we are using the `touch` command to create example files. You, of course, can create files in many ways.**

Appending a Date to a Filename

Although appending a process number to a filename will produce a unique filename, some people prefer the suffix to tell them a little more about the file. This reason is why the other method, appending a date as the suffix, is popular.

But before we discuss the appending date suffix method, let's take a closer look at the possible output of the `date` command. The following is a typical shell response to the `date` command when used by itself:

```
$ date<Enter>
Wed Jul 14 12:48:32 CDT 2001
```

When creating filenames, though, you probably do *not* want to include the text names of the day or month (such as Wed or July). Instead, you want to use their numerical equivalents (such as 07 for July). The numerical equivalent of the preceding date is generated as follows:

```
$ date +'%y%m%d%H%M%S'<Enter>
000714124832
```

Here, we ask the shell for the date in the format of year (`y`), month (`m`), day (`d`), hour (`H`), minute (`M`), and second (`S`).

As seen in Example 9.21, the filename is formatted as a basic filename, followed by a dot as a separator, followed by the date as an extension in the month-day-hour format. If you will be generating more than one filename per hour, though, you should add more precise date fields (minute or even second) to the format to prevent files from being overwritten (and thus, losing valuable information). The syntax is

```
$ touch filename.`date +'[options]'`<Enter>
```

No matter what you decide, be aware of the command syntax. Note that two types of metacharacters are used: the single quotation mark (`'`) and the back quote (`` ` ``).

Example 9.21 Appending the Date to a Filename

After just a few weeks, the *citdiv_order$$* files are starting to confuse Pedro and his other Citrus Division colleagues. Freston suggests that they use the

following date-based technique to assign unique filenames to their order files:

```
$ touch citdiv_order.`date +'%m%d%H'`<Enter>
$ ls testfile.*<Enter>
citdiv_order.072113 citdiv_order.072215 citdiv_order.072313
```

Everyone agrees that it is a little easier now to figure out when orders were created just by looking at the filenames.

Exercises

1. Log into the system and ensure that you are in your home directory. Create a new subdirectory called *newdir*. Change to this new subdirectory and create five zero-length files in it. Name the files `1`, `2`, `3`, `4`, and `5`:

   ```
   $ mkdir newdir<Enter>
   $ cd newdir<Enter>
   $ touch 1 2 3 4 5<Enter>
   ```

2. List the contents of the *newdir* subdirectory. Then, pass the output to `xargs` to copy the files and rename them with the prefix `file` so that the resulting copied filenames are *file1*, *file2*, and so on. Finally, verify that the files were copied and that their names were assigned accordingly:

   ```
   $ ls<Enter>
   $ ls | xargs -t -i cp {} file{}<Enter>
   $ ls<Enter>
   ```

3. Copy the */etc/passwd* file into your *newdir* subdirectory so that you have one file in the directory containing text. All other files in the subdirectory will be zero length. Using `find`, `xargs`, and `grep`, `cat` the contents of the files that contain text:

   ```
   $ cp /etc/passwd .<Enter>
   $ find . -type f | xargs -t grep '.*' | cat<Enter>
   ```

4. Identify the directory in which the `find` command is located:

```
$ whereis find<Enter>
```

or

```
$ which find<Enter>
```

5. Determine what type of file (such as executable, ASCII, or directory) the `find` command is:

```
$ file /usr/bin/find<Enter>
```

6. Using the `find` command, create a recursive listing of all files starting in your home directory and redirect the output to a file named *myfiles*. Using the redirected *myfiles* file, determine the type of files located in the list:

```
$ find /home/teamxx > myfiles ; file -f myfiles | less<Enter>
```

or

```
$ ls -R > myfiles<Enter>
$ file -f myfiles | less<Enter>
```

or

```
$ find /home/teamxx | xargs file | less<Enter>
```

7. Create a file called *list1*. In *list1*, list the names of several people you know. Copy *list1* to a file called *list2*:

```
$ vi list1<Enter>
i
Name 1
Name 2
Name 3
.
.
.
<Esc>
:wq<Enter>
$ cp list1 list2<Enter>
```

Edit *list2* and make the following changes: Change the spelling of one of the names; remove one of the names; and add a new name.

```
$ vi list2<Enter>
i
.
.
.
<Esc>
:wp<Enter>
```

8. Using `diff`, compare the contents of *list1* and *list2*:

```
$ diff list1 list2<Enter>
```

Did you notice anything?

9. Using `cmp`, compare the contents of *list1* and *list2*. Then, invoke a complete or long comparison of the contents of both files.

```
$ cmp list1 list2<Enter>
$ cmp -l list1 list2<Enter>
```

10. Copy the */usr/share/magic* file to a file in your home directory named *mymagic*. Do a long listing of *mymagic* and record the number of bytes in the file.

```
$ cp /usr/share/magic mymagic<Enter>
$ ls -l mymagic<Enter>
```

11. Using the verbose option with `gzip`, compress *mymagic*. Record the percentage of compression and the name of the new compressed file.

```
$ gzip -v mymagic<Enter>
```

12. Do a long listing of the file and record the number of bytes. Compare this number to the number in the preceding instruction. Is there an approximate match on the percentage of compression?

```
$ ls -l mymagic.gz<Enter>
```

13. Using `zcat` or `zless`, expand and view the contents of the file you compressed in the preceding exercise. (Use `zcat` if it is a small file or `zless` if it is a large file.)

```
$ zcat mymagic.gz | less<Enter>
```

or

```
$ zless mymagic.gz<Enter>
```

14. Using `gunzip` with its verbose option, restore the compressed file back to its original characteristics.

```
$ gunzip -v mymagic.gz<Enter>
```

15. Invoke a long listing of the uncompressed file and record the number of bytes. Is the number the same as, greater than, or less than the number you recorded for Exercise 12?

```
$ ls -l mymagic<Enter>
```

16. In your home directory, create a file named *invis* and type a few lines that include random Tab, space bar, and Ctrl-G key presses between the words. (Feel free to replace the sample file contents with your own.) Then, display the file.

```
$ vi invis<Enter>
i
<Tab> this<Tab>file has<Enter> several <Tab> mysterious
spaces<Enter> and^G non-^Gprint<Tab>able<Enter>
characters.<Tab>^G <Esc>
:wq<Enter>
$ cat invis<Enter>
```

17. Note in the preceding exercise that when you displayed the contents of `invis`, it did not look quite right. Display and locate all the nonprintable characters to determine where you used tabs, spaces, and other nonprintable characters.

```
$ cat -etv invis<Enter>
```

See Appendix B for answers.

Quiz

1. True or false: The most important characteristic of the `find` command is its capability to travel both ways through the tree-like hierarchy of a file system.
2. When you use quoting metacharacters with `find`, which of the following interprets the wildcards?

   - `find` itself
   - The shell
   - Depends on the shell you are using at the time

3. Which of the following commands is used to determine the type of data in a file?

   - `cmp`
   - `diff`
   - `find`
   - `grep`
   - `file`

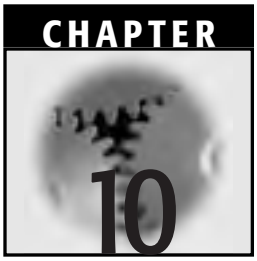
4. True or false: The `gzip` command deletes the original file and replaces it with a compressed copy of the file and then renames the copy with the original name but appends a `.z` extension to it.
5. To display nonprintable characters in a file, which of the following command lines would you use?

   - `ls -la`
   - `cat -etv`
   - `find -inv`
   - `grep -s`
   - `file -v`

6. True or false: The `diff` command compares only text files.

See Appendix C for answers.

CHAPTER



10

The vi Editor

Linux provides several text editor programs, from the simplistic `vi` to the more elegant `Emacs`. The main reason for discussing `vi` (the visual front end for the actual editor, `ex`) in its own chapter is that you are likely to run into it no matter what type of UNIX-related system you encounter. The `vi` editor is on almost every system because it requires comparatively little space and still does the job adequately. When you look for it, please remember that it might not be called `vi` exactly, but it will be called something similar (such as `vim`).

The `vi` editor can be a bit of a tough slog, but you will need it to deal with Linux or with any other UNIX-based operating system on a regular basis. Another reason for dedicating a chapter to `vi` is that it does not have a lot of interactive help (or any other kind). It does have a `man` page, however. Because `vi` fronts for `ex`, you can also check the `man` page for `ex`.

Before you begin editing text, you might want to determine which other text editors are available on your system. Some are installed automatically as part of the base system; others are installed only during custom (or expert) installations. Check your */bin* directory for names that might be familiar to those we will mention here. Or, check your installation guide or

user guide. You will probably find that every Linux distribution has a specific command that you can enter to check whether a program has been installed, but you have to be able to enter the precise program name or something very close to it (using wildcard symbols, perhaps).

Other text editors you might have at your disposal are `Emacs`, `ed`, `joe`, `jed`, or variations on those names, depending on the environment you will be using them in (for example, `jed` versus `jed-xjed`) or the extra features one might have compared with another (for example, `vim-minimal` versus `vim-enhanced`).

One last word: `vi` and other text editors are *not* word processing programs, so do not set your expectations too high. They are valuable tools nonetheless because they facilitate administration of UNIX-like operating systems.

An Introduction to vi

As mentioned, `vi` is the standard editor in all UNIX-related systems. The original `vi` editor was distributed with the Berkeley Software Distribution; the various Linux distributions use a form of *Visual Editor Improved* (`vim`), which claims to be an improvement over the classic `vi`. When you enter `vi` *filename* to invoke the editor, you are actually using a symbolic link to `vim`. `vim` then emulates the classic `vi` editor (an instance of upward compatibility).

The following features are fairly standard across `vi` versions and types:

- Full-screen editor
- Two modes of operation: Command and Insert
- Use of one-letter commands
- Unformatted text
- Flexible search-and-replace facility with pattern matching
- User-defined editing features using macros

This chapter does not pretend to be a comprehensive summary of `vi`. Consult the various Linux information sources for more detailed descriptions of `vi` features. In addition, we have provided a command summary in Appendix A.

We will describe and use two `vi` modes: Command and Insert. Some Linux/UNIX gurus claim that `vi` has three modes: Command, Insert, and Last-line. Users are in Last-line mode when they have used Esc to leave

Insert mode and then typed a colon (:) so that they can enter specific single-letter commands to quit, save, and so on. In this chapter, we fold the Last-line mode in with Command mode.

Starting vi

Assuming that you have logged into Linux and are now facing the command-line prompt, enter the following to invoke vi:

```
$ vi filename<Enter>
```

If the specified file already exists, vi creates a copy of it and puts the copy into a buffer in the */tmp* directory for you to work on. If the file does not exist, vi opens an empty buffer in the same directory and gives the file the name specified in your vi command.

When invoked, vi checks for a file called *.exrc* and incorporates any specifications found there (we discuss *.exrc* later in this chapter in a section titled *Options for Changing How vi Operates*). Then, vi starts in Command mode and waits for directions from you. What you see on the screen is a flashing cursor and the filename at the bottom of the screen. If the file is new, the editor tells you so on the last line. For instance, if you invoke vi to create a new file called *RFI_trip_laundry*, you would see an empty screen similar to that in Figure 10.1. Dashes represent blank lines, and like we said, the filename appears on the last line.

If the file already exists, Linux gives you the filename in double quotes as well as the number of lines and number of characters in the file. Figure 10.2 shows you what might result if you had created *RFI_trip_laundry*, exited vi for a while, and then invoked vi again to edit the same file. As you can see from the bottom of the screen, the file has 13 lines with 172 characters (equal to all the characters, blanks, and line feeds).

Exiting vi

To exit from the vi editor, you must be in Command mode. To ensure that you are in Command mode before inputting commands at any time, press Esc. You can exit vi in one of the following ways (all end with pressing Enter):

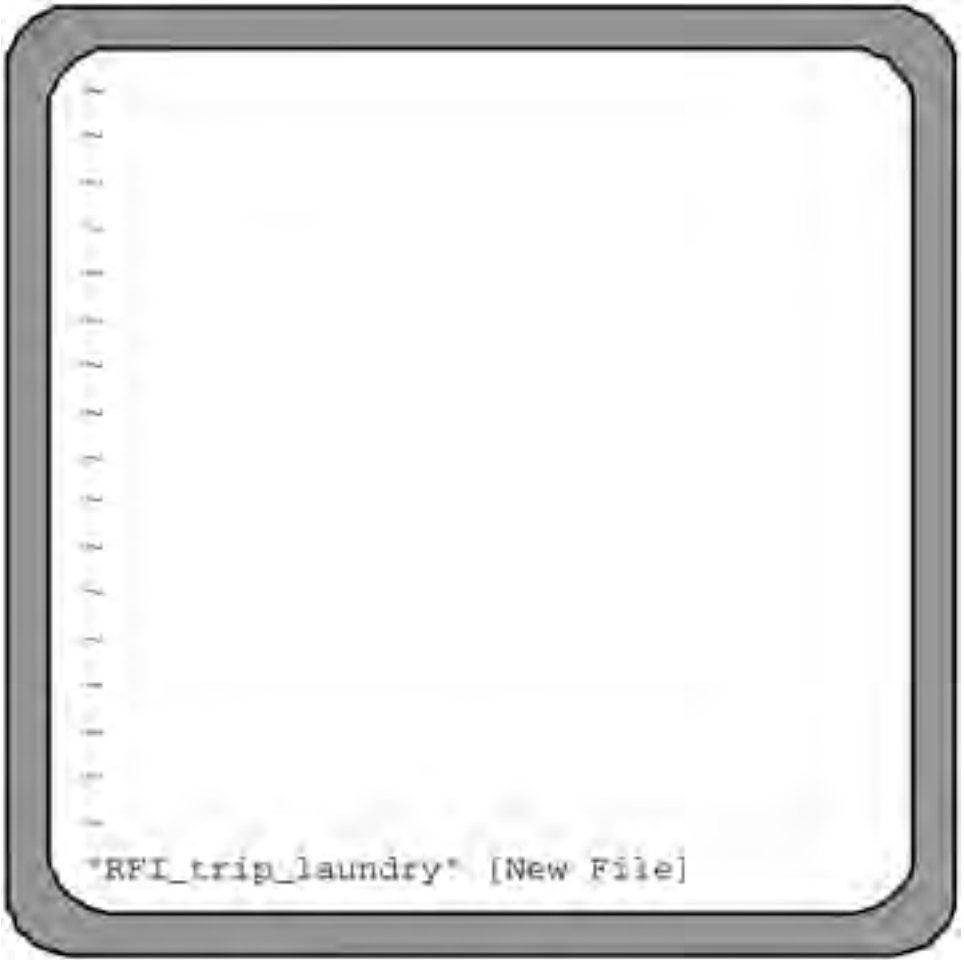


Figure 10.1 Starting vi.

- `:q` quits `vi` without saving your file.
- `:wq` writes changes and quits.
- `<Shift>-zz` writes changes and quits.

The `:q` option works only if you have not made any changes. If you have made changes and you try to quit this way, Linux gives you the following message:

```
No write since last change (use ! to override)
```

Figure 10.2 Editing with vi.

If you want to exit at this point, type one of the following:

- `:wq` writes changes and quits (as we mentioned earlier).
- `:q!` quits without writing changes.
- `:x` also writes changes and quits.

If you choose `:wq` or `:x`, `vi` will display the updated status of the file (in other words, `"filename" Line count, Character count`) then quit and return you to a shell prompt. If, on the other hand, you enter `:q!`, then `vi` will simply quit and return you to a shell prompt.

Adding Text in Insert Mode

Let's say that you open a file in `vi` and want to enter text. Then, you must be in Insert mode (also known as input mode or text mode). Take the following steps to get into Insert mode and then insert text:

1. Using the up and down arrows, position the cursor at the point in a new or existing file where you want to begin inserting text.
2. Use one of the following single-letter commands:
   - `a` adds text immediately after the cursor.
   - `A` (that is, Shift-A) adds text beginning at the end of the line on which the cursor is sitting.
   - `i` inserts text beginning at the same position presently underlined by the cursor.
   - `I` inserts text at the beginning of the line on which the cursor is sitting.
   - `<Insert>` also inserts text beginning at the same position presently underlined by the cursor.
3. Add your text.

The difference between `a` and `i` will become apparent with practice. Note that while you are adding text, the filename, which was at the bottom of the terminal screen, has been replaced by `- INSERT -`. This message is a reminder that you are in Insert mode.

If you want to save your text as you go, press Esc to re-enter Command mode and then type the following:

```
:w<Enter>
```

This action saves the text you have entered thus far, updates the file's status for you (`"filename" Line count, Character count`), and keeps the file available for input. To resume entering text, use one of the single-letter commands just listed.

When you finish adding text, you need to leave Insert mode by pressing Esc. You are then returned to Command mode. Then, you will see that your file is still on the screen, but the filename is no longer displayed at the bottom.

Manipulating Text in Command Mode

The first method you need to know in order to manipulate text is how to move around in the `vi` editor. Table 10.1 provides a list and corresponding descriptions of typical methods for moving your cursor in `vi` while in Command mode. There are even more maneuvering commands available than those listed in Table 10.1, too. Check `vi`'s `man` pages or other information sources for further help.

Deleting Text

There are many commands available for deleting text in Command mode. We list several ways of doing so in Table 10.2 along with two commands to use if you want to undo the deletion. For additional commands, check Appendix A as well as the `man` pages and other information sources.

Searching for Text Strings

When you are in Command mode, pressing the forward slash (`/`) button automatically puts you in text search forward mode and takes you to the last line on the screen, where `vi` has placed a forward slash prompt. Similarly, pressing the question mark (`?`) automatically puts you in text search backward mode and takes you to the last line on the screen, where `vi` has placed a question mark prompt.

After the prompt, enter the string of text you want to search for and press Enter. The search begins in the chosen direction from the position of the cursor. The cursor stops underneath the first character of the first found text string. If you want to continue searching in the same direction, press `n`. If you want to search in the opposite direction, press `N`. Eventually, you will reach the bottom or top of the file, and `vi` will notify you when you do.

To exit from text search, simply enter any other command. You do not press Esc first.

Searching for and Replacing Text

Manually cutting and pasting can be tiresome, inefficient, and occasionally inaccurate. The following is a command for automating it when you are in Command mode in `vi`:

Table 10.1 Cursor Movements in vi

MOVEMENT WITHIN A LINE	CURSOR MOVES
<Left Arrow> or h	One character to the left
<Right Arrow> or l	One character to the right
0(zero)	To the beginning of the line
$	To the end of the line
MOVEMENT AMONG WORDS	**CURSOR MOVES**
w	To the next word
b	Back to the previous word
e	To the end of the existing word; if the cursor is already at the end of a word, then it will move to the end of the next word
MOVEMENT WITHIN THE SCREEN	**CURSOR MOVES**
<Up Arrow> or k	One line up
<Down Arrow> or j	One line down
H	To the beginning of the top line on the screen
M	To the beginning of the middle line on the screen
L	To the beginning of the last line on the screen
SCROLLING AMONG SCREENS IN THE SAME FILE	**CURSOR MOVES**
<Ctrl>-f	Forward to the top of the next screen
<Ctrl>-b	Backward to the bottom of the previous screen
GENERAL MOVEMENT WITHIN THE FILE	**CURSOR MOVES**
1G	To the first line of the file
xxG	To line number xx of the file. Substitute the line number you want to travel to.
G	To the last line of the file

Table 10.2 Key Sequences for Deleting Text in vi

KEY SEQUENCE	DELETE ACTION APPLIED
`x`	To a single character
`dw`	To the end of the current word
`d$`	To the end of the line
`d0`	To the start of the line
`dd`	The entire line
`dG`	To the end of the file
`u`	Undo the last change
`U`	Restore the entire line (only if the cursor has not left the line)

```
:g/searchstring /s//replacementstring /g<Enter>
```

The command works as follows:

- The first `g` (global) tells the editor to perform a search for the first occurrence of the text string in every line of the file.
- The two forward slashes bracket the search string. Please note that in this case, with */searchstring /*, the characters are followed by a space. If the search string is found ending a line or immediately followed by punctuation, it is not replaced.
- The `s` means substitute.
- The forward slash following the `s` tells the editor to use the text string preceding the `s` as the target for the substitution (that is, the preceding text string will be the one replaced).
- The next two forward slashes bracket the text string to be substituted for the replaced text string (in this case, `/replacementstring /`; again, the characters are followed by a space).
- The last `g` tells the editor to make the substitution at every occurrence in each line found by the first `g`.

An extra feature we could have added to the syntax example is a `c` before the last `g` (that is, `/cg`). The `c` tells the editor to ask for confirmation before making each change.

Another of many ways you could modify the command is to make changes to only one line or to the first occurrence on every relevant line.

See your information sources for additional ways to carry out a search and replace.

Example 10.1 Searching for and Replacing Text in vi

Inspired by Quixote, Sancho devotes what spare time he has—that is, when he is not reading about knights and chivalry—to working on his own poetry skills. In Figure 10.3, you can see the latest version of his latest magnum opus, "Fleas," which he admits freely is still a "work in progress." Sancho is not satisfied yet. First, he wants to replace the four occurrences of the contraction *I'll* with the simpler *I*. Here is how he does it:

```
Fleas

You bite those bugs and give a yelp
You scratch anon, sometimes it helps
Oh, but you suffer, my dear old mutt!
You chase your tail, you drag your butt

I'll rub your belly and your back
To stem that burning itch attack,
I'll scratch your ears, I'll scratch your head,
But those cursed fleas fill me with dread!

They are entrenched, they won't go 'way!
Try as we might, they aim to stay
They live thru the "powders", they live thru the
"collars"
Makes you want to spit and holler!

Ahhh! You know what? What we missed, amigo?
To rout the fleas, for that final "flea-go"?
There is just one way to end their wrath
Yes, amigo, you must take a bath!
~
"Fleas" 21L, 654C
```

Figure 10.3 Searching for and replacing text in vi.

- He opens "Fleas" with `vi`.
- He is probably already in Command mode, but he makes sure by pressing Esc.
- Then, he enters his search and replace command.

Here are his commands:

```
$ vi fleas<Enter>
<Esc>
:g/I'll /s//I /g<Enter>
```

In this example, `vi` found all the occurrences of the first string `I'll` and replaced them with the string `I`.

Moving and Copying Text by Characters and Words

As we mentioned previously, `vi` utilizes 36 buffers into which you can cut or copy (or, as `vi` says, `yank`) file data. Unless you specify a buffer number or letter, the data is cut or copied into buffer 0 by default.

There are also many commands available for copying or moving text in Command mode. We list several ways of doing so in Table 10.3, along with two commands to use if you want to undo the action. These commands are handy enough for copying or moving text but are not as popular as the "line by line" technique we will describe next. Meanwhile, for additional commands to those we list in Table 10.3, check Appendix A as well as the `man` pages and other information sources.

Moving or Cutting Text Line(s) by Line(s)

The `vi` editor utilizes 36 buffers (numbered 0 through 9 and lettered `a` through z) into which you can cut or copy file data. Unless you specify a number or a letter, the data is cut or copied into buffer 0 by default. Here is the general line-by-line text moving procedure:

- In Command mode, move the cursor to the line you want to move.
- Press `dd`. The specified line disappears into buffer 0 (the default buffer), and the cursor moves to the next line.

Table 10.3 Key Sequences for Moving and Copying Text in vi

KEY SEQUENCE	ACTION APPLIED
`cw`	Remove all characters to the end of the current word. Puts you in `- - Insert - -` mode. Press Esc and `p` as often as you like to paste the word into the file.
`c$`	Remove all characters to the end of the line. Then, same behavior/actions as `cw`.
`c0`	Remove all characters to the start of the line except the character the cursor is under. Then, same behavior/action as `cw`.
`cG`	Remove all characters to the end of the file. Then, same behavior/actions as `cw`.
`u`	Undo the last change.
`U`	Restore the entire line (only if the cursor has not left the line).

- Move the cursor to the line after which you want to place the specified line.
- Press `p`. The original line appears after the line on which you placed the cursor.

If you press `P` instead of `p`, the original line appears *above* the line on which you had placed the cursor.

If you want to do more complicated moving of text, such as moving several strings at once, try the following:

To cut a line into buffer 2, move the cursor to that line and type

```
"2dd
```

To cut a word into buffer 3, move the cursor to that word and type

```
"3dw
```

To cut a line and the three lines following it into buffer c, move to that line and type

```
"c4dd
```

Move to the various target locations and type "2p, "3p, and "cp, respectively (remember, the double quotation mark is essential), to place those text strings where you want them. You can do so repetitively, out of order, or however you want because copies of the text strings remain in their respective buffers until they are replaced.

NOTE ***Preceding the alphanumeric key combinations with one double quote is essential when specifying buffers.*** **If you do not use the quotation mark, the system interprets the rest of the string as some type of bungled command and either does not respond or gives you an error message.**

Example 10.2 Moving Lines of Text in vi

Here is a simplistic example of moving text. In Command mode, Sancho moves the cursor to the line in "Fleas" that he wants to move (note the underlined O in the third line of text):

```
You bite those bugs and give a yelp
You scratch anon, sometimes it helps
Oh, but you suffer, my dear old mutt!
You chase your tail, you drag your butt
```

Press dd. The specified line disappears into buffer 0 (the default buffer), and the cursor moves to the next line. If necessary, move the cursor to the line after which you want to place the specified line. (The cursor is already in the correct place in this example, under the Y in the now second line.)

```
You bite those bugs and give a yelp
You scratch anon, sometimes it helps
You chase your tail, you drag your butt
```

Press p. The original line appears after the line on which you placed the cursor. Now, the cursor is on the newly placed line.

```
You bite those bugs and give a yelp
You scratch anon, sometimes it helps
You chase your tail, you drag your butt
Oh, but you suffer, my dear old mutt!
```

Copying Text Line by Line in vi

Here is the technique for copying text line by line:

- In Command mode, move the cursor to the line you want to copy.
- Press `yy`. The line is copied into buffer 0 (the default buffer), but the original line remains on the screen. The cursor, meanwhile, stays where it was.
- Move the cursor to the line below which you want to place the yanked line.
- Press `p`. The yanked copy of the original line appears below the line where you placed the cursor. Now, the cursor moves to the newly placed (in other words, the copied) line.

As with moving text, if you want to do more complicated copying—say, copying several strings at once—do the following:

To copy a line into buffer 2, move the cursor to the line and type

```
"2yy
```

To copy a word into buffer 3, move the cursor to that word and type

```
"3yw
```

To copy a line and the three lines following it into buffer c, move to that line and type

```
"c4yy
```

Move to the various target locations and type `"2p`, `"3p`, and `"cp`, respectively, to place those text strings where you want them.

As with moving text, you can copy text repetitively or out of order because copies of the text strings remain in their respective buffers until they are replaced. In the multiple-copy example, just as in the multiple-move example (Example 10.2), `"2yy`, `"3yw`, and `"c4yy` are all preceded by one quotation mark. Remember, you must precede alphanumeric key combinations with one double quote when specifying buffers.

Meanwhile, notice that `vi` echoes, at the bottom of the screen, the actions you have taken (for example, `xxx lines yanked; xxx more lines`) as they are completed.

Example 10.3 illustrates a simple example of copying text.

Example 10.3 Copying Text Line by Line in vi

Sancho has heard that repeating the last line in a stanza is a poetic and songwriting technique used for emphasis, and he wants to see whether it works in "Fleas." So, in Command mode, he moves the cursor to the line he wants to copy (there is the cursor—just below the O in the second line):

```
You chase your tail, you drag your butt
Oh, but you suffer, my dear old mutt!
```

He presses `yy`. The line is copied into buffer `0` (the default buffer), but the original line remains on the screen. Also, the cursor stays where he placed it.

Now, he moves the cursor to the line below which he wants to place the yanked line, which is still in the buffer. (In this case, because he just wants to repeat the last line of the stanza, the cursor is already in the correct location. So he keeps it where it is.)

```
You chase your tail, you drag your butt
Oh, but you suffer, my dear old mutt!
```

Then, he presses `p`. The yanked copy of the original line appears below the line where he had left the cursor. Now, the cursor is on the newly placed (in other words, the copied) line.

```
You chase your tail, you drag your butt
Oh, but you suffer, my dear old mutt!
Oh, but you suffer, my dear old mutt!
```

"No," he concludes, "it was okay the way it was."

The Undo Buffer

If you are deleting, moving, or copying text, the most recent buffer you have put a text string into is the undo buffer, whether you have specified it (such as the `2` or `3` or `c` in the multiple-move example) or whether it is the default `0` buffer (if you have not specified one).

Using the multiple-move example, after you have cut the three text strings, the undo buffer is the `c` buffer, not the `2` or `3` buffer. The only action you would be able to undo with the `u` command is `"c4dd`. To undo your last command, simply press `u`. If you have made several changes on

one line *and* you want to undo all of those changes *and* you have not moved off that line yet, press `U`.

Executing Linux/UNIX Commands in vi

Suppose that while you are working in `vi`, you realize that you need to exit `vi` to run a command. But you do not really want to exit `vi`, run the command, and then re-enter `vi`. What is your solution? Using the exclamation point (`!`) command within `vi` creates an appropriate shell to execute the chosen command, prompts you for the command, executes it, and displays the results.

Here, we will show how to copy a file while remaining in the `vi` editor instead of having to exit `vi`, use the `cat` command on the file, write down the information, re-enter `vi`, and type the information in the file.

To copy the contents of one source file into a target file, enter the following:

```
$ vi targetfilename<Enter>
```

You will see the contents of *targetfilename* on the terminal screen.

Press Esc to enter Command mode.

Move the cursor down to where you want to add the information from *sourcefilename*.

Now, list the contents of the directory where *sourcefilename* resides:

```
:!ls /directorypath<Enter>
```

`vi` will respond with the following:

```
[No write since last change]
"- - other files and directories - -" RFI_trip_laundry
Hit ENTER or type command to continue _
```

Type the following:

```
:r /directorypath/sourcefilename<Enter>
```

`vi` will respond with

```
"sourcefilename" Line count, Character count
Hit ENTER or type command to continue _
```

To complete the procedure, press Enter to insert the contents of the *sourcefilename* file into *targetfilename*. By default, the text from *sourcefilename* will be inserted just below where you left your cursor in *targetfilename* before you pressed `:!ls etc.`

If beforehand, line numbering had been set to `On` in the *targetfilename* file, you could have inserted a line number before `:r /directorypath/sourcefilename` to tell the `vi` editor to place the contents of *sourcefilename* there instead. (Line numbering is discussed later in this chapter; the line numbering command is described in Table 10.3.)

By the way, this method is a common technique for dumping the output of the `date` command into letters, memos, and other documents.

Please note that when you are prompted with `Hit ENTER or type command to continue`, if you need to run a series of commands without returning to `vi` after the execution of the first command, type

```
:sh
```

Then, you can run all your commands in the shell. When you want to exit the shell and return to `vi`, press Ctrl-D.

Example 10.4 Executing Linux/UNIX Commands from vi

Don Quixote is producing a staff notice called *Knight_Laundry*, a list of the minimal protective and other clothing that knights-errant should take with them on their travels. He has already created an up-to-date list of such materials in the *RFI_trip_laundry* file, which we saw in Figure 10.2 (were you wondering when you would see that list again?). To copy that file into his *Knight_Laundry* notice, he enters the following:

```
$ vi Knight_Laundry<Enter>
```

He sees, so far, that the *Knight_Laundry* file reads like the following on the screen:

```
All RFI knights-errant should obtain the
following beforehand and bring it with them
on their Noble Deeds Division missions:
"- - blank line - -"
```

```
~
~
```

He presses Esc to enter Command mode. Then, he moves the cursor down to the blank line below "`. . . Noble Deeds Division missions:`", where he wants to append the information from *RFI_trip_laundry*.

Now, he lists the contents of the directory where *RFI_trip_laundry* is located:

```
:!ls<Enter>
[No write since last change]
"- - other files and directories - -" RFI_trip_laundry
Hit ENTER or type command to continue _
```

He types

```
:r RFI_trip_laundry<Enter>
"RFI_trip_laundry" 6 lines, 40 characters
Hit ENTER or type command to continue _
```

Finally, now he presses Enter to insert the contents of the *RFI_trip_laundry* file into *Knight_Laundry*. By default, the text is inserted just below where Don Quixote had left cursor in that file (in other words, on the blank line below "`. . . Noble Deeds Division missions:`"). So, *Knight_Laundry* now resembles Figure 10.4.

Options for Changing vi Functions

You can change several appearance and behavior characteristics of `vi` to make it more useful or convenient. We list some of them in this section, and we also discuss how you change characteristics.

The two basic approaches to changing `vi` characteristics are to:

- Set default options that remain in place after you close the current session (with the hidden *.exrc* file).
- Specify options for single-session use.

When you invoke `vi`, it searches for your own specific default characteristics in a hidden file called *.exrc* in the current directory. If `vi` finds an *.exrc* file, it follows whatever default specifications are listed there. If it does not find the *.exrc* file in the current directory, it checks your home directory for it. Please note, though, that a *.exrc* file might not even exist. If

Figure 10.4 Executing Linux/UNIX commands from vi.

no one has specifically configured `vi` to alter its default behavior, then there is no need for *.exrc*.

At any rate, have a look in your current directory and/or home directory for a hidden *.exrc* file. If you cannot locate one, you can actually use `vi` to create one. Invoke `vi` with *.exrc* as its argument (in other words, to create *.exrc*), get into Insert mode, and enter any of the options listed in Table 10.4 or in any other `vi` information source. We have only listed a dozen or so options in Table 10.4; there are many more. Some affect the way text is presented; others facilitate editing (especially for new users).

If you want to invoke any of the Table 10.4 options on the fly while you are already in a `vi` session, then simply press Esc to get into Command

Table 10.4 vi Customizing Options

OPTION	DESCRIPTION
`set all`	Display all settings
`set`	Display settings different than the default settings
`set ai`	Turn on autoindent
`set noai`	Turn off autoindent
`set nu`	Turn on line numbering
`set nonu`	Turn off line numbering
`set list`	Display nonprintable characters
`set nolist`	Hide nonprintable characters
`set showmode`	Show the current mode of operation
`set noshowmode`	Hide the current mode of operation
`set ts=4`	Set tabs to four character jumps
`set ic`	Ignore case sensitivity
`set noic`	Set case sensitivity
`set wrapmargin`	Set the margin for automatic word wrapping from one line to the next; a value of 0 turns off word wrapping

mode and then type a colon (`:`) followed by the option as you see it in Table 10.4. We must provide a note of clarification, though: when you list any of the Table 10.4 options in the *.exrc* file, *do not* put a colon at the beginning of any of them. Again, the colon is used *only* when you are doing an interactive-style specification in Command mode when you are already in a `vi` session.

You can also set options for a specific session rather than every time you invoke `vi`. For example, to number lines as you enter text, go to Command mode and type the following:

```
:set nu<Enter>
```

At the end of the session, you will (naturally) leave `vi`. The next time you invoke the `vi` editor, though, line numbering will be off as usual (unless you or someone else has entered `set nu` in the *.exrc* file. Table 10.4 lists com-

monly specified options. For more comprehensive listings, check your information sources.

Entering and Editing Commands at the Command Line

Before we discuss editing at the command line, it might be useful to mention Linux/UNIX shells again. Shells are, among other things, Linux/UNIX command interpreters, and we described them in more detail in Chapter 7, "Shell Basics."

Let's quickly summarize by stating that when a user logs in to Linux, one of the several available shells is invoked by default according to what is found in the user's profile in the */etc/passwd* file. Because many principles, utilities, and commands are similar from shell to shell, we will focus here on the Bourne Again Shell, or Bash shell. (We also learned in Chapter 7 how a user can switch from one shell to another by using some simple commands, however.)

Basic Command-Line Navigation and Editing

With Bash, you type commands by using letters, numbers, and some metacharacters and then execute the commands by pressing Enter. To navigate back and forth along the command line, you can use the right arrow and left arrow keys. To delete text, you can use the Delete and Backspace keys. You can use some of the other handy keys, too, such as Home and End . The Shift, Alt, Ctrl, and function keys (that is, F1, F2, and so on) are a different story, however. Using them can provide some utility (examples: Ctrl-o presents the `ls` command automatically; Ctrl-p presents the last command executed; the Alt key, used with characters such as 2, 4, 7, 8, 9, 0, -, c, e, or }, can create other special characters, and so on). But sometimes, using them can produce unpredictable or inconvenient results as well.

The Up and Down Keys and Command History: A Review

Remember back near the end of Chapter 7 when we introduced you to the power and convenience of the up and down keys? We can use them to tap into our command history, starting with the RAM buffer where commands

from the current session are saved, and then proceeding into the hidden file called *.bash_history* in our *$HOME* directory. Using Up or Down allows us to recall and re-execute, then, commands that we have executed previously.

When you back up enough to enter the hidden *.bash_history* file, your initial position in that file is at the bottom, or the most recent command before your last logout. You can specify the number of old commands that will be placed in the *.bash_history* file, so you can travel back a fair distance in your command history (we will discuss how to do that in Chapter 11, "Shell Variables and the User Environment"). But pressing the Up key once you reach the top of *.bash_history*, however, results in a shell error beep.

To summarize, the benefit of using Up and Down is that you can repeat procedures by recalling commands and then pressing Enter to re-execute. That can save you a lot of typing, especially with respect to complicated syntax. You can also perform sensitivity analyses by re-executing commands while changing options or arguments with each execution.

Invoking Text Editor Features at the Command Line

Linux also enables you to invoke, at the command line, the editing features from its various text editors. For example, if you type the following at the command line, Linux enables you to use some `vi` commands and keystrokes:

```
$ set -o vi<Enter>
```

To turn the features off, type

```
$ set +o vi<Enter>
```

The "invoking other editor features" facility can be handy for UNIX-based operating system users and administrators when they find themselves at different locations (especially at different keyboards). The feature can bring some predictability to command-line activities.

For example, invoking `vi` features might let you use the `h`, `j`, `k`, and `l` keys to emulate the left, down, up, and right arrow keys, respectively, to navigate the command line or to recall previous commands by tapping into the buffer and your *.bash_history* file.

With the flexibility of the text editor features, there are issues, though. Please read the warning carefully.

WARNING **Be careful when invoking editor features at the command line. You might get unpredictable (and thus, inconvenient) results. Plus, if you try to turn off `vi` features (with `set +o vi`, for instance), you might succeed in turning off `vi` features but you might also turn off previous bash command-line features. It could leave you almost helpless at the command line. You might have to log out and log back in again to regain your previous command-line utility. That sounds easy, but because your command-line functionality is severely reduced, you will have trouble keying in `logout`. Try using the `j` and `k` keys to move up and down through your command history to find a `logout` command somewhere.**

The "ambush" described in the warning is the reason why we approach even the updating of this section of the book with caution. It seems like just checking on these commands can get us into trouble. Hopefully, you will not suffer any similar inconvenience.

Related vi Editors

Here is a quick summary of other editors loaded by Linux that are related to the `vi` editor:

- `view` is the read-only version of `vi`. Changes cannot be saved unless overridden by using an exclamation mark.
- `ex` is a line-oriented text editor, but it can access `vi`'s screen-editing capabilities.
- `ed` is a line-oriented text editor with no visual or screen capabilities.

Check the */bin* directory for any of those names. You can then experiment with them by invoking them at the command line by using a syntax like the following:

```
$ ex filename<Enter>
```

You also might be fortunate enough to have `Emacs`, an elegant and powerful visual screen text editor. Unfortunately, `Emacs` uses a lot of disk storage, so it is not installed universally. But it ships with some Linux distributions and is available on the Internet.

Do not be fooled by the presence of an *emacs* directory (for example, */usr/share/emacs*) on your system. It does not necessarily mean that `Emacs` is installed.

Exercises

1. Go to your home directory and create a file in it called *vitest*:

```
cd<Enter>
vi vitest<Enter>
```

2. When you open a `vi` file, you are automatically placed in Command mode. Press the `i` (insert) key to switch to Insert mode. You can also press the `a` (append) key. The use of `i` or `a` simply determines whether typing will start before or after the cursor. Although there is no indicator when you are in Command mode, the text string `-INSERT -` appears at the bottom of the screen when you are in Insert mode.

   Switch from Insert mode to Command mode by pressing the Esc key. Press Esc a second time. Note that if you press Esc twice, you will probably get a beep from your terminal (be careful, though: some ASCII terminals do not beep). The beep indicates that you are already *in* Command mode. Now, press `i` again to go back into Insert mode, and continue to the next exercise.

3. Insert the following text *exactly* as it is presented (otherwise, you have to alter the steps we will present later), line by line. Then, type the alphabet one character per line. The following shows `a` through `c` only, but please continue on to `z` on your terminal. Adding the alphabet, one letter to a line, is an easy way to fill a few screens of information quickly and easily, which you will need for later exercises.

```
This is a training session about the usage of the vi editor. We need
some more lines to learn about the most common commands of the editor.
We are now in the Insert
mode and we will switch right after this to the Command mode.
a
b
```

```
c
.
.
.
```

4. Return to Command mode. Write and quit this new file. Note that as soon as you press the colon (:) key, it appears below the last line of your input area. After the buffer is empty and the file is closed, you see a message giving the number of lines and characters in the file.

```
<Esc>
:wq
```

or

```
<Shift>-zz
```

5. Open the *vitest* file by using `vi`. Note that the bottom line of the screen indicates the name of the file and the number of lines and characters it has found in the file.

```
$ vi vitest<Enter>
```

6. Using both the arrow keys and the `h`, `j`, `k`, and `l` keys, practice moving the cursor down one line, up one line, to the right a couple of characters, and back to the left a couple of characters.
7. You might not want to move the cursor one character or one line at a time throughout an entire file. Practice by using cursor movement keys to navigate page by page or line by line. Using the cursor movement keys mentioned in Exercise 6, position your cursor at the first line of the file. While in Command mode, do the following:

   Move forward one page:

```
<Ctrl>-f
```

   or

```
<PageDown>
```

NOTE **There is no PageDown key on ASCII terminals.**

Move back one page:

```
<Ctrl>-b
```

or

```
<PageUp>
```

Scroll the screen up to one-half the screen size:

```
<Ctrl>-u
```

Move the cursor to the last line in the file:

```
<Shift>-g
```

Move the cursor to the first line in the file:

```
1<Shift>-g
```

or

```
:1<Enter>
```

Move the cursor to line 4 of the file:

```
4<Shift>-g
```

or

```
:4<Enter>
```

Move the cursor to the end of the line:

```
$
```

Move the cursor to the beginning of the line:

```
0 (zero)
```

8. Move your cursor to the top of the file.

```
1<Shift>-g
```

or

```
:1<Enter>
```

Search for the word *entry*. Your cursor should be on the *e*. Switch to Insert mode and add the word *text* with a space after the word.

```
/entry<Enter>
i
text
```

9. Move the cursor to the space after the word *mode* on the same line. Insert a comma. Remember, you are still in Insert mode.

```
<Esc>
```

Position the cursor to the space after the word *mode*.

```
i
,
```

10. Enter Command mode. Position the cursor anywhere on the line beginning with *learn the most*. Insert a blank line to form two paragraphs.

```
<Esc>
```

Position the cursor on the line that starts with *learn the most*.

```
o
```

The lowercase `o` opens the blank line following the line beginning with *learn the most*.

11. Opening up a blank line, as in the preceding exercise, automatically puts you in Insert mode. Therefore, return to Command mode. Now, save the changes you have made thus far, but do not exit the editor.

```
<Esc>
:w<Enter>
```

12. While still in Command mode, remove the characters `c`, `e`, and `g` but leave the blank lines in their place; in other words, do not delete the entire line, just the characters. Then, go back and remove the blank lines. This situation is an opportunity to use two of the delete functions.
    - Position the cursor on the `c`; press `x`.
    - Position the cursor on the `e`; press `x`.
    - Position the cursor on the `g`; press `x`.
    - Do the following *twice*: Position the cursor on a blank line and press `dd`.
13. Now, replace the character `h` with `z`. Position the cursor on the `h`.
    - Press `r`.
    - Press `z`.
14. Assume that you decided you do not want to save the changes to the characters. Quit the editing session without saving the changes made since the last save.

```
:q!<Enter>
```

15. Edit *vitest* one more time.

```
$ vi vitest<Enter>
```

Do the following:

- Copy the first paragraph one line at a time to the end of the file.

- Position the cursor on line 1; press `yy`.
- `<Shift>-g;` press `p`
- `2<Shift>-g`; type `yy`
- `<Shift>-g`; press `p`
- `3<Shift>-g`; type `yy`
- `<Shift>-g`; press `p`

When you complete the preceding actions, copy the second paragraph all at once to the end of the file:

- `4<Shift>-g`; type `3yy`
- `<Shift>-g`; press `p`

16. Assume that you decided the lines you just added to the end of the file do not look right. Delete all of them with a single command. Position the cursor on the first copied line to be deleted at the bottom of the file. Count the number of lines to delete.

    Press `5dd`.

17. Now, before you do anything else with this file, assume that you need to embed the current date and time as the first line of the file. Do this action without leaving the `vi` editor.

```
:!date > datefile<Enter>
Press RETURN or enter command to continue<Enter>
:0r datefile<Enter>
```

18. Options can be set temporarily in an editing session by using the `set` command. Return to the top of the file.

```
1<Shift>-g
```

    Ensure that you are in Command mode, and set the following commands:

    - Set automatic word wrap to 15 spaces before the right margin.
    - Display the Insert mode message when in Insert mode.
    - Turn line numbering on.

```
<Esc>
:set wrapmargin=15<Enter>
:set showmode<Enter>
:set number<Enter>
```

19. Test each of the options set in Exercise 18. You should see that the lines in the file are automatically numbered just after the command was entered. Try entering Insert mode by typing `i` or `a`. You will see the `- INSERT -` message at the bottom-left of your screen. Type a few continuous lines of text to test the automatic word wrap feature.

    Enter Command mode by pressing Esc. The `- INSERT -` message disappears from the bottom of the screen.

20. Write the file and quit the editor.

```
:wq<Enter>
```

21. To set up a command-line editing session, use the `set -o vi` command.

```
$ set -o vi<Enter>
```

22. Now, you can recall previously executed commands, edit them, and resubmit them. Let's build a command history. List the contents of the */usr* directory (do a simple list, not a long list). Display the contents of the */etc/filesystems* file. Echo *hello*.

```
$ ls /usr<Enter>
$ cat /etc/filesystems<Enter>
$ echo hello<Enter>
```

23. Assume that you want to edit one of the commands you just executed. Press the Esc key to get to `vi` Command mode. Try pressing the `k` key several times to go up the list of commands. Try `j` to go down. This recall of commands is essentially a browse through a buffer of commands that you previously executed in this session. After you log out, the commands are stored in your *.bash_history* file in your home directory.
24. Retrieve the `ls` command. Use the `l` key to move your cursor to the slash in */usr*. (Note: The arrow keys tend to wipe out your line. You have to use the `l` key for right cursor movement and `h` for left cursor

movement.) Use the `j` key to insert text and change this command to a long list, and then execute by pressing Enter.

- `k` to get to the `ls /usr` command
- `l` to get to the `/`
- `i` to get into Insert mode; you could also use an `a` to append if the cursor was on the space before the `/`
- `-l` to change the `ls` to `ls -l`

25. Recall the `cat` command. This time, list the contents of the */etc/passwd* file:
    - Press Esc.
    - Press `k` (to get to the previous `cat` command).
    - Move the cursor to the `f` in `filesystems`.
    - Press `D` (to erase from the `f` to the end of the line or `dw` to simply erase the word).
    - Press `a` (to begin appending text).

    To execute the */etc/passwd* file, type

    ```
    passwd<Enter>
    ```

26. Recall the `cat` command. Go to the end of the line (use `$`). Add to the end of that command by piping the output of the `etc/passwd` command to `wc`, which counts the lines in the *etc/passwd* file.
    - Press Esc.
    - Press `k` (to get to the previous `cat` command).
    - Press `$` (to get to the end of the line).
    - Press `a` (to begin appending text).

    Type the following to execute the *etc/passwd* file again and to get `wc` to count the number of lines in *etc/passwd* and print the number to the screen:

    ```
    | wc -l<Enter>
    ```

See Appendix B for answers.

Quiz

1. When using the `vi` editor, what are the two modes of operation?
2. While using `vi`, how do you get into Command mode?
3. Which of these single-letter commands could you use to enter text?
   - `a`
   - `x`
   - `i`
   - `dd`
4. True or false: While in Command mode, pressing `u` repeatedly will undo all previously entered commands.
5. True or false: The `vi` editor can be used to globally change the first occurrence of a pattern on every line with a given pattern.
6. Which of these will enable you to quit `vi` while in Insert mode?
   - `:qw`
   - `:wq`
   - `<Shift>-zz`
   - `<Shift>-ss`
   - `:q!`
   - All of the above
   - None of the above

See Appendix C for answers.

Shell Variables and the User Environment

This chapter describes many of the variables found in both the terminal and the shell environments. Linux supports three types of variables:

- Terminal environment variables
- Built-in variables
- User-defined variables

We will also show you the two principal methods of setting variables: variable substitution and command substitution.

Variables can be set so that all or only specified processes can use them. Note, however, that some variable setting commands differ from shell to shell.

Meanwhile, sometimes you might wish you had more control over your Linux/UNIX environment. Perhaps you would like to execute some commonly used commands but would rather use the DOS names because, well, you know them already. Or perhaps you would prefer to use a different default shell at login than bash. How could you do that? We will also discuss the evolution of your Linux environment to set the stage for where

and why you might, or might not, customize it. Topics covered include techniques to undertake any desired Linux environment customization, the use of aliases for certain commands so that you can remember or execute them more easily, and methods for manipulating and re-executing previously executed commands in a fast and easy manner.

Variables and the Terminal Environment

Before we can discuss variables in detail and their influence on the Linux terminal and shell environments, it is worthwhile to first define variables and the terminal environment.

A *variable* is a name that is known to a program and that represents data. Although the value of the data might remain constant, it more likely changes one or more times while the program is running.

The *terminal environment* is basically the set of all variables and shell functions to which all your shells and their commands, utilities, and other processes have access. The terminal environment is thus the set of variables that all those commands, utilities, and processes have in common, but it is not some all-encompassing set of all variables from all shells. This definition varies from the more classic definition of a terminal environment; that is, the list of variables pertaining only to your computer terminal (the ones you can see by typing `stty -a` at the command line). For our purposes, the variables in the terminal environment include the `stty` variables and variables exported to the environment by various profiles, daemons, startup files, and even the shells themselves as your system was configured (as it starts up and as you log in).

As you learn more about Linux, you will no doubt customize your terminal environment by altering the values of existing variables and by defining and exporting other variables into your environment. You can rely on the variables that shipped with your distribution and that were initialized during your installation and configuration, or you can add to or change them as you become more familiar with variable and process functionality.

How do you identify the variables in your terminal environment and their respective values? Go to a command line and type `env`.

At some point, you might decide that one or more of your shell variables should be added to your environment so that you do not have to keep defining them from session to session or from shell to shell. How do you add variables to your terminal environment? If you are in the `bash` shell, you first set the shell variable by typing

```
$ variablename=value<Enter>
```

(See the *Equals Sign* section later in the chapter for an explanation of this syntax.) Then, to export that variable, you type

```
$ export variablename<Enter>
```

Now, if you are in the `tcsh` shell, the `export` command is different. After you have set the variable, you type

```
$ setenv variablename value<Enter>
```

The `set` and `setenv` commands are discussed again later in this chapter. See Chapter 12, "Linux Processes and Process Control," for more information about exporting variables.

Shell Variable Types

As discussed in Chapter 7, "Shell Basics," the shell is the interface between the user and the operating system, and it passes information about the terminal environment to applications, commands, and processes. Your shell must have access to several variables so that it can control your Linux session. When you check your shell for all its variables, you will find that it has inherited all the terminal environment variables plus all the shell variables that have not been exported to the environment.

Thus, the set of shell variables (that is, the shell environment) is larger than the set of terminal environment variables. Have a look at Figure 11.1; it illustrates the relationships among the environments that we have discussed to this point. It is *not* meant to show the relative sizes of those environments, just the relationships.

Linux supports the following three types of variables:

Terminal environment variables. Part of the operating system environment. Users do not necessarily have to define these, but they can use them in shell programs; some can be modified within shell programs (for example, `PATH`, `USER`, and `SHELL`).

Built-in variables. Provided by Linux's operating system and are automatically set by the shell to make decisions within a shell script or program (for example, `$#`, `$?`, `$0`, `$*`, `$$`). Users cannot modify these variables.

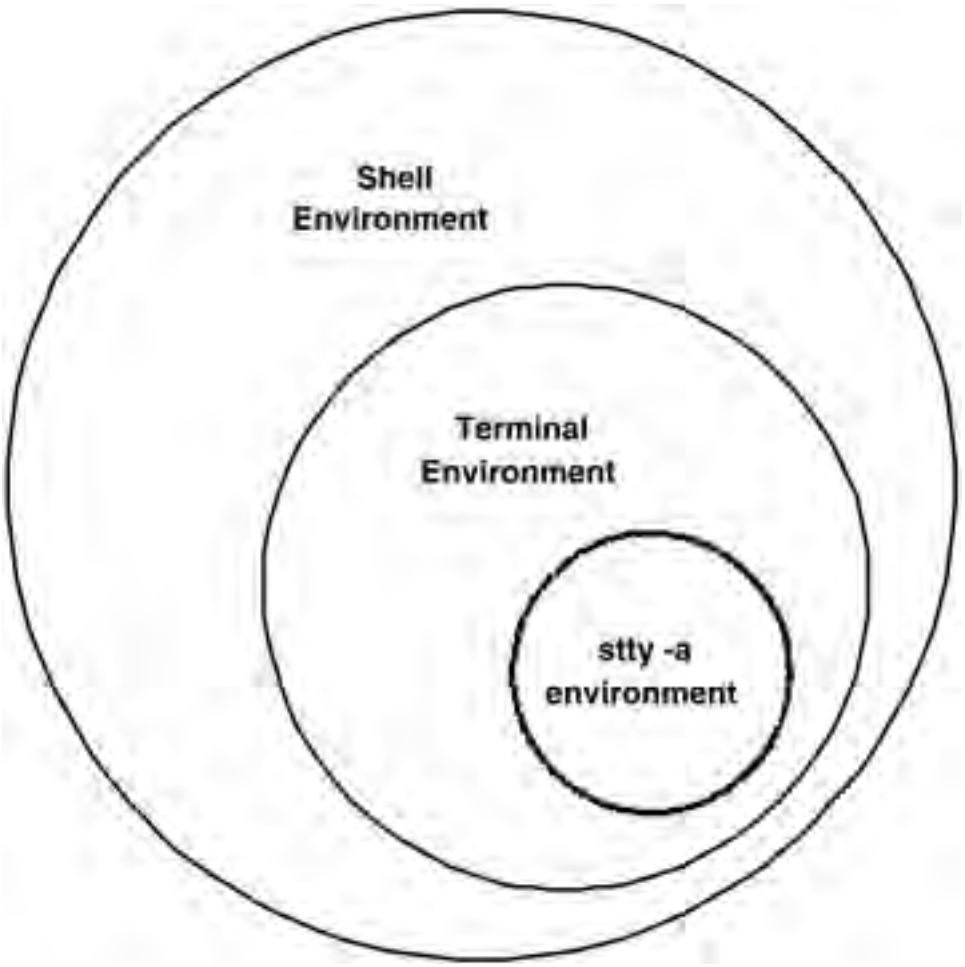


Figure 11.1 Relationships between environments.

User-defined variables. Defined by a user when a shell script is written. Users can use and modify these variables anytime within a shell program.

The shell uses all these variable types to define a user's Linux session. All shell environment variables are case sensitive. For instance, if you define a variable called `path`, it will not be the same as `PATH`. Built-in variables and terminal environment variables are generally uppercase, and those defined by the user are generally lowercase. This situation is not a rule but rather a convention. When you observe this convention, however, user-defined variables will not interfere with the operation of terminal

environment or built-in variables, and you will find it easier to keep them straight when reading your variable listings or doing your own shell programming.

Listing Variable Settings: set Command

To identify your shell environment variables, you can type

```
$ set<Enter>
```

It might be advisable to type the following instead in case the listing of variables is longer than one screenful:

```
$ set | more<Enter>
```

or

```
$ set | less<Enter>
```

Actually, the customary use of the `set` command is to assign values to variables, as we will see later in the *Variable Substitution* section, as follows:

```
$ set variablename=value<Enter>
```

If you do not use options or arguments with the `set` command, however, the shell reports back with a listing of all shell variables as well as their values (as you will see in Example 11.1).

The `set` command is a shell-related command and thus provides a different listing according to the shell that is currently being used. Because the `set` command reports the variables set in the current shell, it also reports the variables from the terminal environment. As mentioned, the terminal environment variables are a subset of the shell environment variables. Although your current shell has access to the values of these variables, all your commands might not have the same access because they might be invoked from different shells. The `export` command enables you to export the variables and their values to the terminal environment so that all commands will have that access, too.

Near the end of the `set` variable listing, you will probably notice a line that begins with an underscore (_). This line is related to the command you executed before `set`. The next time you execute `set`, different characters will likely appear following the underscore because you will have entered a different command before executing `set`.

Example 11.1 Listing Environment Variables with set

Sancho wonders why he always has to specify the directory path whenever he invokes the `banner` command (as we saw in Chapter 3, "Getting Started Using the Linux System"). Freston says the reason is probably because `banner`'s directory path is not specified in Sancho's `PATH` variable. To check the `PATH` variable, Freston tells Sancho to go to his terminal and type

```
$ set<Enter>
COLORS     /etc/DIR_COLORS
               set

addsuffix
argv     ()
autologout   60
cwd     /home/panzasan
dirstack     /home/panzasan
dspmbyte     euc
echo-style     both
edit
file     /home/panzasan/.i18n
gid     603
group     knights1
history     1000
home     /home/panzasan
.
.
.
path (/usr/local/bin /bin /usr/bin /usr/X11R6/bin)
.
.
.
uid     604
user     panzasan
version     tcsh 6.10.00 (Astron) ... etc.
```

(If you are in the `bash` shell and you type the same command, be prepared to see something similar but still quite different. Remember, Sancho works in the `tcsh` shell). So, Freston was right. The */usr/games* directory is not specified as part of Sancho's `PATH` variable.

Listing the Values of Individual Variables: The echo Command

If you want to check to see what a variable is set to, simply enter the `echo` command followed by a dollar sign and the name of the variable, as such:

```
$ echo $variablename<Enter>
```

Note that you add a space between the `echo` command and the dollar sign but *not* between the dollar sign and the name of the variable being referenced.

Example 11.2 Using echo $ to Check Variable Settings

Sancho works in the `tcsh` shell. He asks Don Quixote, who he knows works in the `bash` shell, whether the Don's path variable contains */usr/games*. The Don enters

```
$ echo $PATH<Enter>
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/quixoted/bin
```

Both realize that */usr/games* is not specified in either of their `PATH`s.

Setting Shell Variables

User-defined variables can hold any type of data (for example, integer numbers, single words, text strings, or complex numbers). It is up to the application referencing the variable to decide what to do with the contents. In contrast, the content of system-defined variables is fairly static and inflexible. For example, the `HOME` variable can contain only a directory and *not* a file.

When we say "set shell variables," we mean "specify, or set, the *value* of a variable." The values of variables can be set in two ways:

- By a form of direct definition called *variable substitution*
- To the output of a command or a group of commands, which is called command substitution. Command substitution is discussed later in this chapter.

We will look at each technique in turn.

Variable Substitution: Setting Variable Values with the Equals Sign

There are two ways to use variable substitution. This first method is simple and straightforward and does not really rely on the "character substitution" aspect of variable substitution, which we will illustrate in the next section.

When setting the values of variables, the commands differ depending on the shell you are using. To set a variable when you are in the `bash` or `pdksh` shell, use the equals sign with the name of the variable (existing or new) on the left side and the value you want to set for the variable on the right side:

```
$ variablename=value<Enter>
```

There are *no* spaces between the characters. To set a variable in the `tcsh` shell, on the other hand, use the `set` command:

```
$ set variablename = value<Enter>
```

In the `tcsh` case, spaces appear on both sides of the equals sign. See Example 11.2 for setting variables in all three shells.

NOTE **The `man` page for each shell generally describes the syntax used to set their respective prompts. Do not forget—if you are in the `tcsh` shell and you want to export variables and their values to the terminal environment, you have to use `setenv` instead of `export`.**

Example 11.3 Setting Variable Values with the Equals Sign (=)

Both Don Quixote and Sancho find that the commands they type tend to require more than one line. But the secondary prompt (`PS2`) variable, the simple right angle bracket symbol, tends to confuse them occasionally and they make mistakes on the second line. Here is how they each change their respective secondary prompts. Because he works in the `bash` shell, the Don sets the `PS2` variable as follows (similar to the `pdksh` shell):

```
$ PS2="Please Continue >"<Enter>
```

Sancho, who uses the `tcsh` shell, types

```
$ set PS2 = "More >"<Enter>
```

The variable substitution in Example 11.3 is reminiscent of the latter part of Chapter 7, where we used a backslash followed by Enter to interrupt a long command on one line so that we could continue typing the command on the next line. We mentioned that the greater-than sign (>) was assigned to the secondary prompt through the value of the `PS2` variable.

Thus, in Example 11.3, the Don and Sancho changed from the secondary prompt > to a text string to avoid confusion with the standard out redirection symbol. Note that to do so, they each had to include double quotation marks at each end of the text string. If they did not use the double quotation marks, they would have received an error message beginning with `bash:syntax error`. So, whenever you set a variable with a text string, *always* enclose the text string in double quotation marks.

Variable Substitution Using Character Substitution

This method is a handy technique that can save you keystrokes almost every day. We will define it the best we can here, but to get the best grasp of it, we refer you to Example 11.4. In this case, examples work best.

Begin by defining variables and specifying values for them as you have done before. For the `bash` shell,

```
$ variablename=value<Enter>
```

Then, as you create correspondences, memos, and such, you insert the variable names right into your text using a dollar sign as the first character of the variable name. Then, when the time comes, the shell will substitute the value of the variable for the variable name you typed.

Look closely at Example 11.4. It illustrates some typical and successful variable definition and substitution scenarios. But one scenario produces unexpected and incorrect results.

NOTE **To set a variable equivalent to a string of text, surround the text string with double quotation marks.**

Example 11.4 Setting Variables by Character Substitution

Juana is learning to use the character substitution technique to save her some time when she is organizing Lady Dulcinea's schedule. She finds that she uses the word *day* in many situations. So, she will try to create a variable called `abc` and assign the value `day` to it. Here, she sets and verifies the `abc` variable:

```
$ abc=day<Enter>
$ echo $abc<Enter>
day
```

Now, she checks to see whether it works by substitution:

```
$ echo Tomorrow is Tues$abc <Enter>
Tomorrow is Tuesday
```

So far so good. It works when no space appears *before* the variable. The substitution is not hampered.

But here, something seems to go wrong. The shell tries but fails to substitute:

```
$ echo There will be a $abclong meeting<Enter>
There will be a meeting
```

When no space is encountered *after* the variable name, the shell was tricked into looking for the wrong name. In this case, it searched in vain for a variable it thought was called `abclong`, which we know did not exist because we did not define it. When the shell could find no definition for this (wrong) variable name `abclong`, it returned with a null string. That is, the shell did not make a substitution but rather, removed the portion of the statement containing the wrong or undefined variable name.

To try again to substitute, this time successfully:

```
$ echo There will be a ${abc}long meeting<Enter>
There will be a daylong meeting
```

When you cannot have a space following the variable name, you can preserve the proper variable name by surrounding it with curly brackets ({ }). Note, however, that `$` must remain *outside* the curly brackets. Otherwise, the shell will not be "triggered" to perform the substitution.

Setting Shell Variables by Command Substitution

In the preceding discussion of variable substitution, you saw how to specifically place a value into a variable. In this section, we discuss how to place the result of command execution into a variable.

In Linux's `bash` shell, you can accomplish command substitution by using two types of syntaxes:

```
$ variablename=`command (-options) [arguments]`
```

and

```
$ variablename=$(command (-options) [arguments])
```

The first syntax example uses *back quotes* (*not* single quotes) to surround the command whose output you want to assign as the value of the variable. The back quote key (`) is located immediately to the left of the 1 key. Do not confuse the back quote with single quotation marks.

Back quotes are supported by the classic Bourne and C shells, so they are supported also by the `bash` and `tcsh` shells. Back quotes are supported by the Korn shell for backward compatibility.

The second type of syntax, `variablename=$(command . . . etc.)`, was originally specific to the Korn shell.

Example 11.5 Command Substitution: Variable Values Set to Command Results

Freston, while performing several of his routine administration functions, makes use of command substitution. In the following code, he will define a variable called `now` and for its value substitute the output of the `date` command:

```
$ date<Enter>
Mon Jul 16 16:14:34 CST 2001
```

He sets the variable to contain the results of the command:

```
$ now=`date`<Enter>
```

or

```
$ now=$(date)<Enter>
```

Now, he checks to see whether the variable works:

```
$ echo $now<Enter>
Mon Jul 16 16:14:34 CST 2001
```

Note that you can carry out the command only once. The result is then available for use until the variable is removed or reassigned.

Changing Variable Settings with the unset Command

At some point, you might no longer need or want some variables. To remove them, you use the `unset` command as follows:

```
$ unset variablename<Enter>
```

After using `unset`, it is a best practice to check that the variable is indeed gone by `echo`ing it (do not forget to add the dollar sign):

```
$ echo $variablename<Enter>
```

It will seem to you that the shell did not react to the command. But that lack of a response is a good thing. The shell is telling you that `variable-name` no longer has a value set for it.

NOTE **Similar to `echo $variablename`, you can also `unset variablename` on variables whose values were set through command substitution.**

Example 11.6 Changing Variable Settings with the unset Command

After creating the variable `abc` and assigning the value "day" to it, Juana finds that she does not use the variable as much as she thought she would and that the syntax is a little cumbersome those few times she uses it. So, she decides to remove `abc`. Then, she checks to see whether it has really been removed.

```
$ unset abc<Enter>
$ echo $abc<Enter>
```

The shell simply returns a command prompt to her. "Good," she thinks, "it's gone."

Customizing the User Environment

In this section, we describe the sources of most environment variables and mention the roles of several programs, initialization scripts, and other files that you use from the time you boot your system until you successfully log in. Then, we will look at how you can customize your environment depending on your job functions, maybe, or depending on your other needs or preferences.

Setting Default Shell Variables During Bootup and Login

We will focus on logging into a `bash` shell; the other shells use a slightly different but similar process. The compiling of environment variables from their respective files is shown in Figure 11.2. It is not meant to be totally comprehensive, but it is sufficient for this introductory-level discussion. As you can see from Figure 11.2, the values for some variables can be "updated" as additional files are consulted during the login process. From version to version, we have noticed that Linux developers have worked to eliminate this redundancy.

But, if you have logged in and you notice that the value of one or another variable is not what you expect, then it might be worth checking which files contain the variable and in what order the environment was configured. (Here, we use the broadest definition of *files* as collections of data or byte streams; some of these files are programs, scripts, and the like and are not strictly data files.)

Okay, assume that you are logging in at your terminal, which you have already booted. Already, during the course of booting, the `init` program (commonly regarded as the parent of all system processes because it is the first program that the kernel executes; it is even given the PID of 1) has already executed, or caused to be executed, several system initialization scripts. Those scripts—the primary one being */etc/rc.d/rc.sysinit*—define the system initialization environment and other parameters regarding security and other system features that reflect your or your company's policies. The */etc/profile* shell script (which contains system-wide environment variables

and commands) and its companion, the */etc/bashrc* shell script (which contains system-wide functions and aliases) can be changed only by the system administrator or root user. The `init` program has also invoked a `mingetty` program.

Normally, a `getty` is a program that is invoked by `init` to connect a user with the Linux/UNIX system. Initially, `getty` prints the contents of */etc/issue* (if it exists), then prints the login message field for the entry it is using. When you log in by supplying your username, the `getty` (in our case, `mingetty`, which is a scaled-down `getty` program) program invokes the `/bin/login` program, which carries out a number of functions and prompts you for a password. Also, while reading your login name, `mingetty` attempts to adapt the system to the speed and type of terminal being used. (If you type the `ps a` command at your terminal, you will see copies of */sbin/mingetty* waiting at your other virtual terminals).

If you provide a password that matches the now-encoded password on file for you, *login* authenticates you and then runs the `login` command with the */etc/passwd* profile. Now that you have been authenticated, your login shell—the one listed alongside your username in */etc/passwd*—is invoked.

Your *$HOME/.bash_profile* and *$HOME/.bashrc* shell scripts are used to initialize the `bash` shell and add more variables and values to your shell environment. The environment variables in those scripts override any of the same variables set in previous scripts (for example, `PATH` as it might have been defined by */etc/profile* or */etc/rc.d/rc.sysinit*). In addition, you can change or add to the *$HOME* script files to customize your environment.

The difference between *.bash_profile* and *.bashrc* is that *.bash_profile* is used when you are in your login shell and *.bashrc* is read when you invoke any other child process (such as a subshell) that is *not* your login shell. Both bash and tcsh distinguish between the login shell and other invocations of the shell. If *.bash_profile* is not present at login, *.profile* is used if it exists.

Now that you are authenticated and your shell is initialized, `login` displays the message of the day (*/etc/motd*), if there is one, checks for any e-mail messages awaiting you, and notifies you whether e-mail messages are available.

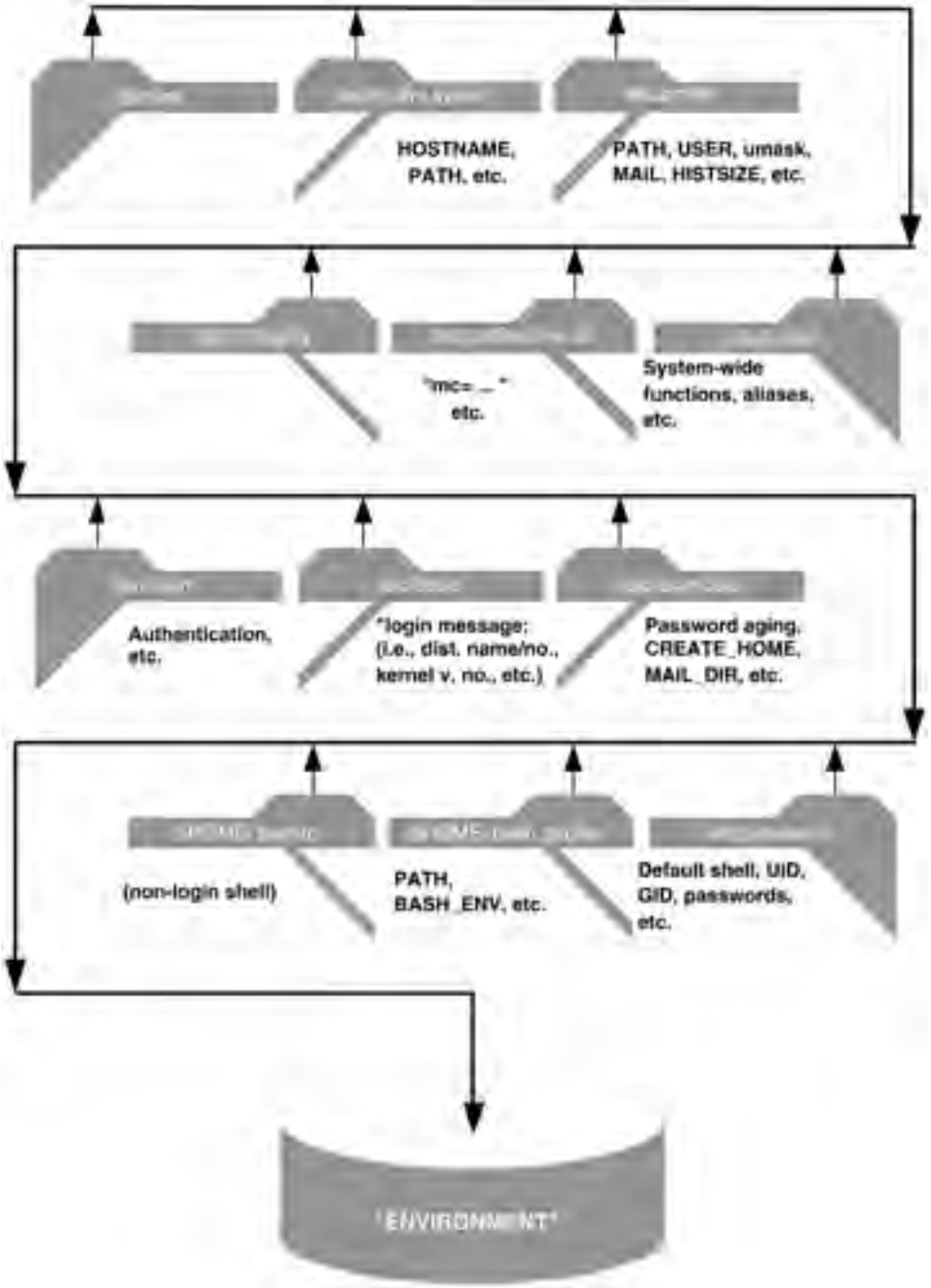


Figure 11.2 Building the shell environment during bootup and login.

Samples of Environment-Building Files

Let's look at some of the files illustrated in Figure 11.2 in some detail. Some are system-wide in scope; some pertain only to individual users.

A Sample /etc/profile File

The */etc/profile* file is the default initialization file for the `bash`, `ksh`, and `sh` shells. Because it contains the environment commands and variables that are invoked when every user logs into the system, this file is also called the *global profile*. Only the system administrator can change the file, but individual users can override the variables by modifying them in their own *$HOME/.bash_profile* files. The */etc/profile* file, however, is *not* intended to replace the individual user's *$HOME/.bash_profile* script file. All customizing should be performed in those individual scripts.

The */etc/profile* should be kept as small as possible because it is used by other scripts as well as by users. Example 11.7 shows the contents of a sample global profile file. An explanation of some of its variables is contained in Table 11.1, which follows it.

Example 11.7 Examining an etc/profile File

Freston is in charge of maintaining and altering, if necessary, to reflect new corporate policies, the system-wide environment. So occasionally, he has a look at the "global profile"; that is, at the */etc/profile* file.

```
# cat /etc/profile<Enter>
# /etc/profile
# System-wide environment and startup programs
# Functions and aliases go in /etc/bashrc

if ! echo $PATH | /bin/grep -q "/usr/X11R6/bin"; then
        PATH="$PATH:/usr/X11R6/bin"
fi

PS1="[\u@\h \W]\\$ "

ulimit -S -c 1000000 > /dev/null 2> &1
if [ `id -gn` = `id -un` -a `id -u` -gt 14 ]; then
umask 002
else
umask 022
fi
```

```
USER=`id -un`
LOGNAME=$USER
MAIL="/var/spool/mail/$USER"

HOSTNAME=`/bin/hostname`
HISTSIZE=1000

if [ -z "$INPUTRC" -a ! -f "$HOME/.inputrc" ]; then
     INPUTRC=/etc/inputrc
fi
export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE INPUTRC
for i in /etc/profile.d/*.sh ; do
if [ -x $i ]; then
. $i
fi
done
unset i
```

Selected variables from Example 11.7 are described in Table 11.1. The `PATH` variable lists the directories through which the shell will look for commands that have been issued to it. Note the format of `PATH` in Example 12.1, especially `$PATH`, which tells the shell to include all directories listed in previous initialization scripts. Notice also that the shell was told to add the */usr/X11R6/bin* directory to those directories.

The `PS1` variable, as discussed previously, sets the format of the primary shell prompt. In Example 11.7, the shell is told to put the username, the name of the user's computer, and the user's default home (file) directory in square brackets. The dollar sign (outside the brackets) is the prompt symbol.

With the `ulimit` command, the shell is provided with the upper limit on system resources that commands and files can use. Check the `ulimit man` pages or other sources for further information.

As discussed in Chapter 6, "Linux File Permissions," `umask` sets the user's default permissions as owner, group member, and member of "others" for user-created files and directories. In the example, the user ID is subjected to an if-then-else process to set the `umask` for a root user to `002` and for an ordinary user to `022`.

Next, the shell is told that the format for the user account name will be the user's name on the system, not the user's ID number as found in */etc/passwd*. In other words, any command or script that calls for the variable `USER` will be given that name in the format specified here.

The `LOGNAME` variable tells the shell that the user's login name will be the same as the user's name stipulated by the `USER` variable.

Table 11.1 Selected Variables in the Global Profile File

VARIABLE OR NUMBER	COMMAND NAME	DESCRIPTION
1	`PATH`	List of directories the shell will look through for commands
2	`PS1`	Primary system prompt
3	`ulimit`	Upper limit on system resources that a file or command can use
4	`umask 002;`	Permissions on new files or directories for the root user
	`umask 022`	Permissions on new files or directories for the ordinary user
5	`$USER`	Format of user's name as it is applied to commands and scripts
6	`LOGNAME`	Format of expected login name
7	`MAIL`	Directory where the user's mail messages are found
8	`HOSTNAME`	Name of the computer
9	`HISTSIZE`	Maximum number of previous commands displayed in response to the history command

The `MAIL` variable tells the shell where to look for mail messages for `USER`. The mail will be found in the directory whose name is the same as the `USER` within the */var/spool/mail* directory.

The `HOSTNAME` variable tells the shell where to find the name of the computer (in this case, */bin/hostname*).

The `HISTSIZE` variable sets the number of lines of command history (that is, the number of previous commands) that the shell will display when the user types `history`. This feature can be handy, as discussed previously, for retyping previous commands, doing sensitivity analyses around command arguments, or even troubleshooting. As seen in Example 11.7, the default is 1,000 commands!

A Sample /etc/bashrc File

The *etc/bashrc* file is another system initialization script invoked by `init`. It is the companion to */etc/profile*, but where */etc/profile* is intended to hold

environment variables and commands, */etc/bashrc* is intended to hold other system functions and aliases for commands or procedures. Remember that a *function* is a procedure that transforms a value or performs other actions and returns the results. Invoking the action or transformation is generally referred to as *calling* the function.

In Example 11.8, we see a reminder that environment information will be placed in */etc/profile*. We then ensure that the primary system prompt for users will be consistent no matter where the user is operating. Finally, an alias is defined (discussed later in this chapter), which enables any user to use the newly defined command when trying to determine the absolute path location of any command they name. For example, `which man<Enter>` returns `/usr/bin/man`, the location of the executable file that invokes `man` pages.

Example 11.8 System Initialization with /etc/bashrc

Along with checking the global profile, Freston also periodically checks the */etc/bashrc* file, with its system-wide functions and aliases:

```
# cat /etc/bashrc<Enter>
# /etc/bashrc

# System wide functions and aliases

# Environment stuff goes in /etc/profile

# are we an interactive shell?
if [ "$PS1" ]; then
     if [ -x /usr/bin/tput ]; then
            if "x`tput kbs`" != "x" ]; then #We can't do this with
"dumb" terminal
                    stty erase `tput kbs`
            elif [ -x /usr/bin/wc ]; then
                    if [ "`tput kbs/wc -c `" -gt 0 ]; then #We can't do
this with "dumb" terminal
                    fi
            fi
fi
     case $TERM in
            xterm*)
                    PROMPT_COMMAND='echo -ne
"\033]0;${USER]@${HOSTNAME}: ${PWD}\007"`
                    ;;
            *)
                    ;;
```

```
    esac
    [ "$PS1" = "\\s_\\v\\$ " ] && PS1="[\u@\h \W]\\$ "

    if [ "x$SHLVL" != "x1" ]; then #We're not a login shell
            for i in /etc/profile.d/*.sh; do
                    if [ -x $i ]; then
                            . $i
                    fi
            done
    fi
fi
```

The file still functions, and there have been no complaints or requests, so Freston leaves it alone for now.

A Sample $HOME/.bash_profile File

In Example 11.9, we see the variables set for an individual user in his or her personal *.bash_profile* script file. Every time the user (in this case, Don Quixote) logs in, this script is read and its variables and commands are adopted or executed.

Example 11.9 Setting Variables for an Individual User

Curious about why his `PATH` has his home directory in it and Sancho's does not, Don Quixote calls again on Freston. Freston tells him to check his hidden *.bash_profile* file. "It's probably specified there," Freston says. So the Don goes to his home directory and `cat`'s his *.bash_profile*:

```
$ pwd<Enter>
/home/quixoted
$ cat .bash_profile<Enter>
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
. ~/.bashrc
fi

# User-specific environment and startup programs

PATH=$PATH:$HOME/bin
BASH_ENV=$HOME/.bashrc

export BASH_ENV PATH
unset USERNAME
```

The Don sees how the first conditional if-then-else decision structure calls for the shell to check whether a hidden script file called *.bashrc* (containing aliases and functions specific to this user only and not to all users, such as those in */etc/bashrc*) exists in this directory, and if so, to execute it. Note that the *.bashrc* file, which contains user-specific functions and aliases, could be called anything, but the convention is to name it similarly to the */etc/bashrc* file, which has global functions and aliases.

Next, he sees how his user's `PATH` statement is modified. The existing `PATH` directories are adopted, and the */bin* directory in the user's home directory is added to them (please note that this scenario is a bash shell situation; Sancho's environment, being tcsh, does not include this addition of the home directory to the `PATH`). If you want to ensure that the current directory is included in the `PATH` variable, verify that the variable contains two or more adjacent colons or a colon followed by a period.

Don Quixote now has an idea why his `PATH` and Sancho's are not identical during normal operations. Freston tells him, later, that the `PATH`s are identical when Sancho works in the `bash` shell.

Meanwhile, the `BASH_ENV` variable causes the *.bashrc* script in the user's home directory to run every time another new `bash` shell is invoked. This script is not exported because the settings, aliases, and other functions in *.bashrc* might not be variables and thus are not subject to export.

Note that the `BASH_ENV` and `PATH` variables are exported so that these variables and their respective values will be adopted by child processes.

Finally, the variable `USERNAME` (which has fallen out of favor with recent Linux versions), if it has survived this far, is removed. It used to cause confusion with users and administrators alike.

Just as */etc/profile* has a companion in */etc/bashrc* for system functions and aliases, *$HOME/.bash_profile* has, by convention, its companion *$HOME/.bashrc* for user-defined functions and aliases.

A Sample $HOME/.bashrc File

Example 11.3 contains the `BASH_ENV=$HOME/.bashrc` variable, which causes *$HOME/.bashrc*, with its extra specifics, to be executed every time the users starts a new `bash` shell. The *$HOME/.bash_profile* file is executed only once when the user logs in.

In Example 11.10, we see that aliases and functions specific to the user are recommended, and then the global aliases and functions are searched for and invoked by the if-then-else decision string. Otherwise, those functions, aliases, and so on would not be invoked.

Example 11.10 Script for User-Defined Functions and Aliases

Here, we eavesdrop on the Don when he checks his *.bashrc* file:

```
$ pwd<Enter>
/home/quixoted
$ cat .bashrc
# .bashrc
# User specific aliases and functions
# Source global definitions
if [ -f /etc/bashrc ]; then
. /etc/bashrc
fi
# Begin aliases written by Quixote de La Mancha
# Number 1 - Alias similar to old DOS command; written on July 23
alias dir-p="ls -l | less"
# End of aliases written by Quixote de La Mancha
```

In Example 11.10, we see how Don Quixote has written his own alias. It appears that the Don is a DOS veteran, preferring the `dir` style of commands to the UNIX `ls` style. Note, however, that he cannot use `dir /p`, which is the proper DOS syntax. He has to modify the command. Why? The shell would not interpret the forward slash in the same way as DOS and so would return an error message.

Exercises

1. Ensure that you are in your home directory and then display the shell variables.

```
$ cd<Enter>
$ pwd<Enter>
$ set<Enter>
```

2. Create a variable named `lunch` and set its value to `pizza`. Create another variable named `dinner` and set its value to `ham`. Display the value of the variables by using `echo`.

   Locate them in the list of variables.

```
$ lunch=pizza<Enter>
$ dinner=ham<Enter>
```

```
$ echo $lunch ; echo $dinner<Enter>
$ set<Enter>
```

3. Using the variables you just defined, display the message `Lunch today is pizza and dinner is ham`.

```
$ echo Lunch today is $lunch and dinner is $dinner<Enter>
```

4. Using the variables you just defined, display the message `Lunch today is hamburgers`.

```
$ echo Lunch today is ${dinner}burgers<Enter>
```

5. Remove the value of both variables. Check to make sure they are no longer included in your list of variables.

```
$ unset lunch<Enter>
$ unset dinner<Enter>
$ set<Enter>
```

6. Display the value of your primary and secondary prompt strings.

```
$ echo $PS1<Enter>
$ echo $PS2<Enter>
```

7. Change the primary prompt string to `You Rang?`

```
$ PS1="You Rang?"<Enter>
```

Why is it necessary to use quotes with `You Rang?`

NOTE **Single quotation marks will work, too.**

8. Change your secondary prompt string to `What Else?` Test it with the `ls` command by using line continuation. End the command. Reset both prompt strings back to their original values.

```
You Rang? PS2="What Else?"<Enter>
You Rang? ls -l\<Enter>
What Else? <Enter>
What Else? <Ctrl>-c
You Rang? PS1= "$"<Enter>
$ PS2=">"<Enter>
```

Why are quotes needed around the left angle bracket (>) when resetting the PS2 variable?

9. Check the value of the variable related to your home directory. Reset that variable to change your home directory to */bin*. Use the `cd` and `pwd` commands to test the effects of this change.

```
$ echo $HOME<Enter>
$ HOME=/bin<Enter>
$ cd ; pwd<Enter>
```

NOTE **You can check the value also by using `set`.**

10. Log out and then log back in.

```
$ exit<Enter>
hostname login: username<Enter>
Password:
```

What is your home directory?

```
$ cd ; pwd<Enter>
$ echo $HOME<Enter>
```

Why is your home directory what it is?

11. Display your list of variables. Reissue the command but send the output to the `wc` command to get the number of variables that are currently set.

```
$ set<Enter>
$ set | wc -l<Enter>
```

12. Using command substitution, echo the following:

```
# variables are currently set
```

where # is the number of variables.

```
$ echo `set | wc -l` variables are currently set<Enter>
```

13. Each user ID configured on the system is represented by one line in the */etc/passwd* file. Applying your knowledge of command substitution, `echo` a message that displays

```
There are # users created on the system
```

where # is the number of line entries in */etc/passwd*.

```
$ echo There are `cat /etc/passwd | wc -l` users created on the
system<Enter>
```

14. To customize your environment and have that customized environment take effect every time you log in, you must incorporate your modifications in a file that is read at login. First, log into the system and ensure that you are in your home directory.

```
$ cd<Enter>
$ pwd<Enter>
```

15. Now, edit your *.bash_profile* file as follows: Change your primary system prompt string to reflect your current directory; display a message at every log in that contains your login name and the time of your login; and define an alias named `dir` that invokes the `ls -l` command.

```
$ vi .bash_profile<Enter>
i (to enter Insert mode)
PS1='$ PWD =>'<Enter>
echo User $LOGNAME logged in at `date` <Enter>
alias dir='ls -l'<Enter>
set -o vi<Enter>
<Esc> (to enter Command mode)
:wq<Enter> (to save .bash_profile and leave vi)
```

16. Test your customization by re-executing your *.bash_profile*. You can do that by logging out and then logging in or by simply rerunning the file using dot notation.

```
$ logout
Login: username
Password:
```

or

```
$ . .bash_profile<Enter>
```

Execute and answer the following:

- Did your message appear?
- Is your prompt identical to the name of your home directory?
- Change to the */etc* directory. Did your prompt change?
- Use `dir`. Did you get a long listing of your current directory?

If you answered no to any of these questions, edit your *.bash_profile* and correct it.

17. After your customized *.bash_profile* is set up and functioning properly, open a subshell.

```
/home/teamxx => bash<Enter>
```

Is your prompt identical to the name of your current directory?

Does the `dir` alias still work the way you set it?

18. Exit from the subshell and return to your home directory.

```
$ <Ctrl>-d
/home/teamxx => cd<Enter>
```

19. Most settings, with the exception of system variables, apply only to the current environment they are set in and are not passed to subshells (which are child processes). To pass customized settings down to subshells, you must set the `BASH_ENV` variable appropriately in your *.bash_profile*, and you must have a properly customized *.bashrc* file.

    Revise your *.bash_profile* and create the appropriate *.bashrc* file to support the customization you implemented in Exercise 15. Remove *only* what you previously added to the *.bash_profile* in Exercise 15 except for the `echo` statement and the `PS1` variable assignment.

    Add the `BASH_ENV` variable assignment. Export both the `PS1` and `BASH_ENV` variables and their values.

```
/home/teamxx => vi .bash_profile<Enter>
i (to enter Insert mode)
BASH_ENV=/home/teamxx/.bashrc<Enter>
export PATH PS1 BASH_ENV<Enter>
<Esc> (to enter Command mode)
:wq<Enter> (to save .bash_profile and leave vi)
/home/teamxx => vi .bashrc<Enter>
i (to enter Insert mode)
alias dir='ls -l'<Enter>
<Esc> (to enter Command mode)
:wq<Enter> (to save .bashrc and leave vi)
```

20. Test your customization by re-executing your *.bash_profile* file. Open a subshell.

```
/home/teamxx => . .bash_profile<Enter>
/home/teamxx => bash<Enter>
```

    Is your prompt identical to the name of your current directory? Is the value of the `dir` alias still working?

21. Exit the subshell and return to your login shell. Display a listing of all currently set alias names and locate the `dir` alias.

```
/home/teamxx => <Ctrl>-d
/home/teamxx => cd<Enter>
/home/teamxx => alias<Enter>
```

22. Temporarily `unalias dir` without editing the *.bashrc* file. Then, display the list of alias settings again and ensure that `dir` is no longer defined. Try executing `dir`.

```
/home/teamxx => unalias dir<Enter>
/home/teamxx => alias<Enter>
/home/teamxx => dir<Enter>
```

23. The `dir` alias is still in your *.bashrc* file, but it is not set. The `unalias` command removed it from the list of current alias names. Invoke *.bashrc* to automatically add `dir` back to the alias list.

```
$ logout
Login: teamxx
Password:
```

or

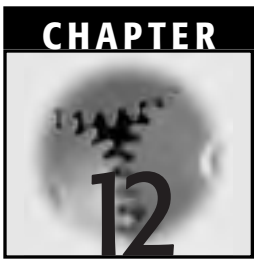
```
$ . .bashrc<Enter>
Execute dir.
/home/teamxx => dir<Enter>
```

See Appendix B for answers.

Quiz

1. True or false: The listing of terminal environment variables contains all the variables found in your current shell.
2. True or false: When creating built-in variables, you have to ensure that their names are in uppercase. Otherwise, the operating system does not process them properly.
3. What methods can you use to set variable values?
4. What are the differences between the *.bash_profile* and *.bashrc* files?
5. True or false: You have to have a *.bashrc* file when you are using the `bash` shell.
6. True or false: The `init` process is considered the parent of all processes.
7. Which file is called the global profile? Why?
8. Define the following:
   - HISTSIZE
   - HISTFILE
9. True or false: After an `alias` is defined in your *.bashrc* file, you cannot undo its functionality with `unalias`.
10. Fill in the blanks. In the `bash` shell, _____ is to */etc/profile* as _____ is to *.bash_profile*.

See Appendix C for answers.



Linux Processes and Process Control

Processes are the essence of computing. Each command during execution represents at least one process. Some processes run in the foreground, where you can monitor and feed them, while some run in the background. Some, like daemons, are initiated at boot time to run all the time or according to an on-demand regime. Batch processes are generally scheduled for execution at a specified time, such as only once, periodically, or somewhere in between.

This chapter describes processes and their environments. Topics also include relationships between processes (for example, how their ID numbers indicate who is who) and the exporting of variables between parent and child processes. Then, we discuss return codes, which are used to indicate whether commands have completed successfully. Later in the chapter, we discuss monitoring and controlling processes as well as foreground and background processes. We then turn to elegant and brutal process termination.

Process Environments

A *process* is a single program or task running in its own virtual address space. A command or a job can consist of many processes, but (contrary to some opinions) a process is not necessarily the same as a command or a job. Very simple commands, however, can be considered single processes.

From a strict technical interpretation, a program is just a set of instructions, but a process is a dynamic operation that uses the running system's resources. As we discussed before, when you log in, Linux places you in your home (file) directory and initiates a program called a shell. The shell itself is a process. After that, whenever you invoke commands or applications, you are initiating them from within the shell (that is, from within the shell process).

In Linux, you can run more than one process at a time, and you can run several copies of the same program simultaneously. You can also track the environments of all the processes while they run. Table 12.1 summarizes the environment, which consists of variables and parameters that a typical process might require to run. Some processes require fewer variables and parameters; others require more.

We have already mentioned that the shell is a process. Example 12.1 shows you how to check the `process ID,` or `PID,` of the shell process.

Table 12.1 Process Environment

FIELD	EXPLANATION
Terminal (TTY)	Terminal ID from which the process was launched
Open files	Files the process is using
Current directory	The directory from which the process was invoked
User ID (UID)	ID of the user who invoked the process
Group ID (GID)	ID of the group to which the user belongs
Process ID (PID)	A unique number that the kernel randomly assigned to the process
Parent process ID (PPID)	A unique number randomly assigned by the process that launched the subject process, if applicable
Flags	Any options and arguments appended to the command that launched the process

The built-in variable $ (which is, by no coincidence, also the default user prompt) is the symbol for the current shell.

To view the PID of the current shell, use the following syntax:

```
$ echo $$<Enter>
```

Example 12.1 Checking the Shell's PID

To view the PID of the current shell, type

```
$ echo $$<Enter>
340
```

The Login Process

The user login process creates a Linux session that continues until the user logs out. The process environment contains all the information necessary to enable the process to run. We have seen before how Linux creates the shell environment and initiates the shell process. Generally, by default, users are placed in their own home directories and the bash shell is initiated.

Example 12.2 Variables and Parameters for Logging In

Example 12.2 shows the typical variables and parameters required for the user login process:

Command: `-bash`

UID: `quixoted`

GID: `knights1`

TTY: `tty1`

PID: `340`

All of the information in Example 12.2 is unique to the process environment as long as Don Quixote remains logged in. After he logs out, the information is forgotten. When he logs in again, another new process environment is created for him.

Parent-Child Relationships

When you execute a command, it is important to remember that one or more processes are already running. Thus, every process running in Linux is invoked or managed by another process. All processes exist in this parent-child relationship (or parent-child hierarchy, if you prefer). The process started by a program or command is known as the *parent* process. The parent process can invoke other processes; those processes are called *child* processes. A parent process might have several child processes, but any child process can have only one parent process.

The child process environment is a local one that it has inherited from the parent. That environment tells the child what invoked the child process, how the child process should handle output, and so on. The child can modify its inherited environment and pass the modified environment down to its own children (if applicable), but it cannot pass the modified environment back to its parent. The exception is the child process that is invoked through the use of a dot (.), which we will discuss later.

Invoking Shells

A common parent-child propagation is the invocation of one shell from another—usually the invocation of a non-login shell from a user's login shell. There are two basic ways to invoke new shells:

- Invoke a new shell. The original shell becomes a "sleeping shell."
- Invoke a new shell. The original shell is stopped.

Almost all of our examples in this chapter will involve the first method: invoking a new shell while the original shell goes to sleep. In the meantime, let's look at the two methods.

Invoking a New Shell and Interrupting the Original Shell

To invoke a new shell while only interrupting the original shell, use the following syntax:

```
$ newshellname (-options) [arguments]<Enter>
```

When a new shell is invoked in this manner, the previous shell (usually, but not always, the login shell) will "sleep" until you go back to it by entering the `exit` command.

You can also invoke a new shell by entering the following:

```
$ newshellname (-options) commandname (-options) [arguments]<Enter>
```

or

```
$ newshellname (-options) scriptname<Enter>
```

This action spawns a new shell, which in turn will execute either a new command/program or a new script. Once the command/program or script has executed, then control will pass from the new shell back to the original shell. Until then, the original shell will "sleep."

Table 12.2 lists a few of the several options that can be invoked along with either the `sh` or `bash` shell commands.

You will notice that when a new shell is invoked that is the same as the login shell, the user prompt will not change (have a look at Figure 12.1). As it is invoked, the new shell will read the same environment files as the previous shells. You can check that a new shell has been invoked by entering the `ps` command with no options or arguments. The new shell will have the same name as the previous shell but will have a different process ID number (`PID`).

Table 12.2 Options Available for Use with bash or sh Shells

OPTION	EXPLANATION
`-c`	The additional commands following the `-c` can be run with the shell command.
`-r`	Start a restricted shell; that is, a shell that has approximately a dozen restrictions. Examples: you cannot `cd` between directories; you cannot set or unset the SHELL, PATH, ENV, or BASH_ENV variables (and so on).
`-i`	The shell will be run interactively; you can test this capability with startup files or scripts.
`--`	(no spaces between them) indicates "that's the end of the options"; any arguments after the `- -` will be treated like filenames or other arguments (not as options). You can use `-` as well.

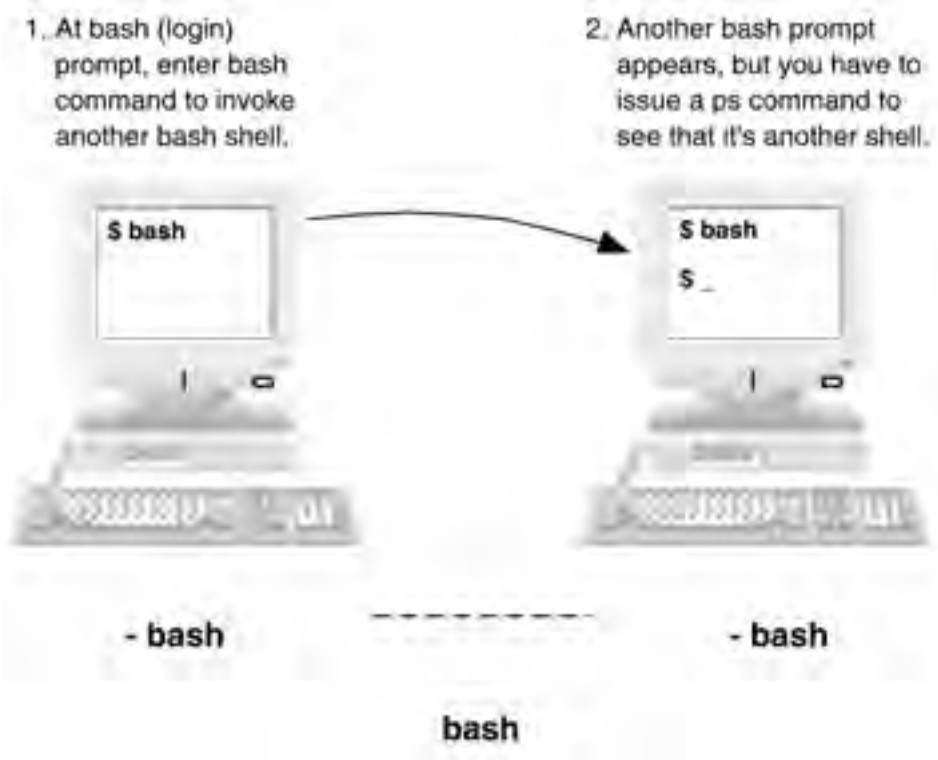


Figure 12.1 Invoking a shell while interrupting the original shell.

Meanwhile, if you invoke a new shell that is different from the existing shell, the prompt will change. The shell, when invoked, will have read its own specific environment files—and chances are that a different `PS1` style was specified in them.

Invoking a New Shell and Stopping the Original Shell

To invoke a new shell while stopping the original shell, use the following syntax:

```
$ exec newshellname (-options) [arguments]<Enter>
```

When a new shell is invoked in this manner, the previous shell (usually, but not always, the login shell) will stop completely and the new shell will actually take the previous shell's process ID number. The only way to return to the previous shell would be to invoke a new one. And, even then,

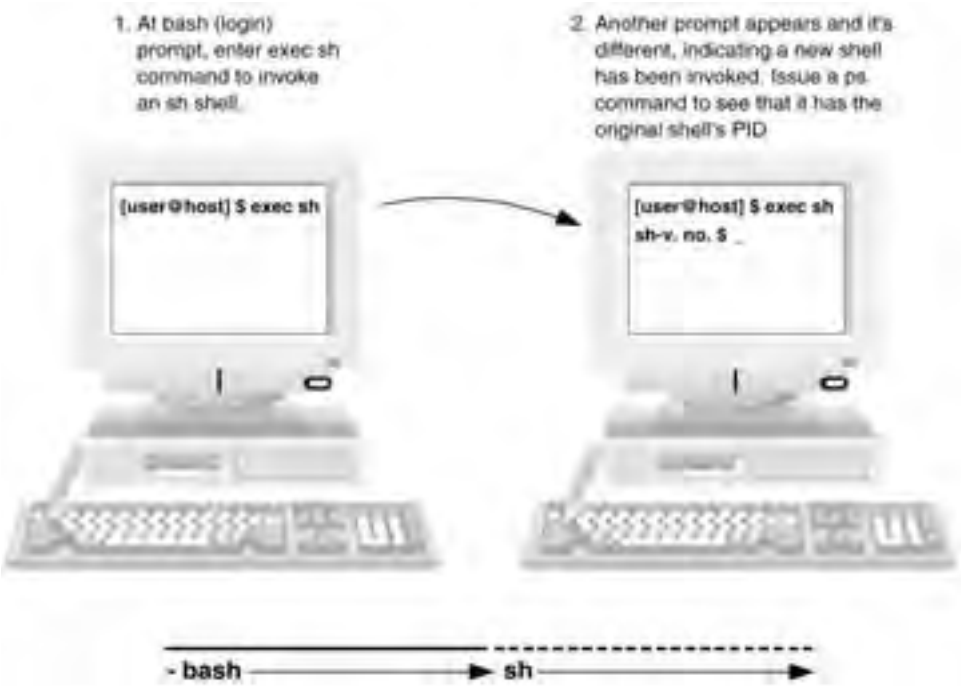


Figure 12.2 Invoking a new shell and stopping the original shell.

that shell would have a new `PID`. Here, you will notice that when a new shell is invoked that is different from the login shell, the user prompt has changed (see Figure 12.2).

As it is invoked, the new shell will read its own environment files, which are usually different from the previous shell's files. You can tell that a new shell has been invoked just by looking at the screen. This time, though, when you enter the `ps` command, the new shell will naturally have a different name from the previous shell but will have inherited the previous shell's `PID`.

Example 12.3 The ps -auxf Command to Examine Parent-Child Processes

To illustrate some parent-child process relationships, suppose that Lady Toloso logs into her terminal and types the following command and options at the command line:

```
$ ps -auxf | less<Enter>
```

We will discuss the command in more detail later. But for now, the system normally responds with a list of processes that are currently running. The output will probably come to more than one screenful of output. We have piped the output of the `ps` command and options to the `less` command, however, so the system will stop and wait for user input after each screenful of output is displayed. Let's look at some of the output; see Table 12.3.

We included the `init` process in the listing here to illustrate that it is indeed the parent of all processes. Now, let's look at `PID 637`, which shows us when the root user logged in. The system provided the capability to log in at 09:32, but the root user actually logged in at 12:57. We also see that root's login process automatically spawned the child process `PID 694`, a (default) bash shell, at 12:57. We know that the bash shell is the child of the login because of the slash mark `( \ )` that connects the `bash` command to `login - - root`.

Moving along, we see that our user Lady Toloso logged in at 14:22, which caused the spawning of a child process bash shell, as well. Notice the same slash connection between that bash shell and Lady Toloso's login process. Later, at 14:37, we see that Lady Toloso entered the `ps -auxf | less` command. Two child processes were spawned as a result of Lady Toloso entering that command: the `ps -auxf` process itself and the (piped) `less` process. Those two processes are children of the bash shell because each has a slash connection to it.

Example 12.4 Examining PIDs and PPIDs

Have a look now at Figure 12.3. This figure shows an illustration of Don Quixote's login session, which we briefly discussed earlier in this chapter. Remember that the `init` process—the first process to start in a system—has a process ID number of `1`. The Don logged into the system and the kernel assigned his login shell a `PID` of `340`. To check the `PID` number, he could have used the `echo $$` command. Incidentally, the `echo` command is one of several built-in commands; thus, when it is invoked, there is no need to create a subshell to run it.

Although Don Quixote used the special `$$` variable as an argument to `echo`, `$$` is mostly used within shell scripts to distinguish between multiple instances of the same shell script. The value of that variable changes as you move from one (current) process environment to another.

At any rate, there he is in the login shell. The `PID` is `340` and the parent process ID (`PPID`) is `1` (`init` process). To further illustrate parent-child

Table 12.3 Results of the ps -auxf Command

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.7	1120	472	?	S	09:31	0:04	init
.										
.										
root	637	0.0	1.6	2224	1036	tty1	S	09:32	0:00	login - - root
root	694	0.0	1.4	1702	944	tty1	S	12:57	0:00	_ -bash
root	638	0.0	1.7	2232	1128	tty2	S	09:32	0:00	login - - Toloso
Toloso	749	0.0	1.4	1700	940	tty2	S	14:22	0:00	_ -bash
Toloso	816	0.0	1.2	2488	788	tty2	R	14:37	0:00	_ ps -auxf
Toloso	817	0.0	1.0	1584	676	tty2	S	14:37	0:00	_ less
.										
.										

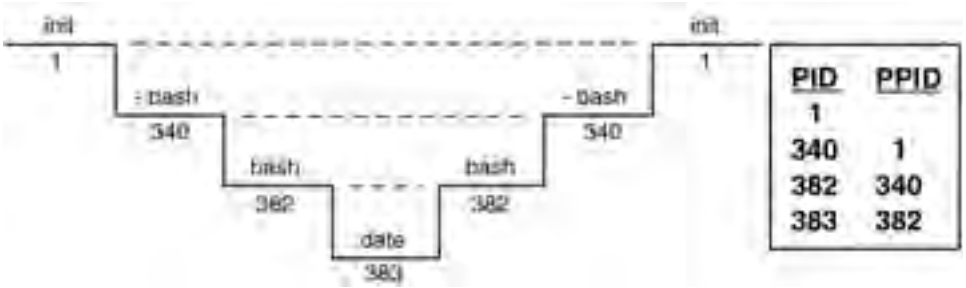


Figure 12.3 Parent-child processes.

relationships, he starts another bash shell. He could have started the tcsh or any other shell depending on whether he needed a specific shell to run a certain application or shell script, but here he was content to just open another bash session. The `PID` of the new shell is `382` and the `PPID` is `340`, indicating that the parent process was the previous bash shell.

While in the second bash shell (that is, the subshell), the Don invoked the `date` command, which is yet another process. That `date` process is `PID 383`, and its parent's `PPID` is `382` (the process ID number of the bash subshell).

Next, Don Quixote exited from the bash subshell by pressing Ctrl-D. He moved to the original bash login shell, which he verified by executing `echo $$` one last time. Note that had he invoked more complicated commands or maybe a shell script or two, their processes might have spawned additional child processes. Shell scripts often launch additional shells wherein, in turn, more shell commands are executed.

WARNING **Although it executes very quickly, you must have noticed that while the `date` command executes, you cannot interact with the shell. You have to wait until the command is finished before running other processes. Thus, the parent (the bash subshell) "sleeps" while the child (`date`, in this case) executes. Also, when the child process has terminated, the parent (in other words, the shell) "reawakens."**

Processes and Variables

When a user defines his or her own variables, those user-defined variables are local to the shell or process in which they are set. Child processes do not automatically inherit user-defined variables and their values. The two basic principles involved here are as follows:

- Variables form part of the environment of a process.
- Processes cannot access or change variables in the environment of another process.

Example 12.5 illustrates these points.

Example 12.5 Setting User-Defined Variables

Set a value of `x` in the bash shell:

```
$ x=4<Enter>
```

Now, create another bash subshell:

```
$ bash<Enter>
```

Search for the value of `x` in the subshell.

```
$ echo $x<Enter>
$
```

No record of `x` is found. So far, it looks like neither the variable nor its value was inherited. So, now, set the value of `x` in this subshell to `1`:

```
$ x=1<Enter>
```

Exit from the bash subshell to the original bash shell and check the value of `x`.

```
$ <Ctrl>-d
$ echo $x<Enter>
4
```

The original value of 4 is returned. As expected, the subshell's variable `x` and its respective value `1` were not inherited backwards. They also had no effect on the original variable `x` and its value, `4`.

Exporting Shell Variables

In Chapter 11, "Shell Variables and the User Environment," we encountered the `export` command when we discussed how to ensure that variables are inherited from the terminal environment to the shell environment.

In this section, we discuss the inheriting of variables and their values from parent processes to their respective child processes—the same principle, only extrapolated.

As we stated previously, every process has an operating environment inherited from its parent process. User-defined variables, however, are valid only in their respective processes; they are not by default inherited from parent to child, nor are they passed back from child to parent. But sometimes, you are in a shell and you need to set variables so that they can be used by other programs or shell scripts (that is, by other processes). To make one or more variables and their respective values available to any child processes, you export them. But before you export, you have to set the value of the variable, as we will show in Example 12.6. The syntax for the `export` command is

```
$ variablename=value<Enter>
$ export variablename<Enter>
```

If you do not specify options after the `export` command, the screen displays the variables that are already exported. This feature can be handy sometimes. Also handy are the `set` and `env` commands for displaying the shell environment and the inherited environment (referred to as the terminal environment in Chapter 11), respectively.

Example 12.6 -export

Once again, the `echo` command lets us check the `PID` of the process that is currently running:

```
$ echo $$<Enter>
340
```

The system responds and tells us that the PID is 340. We now set the value of the variable `x` to 4:

```
$ x=4<Enter>
```

We now tell the system that any user-defined variables or values, such as `x = 4`, will be inherited by child processes.

```
$ export x<Enter>
```

Now, we start a bash subshell, which is a child process of the original bash shell.

```
$ bash<Enter>
```

Now, we check the PID of the new bash subshell:

```
$ echo $$<Enter>
395
```

The system tells us that the subshell has a PID of 395.

And now, the moment of truth: We ask what the value of the user-defined variable x is.

```
$ echo $x<Enter>
4
```

The value of x has indeed been inherited by this child process. Now, at this subshell process level, we change the value of x to 400.

```
$ x=400<Enter>
```

As a check, we request the value of x.

```
$ echo $x<Enter>
400
```

As expected, the value of x in this subshell is now 400. We now leave the bash subshell and return to the parent bash shell.

```
$ <Ctrl>-d
```

We check to see whether we have indeed left the subshell by requesting the PID of this shell.

```
$ echo $$<Enter>
340
```

This situation is no surprise. We are back in the parent bash shell. Now, we request the value of x to see whether the 400 has been inherited upward to the parent process.

```
$ echo $x<Enter>
4
```

No, the value of x in the parent process is still 4. Thus, variable values are inherited from parent to child but not vice versa.

NOTE **The `export` command (alone) and the `env` command show all inherited variables. Use `set` to show all shell variables.**

To paraphrase Example 12.6, the value of x was set to 4 in the login shell and then exported to the terminal environment. Another bash subshell was then created. The inherited value of x was, as predicted, 4 here, too. Then, we changed the value of x to 400 in the subshell to see whether it would affect the value of x in the parent login shell. Then, we exited to the login shell and checked the value of x. We discovered that despite the fact that we changed x from 4 to 400 in the bash subshell, the value of x remained unchanged in the login shell (it was still 4). In other words, then, changing the value of a variable in a subshell does not affect the parent process.

Assume that you are still in the bash subshell and have just changed the value of x from 4 to 400. If you open another subshell within this subshell, will the value of x be 4 or 400? The answer is 400. The child process inherits the parent process value in this case. After a variable has been exported, it does not have to be exported again. If you continued to open subshells within these subshells, the value of x would remain 400 until you saw fit to change it.

If you exit from the string of subshells back to the login shell and then open subshells below the login shell, the value of x would return to and remain at 4. Changes made by a child process to an exported variable are available only to subsequent child processes. The variable value is not inherited back up to the login shell or down any other process.

Return Codes from Commands

After a command has attempted or completed execution, it sends a *return code* (also known as an *exit code*) to its parent process. Note that a return code is a return message from a command, *not* from each individual process within the command. Also, please note that piped commands (discussed in Chapter 7, "Shell Basics") send only one return code to their parent.

The return code is stored in the parent process environment as a value for the built-in question mark (?) variable, as shown in Example 12.7. Suc-

cessful completion of a command returns a value of 0; unsuccessful completion returns a value ranging from 1 to 255.

How do you find the return code to a command? After sending the command for execution, type

```
$ echo $?<Enter>
```

Example 12.7 Using $? to Find a Command's Return Code

```
$ date<Enter>
Fri Mar 19 12:05:39 CST 1999
$ echo $?<Enter>
0
```

NOTE **If you are contemplating using this method by default, be aware that after settings are made in the current shell (you are usually in the login shell), the only way of returning to the original default settings is to log out and log back in again.**

Process Monitoring: The ps Command

Users (occasionally) and system administrators (more frequently) have to check how their Linux processes are running. The ps (process status) command is commonly used to see how many processes are running, whether a specific process is running or has gotten hung up, and how many resources the various processes are using. In addition, the command checks priorities, who is doing what, and so on. A frequent use of the ps command is to monitor background jobs and other processes that do not regularly communicate with your terminal. The syntax is as follows:

```
$ ps (-options)<Enter>
```

Earlier in this chapter, we briefly introduced you to the ps command when we illustrated parent-child process relationships. At that time, the ps command, with its a, u, x, and f options, showed us which processes were running, who they belonged to, and which processes were children of other processes.

The default ps listing, illustrated in Example 12.8, displays only minimum information about the processes started from your terminal. These

Table 12.4 Response to the ps Command

FIELD	DESCRIPTION
PID	Process identification number assigned by the kernel
TTY	Terminal where the process originated
STAT	Current status of the process (for example, S = Asleep, R = Runnable)
TIME	Cumulative execution time for the process (min:sec)
COMMAND	Name of the process being executed

processes are described in Table 12.4. On a system that has more than one terminal, the TTY field indicates which terminal initiated the process.

Example 12.8 Monitoring Processes with ps

```
$ ps<Enter>
PID TTY STAT TIME COMMAND
340 1 s 0:00 -bash
356 1 S 0:00 bash
363 1 S 0:00 ls -R /
364 1 S 0:00 ps
```

The ps command options are fully detailed in the man pages and in other information sources. A few options are explained in Table 12.5.

Table 12.5 Some ps Command Options

OPTIONS	DESCRIPTION
e	Environment of the process
f	Processes and subprocesses spawned from the parent processes
l	Long listing (along with basic ps flags, including FLAGS, UID, PPID, PRI, NI, SIZE, RSS, WCHAN)
j	Jobs format (along with basic ps flags, including UID, PPID, PGID, SID, TPGID)
a	Other users' processes, too
u	User's name and start time
x	Processes without an associated terminal

Invoking Foreground and Background Processes

Typically—and so far in this book—you invoke processes interactively from the command line. You can also invoke processes from the foreground and background states, however.

Processes that are started and require interaction from the terminal are called *foreground processes.* They are invoked in the foreground when you are fairly sure that they will finish in a short time or when you have to interact with them before and during execution. The processes themselves take over the terminal while they run. As we have seen before, the syntax for a foreground process is

```
$ command [-] options arguments<Enter>
```

Processes that run independently from the initiating terminal are called *background processes.* You run processes in the background when, for example, you want to use your terminal for other tasks. But they are most useful for executing commands (or scripts or batch files containing numerous commands) that will take a long time to run, regardless of whether or not you are using your terminal. The syntax for a background process, then, is

```
$ command [-] options arguments / > backfile &<Enter>
```

NOTE **A process can be run in the background only if it does not require keyboard input and only if you observe the invocation syntax—the ampersand** `(&)` **at the end of the command line.**

Terminating Processes

You might want to terminate a foreground or background process for several reasons:

- You no longer need the program or process, and there is no other way to stop it.
- You are not getting the results you expected.
- You are not getting any results and your system seems "frozen."

- Conversely, your output facilities (screen or printer) are overwhelmed by program or process output.
- The process or program is using too many system or network resources (such as memory, CPU usage, or bandwidth usage).
- The process or program is not behaving properly or predictably.

The first reason likely affects an advanced user or system administrator who might have altered the configuration of one or more systems and now finds that daemons or other background processes are no longer relevant to the system. The other reasons are pretty much self-explanatory.

Termination Methods

Foreground processes run on the user's terminal and can usually be terminated by some type of quit signal. The three most common methods are as follows:

- The Ctrl-C key sequence
- The `kill` command
- Powering the system down

The Ctrl-C key sequence is the most common method. It stops a foreground process and returns you to a shell prompt, generally a dollar sign (`$`) for users or a pound sign (`#`) for the administrator, root, and superuser. But remember that Ctrl-C does not work with `man`; use `q` or Ctrl-Z instead. Also, some shell scripts or other programs might ignore the quit signal.

In some cases where the processes ignore the common quit signals, you might have to enter the `kill` command, which we discuss in detail in the next section. You always have to use `kill` to terminate background processes.

Powering down a local system is a reasonably sure way of terminating foreground or background processes running on the system. You should always be concerned with the effects that suddenly powering down might have on individual files or programs, file systems, other system features or processes, and even other users, however. So powering down should be a last resort alternative.

The kill Command

An ordinary user has the ability to kill any process that he or she has initiated. By comparison, a root user can kill *any* process. The syntax is

```
$ kill PIDs<Enter>
```

or

```
$ kill signal PIDs<Enter>
```

If a process is successfully terminated, you get no message from the shell except a prompt. If you try to kill a process that you have no right to terminate, or if you try to kill a process and the process you specify does not exist, you will see error messages.

NOTE **Be careful when specifying PIDs with the `kill` command: if you kill the wrong process by mistake, you might end up in a lot of trouble.**

Example 12.9 Terminating a Process with the kill Command

In this example, the `yes` command is invoked in the background. Then, the `ps` command finds its `PID` and a `kill` command is sent to the `yes PID`.

```
$ bash<Enter>
$ yes > /dev/null &<Enter>
$ ps f<Enter>
PID TTY STAT TIME COMMAND
201 1 s 0:00 -bash
206 1 S 0:00 \_ bash
209 1 R 0:00 \_ yes
210 1 R 0:00 \_ ps f
$ kill 209<Enter>
```

NOTE **Another way to kill a process is `kill %jobnumber`. This method is discussed later in this chapter.**

If your terminal hangs up, locks, or freezes (all these names describe the same affliction) and Ctrl-C does not work, you might have to log in to a different terminal (or to a different virtual terminal). From there, you might use the `kill` command on the login shell of the hung-up terminal. This feature is just one advantage of UNIX operating systems over that popular GUI-related operating system we all know.

Table 12.6 Selected kill Command Signals

NUMERIC SIGNAL	BASH SIGNAL	TCSH SIGNAL	DESCRIPTION
1	SIGHUP	HUP	Hang-up signal sent to a process if its parent is terminated (such as logging off while the process is running). The process can terminate gracefully.
2	SIGINT	INT	Interrupt signal sent when <Ctrl>-c is sent from the keyboard.
3	SIGQUIT	QUIT	Quit signal sent when <Ctrl>-\ is sent from the keyboard.
9	SIGKILL	KILL	Unconditional kill, which cannot be caught or ignored. Stops processes before they have completed.
15	SIGTERM	TERM	Termination signal, which is the default that instructs a process to terminate.

kill Command Signals

With the `kill` command, you can specify one of several signals to terminate processes in a prescribed manner. Table 12.6 shows just a few of the 30-odd signals available. To get a complete listing of Linux `kill` command signals, type the following at the command line:

```
$ kill -l<Enter>
```

As we stated previously (and as can be seen in the `kill -l` listing), these signals are used to communicate a change of state to running commands. Depending on the signal used, the change of state ranges from an orderly stop to an immediate stop to a stop with the dumping of information to a file for debugging or even to a rereading of parameters.

Let's discuss the signals tabulated here in a little more detail. First, the *hang-up signal* (that is, `1`, `SIGHUP`, or `HUP`) is sent to a process if its parent process is terminated. For example, when you log out—that is, when you terminate the login shell—a hang-up signal is sent to any running background processes.

The *interrupt* signal (that is, `2`, `SIGINT`, or `INT`) is sent when a user presses the interrupt key sequence, Ctrl-C. A process running in the foreground stops unless its program is set to ignore interrupts. Then, a more

drastic step might be required. Interrupts do not work on background processes.

The *quit* signal (that is, `3`, `SIGQUIT`, or `QUIT`) is sent when the quit key is pressed. Generally, the quit key sequence is Ctrl-`\`, but it might vary by system. This signal produces a core file.

The *unconditional kill* signal (that is, `9`, `SIGKILL`, or `KILL`) is sent when no other signal can stop the process. No process can catch, or ignore, this signal. It should be used with caution because the target process stops immediately. It does not finish what it was intended to do, which could cause a loss or damage to files or information already generated by the process. For example, if you unconditionally kill a process that is updating a file, the updated material or even the entire file could be lost.

The *termination* signal (that is, `15`, `SIGTERM`, or `TERM`) is the default signal that tells the command it should proceed with an orderly shutdown. Some commands or shell scripts, however,

- Might be waiting for device operations to finish
- Might be attempting to interact with unavailable NFS resources
- Might contain statements that enable them to continue executing despite being sent a kill signal (that is, they contain statements that enable them to catch the kill signals)

In those cases, a more drastic `kill -9` might be necessary.

You can specify any one of several signals by using the syntax we show you in Example 12.9. But if you do not specify a signal, `kill` sends the default `-15` (also called `SIGTERM` in the bash shell or `TERM` in the tsch shell) to instruct all processes to terminate themselves.

Example 12.10 kill Signals

In this example, we use the unconditional `-9/KILL/SIGKILL` to show you the two basic ways to specify kill signals: using their numeric names or using their shell-specific names.

Numeric signals for the bash or tcsh shell:

```
$ kill -9 PID<Enter>
```

The bash shell signal name:

```
$ kill SIGKILL PID<Enter>
```

The tcsh shell signal name:

```
$ kill KILL PID<Enter>
```

NOTE **The number assigned to a signal by Linux/UNIX has no bearing on its strength or priority. In other words, a higher or lower number does not indicate potency.**

WARNING **Sometimes when a process is killed—especially if it has been terminated with a `kill -9`—a child process might be terminated. But the parent process, although notified of the child's termination, does not acknowledge the termination. The now-dead child process becomes what is called a zombie and will appear as `defunct` under the `COMMAND` column when you issue a `ps` command. You might or might not be able to `kill` a zombie; it will continue to be listed in the `COMMAND` column because it holds onto a process slot (that is, a record in the process table) until its parent process acknowledges its termination. A few zombies do not present a major problem, but if the `ps` table begins to fill up with them, your ability to execute legitimate processes declines. Zombies can be eliminated by a system reboot or occasionally by a `cleanup` initiated by the `init` process.**

Running Long Processes: The nohup Command

The `nohup` (no hang-up) command tells a background command or process to ignore `kill` signals `1` (hang-up) and `3` (quit). This command enables the background process to continue executing after the owner logs out of the system. The `nohup` command itself takes over control of the command. The syntax is as follows:

```
$ nohup command -option argument > filename &<Enter>
```

Because the `nohup` command is designed to shepherd a process *after* the user/owner has logged out, output from the command cannot go to the terminal screen. Therefore, the user/owner should redirect the output to a destination of his or her choice. If the user/owner does not redirect output, `nohup` automatically redirects the output to the *nohup.out* file in the directory in which the `nohup` command was originally invoked. See Example 12.11 for both types of redirection.

Example 12.11 nohup

When the user redirects output,

```
$ nohup ls -R > fileout &<Enter>
[1] 384
```

When the nohup command redirects output,

```
$ nohup ls -R &<Enter>
[2] 574
nohup: appending output to 'nohup.out'
```

NOTE **If more than one background process is started with** `nohup` **in the same current directory and the owner/user has not deliberately redirected output, the *nohup.out* file contains the output from all of those processes (either mixed or appended). This situation might create unpredictable results. If no output is required, the output could be directed to a log file or even to the null device** `(/dev/null)`.

Because all processes require affiliation with a parent process, commands started with nohup are affiliated with the `init` process as their parent after the owner/user logs out of the system.

After the nohup command is sent for execution, the shell replies by displaying numbers (such as `[1] 384` in Example 12.5). In this case, the numbers translate to "This is the first command this user is running in the background, and the `PID` is `384`."

Job Control in the bash and tcsh Shells

There may be times when you, in your role as administrator, have to change the priorities of the jobs running on your system. For example, in some organizations, routine processes are periodically suspended or given lower priorities when payrolls are calculated and checks are generated. Some organizations will interrupt otherwise routine processing for special technical tasks or for special reports. At those times, it is valuable to know how to access, suspend, resume, terminate, and otherwise manipulate the jobs on the system, whether they run in the foreground or the

background. This section introduces several handy commands which enable you to do so.

Creating a List of Background or Suspended Jobs

When you are running multiple processes, it is important to be able to identify which processes are running in the background. You cannot always determine which jobs are in the background with the `ps` command; that is where the `jobs` command comes in handy.

Looking at Example 12.11. You can see that the results from the `jobs` command indicate that there are two jobs and that each has been given a job number (in square brackets). The first is still running, and the second has run and completed. (If we invoked `jobs` again, the second job would not appear.) The `jobs` command does not list jobs that were started with the `nohup` command if the user has logged out of the system and then logged back in. If the user invokes a `nohup` job and issues the `jobs` command before logging out, however, that `nohup` job is displayed.

Example 12.12 Listing Background Processes with the jobs Command

```
$ jobs<Enter>
[1]- Running yes >/dev/null
[2]+ Done nohup ls -R
```

Suspending and Resuming a Foreground Task

Ctrl-Z is used to suspend, not terminate, a foreground process. No CPU resources are used for the suspended process, although it is still a process—it still occupies system memory and is subject to swapping to the hard disk. In other words, the suspended process's data, functions, scripts, and so on remain mapped in RAM until higher-priority processes are invoked that will occupy the same RAM space. If that happens, that will cause the suspended process's attributes to be saved in the swap space on the hard disk until the suspended process is resumed or terminated. You can tell the job to continue as needed in the foreground or background.

On some older ASCII terminals, Ctrl-Z might not be capable of suspending a foreground task. If that is the case, try entering the following to force the shell to use the key sequence to suspend processes:

```
$ stty susp <Ctrl>-z<Enter>
```

As seen in Example 12.13, to resume a foreground suspended task, simply enter `fg` at the prompt.

Example 12.13 Foreground Task Control

In this example, we use the `yes` utility that, in this case, simply enters the characters `y` and `newline` (that is, Enter) ad infinitum, using system resources until it is deliberately terminated. Thus, we are using `yes` as a utility with predictable and controllable outputs. If you are curious about more useful ways to use `yes`, consult your information sources.

So, to suspend a foreground task,

```
$ yes >/dev/null<Enter>
<Ctrl>-z
[1]+ Stopped yes >/dev/null
```

And then, to resume the suspended task,

```
[1]+ Stopped yes >/dev/null
$ fg<Enter>
yes ./dev/null
```

Suspending and Resuming a Background Task

If the job you want to suspend is running in the background, you will find that Ctrl-Z does not work because it is strictly a foreground-related command. In this case, use the `fg` command to bring the command into the foreground and then suspend it with Ctrl-Z. You might have to use the job number as supplied by the `jobs` command.

As shown in Example 12.13, you would issue `fg %jobnumber`, press Enter, and then press Ctrl-Z. Assume that you want to resume a suspended job and return the job processing to the background. Simply type `bg` (background) at the prompt. The shell returns a job number in square brackets followed by a plus sign (+). The plus sign indicates that the job is the most recently started or stopped. The returned display also includes the ampersand (`&`) at the end of the listed job name, indicating that the job is in the background.

Example 12.14 Background Task Control

To suspend a background task, use the `fg` command to move it to the foreground. Then, suspend it:

```
$ fg %jobnumber<Enter>
<Ctrl>-z
[1]+ Stopped yes >/dev/null
```

To resume the same suspended task and move it to the background,

```
[1]+ Stopped yes >/dev/null
$ bg<Enter>
[1]+ yes >/dev/null &
```

NOTE When you execute jobs to check background processes, the shell assigns a job number (in square brackets) to each job. With the `kill` command, you can use the job number or the `PID`. With the `bg` or `fg` command, you must use the job number; the `PID` will not work.

More Job Control Examples

Example 12.15 illustrates how a job can be created and controlled by using the commands covered in this chapter.

Example 12.15 Job Creation and Control

A job is initiated in the background, and the shell assigns it job number `1` and `PID 273`.

```
$ ls -R > out &<Enter>
[1] 273
```

The `jobs` command indicates that the job is running in the background:

```
$ jobs<Enter>
[1]+ Running ls -R > out &
```

The job is brought to the foreground:

```
$ fg %1<Enter>
ls -R > out
```

The job is stopped with Ctrl-Z, and the shell reports that it has indeed stopped:

```
<Ctrl>-z
[1]+ ls -R > out &
```

Again, `jobs` indicates that the job is running in the background:

```
$ jobs<Enter>
[1]+ Running ls -R > out &
```

`kill` terminates the job in the background:

```
$ kill %1<Enter>
```

The `jobs` command verifies that the job has terminated:

```
$ jobs<Enter>
[1]+ Terminated ls -R > out
```

Daemons: Never-Ending System Processes

Daemons are constantly running processes that were not started by the user and are not associated with the terminal. They start when you start your system and run until you shut down your system.

Daemons wait for a specific event to take place, such as the submission of a print job to a print queue. The printing daemon detects the event and then takes responsibility for the task, seeing that it gets processed. More specifically, Linux's printing daemon, called `lpd`, tracks print job requests as well as the printers available to handle those requests. The daemon maintains queues (that is, spool directories) of outstanding requests and sends each request to the appropriate device at the proper time.

In Linux, most of the filenames associated with daemons end with `d` and are found in configuration file directories (such as */usr/bin*). You can usually view them by entering the following command:

```
$ ps -guax<Enter>
```

If you have a moment, take a look at them. You will see common daemons such as `lpd`, `syslogd`, `inetd`, and `crond`. System administrators typically maintain these directories and files.

Exercises

1. Ensure that you are in your home directory, and then display your current process ID (`PID`).

```
$ cd<Enter>
$ pwd<Enter>
$ echo $$<Enter>
```

2. Create a subshell by entering `bash` at the prompt and then request the `PID` of the subshell. Is the subshell `PID` different than your login process?

```
$ bash<Enter>
$ echo $$<Enter>
```

3. Enter the `ls -R / > outfile 2> errfile &<Enter>` command and then execute the command that displays all your running processes. (The `ls` command terminates when it gets to the end of the file system.)

```
$ ls -R / > outfile 2> errfile &<Enter>
$ ps a<Enter>
```

4. Terminate your child shell.

```
$ exit<Enter>
```

   What happens if you type `exit` from your login shell?

5. Display all variables in your current process environment.

```
$ set<Enter>
```

6. Create a variable named `x` and set its value to `10`. Check the value of the variable. Again, display all current variables.

```
$ x=10<Enter>
$ echo $x<Enter>
$ set<Enter>
```

7. Create a subshell.

```
$ bash<Enter>
```

   Check to see what value the x variable holds in the subshell.

```
$ echo $x<Enter>
```

   What is the value of x? List the subshell's current variables.

```
$ set<Enter>
```

   Do you see a listing for x?

8. Return to your parent process.

```
$ exit<Enter>
```

   Set the value of the x variable so that its value will be inherited by your child processes.

```
$ export x=10<Enter>
```

   Verify by creating a subshell and checking the value of x in the subshell.

```
$ bash<Enter>
$ echo $x<Enter>
```

9. Change the value of x in the subshell to 200.

```
$ x=200<Enter>
```

   Verify that the value was changed.

```
$ echo $x<Enter>
```

10. Return to the parent process.

```
$ exit<Enter>
```

Check the value of x in this environment.

```
$ echo $x<Enter>
```

Was the change in the subshell exported back to the parent?

11. Create a shell script and name it `sc1`. It should read `pwd, cd /, pwd`

```
$ vi sc1<Enter>
i (to enter Insert mode)
pwd
cd /
pwd
<Esc> (to leave Insert mode)
:x<Enter> (to save scl and leave vi)
```

12. Make `sc1` executable and run the program.

```
$ chmod 700 sc1<Enter>
$ sc1<Enter>
```

What directory are you in now? Why?

```
$ pwd<Enter>
```

NOTE **If you received a response such as** `bash: sc1: command not found`, **rerun the program with the command** `/home/username/sc1`**. It is likely that** ***/home/username*** **was not in your** `PATH` **environment variable, so the shell could not find the** `sc1` **command.**

13. Create another shell script and name it `sc2`. Have it read as follows:

```
var1 = hello; var2=$LOGNAME; export var1 var2.
$ vi sc2<Enter>
i (to enter Insert mode)
var1=hello
```

```
var2=$LOGNAME
export var1 var2
<Esc> (to leave Insert mode)
:x<Enter> (to save sc2 and quit vi
```

14. Make `sc2` executable and run the program.

```
$ chmod 700 sc2<Enter>
$ sc2<Enter>
```

When the script has finished, examine the values of the `var1` and `var2` variables.

```
$ echo $var1 $var2<Enter>
```

What values do `var1` and `var2` have? Why?

15. Run the `sc2` program again, this time by forcing it to run in the current shell.

```
$ . sc2<Enter>
```

When it is finished, check the values for `var1` and `var2`.

```
$ echo $var1 $var2<Enter>
```

What values do `var1` and `var2` have now? Why?

16. Execute the `ls -R / > outfile 2> errfile<Enter>` command in the foreground.

```
$ ls -R / > outfile 2> errfile<Enter>
```

17. Suspend the job you just started.

```
$ <Ctrl>-z
```

18. List all the jobs that you are running on the system, and restart the preceding job in the background.

```
$ jobs<Enter>
$ bg %jobnumber<Enter>
```

19. Bring the job back to the foreground.

```
$ fg %jobnumber<Enter>
```

20. After the `ls` command finishes executing, restart it again in the background. Display the process ID and log out.

```
$ ls -R / > outfile 2> errfile<Enter>
$ ps a<Enter>
$ exit<Enter> (You will see a message telling you that you have jobs
running.)
$ exit<Enter>
```

21. Log in. Check to see whether the process is still running.

```
Login: username<Enter>
Password: <Enter>
$ ps a<Enter>
```

22. Create a shell script and name it `sc3`. It should read as follows:

```
sleep 60; ls -R / > outfile 2> errfile &.
$ vi sc3<Enter>
i (to enter Insert mode)
sleep 60
ls -R / > outfile 2> errfile &
<Esc> (to leave Insert mode)
:x<Enter> (to save sc3 and quit vi)
```

Make the script executable.

```
$ chmod 700 sc3<Enter>
```

Start the script with the `nohup` command, reference it by using an explicit path, and put it in the background. Do not forget to redirect the output from `sc3` and then log out.

```
$ nohup ./sc3 > sc3.out 2> sc3err &<Enter>
$ exit<Enter> (You will see a message telling you that you have jobs
running.)
$ exit<Enter>
```

23. Log in. Check to see whether the process is still running.

```
Login: username<Enter>
Password: <Enter>
$ ps a<Enter>
```

24. When the process has finished, display the file that contains your output. (Hint: If you did not specify an output file, `nohup` sends the output to `nohup.out`.)

```
$ more /home/directoryname/sc3.out<Enter>
```

25. Use the `ls -R /<Enter>` command to start a long-running job in the background.

```
$ ls -R / > outfile 2> errfile &<Enter>
```

Note the process ID that is provided when you begin the background process.

26. If you did not record the process ID when you first started the command in the background, how would you find it?

After you know the process ID, `kill` the process. Check to be sure it was killed.

```
$ ps a<Enter>
$ kill PID<Enter>
```

(For this step, enter the `PID` you recorded in Exercise 25 or found with the `ps a` command you just executed.)

```
$ ps a<Enter>
```

27. Repeat Exercise 25. Kill the process using the job number rather than the process ID. Check to be sure the job was killed.

```
$ ls -R / > outfile 2> errfile &<Enter>
$ jobs<Enter>
$ kill %jobnumber<Enter>
$ ps a<Enter>
```

or

```
$ jobs<Enter>
```

See Appendix B for answers.

Quiz

1. Which command is used to pass the value of a variable from a parent shell to a subshell?
2. Determine the value of `x` at the end of the following procedure:

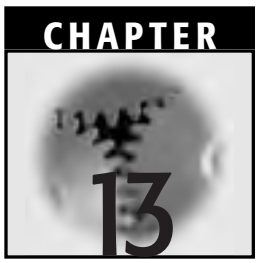
```
$ (starting in the login shell)
$ x=5<Enter>
$ bash<Enter>
$ x=50<Enter>
$ export x<Enter>
$ <Ctrl>-d
```

3. True or false: each process returns an exit code to its parent process after successful completion.
4. Which of the following is the proper syntax for checking a shell's process identification number?
   - `$ echo $$<Enter>`
   - `$ echo $?<Enter>`

- `$ echo $pid<Enter>`
- `$ ps ef<Enter>`

5. When you export a variable, the variable and its value become part of what environment?
6. What option would you use with the `ps` command to show the relationships between your running programs and their parent processes?
7. True or false: As an ordinary user, you can `kill` only your own jobs and not those of other users.
8. Which of the `kill` command signals is strongest and cannot be caught or ignored?
9. It is always a good idea to start long jobs in the background with the `nohup` command. Give two reasons why.
10. What are never-ending Linux/UNIX processes called?
11. True or false: When users execute a command in the normal way and the command involves navigation of the directory structure, they will find themselves in the last directory mentioned in the command unless some type of return command (such as `cd`) is included in the program or script.

See Appendix C for answers.



Shell Programming

This chapter does some cool stuff and gives you an opportunity to test your shell command knowledge. At this point, we do not expect you to be able to construct a shell script by yourself, but you should be able to read one. We have learned from past experience that the best way to introduce someone to scripting is to use lots of examples. This chapter contains some very basic shell scripts with some discussion of what they do and what they can be used for.

Shell Scripts

A shell script is:

- A readable text file that can be edited with a text editor like `vi` or the others we mentioned in Chapter 10, "The vi Editor"
- A program that contains system commands, variable assignments, flow control syntax, and shell commands
- A program that contains comments to let you or some other reader or developer in the future or in some unknown place know what the

objectives, subroutines, and expected outputs were of the script itself (sometimes the comments are as important as the script commands themselves)

Thus, a *shell script* is a collection of system commands stored in a text file that the shell reads and executes in sequence. A script can enable you to do anything that you could normally do from the shell prompt. Therefore, in its simplest form, it will contain at least one Linux/UNIX command, as shown in Example 13.1. When the shell processes a shell script, it reads the script file one command at a time, parses the commands, and sends them to the operating system for execution. The commands are executed in turn, just as if you had typed them at the terminal command line.

You can execute any Linux command in a shell script. In fact, some shell features that you can use in a script cannot be accessed at the command line. They are built-in facilities that enable more complicated functions to be performed.

Meanwhile, you can use any Linux/UNIX text editor to create a shell script.

Example 13.1 Using the cat Command to Create a Simple Shell Script

Lady Molinera wants to execute the `date` command, but she does not want to enter it on the command line every time she wants to invoke it. So, she creates a shell script called `script1` using the `cat` command as her text editor:

```
$ cat > script1<Enter>
$ date<Enter>
<Ctrl>-d
```

Example 13.2 Creating a Simple Shell Script with vi

Lady Molinera also likes to keep close track of her directory contents. She wants to execute `date`, `pwd`, and `ls -l` periodically but does not want to enter the commands at the command line one after the other every time. So, with `vi`, she creates another shell script called `script2` to handle this function:

```
$ vi script2<Enter>
i
```

```
date
pwd
ls -l
:wq
```

Example 13.3 Creating the Same Simple Shell Script with cat and ;

Lady Molinera could have used `cat` to create `script2`, too. There is really no difference between the scripts created with `vi` and the syntax shown next. You will come across (and use, likely) several styles of script creation when reviewing others' shell scripts. Here, the Lady creates `script3`, which does the same as `script2` with `cat`:

```
$ cat > script3<Enter>
date;pwd;ls -l<Enter>
<Ctrl>-d
```

Executing Shell Scripts

Before executing a shell script, you have to determine two things:

- Can the users access the script file?
- How will the script be executed? Are the permissions on the shell script file set so the script can be executed in the manner you intended?

Can Users Access the Script File?

In order for users to be able to execute the script file, they have to be able to access it. If they cannot, then naturally they will never be able to execute it.

One way to make it accessible is to put the script file in a directory to which there is a path. So, try to make sure its directory is listed in the `PATH` environment variable: yours, your group's, or everyone's. If there is no path to the directory, then whoever tries to invoke it will get a message from Linux similar to the following:

```
bash: script1: No such file or directory
```

If the shell script cannot be located in the user's path, then they will have to provide a directory path to the script (relative or absolute) when they execute it.

Executing the Shell Script

There are two basic ways to execute a shell script:

- Use the `bash` command (if you are in the bash shell; otherwise, you use the respective shell execution command).
- Make the script file an executable file all its own. That way, the user only has to use its name to execute it.

Using the bash (or Equivalent) Command to Execute the Script File

If the script's permissions are not set as executable, you can still execute the script, but you have to use the `bash` command. In this case, no one needs the `x` (in other words, execute) permission on the file—not even the owner. `bash` will take care of it. Later in this chapter, we will show you what is actually happening when scripts are executed in this manner from a process standpoint.

Example 13.4 Using the bash Command to Execute a Shell Script

Here, Lady Molinera executes her `script2` script by using the `bash` command:

```
$ bash script2
Sat Jul 28 13:45:25 MDT 2001
/home/molinera
total 12
-rwxrwxr-x   1   molinera   knights3   110  Jul 14  12:47  apples
drwxr-xr-x   2   molinera   knights3  4096  Jul  4  12:36  Desktop
-rw-rw-r--   1   molinera   knights3     8  Jul 25  13:24  script1
-rw-rw-r--   1   molinera   knights3    15  Jul 25  13:36  script2
```

Note that in this example, we presumed one of two things: 1) that the script file is in the same directory as Lady Molinera (which the response actually substantiates), or 2) that the script file could have been somewhere in the directories defined by the Lady's `PATH` variable.

To check her PATH environment variable value(s), let's have Lady Molinera echo her PATH:

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:.:/home/molinera/bin:
```

Making Shell Script Files Executable

The alternative (the preferred alternative, really) for script execution relies on the owner of the shell script file making that an "executable" by using the chmod command (we discussed chmod in Chapter 6, "Linux File Permissions"). Users of the shell script file must have both read and execute permissions for the shell script file. Why? The answer is, because the shell needs to open the script file to read the commands within it. If you give "group" the execute permission, however, you need not give it to "others" and vice versa (although if someone is in "group," he or she is probably in "others," too). You can be selective.

The owner can use either of the following two syntax formats:

```
$ chmod ugo+rx scriptfilename<Enter>
```

or

```
$ chmod 755 scriptfilename<Enter>
```

or

```
$ chmod +x script2<Enter>
```

In those three examples, "execute" permission is given to everybody. We could have narrowed the permissions somewhat to "ug+rx" or "754" for just the "same group" members or just "u+x" or "744" so that the owner is the only one who can execute it like that.

At any rate, after you complete the previous step(s), the designated user(s) can invoke the script2 shell script as if it were an ordinary command (in other words, without bash or any other shell execution command) as follows:

```
$ script2<Enter>
```

Similar to executing with the `bash` command, later in this chapter we will show you what is actually happening from a process standpoint when scripts are executed in this manner.

Example 13.5 Using the bash Command to Execute a Shell Script

Lady Molinera wants to make her `script2` file executable so that she can execute it without relying on the shell execution commands all the time. Plus, she would like to make `script2` available to others in her group for their convenience, too. Here is how she does both at once:

```
$ chmod 754 script2<Enter>
```

She can now invoke it in the following manner:

```
$ script2<Enter>
Wed Jul 25 13:54:25 MDT 2001
/home/molinera
total 12
-rwxrwxr-x   1  molinera  knights3    110  Jul 14  12:47  apples
drwxr-xr-x   2  molinera  knights3   4096  Jul  4  12:36  Desktop
-rw-rw-r--   1  molinera  knights3      8  Jul 25  13:24  script1
-rwxr-xr--   1  molinera  knights3     15  Jul 25  13:36  script2
```

Before we move on, see how the permission bits were changed on `script2`?

Shell Script Invocation from a Process Standpoint: Three Options

We have already seen the results of two of the following three options for invoking shell scripts. Here are all three:

- Invoking shell scripts by using the `bash` command
- Invoking shell scripts that are executable
- Invoking shell scripts inside the current shell

Basically, all three methods seem to create the same results. But they might cause the shell(s) to behave differently. To illustrate, we will use Lady Molinera's very simple `script1` file created in Example 13.1.

Option 1: Invoking Scripts by Using the bash Command

When a shell script like `script1` is invoked with `bash`, a `bash` subshell is invoked by the login `-bash` shell to execute the shell script file.

That other `bash` subshell runs as a child to the parent login shell. When that subshell has finished executing the commands in the `script1` file, it terminates and control is passed back to the parent login `-bash` shell.

Example 13.6 Processes Invoked When the bash Command Executes a Shell Script

The only way that we can really demonstrate this behavior is to

- Identify and echo the current shell's `PID`.
- Ask `script1`, as it is being executed, to `echo` back the `PID` of the shell in which it is actually being executed.

To do so, we have asked Lady Molinera to modify her `script1` to `echo` the current subshell's `PID` as it runs. The script now looks like the following (the `echo $$` will display the `PID` of the subshell that actually invokes `script1`):

```
$ cat script1<Enter>
date
echo $$
```

Before running the script, she displays the current shell's `PID` to demonstrate that it is different from the subshell in which the script will eventually execute:

```
$ echo $$<Enter>
2393
$ bash script1<Enter>
Wed Jul 25 14:35:59 MDT 2001
2413
```

The output shows that the original login `-bash` shell has a `PID` of 2393 and that `script1`, when executed, returned the date and a `PID` of 2413. This concept is very important to remember. In this case, `PID` 2413 is a child of `PID` 2393.

Figure 13.1 illustrates that the login shell called `-bash` opens another bash subshell to run `script1`, and the `date` command itself will run in yet another subshell. All of the subshells exist *only* for the time it takes to run their respective commands. Also, when they are finished, control is passed back to the login `-bash` shell, which has been asleep while the subshells performed their functions.

Option 2: Invoking Scripts by Making Them Executable

When `script1` is made to be executable on its own and then invoked, a `bash` subshell is invoked again by the login `-bash` shell to execute the shell script file (basically the same process as Option 1 when the bash command was used).

The only difference is—and this fact is probably obvious to you by now—that the shell script has been flagged as executable. Then you can execute the script directly, like such:

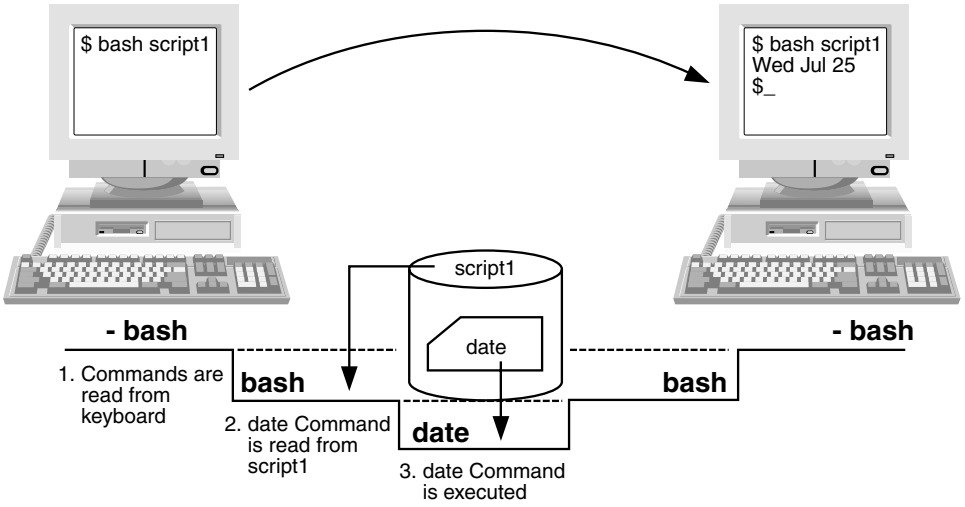


Figure 13.1 Invoking shell scripts by using the bash command.

```
$ script1<Enter>
Wed Jul 25 15:07:51 MDT 2001
```

As illustrated in Figure 13.2, invoking `script1` on its own in the login shell invokes another `bash` subshell that runs as a child to the login shell. When that subshell has finished executing the commands in the `script1` file, it terminates and control is passed back to the parent login `-bash` shell.

Example 13.7 Processes the Shell Script Invoked Itself

We will demonstrate this behavior in a manner identical to Option 1 (Example 13.6):

- Identify and echo the current shell's `PID`.
- Ask `script1`, as it is being executed, to `echo` back the `PID` of the shell in which it is actually being executed.

Using `$ chmod 775<Enter>`, Lady Molinera has now made `script1` executable. Its long listing now looks like the following:

```
$ ls -l script1<Enter>
-rwxrwxr-x   1  molinera  knights3  13 Jul 14  15:10  script1
```

Before running the script, she again displays the current shell's `PID` to demonstrate that it is different from the subshell in which the script will eventually execute:

```
$ echo $$<Enter>
2393
$ bash script1<Enter>
Wed Jul 25 14:35:59 MDT 2001
2443
```

This output shows that the original login `-bash` shell has a `PID` of 2393 and that `script1`, when executed, returned the date and a `PID` of 2443. Again, this concept is important to remember. `PID` 2413 is the child of `PID` 2393 this time.

Figure 13.2 illustrates that the login shell called `-bash` opens another bash subshell to run `script1` and that the `date` command itself will run in yet another subshell. When they were finished, control was passed back to the login `-bash` shell, which was asleep while the subshells performed their functions.

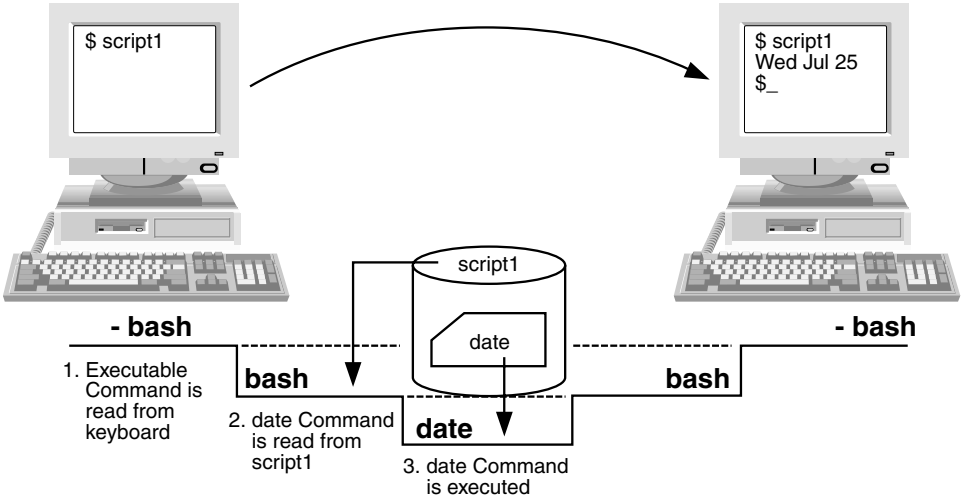


Figure 13.2 Invoking shell scripts that are executable.

Option 3: Invoking Scripts in the Same Shell

It is possible to force shell scripts to execute in the current shell instead of opening all the other subshells. You perform this task by altering the syntax to what is shown here:

```
$ . scriptfilename<Enter>
```

or

```
$ source script1<Enter>
```

In the first syntax example, the "." denotes "in the current shell." The process will ultimately resemble that shown in Figure 13.3.

Example 13.8 Invoking Scripts in the Same Shell

Freston tells Lady Molinera that she can save a few CPU cycles and some nanoseconds if she forces her `script1` to execute in the same shell she is in now. But to prove it, he has her check her shell's `PID`'s again, just like she did for Options 1 and 2.

```
$ echo $$
2393
$ . script1
Wed Jul 25 15:55:28 MDT 2001
2393
```

She executed script1 by running it with a "." in front of it. Remember, the "." means, "Do this in the current shell." The original shell returned a `PID` of 2393, and the shell script also returned a current `PID` of 2393. The alternative syntax to the "." is "source." These two methods are the same, and Figure 13.3 illustrates both of them.

```
$ echo $$
2393
$ source script1
Wed Jul 25 16:05:53 MDT 2001
2393
```

Example 13.9 Changing .bash_profiles "On the Fly"

Where else do you think this method could be useful? Assume that Freston has just made changes to his hidden *.bash_profile* file. Using the "`.`" or `source`, he can make the shell re-execute his user profile without him having

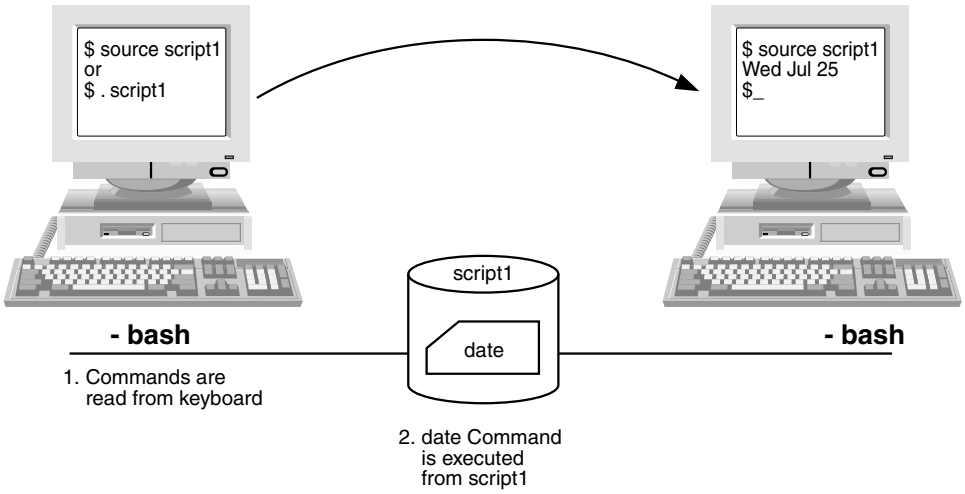


Figure 13.3 Invoking shell scripts inside the current shell.

to log out and log in again. After he has made the changes to the file itself, all he has to do is type either of the following:

```
$ . .bash_profile<Enter>
```

or

```
$ source .bash_profile<Enter>
```

Creating Scripts: Some Practical Examples

Example 13.10 Exporting Variables and Values for Use with Scripts

The following script shows a variation on the classic Hello World script: Sancho is sending a greeting to his burro, Dapple (well, maybe one of the trainers will read it to him. Who knows?). We are going to use this simple script to illustrate the exporting of variables and how they can be accessed in subshells.

```
$ cat > hello<Enter>
1 #!/bin/bash
2 #This is a simple Hello World type of script
3 #
4 var1="Hi Dapple! Ready for our August journey?"
5 echo $var1
<Ctrl>-d
```

Now, to make the script executable,

```
$ chmod 755 hello<Enter>
```

When it is executed, the `hello` script will return the following:

```
$ hello<Enter>
Hi Dapple! Ready for our August journey?
```

The `hello` script has been listed with line numbers for reference. As we examine it, we will not mention all the lines. Actually, if you are using these scripts as example, leave the line numbers off.

Line 1, although it has a tic-tac-toe symbol, is actually a shell declaration (the ! ensures that). Putting a declaration in such as this

one is considered proper practice, because you never know what shell the user will be in (and this addition guarantees that this script will execute in a `bash` shell environment).

Line 2 is a comment. It is also considered best practice to describe what the script does and sometimes who to contact for support. Quite often, these lines include information about script versioning in large environments.

Line 3 is a comment spacer line. This line makes scripting "easier on the eyes" (in other words, easier to create; easier to debug; easier to analyze; and even easier to copy).

Line 4 sets a user-defined variable called `var1` to the text string "`Hi Dapple! Ready for our August journey?`". When a variable is set inside the script, there will never be any question on what var1 is equal to.

Line 4 will `echo` the string to standard out, which is the terminal screen.

We will work with this example to demonstrate how to access variables outside the script. In the sample code coming up next, we will set another variable. But this time, it will be set outside the script in the current shell environment. The thing to note here is that it will not find the variable. This situation is intentional, and we will fix it in the next sample code:

```
$ cat >> hello<Enter>
echo $var2<Enter>
<Ctrl>-d

$ var2="Yes I am. Let's go!"<Enter>
$ echo $var2<Enter>
Yes I am. Let's go!
$ cat hello<Enter>
#!/bin/bash
#This is a simple Hello World type of script
#
var1="Hi Dapple! Ready for our August journey?"
echo $var1
echo $var2
$ hello<Enter>
Hi Dapple! Ready for our August journey?
```

Notice that `var2` did not display. The reason why is because the script executed in a subshell. The `var2` variable, created in the parent login shell,

was not exported to the child subshell. Therefore, the subshell did not have access to it. Let's fix that now by using the `export` feature.

```
$ export var2<Enter>
$ hello<Enter>
Hi Dapple! Ready for our August journey?
Yes I am. Let's go!
```

For this reason, many system administrators choose to set all their variables inside a script that needs them. In the event that you are in a situation where you are relying on a variable outside your shell, you should always conduct a test to see whether the variable is null or not before proceeding with whatever processing you wish to perform.

Example 13.11 A Simple Backup Script

A more practical, but still simple, script would be a backup script. But first, let's look at Freston's private directory:

```
$ ls -l<Enter>
total 56
-rwxrwxr-x   1   freston   knights4    110   Apr 14  12:47  monthly.rept
drwxr-xr-x   2   freston   knights4   4096   Jan  4  12:36  Desktop
-rwxrwxr-x   1   freston   knights4    102   Mar 14  16:26  hello
-rwxrwxr-x   1   freston   knights4     74   Jul 14  16:43  private_bu
-rwxrwxr-x   1   freston   knights4     13   Jul 25  14:30  script1
-rwxr-xr-x   1   freston   knights4     15   Jul 25  13:36  script2
-rw-rw-r--   1   freston   knights4   2288   Jul 26  16:44  typescript
```

See if you can interpret what the following script does:

```
$ cat private_bu<Enter>
#!/bin/bash
BUF=/nightly/$USER-backup-$(date +%Y%m%d).tgz
tar -cZf $BUF ~
```

The script is called `private_bu`, and let's say that its permissions are executable. The script is directed to run in a `bash` shell environment. A variable called `BUF` is set to a string that will represent the name of the backup file used by the `tar` command. The string for `BUF` will resolve another variable called `$USER` to Freston and will set the date with a specified format of year-month-day to keep the filename unique. The `tar` command will use this `BUF` variable to create the file inside a directory called *nightly*. Basically, Freston is creating a one-user backup file in a special directory.

Does it work? Let's invoke it:

```
$ private_bu<Enter>
$ ls -l /nightly<Enter>
total 12
-rw-rw-r-- 1 freston knights4  8299 Jul 14 16:44 freston-backup-
20010714.tgz
```

Well, we can tell you that it worked. Try it.

Conditionals

We mentioned earlier that sometimes you need to test for a condition—that is, to see whether it something true or not—before proceeding with a script. The use of conditionals like this is quite simple and the conditionals generally use combinations or variations of "if," "then," and "else" expressions. These conditionals can use many forms but at the same time must abide by some simple rules.

If, Then, Else

The simplest form of a conditional is just *if* and *then*, as follows: "*If* this happens, *then* do that." The *if* part has to be true, though. Otherwise, it is ignored. Sometimes this simple "*If . . . then*" is all you need.

When this statement is not enough, you can add *else*: "*If* this happens to be true, *then* do the following. For all those that are false, then do *else*." Think of each of these elements as being people. Mr. *If* decides whether it is true or false. If it is true, Mr. *If* will give the work to Mr. *Then*. If it is false, *then* Mr. *If* will give the work to Ms. *Else*. The beauty of Mr. *If*'s job is that everything is either true or false. No matter what the response is, he gets to hand the rest of the job off to somebody else.

The only other thing that we will mention before showing some examples is that there are many forms of the use of *if*, *then*, and *else* in other programming environments. Unless you want to make a career out of memorizing when a finish is required and when it is not required, just use it. At the end of every *if*, *then*, and *else* should be an *fi*.

Example 13.12 The Modified Simple Backup Script

Freston modified the script from the previous example to check for "disk full" before backing up files in the home directory. Look for the "*if . . . then . . . else*" sequence:

```
$ cat private_bu
#!/bin/bash
var3=`df -h | grep hda6 | grep % | cut -c 40-42`
BUF=/nightly/$USER-backup-$(date +%Y%m%d).tgz
if [ $var3 -lt "85" ] ; then
tar -cZf $BUF ~
else
echo "The file system is too full! Call system administrator NOW!"
fi
```

This script also sets a variable called `var3` to the results of the piped commands. This method is a very nice way to supply information to a script. This specific case takes the results of the `df- h` command, `greps` out the "% used" number, and then uses `cut` to single out the value. If the value exceeds 85 percent, then it is to issue a console message to contact the system administrator.

For, Cron, While, and Until Loops

Nothing is handier than a little mechanism that can save you lots of redundant keying. We are going to show you how to use basic *for, while*, and *until* loops to aid your productivity. We also mention `crontab`, too.

For Looping

For is good for iterating over strings of data and performing basic functions.

Example 13.13 "For Looping" the Scripts to the Boss

Suppose that Lady Dulcinea wants a copy of Freston's script files at the end of every day. Freston would write a script with a *for* loop to accommodate this request. Here he goes:

```
$ cat > cp4boss<Enter>
#!/bin/bash
#
# Copy files to the bosses directory
#
for i in $( ls script* ) ; do
   cp $i /home/dulcinea
   echo $i copied to Lady Ds directory
done
<Ctrl>-d
```

```
$ chmod 755 cp4boss<Enter>
$ cp4boss<Enter>
script1 copied to Lady Ds directory
script2 copied to Lady Ds directory
private_bu copied to Lady Ds directory
cp4boss copied to Lady Ds directory
```

Example 13.14 "Cron-ing" the Scripts Every day

Because it is a regular activity, Freston `crons` the job to run at 7 P.M. every day. This way, he knows he will not miss any, and he does not have to dial up from home when he forgets.

```
$ crontab -l<Enter>
# DO NOT EDIT THIS FILE - edit the master and reinstall.
# (/tmp/crontab.2886 installed on Sat Jul 14 18:37:01 2001)
# (Cron version -- $Id: crontab.c,v 2.13 1994/01/17 03:20:37 vixie
Exp $)
0 19 * * /home/freston/cp4boss  #Copy script files to bosses directory
```

Example 13.15 "While Looping" to Prevent a System Crash

At about 2 P.M. on Mondays in October, Freston knows that everyone in the extended RFI organization will decide to check the local soccer, football, baseball, *and* hockey pool results. He knows that such a load will be just too much for the server to handle. When the number of users exceeds 540, he knows the system will crash. So, in preparation for the inevitable heavy load, he decides that a *while* script would be a worthwhile thing to develop in order to advise all those who are logged in to try another time before the system crashes. The script will send out a warning every five seconds until users respond. He has seen it in action before: It is *very* annoying and therefore works well. His script looks like the following:

```
$ cat > warn_users<Enter>
#!/bin/bash
#
# This script warns users that the system may be slow due to peak usage
# It will urge them to logout if they aren't active
#
var4=`w | wc -l`
while [ $var4 -gt "435" ] ; do
  wall System will be slow because there are $var4 other users logged in.
  wall If you are not active please logout!
sleep 5
done
<Ctrl>-d
```

He will run the script "in the background." To submit the job for background use, he uses the "&" (ampersand symbol; Juana calls it a "pretzel"). Here is how he will do it:

```
$ warn_users &<Enter>
```

In October, when the "crunch" comes, this message is what the users will see every five seconds:

```
Broadcast message from freston (pts/2) Mon Oct 15 14:50:54 2001...
If you are not active please logout!

Broadcast message from freston (pts/2) Mon Oct 15 18:50:54 2001...
System will be slow because there are 456 other users logged in.

Broadcast message from freston (pts/2) Mon Oct 15 18:50:59 2001...
If you are not active please logout!

Broadcast message from freston (pts/2) Mon Oct 15 18:50:59 2001...
System will be slow because there are 456 other users logged in.
```

They will hate him for it. But it will work. And everyone will still be able to work.

Example 13.16 "Until Looping" to Prevent a System Crash

Freston could have taken a different approach to the same problem by implementing an *until* script. With this script, until the number of users logged in is reduced to 425, the messages will continue to appear. That script would look like the following:

```
$ cat > warn_users<Enter>
#!/bin/bash
#
# This script warns users that the system may be slow due to peak useage
# It will urge them to logout if they aren't active
#
var4=`w | wc -l`
until [ $var4 -lt "425" ] ; do
  wall System will be slow because there are $var4 other users logged in.
  wall If you are not active please logout!
sleep 5
done
<Ctrl>-d
```

Some More Tips for Scripts

These scripts that we have presented are very simple yet effective. As your requirements increase, so do the sizes and complexity of the scripts. Although this section is a very brief introduction to scripting, there are some very sound recommendations and best practices for creating robust scripts:

- Document the prerequisites and main sequence for running scripts.
- Divide actions into logical groups.
- Develop an execution sequence based on a common usage scenario.
- Provide comments and instructions in each shell script.
- Make an initial backup to create a baseline.
- Check for input parameters and environment variables.
- Try to provide "usage" feedback.
- Try to provide a "silent" running mode.
- Provide one function to terminate the script when there are errors.
- When possible, provide functions that do a single task well.
- Capture the output of each script while watching the output being produced.
- Inside each script, capture the return code of each line command.
- Keep a count of the failed transactions.
- Highlight the error messages for easy identification in the output file.
- When possible, generate files "on the fly."
- Provide feedback on the progress of the execution of the script.
- Provide a summary of the execution of the script.
- Try to provide an output file that is easy to interpret.
- When possible, provide cleanup scripts and a way to return to the baseline.

Quiz

1. When would you execute a shell script by using the dot (.) command? Why?

2. When invoking shell scripts, which permissions among the following must be given to the script file?
   - `drwsxrwxrwx`
   - `r` but not necessarily `w` or `x`
   - `x` and `r`
   - `755`
   - `-rw-r-xr-x`
   - `r` but not necessarily `w` or `x` and `-rw-r-xr-x`
   - `r` but not necessarily `w` or `x`; `x` and `r`
   - `-rw-r-xr-x`

3. How would you handle making variables available to subshells?
   - Define the variable inside the script.
   - Recreate the variable in the new shell.
   - Export the variable.
   - Place all new variables required in a file.

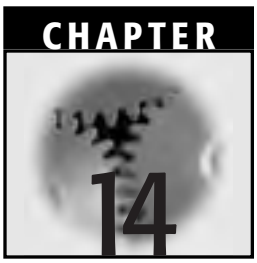
4. How could you guarantee that whomever ran your script would run it in a bash environment?

5. Will the following script print true or false?

```
#!/bin/bash
T1="foo"
T2="bar"
if [ "$T1" = "$T2" ] ; then
    echo This script evaluates as true
else
    echo This script evaluates as false
fi
```

6. Using the same script, if we were to export `T2="foo"`, would it change how the script evaluates?

See Appendix C for answers.



The Linux X Window System

Thus far, we have concentrated on Linux at the command line. Depending on your information source, that is also called

- At the character cell prompt
- At the console
- In the ASCII character environment

Although it seems to be an outmoded way of doing things—especially when there are so many GUIs available for a price or for free—for effective Linux usage and especially administration, we often rely on our ability to use and understand command-line concepts, commands, and utilities.

Linux also has powerful and flexible GUI capabilities, however. This chapter provides an introduction to the GUI foundation, called the X Window System. We wholeheartedly recommend installing the X Window System when you first install Linux on your system (our preference) or later after Linux has been installed. After you properly install and configure X, it will facilitate your productivity and enjoyment and will enable you to use many more Linux/UNIX programs.

WARNING **For those who are planning to install X Window System, we echo the advice of all X installers: Pay special attention to your system's video card and monitor specifications. Failure to do so could result in system or monitor damage.**

Meanwhile, some of the concepts in this chapter (and the way we explain them) will seem to be very basic—even trivial to some (dare we say, "most"?). If so, we apologize. But it is still very important to form a proper knowledge foundation. Also, you never know when you might find a "pearl" among these oysters. Hey, some people like oysters. I know I do.

A Brief History

The first X Window System was developed in 1984 by the *Massachusetts Institute of Technology* (MIT). The X Window System, which is often referred to as X or X11, provides a powerful network-based foundation for a Linux/UNIX GUI environment. X is a collection of programs, including servers, documentation, fonts, programming libraries, and utilities, but its foundation programs are as follows:

- A basic windowing program that provides windowing services
- The X Network Protocol, a protocol for network communication
- A low-level interface called `Xlib`, which sits between the higher-level programs (on top) and the network or base system (on the bottom)
- A "window manager"—an X application to control the type and appearances of the windows

The X Window System enhances UNIX in the same way that the "other GUI operating system" enhances (or enhanced at one time) DOS, only more reliably. What do we mean by *enhance*? Some advantages and features of GUIs are listed next:

- An attractive and intuitive (that is, easy-to-use) human-to-machine interface
- Potentially easier ways of identifying, entering, and modifying data
- Several control devices that require no knowledge of programming or other commands (for example, buttons, icons, menus, and scrollbars)

- Consistency of look and utility, which leads to reduced training requirements for new applications
- Greater ease in managing multiple and simultaneous processes and applications

Now, all is not perfect in the world of GUIs. GUI programs by nature are generally large and rather inefficient because they tend to use a lot of resources. They have to write or call upon many controlling functions for window manipulation, graphics control, and data input from a keyboard and a mouse. In short, GUIs require more resources than the command line but can also make complex programs easier to use.

Since the birth of X in 1984, commercial vendors—particularly the vendor- and researcher-based organization called the X Consortium, whose membership includes IBM, Digital Equipment, and MIT—have effectively made X the UNIX industry standard. X is installed on most UNIX systems around the world.

A free software conversion (in other words, a free "port") of X for Linux and other UNIX variants, called XFree86, was developed by the XFree86 Project Inc. (now a member of the X Consortium). XFree86 is a version of MIT's X Window System v. 11, rel. 6 (that is, X11R6).

If XFree86 is installed on your system, you will see several X11R6 references as part of directory or filenames. The original X version 11 was released in 1987, release 6 came out in 1994, and the latest version is X11R6.4. As of this writing, the latest release of XFree86 itself is 4.1.0. Your version of XFree86 probably has all the binary files, support files, libraries, and tools that you will need. If not, check the XFree86 Project Internet site at www.XFree86.org. You can also check that Web site for the latest version of XFree86.

X Window Networking

In Figure 14.1, we illustrate a simplified X Window System network consisting of various types of machines. In an environment like this one, an X application could run on one processor or even on one type of processor. A user or administrator could sit at any other system on the network and run an X Window application. Thus, we can claim that X is platform independent. X enables a display and keyboard attached to one system to use programs (that is, clients) running on a completely different system—even a completely different type of system.

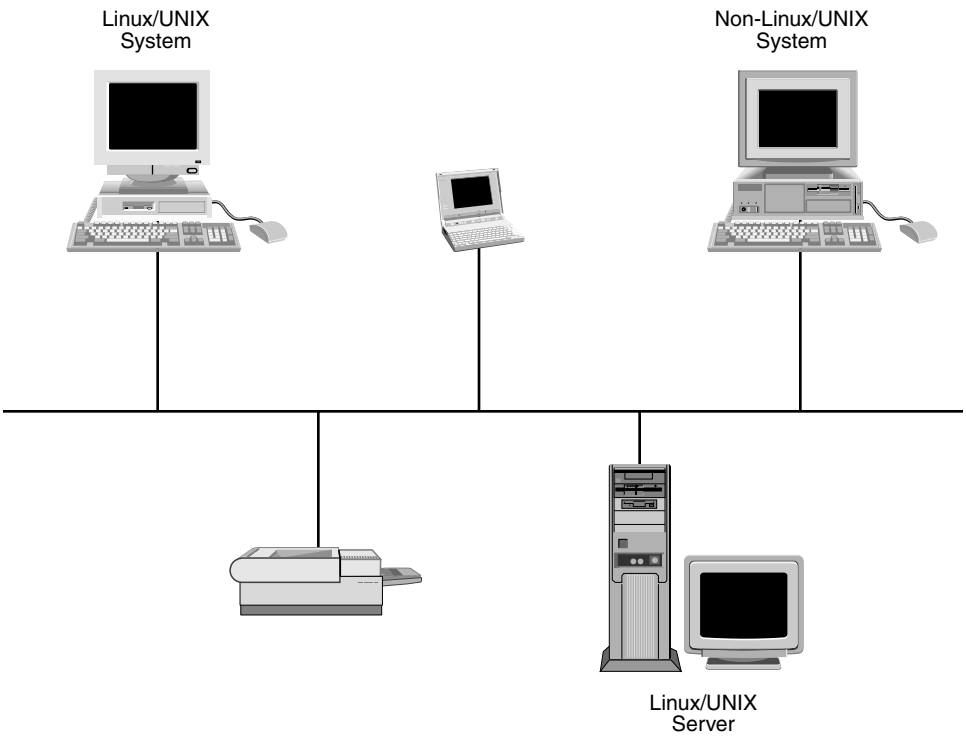


Figure 14.1 A simplified X Window System network.

X Window System functions are split into terminal and application support. Typically, the application support runs on a Linux/UNIX system. The terminal support can run on the same Linux/UNIX system, on a remote Linux/UNIX system, or even on a non-Linux/UNIX system. An example of the last two systems is one that might have a slow CPU of its own or might have no hard disk drive. The client applications would run on a remote, fast central server while only the terminal-supporting X Window server would run on the underpowered terminal. For this reason, we refer to the X Window System as a networking window system. Any connections between the client and the server are TCP/IP protocol-based connections.

Different Client/Server Environment Concepts

For those who do not customarily operate in UNIX X environments, the concepts of client and server are generally understood to be the following:

- A server is a system that provides resources such as applications, hard drive storage, modems, printers, and so on to other systems across one or more networks.
- A client is a system that uses those resources, accessing them across one or more networks. In the Linux/UNIX environment, however, the client and server concepts are different.

In the world of the Linux/UNIX X Window system, however, a *client* is defined as a device-independent program that runs on your local machine or on a remote machine and requires input from the user and the capability to display or otherwise provide output to the user. In other words, the client is the system or machine providing the application support.

Common applications, such as `xcalc` and `xclock` as well as the more complicated database or simulation applications, are clients to a Linux/UNIX X system.

On the other hand, a *server* is a device-dependent program that runs on your local machine and controls the input devices (for example, keyboard and mouse) as well as the terminal display and other output devices. We say that the server is the system or machine providing the terminal support. In many cases—especially in a small network or home environment—the client and the server run on the same system. Even so, they still communicate with each other by using the TCP/IP protocol.

X Client Features

A user interacts with X clients when he or she interacts with applications (such as `xclock`, `xcalc`, and `xterm`) that might be running on either a local machine or on a remote machine. The applications themselves are the X clients that are invoked and run by the users in an X window system.

X clients can be started from the command line by special startup files or from other X clients. For example, you can start window managers manually or invoke them automatically when your machine boots up. To access remote machines, you might need special applications invoked on a local machine (for example, WRQ Inc.'s product, ReflectionXÆ, enables Windows machines to access UNIX machines. Their Web site, if you are interested, is www.wrq.com.

Most X clients share the same or similar options, such as foreground color, background color, display name, fonts, and window geometry. For example, invoke `xterm ?` on the command line. Examine the specifications and options. Then, invoke `xclock ?` and compare those specifica-

Table 14.1 Selected X Client Options

OPTION	DESCRIPTION
`-bg <color>`	Background color of the window
`-fg <color>`	Foreground color of the window
`-bw <pixels>`	Window border width in pixels
`-fn <font_name>`	Font set for the standard text size in the window
`-geometry geom`	Geometry of the window (such as 80x125+0+0)
`-rvc`	Reverse the default image on the screen (that is, if the screen normally appears with black text on a white background, this option changes the appearance to white text on a black background)

tions and options. No doubt, `xterm` has more specifications and options, but some will be identical to `xclock`'s. Some standard options are listed in Table 14.1.

X Server Features

XFree86 is the version of the X11R6 X Window System software that has been converted for use on Intel-based platforms. XFree86, then, is the basis for the X Server. XFree86 has to be configured for each system it runs on, which you can accomplish by specifying certain system attributes in a file called *XF86Config*, which is found in the */etc/X11* (the file itself) or */usr/X11R6/lib/X11* (some examples) directories.

When XFree86 is invoked (either at the command line by the user typing the `startx` command or automatically upon system bootup), it reads the *XF86Config* file and follows the instructions found therein. Customarily, those instructions tell XFree86 how and when to:

- Control and route the keyboard, mouse, graphics pad stylus, or other such input to the correct clients
- Perform basic graphic operations on one or more screens
- Control video-related attributes such as colors, color depth, screen size, fonts, modelines, vertical sync ranges, horizontal sync, video chipsets, video RAM, and clockchips

- Enable simultaneous access by several clients
- Load special software modules
- Control special system actions (such as core dumps, video mode switching, and input device configuration)

The actual installation of the X Window System and the development of the *XF86Config* file are beyond the scope of this book. Consult your sources for this information.

Only after XFree86 has invoked the other server-style software can you say that the full combination of XFree86 and any other necessary software has become your X Server, ready to enable you to use the full features of the X Window System. You might occasionally hear others refer to video servers—such as Mach32, Mach64, or S3—as X servers. Such references occur probably because configuration of the video attributes is by far the most involved and occasionally perilous (for your monitor, if not done correctly) part of X Window System installation and configuration. But the full X server has to control much more than video output.

The X Window System provides the basis for your graphical interface. But the interface itself—the look, feel, and personality of X—is provided by a client application called a window manager.

X Window Managers

New users often believe that X and its various window managers are synonymous, but this situation is not the case. X is in charge of all communication between and among windows, applications, and input and output devices, whether those facilities are located on the local machine or are distributed across a local or *wide-area network* (WAN). On the other hand, the window manager handles local operations such as the movement, resizing, or iconifying of windows. As we stated previously, the window manager provides the look and feel of your X desktop.

Thus, a window manager is an X client, albeit a special one. It is the only one that has no windows itself unless you count the window or menu that you summon when you left-click the empty desktop. But the window manager can move, resize, iconify (hide), de-iconify (restore), map, and unmap windows. Your distribution or other version of Linux might provide several window managers (check your installation information or scout around your file system with `whereis` or with a similar command).

A user can switch from one to another in a given session but can use only one at a time.

Without a window manager, you can still invoke an X session and display windows, but you cannot move or resize them. In addition, if one window completely overlays another, you can access only the one on top. If part of a window is hidden by another window, you cannot access the hidden part of that window.

Several window managers are listed in Table 14.2. For more examples of window managers and desktop environments, or for the latest ones, check the www.plig.org/xwinman Web site. There, you will also find feature comparison charts and resource requirements (if applicable). Plus, there is a lot of good basic X information.

The first six window managers listed in Table 14.2 are still called "window managers." The last three are commonly called "desktop environments" or "desktops" for short. The desktop environments have received the most development over the past few years. Now, they are considered (and are) more complete interfaces. They have a wider range of better-integrated utilities (witness the fact that in Chapter 5, "Using Files in Linux," we actually use KDE to set up printing) and more (and more entertaining, too) applications available. They have become the number one attraction to the "at-home" Linux newbies.

X Window Fundamentals

This section introduces the most basic aspects of the X Window system that a novice user wants to know. Just like learning to ride a bicycle or drive a car, the first things a new user wants to know are: "How do I get it going?" followed by "How do I stop it?" The only extra topic we add is "How do I get it to start automatically?"

Starting X Manually

No matter what distribution of Linux you purchased or downloaded from the Internet, you probably realized that by default, it does not invoke X automatically when it boots up. You are probably left staring at the command-line login prompt. In that case, you have to start it manually.

When you have completed the login procedure, you will be faced with your user or root prompt (`$` or `#`, respectively). At the prompt, type `startx`. The screen goes blank briefly as your X session is initiated, and the window manager appears in front of you.

Table 14.2 Window Managers

WINDOW MANAGER	DESCRIPTION
`twm`	Tab window manager, the classic MIT window manager; included with the standard XFree86 distribution.
`olvwm`	Open Look Virtual Window Manager, a more advanced window manager; used by SunOS and Solaris UNIXes.
`fvwm`	Popular window manager; small, requires less than half the memory used by `twm`; greatly customizable.
`mwm`	Motif Window Manager; basis for CDE.
`Enlightenment`	Popular and well written; originally based on `fvwm`, but recent versions are written from scratch.
`AfterStep`	Descendant of `fvwm`; has floating window for application buttons, icons, and so on, and some animation.
`GNOME`	GNU Network Object Model Environment (pronounced "guh-nome") developed as part of the GNU project; complete user-friendly desktop comprised of utilities and applications. GNOME and related applications are free because of their availability under the GNU Public License (GPL). Described as the future of the graphical X desktop.
`CDE`	Common Desktop Environment, a commercially developed standard desktop/window manager for most versions of UNIX.
`KDE`	K Desktop Environment, a freeware project designed to be similar to CDE but developed and released under the GNU General Public License. KDE emphasizes international support and a standardized appearance and performance with many applications. KDE was formerly available only as a download from the Internet, but due to recent resolutions of copyright issues, some Linux distributions now include KDE. In fact, some now use KDE as their default window manager.

Exit X

You have two ways to exit X inside a window manager: `AfterStep` or `fvwm`. Using the first method, you only need to move the cursor arrow to a

blank area of your desktop (that is, an area without a window) and left-click. You are presented with a root menu. You then scroll down to a selection similar to *Exit <window manager>* and hold the mouse in that position or even left-click. A second part of the menu might appear (depending on the window manager). It might ask you whether your intention is to *Really Quit <window manager name>?*. If this situation happens and it is still your intention to quit, select the *Yes, Really Quit* option. After you have chosen to quit, you are returned to a command-line prompt. Using this method, you will be able to gracefully exit X, which involves stopping all relevant applications and processes in order.

The second method simply involves entering the Ctrl-Alt-Backspace key sequence. Almost immediately, you are returned to a command-line prompt. Alternatively, you could enter Ctrl-Alt-F1, followed by Ctrl-C. But because the two key sequences achieve the same objective as the single-step sequence, you might as well use Ctrl-Alt-Backspace. The single-step method is pretty brutal, though. It stops the X server, but then all applications die ungracefully because they simply lose their connections. If you do not want the single-step sequence used on your machine, you can disable it in the */etc/X11/XF86Config* file.

Meanwhile, if you are in a Desktop environment such as KDE or GNOME, you usually have a Start-like button in the lower-left corner. If that is the case, all you need to do is click it and select exit or logout or something similar. It is pretty intuitive (now watch me get lost looking for it next time).

Start X Automatically

Before you even think of instituting the following procedure for starting X automatically, you must ensure that your X configuration works properly. Otherwise, you might have trouble logging into your system. To carry out this procedure, you have to be the root user.

Test xdm with the nodaemon Argument

The `xdm` program (X Display Manager) can control one or more X display sessions on one or more servers, local and remote. This application presents you with what is called an `xlogin` widget (basically a graphical prompt for a username and password) and then invokes the X Window System after you have been authenticated. You can customize `xdm` with the

use of a configuration file, customarily called *xdm-config* (but that is beyond the scope of this book). All you have to do is ensure that `xdm` works correctly, because it is used in the process of automatically starting X from bootup.

To test `xdm`, type the following at the command prompt:

```
# xdm -nodaemon<Enter>
```

The `nodaemon` option prevents `xdm` from following its normal procedure, which is to put its daemon beyond the control of the terminal. Thus, you are not preventing the daemon from being used by `xdm` but are rather keeping it under some control.

If all goes well, `xdm` presents you with its `xlogin` widget. You type `root` as the login name, press Enter, type root's password, and press Enter again. The `xdm` program then creates the X Window environment with your chosen window manager. Although we have focused on `xdm`, it is not the only display manager. Other display managers such as the Gnome Display Manager and the KDE Manager are also available and have many features that can help you manage your environment.

If you got the results described, `xdm` works for you. It is safe now to move on to the next steps. But first, exit from the window manager by using the root menu or start button. The `xdm` program takes you back to the `xlogin` widget. To get back to the command line, use the key sequence Ctrl-Alt-F1. That takes you back nearly to a command prompt, but you will not be at a prompt per se. You should see a cursor blinking on the line immediately below the `xdm -nodaemon` command. Now, press Ctrl-C to terminate the `xdm` daemon and be left at the command prompt.

Edit the /etc/inittab File to Run Level 5

Using `vi` as your text editor, you will modify the Linux initialization table found in the */etc/inittab* file, which controls (among other things) the run level (in other words, the number and type of services provided by Linux when it boots up). Be very careful. Browse down through */etc/inittab* until you see the line `id:3:initdefault`, and change the `3` to a `5`. Do not change anything else. Save and exit */etc/inittab*. To Linux, running level `5` means that it should no longer boot to full multiuser mode as it did with running level 3. It should now boot to X11 mode.

Reboot the System

At this juncture, all you have to do is reboot the machine. When Linux boots again, the command-line login prompt does not appear. Instead, you are presented with the graphical `xlogin` widget.

Basic X Window Components

The concepts introduced in this section should be easily grasped, especially since almost all of us have used a graphical user interface (GUI) before. However, even though the discussion may seem almost trivial to some, the terms and concepts as defined here are worth remembering. They are common to all GUIs, and provide a foundation for understanding and comparison not only among X Window managers, but also between the X Window system and other GUIs.

Display

Figure 14.2 illustrates what your X Window System *display* might look like when using one of the less-complex window managers. All window managers do not look like this one, however; some have more or fewer applications showing different backgrounds and so on, depending on the preferences of the system administrator or the person who installed X. Moreover, ordinary users will have some ability to add or delete applications or other information.

The basic components of the display are described next:

Root window. Also called the *desktop*, this area fills the entire screen. You can move your mouse to an open part of this area (that is, an area not covered by other windows or other displays) and then left-click to access the root menu, a menu that helps you start other windows or otherwise customize your environment. A sample root menu appears in the bottom-right corner of Figure 14.2. Note that we have chosen a generic name (`winmgr`) for the window manager. Now, if you right-click an open area of the root window, you automatically invoke the Programs menu, which is the same menu you would get if you accessed a root menu and then clicked `Programs` (because of space constraints, we did not show the

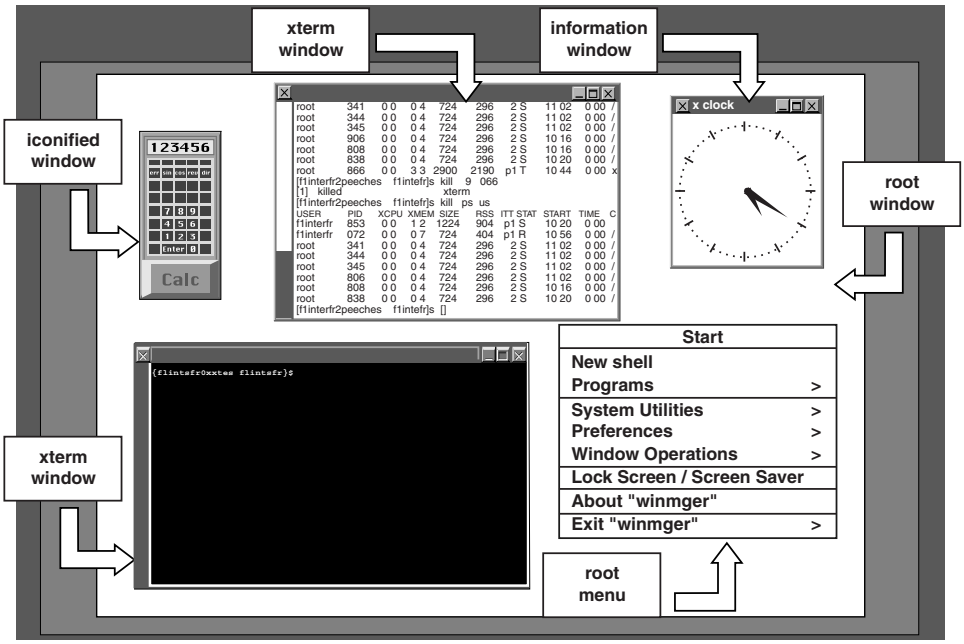


Figure 14.2 An example of an X Window display.

Programs menu in Figure 14.2). Thus, right-clicking to invoke the Programs menu is just a shortcut.

Information window. The `xclock` window is an example of an application window that displays information for the user's benefit.

`xterm` **window.** An `xterm` window is a terminal emulation window that has been opened for any number of purposes. Figure 14.2 shows two terminal windows open, one for each process.

Iconified window. In Figure 14.2, the calculator symbol (with `Calc` across the bottom) is not a window but a representation of a window. The represented process has not been terminated.

root Menu

As we mentioned, when you move the mouse pointer to an open area of the root window and left-click, the `root menu` appears. The menu options are described in Table 14.3. You can either move the mouse pointer over one or more options or left-click to select a specific option.

Table 14.3 root Menu Options

OPTION	DESCRIPTION
New shell	Create another `xterm` window.
Programs	Types of programs available, as well as some specific programs.
System Utilities	Types of utilities available.
Preferences	Types of changes you can make to your environment.
Window Operations	Types of available window operations.
Lock Screen/Screen Saver	Invoke the `xlock` program.
About "winmgr"	Invoke a menu with information about your host machine, the window manager you are using, and your X environment.
Exit "winmgr"	Restart your X session; switch from your present window manager to another; cancel the quit/exit from X; or verify your quit/exit from X.

Mouse Pointers, Input Focus, and Location Cursors

Linux can work with either a two-button or a three-button mouse, but the documentation with most distributions recommends that you use a three-button mouse or invoke three-button mouse emulation (using both buttons at once simulates the use of the middle button) during Linux installation. Meanwhile, moving the mouse causes the movement of a small icon, commonly called a *pointer* or a *cursor*, on the terminal screen. There are several types of pointers.

When used on an X display, the mouse and its pointer can be used to select whatever window you would like to *activate* (making the window capable of receiving input). Note that when you activate a window, its frame changes color (it becomes highlighted). Furthermore, when you direct some input information or data to that highlighted active window, you are *focusing*. There are two types of focusing: explicit focusing, where you move the pointer to a window and left-click the mouse to activate the window, and pointer focusing, where you only have to move the pointer to a window to activate it. Most window managers use explicit focusing.

When you place the pointer in or on windows or other objects, pressing the left button activates the window or object. Pressing the right button or pressing the middle button—or, on a two-button mouse, pressing both the left and right buttons simultaneously—will also cause certain actions to take place, such as the appearance of a menu from which you can select further actions.

In X, the *location cursor* is similar to the cursor on the command line of your character-based screen. Its location determines where to insert your keyboard input on a terminal emulation screen. The location cursor is not activated until you activate the terminal window first, however.

Window Frame

The frame around a window is often taken for granted. Although we generally think of the window contents and frame as one and the same, they are not. Strictly speaking, the *frame* is provided by the window manager application and enables a certain amount of window manipulation without affecting the operation of the application running in the window itself. That window manipulation, however, requires the proper placement and operation of the mouse. The basic components of the window frame might vary slightly from one window manager application to another (for example, one might have an exit button at the far right end of the title bar, but another might not).

The first component of the window frame is the *title bar,* which runs across the top of the window between the window operations button on the left and the minimize and maximize buttons on the right. Predictably, the title bar contains the title of the window. For instance, the window emulating a terminal screen would be titled `xterm`. You can use the title bar to move the window from place to place on the root menu by the drag-and-drop method.

NOTE **The term "drag and drop" refers to the following actions: positioning the mouse cursor over something (such as a title bar or menu option) and pressing but not releasing the left mouse button; then moving the mouse cursor to another location (dragging) and releasing the left mouse button (dropping).**

You can reduce the window to its icon representation (its minimum size) by left-clicking the *minimize button*, which is immediately to the right of the title bar. If the window is smaller than its maximum size, you can maximize it by left-clicking the *maximize button* next to the minimize button (maximum size usually means that the window will fill the entire root

window). Some window managers also provide an `exit` or `close` button, which usually looks like a square with an X in its center, in the upper-right corner of the window frame. Left-clicking that button will close the window, the same as left-clicking the window operations button and selecting the `close/exit` option.

Speaking of resizing the window, there are several ways of making the window larger or smaller. All require proper placement of the mouse pointer. Note that if you place the pointer exactly in the corners of the window frame, the pointer changes to resemble something like an arrow pushing against the corner. By dragging and dropping at that point, you can alter the window's dimensions in two directions (height and width) simultaneously, if you want.

If you just place the mouse on any of the four borders (top, bottom, left, or right) of the window frame at any one time, the pointer again changes, but this time it changes to something resembling an arrow pushing against that border only. Now you can drag and drop again, but you can change only one window dimension at a time.

If you move the mouse pointer to the upper-left corner of the window frame and left-click on what is called the *window operations button,* you will activate a drop-down menu containing several commands. You can move the pointer down the menu and select the operations of your choice.

Icons

You probably already know about the benefits of icons on your desktop. They are a handy way of multitasking without repeatedly having to invoke applications, and they are a handy way to manage your root window (that is, desktop) space. When an application window has been changed to an *icon*, the process remains invoked and ready, but the screen clutter is reduced.

You can create an icon from an active application in two ways. One is by left-clicking the window operations button in the top-left corner of the application window and then scrolling down the resulting drop-down menu to select the minimize (or hide/restore or similar instruction) option. The benefit of using this method stems mainly from having a choice of options presented to you. The result is the same as the second and simpler method, namely left-clicking the minimize button on the window frame.

Depending on the window manager used, the resulting icons for identical applications might be different in appearance. For example, minimizing the calculator window in `fvwm95` results in an icon that looks like a button bar. Minimizing the same window in `lesstif` results in an icon

that looks like a reduced-scale calculator. No matter which window manager you use, the resulting icons are actually bitmap images.

xterm Fundamentals

An `xterm` window is an X client application that simulates a common video terminal, such as a DEC vt100. By now, you have undoubtedly noticed that you can create and use `xterm` windows (commonly called `nxterm`, `AIXterm`, or something similar, depending on the version of UNIX) while in X to enter Linux/UNIX commands in the same manner you entered them at the command line when X was not invoked.

Create an xterm Window

You can create additional windows in X in two ways. First, from an already opened `xterm` window, you can type `xterm &`. Alternatively, you can move the mouse pointer to an open area of the desktop (that is, the area not already occupied by a window) and left-click the mouse. Then, from the resulting root menu, left-click New shell, which is the first selection under Start at the top of the root menu.

Copy Text

The `xterm` window enables you to copy and paste text to another part of the window or to another window. Place the mouse pointer at the first letter of the text you want to copy, and then left-click and drag over all the text you want to copy. The text is highlighted. When you release the mouse button, the highlighted text is copied into a buffer. Then, you move the mouse pointer to a position in the same window or in another `xterm` window. If you are using a three-button mouse, press the center button to copy the text. If you are using a two-button mouse, push both buttons at once.

Create a Scrollbar

If you want to create a window with a scrollbar in it, first create another preliminary window. In that preliminary window, type one of the following command sequences:

```
$ xterm -sb leftbar<Enter>
```

or

```
$ xterm -sb rightbar <Enter>
```

Each of these sequences creates a new window with a scrollbar on the side that you specify.

If you want to create a scrollbar in an existing window, place the mouse pointer inside that window, hold down the Ctrl key, and press the center button on the mouse (on a three-button mouse) or press both mouse buttons at once (on a two-button mouse). The VT Options menu is displayed on the screen. The first, or one of the first, selections on the menu is `Enable scrollbar`. Click that selection, and a scrollbar appears in your window.

To use the scrollbar, place the mouse pointer in the scrollbar itself and then press the left mouse button to move up or press the right mouse button to move down. If you want to remove the scrollbar, redo the instruction and left-click `Enable scrollbar` to remove the check mark next to that selection.

Use the `man` pages, the help options, and other information sources to see what else you can do with the `xterm` command.

Close an xterm Window

To close an `xterm` window, you have several alternatives. At the prompt in the window, press Ctrl-D or type `exit` and press Enter. Double-click the Window-Ops button in the upper-left corner of the window or left-click that same Window-Ops button and then left-click `Close` on the resulting drop-down menu.

Customize xterm

We have already discussed a couple of `xterm` options:

- Running the second `xterm` window in the background so that you can move back and forth from window to window (the `&` option)
- Adding a scrollbar to the new `xterm` window (using the `-sb` option with either the `-leftbar` or `-rightbar` option)

In this section, we present a few more of the dozens of options available to you with the `xterm` command. Combining these options gives you perhaps thousands of possibilities for new `xterm` windows. Take a look at Example 14.1 and Table 14.4 to see how `xterm` can work for you.

Table 14.4 Selected xterm Options

OPTION	EXPLANATION
`-bg color`	Background color of the window
`-cr color`	Cursor color. The default color is the foreground color.
`-display Name:Number`	Host server name and X server display number where the command will run. If not specified, the client program gets the hostname and displays the number from the DISPLAY environment variable.
`-e command`	Command to be executed in the window. This flag executes a command; it does not start a shell. The command and its arguments (if any) must appear last on the `xterm` command line.
`-fg color`	Foreground color of the window
`-fn font`	Font to be used for all normal-sized text
`-fullcursor`	Full height (not the underscore style) text cursor
`-geometry`	Geometry location and dimensions of the window. The default is 80x25+0+0; that is, 80 pixels wide, 25 pixels long, located in the top-left corner (that is, at the 0,0 position) of the screen.
`-help`	Available option flags
`-i`	Display a newly created window as an icon, not a window.
`-keywords`	List the `xdefaults` keywords.
`-l`	Append output from the `xterm` command to the login.
`-lf file`	The location where the output is saved. The default file is `xterm.logxxxxx`, where `xxxxx` is the `PID` of the `xterm` command.
`-n` *`icon name`*	Icon name used by the `xterm` command
`-name` *`application`*	The application name to use for .Xdefaults
`-sb`	Scrollbar. `-leftbar` places the scrollbar on the left.
`-sl`	Maximum number of lines to save that are scrolled off the top of a window. The default is 64.
`-T` *`title`*	Name of the window in the title bar. If `-n` is not specified or a name for the icon has not been specified in .Xdefaults, this name is displayed also on the icon of the window.
`-W`	Move the cursor to the middle of the `xterm` window when it is created.
`-xrm string`	Resource string to be used

Example 14.1 Creating an xterm Window

On host machine SYSTEMB, create an xterm window for X server 0. Allow this process to run in the background:

```
$ xterm -display SYSTEMB:0 &<Enter>
```

On this host machine, create an xterm window with a red background and white foreground with Times Roman 10 normal font, 80 pixels wide and 40 pixels long, and positioned in the top-left corner (the 0,0 position). Allow the window to run in the background.

```
$ xterm -bg red -fg white -fn rom10 \<Enter>
> -geometry 80x40+0+0 &<Enter>
```

NOTE **You can apply the options in Table 14.4 to other X applications such as xcalc and xclock. For further information regarding those applications and their options, check the man pages and other information sources.**

Now that you have had a look at xterm windows in general, it is time to look at a special way of using them: to create a special window for the root user.

Additional Basic X Window Commands

This section introduces some features that you may not be familiar with if you are not already a Linux/UNIX user. These features have been available with UNIX for years and set Linux/UNIX apart from "that other GUI operating platform."

Become the root User

When using xterm windows in an X session, you can become the root user in two ways. First, if you are operating in an ordinary xterm window, you can simply enter su - <Enter>. You are prompted immediately with the root Password: prompt, at which point you type the root password. Then you are prompted with the root prompt, #, and you can conduct root business.

Alternatively, left-click an open desktop area, scroll down the root menu, and select System Utilities. Another drop-down menu called System Utilities Button Bar, appears. Scroll down that menu and select

Root Shell. An xterm window appears. At the top is the title Root Window, and within the window is the root Password: prompt. Type the proper password and press Enter, and you are given the root # prompt.

Let Another User Run a Client on Your System

As we discussed earlier, the X Window System uses the client/server model. Remember, the client is an application (such as xterm or xcalc), and the server is the program that supports the application by controlling input and output. In many cases, the client and server both run on the same system. The last topic in this chapter is xhost, which you must invoke before any of the clients will connect and work.

With X, however, you can run a client on your own system on the network but display the application on another (presumably remote) terminal screen. This arrangement gives others access to software programs running on your system. Others can therefore enter commands in the window and use their mouse, despite the fact that the actual process is running on your system.

Assume that you are trying to run a client application on the sys1 machine, but you want to display the output on the sys2 server. The application is simply xterm (that is, you want to open an xterm window), but it could be any X Window system application begun through the character cell. Example 14.2 shows three possible ways to invoke xterm.

To run the client remotely and display its results locally, you must tell the client process where to display its window. X applications use the value of the DISPLAY environment variable to indicate the TCP/IP host name of the server (that is, the name of the system on which the client should display its output).

If the DISPLAY value has not been set, you must include the -display option in the command to invoke the client application. If the DISPLAY value is set, you can override it by using the -display option and the specified server name when starting the client. Typically, the display value is set to something like :0.0 for the local server or sys2:0.0 to tell the client to display its output on the remote server called sys2. The part before the colon specifies the server's TCP/IP host name or IP address. The part after the colon specifies the server number and display number, but the display number is optional. Unless you run multiple servers on one machine or have multiple displays controlled by a single server, set these values to 0. The TCP/IP protocol must be installed and configured on the network.

Example 14.2 Invoke xterm on Another System

Option 1: In an `xterm` window on `sys1`, redefine the `DISPLAY` variable as `sys2:0.0`. You can then invoke the application, and the shell automatically sends the output wherever that environment variable tells it to go:

```
$ DISPLAY=sys2:0.0<Enter>
$ xterm &<Enter>
```

Option 2: In an `xterm` window on `sys1`, enter

```
$ export DISPLAY=sys2:0.0; xterm &<Enter>
```

Option 3: In an `xterm` window on `sys1`, enter

```
$ xterm -display sys2:0.0 &<Enter>
```

The `export` command is a TCP/IP command that sends another command to a specified remote system for execution.

Run a Client on Someone Else's System

Suppose you want to be able to use applications on someone else's system. To do that, open an `xterm` window, activate it, and enter the following text on the command line:

```
$ rlogin -l username remote_hostname<Enter>
```

You must use a username that has already been created on the remote host. `rlogin` is one of the remote-host execute commands, commonly called `r` commands. The `-l` (the letter "ell," not a number one) option basically clears the way for you to specify the username. The remote host name is the TCP/IP name of the system you are trying to log into. (The `r` commands, which we do not cover in this book, are introduced here only to illustrate how to accomplish this simplified client/server application execution.)

The shell responds with a password prompt. Simply fill it in and press Enter, and you are authenticated to the remote system. Now, use one of the two following command sequences to invoke an application on the remote host and display it on your system:

```
$ xterm -display sys1:0.0 &<Enter>
```

or

```
$ export DISPLAY=sys1:0.0; xterm &<Enter>
```

To accomplish the procedures described in this section and the preceding one, you have to give the systems the capability to allow, restrict, or limit other remote hosts from accessing their displays or applications. To do so, the `xhost` command must be executed on the server(s).

xhost Command

As mentioned, the X Window System enables users at one host system to run a client on another system. But there might be times, for security or other reasons, when it is not desirable to allow a particular user or system to connect to another. One way to deny access to a user or a system is to use the `xhost` command on the system to which access is sought.

At the target machine, the user invokes a terminal emulation screen and enters

```
$ xhost [ + | - ] [name]<Enter>
```

The square brackets indicate that the plus and minus signs are optional, as are the usernames or host names. The examples in Table 14.5 should help to clarify these concepts. If you need further information, consult your information sources. Note that using only `xhost +` will open the machine in question to everyone on the network, and if applicable, possibly to everyone on the Internet. Consequently, if your machine is exposed to a large network or to the Internet, you should install security on the

Table 14.5 xhost Syntax Options

SYNTAX	EXPLANATION
`xhost`	Ask for a message: Is access control enabled or not?
`xhost +`	Turn off access control; grant access to everyone.
`xhost -`	Turn on access control; grant access only to those on the list.
`xhost + name`	Add `name` to the list of users/hosts allowed to connect to this X server (the `+` is optional here).
`xhost - name`	Delete `name` from the list of users/hosts allowed to connect to this X server.

machine. Although `xhost` might not be a comprehensive answer, it can be a first step.

After you use `xhost` to invoke access control to an X server machine, a file similar to `X*.hosts` is created in the `/etc` directory. That file contains the names of the host machines and users who can access the X server machine in question.

Exercises

1. Start the X Window system environment with the `startx` command.

   ```
   $ startx<Enter>
   ```

   What window manager is invoked by default? Are any windows displayed by default? If so, what are they called?

2. If an `xterm` window appears, verify that it is activated or activate it if it is not. If there is no `xterm` window, create one.
   - If an `xterm` window has been created already, check to see whether the window's title bar is highlighted in blue. If it is not, move your mouse pointer to the `xterm` window and left-click. The window frame changes color to indicate that the window is now activated.
   - If there is no `xterm` window yet, you might have to left-click an open space in the root window or press an equivalent start button and then select `New shell`, `xterm`, or a similar item from a root menu. Each distribution of Linux, and each window manager within each distribution, has a different but similar way of conjuring up these `xterm`-like windows.

3. Using the `xterm` window, try some commands such as `ls` (with or without other arguments), `date`, `cal`, and `whoami`.

4. Resize the width of the window.

   Move your mouse pointer to the right edge of the window frame. Note how the pointer changes shape (for example, from a single-headed arrow or I-beam to a double-headed arrow, or from a single-headed arrow to an arrow pushing against a perpendicular line), indicating that the mouse can now be used to change one dimension

of the window (the width or the height, depending on which edge the cursor sits). Press and hold the left mouse button and move the mouse to alter the window's dimensions. An outline might appear, indicating the new window size. Also, a small feedback indicator might appear, which tells you the new dimensions in pixels or in lines and columns. An indication of the location of the upper-left corner of the window might also appear. As we stated previously, each window manager has its own idiosyncrasies. When the window is the desired size, simply release the left mouse button.

If you saw the small feedback indicator, note that it contains the same kind of information you might use if you were to create a new `xterm` window from the command line of an existing `xterm` window, using the `xterm` command with `-geometry` as an argument followed by the dimensions and location.

5. Change the height and width of the window simultaneously.

   Move your mouse pointer to a corner of the window frame. Note how this time the pointer again changes shape (for example, to an arrow pushing against a corner), indicating that the mouse can now be used to change both window dimensions (width and height) at once. Press and hold the left mouse button and move the mouse to alter the window dimensions. You will get the same kind of response feedback you received in Exercise 4. Again, when the window is the desired size, simply release the left mouse button.

6. Drag and drop the `xterm` window from one side of the root window to the other.

   Move your mouse pointer to the title bar of the window frame. Note how this time the pointer again changes shape (for example, to a large dot), indicating that you can now use the mouse to move the window. Press and hold the left mouse button and move the mouse to alter the window's location. You get the same type of response feedback you received in Exercises 4 and 5, except you will not be altering the window size. When the window reaches the desired location, release the left mouse button.

7. Now, use the options in the window operations (button) menu to move and resize the window.

   In the `xterm` window, left-click the window operations button (that is, the button to the left of the title bar) on the window frame. The

window operations menu appears. Left-click the `Move` command and you will notice a similar feedback to the response in Exercise 6, except that you do not have to hold down the left button as you move the window. When the window is in the desired position, simply press the left button and the window will stay where it is.

Now, left-click the window operations button. From the window operations menu, left-click the `Resize` command. You will notice the same type of feedback as in Exercises 4 and 5, except that you do not have to hold down the left button as you alter the window's dimensions. When the window is the desired size, simply press the left button and the window remains that size.

NOTE Instead of left-clicking to hold the window in the desired position or to maintain the new dimensions, depending on which function you are exercising, you can also press Enter.

8. Take a close look at the window operations menu again. Are there any items on it that appear grayed or dimmed? If so, why?

   On the `xterm` window, left-click the window operations button (that is, the button to the left of the title bar) on the window frame again. The window operations menu appears.

9. Here is something you might not have tried. On the window operations menu, type the letter `m` instead of clicking `Move` and then use the arrow keys. This method is another way to move your window. Now, try the underlined letters for some of the other functions.

   On the `xterm` window, left-click the window operations button and the window operations menu appears. This time, instead of left-clicking the `Move` command, just type `m` on the keyboard. You will notice a similar feedback to the response in Exercise 7, and you will not have to hold the left button down as you move the window around. When the window is in the desired position, simply press the left mouse button or press Enter and the window stays where it is.

10. With the window operations menu closed, try some of the underlined letters. Do they work?

    Activate the `xterm` window. Then, without activating the window operations menu, try using the underlined letters you noted when that menu was open.

11. Iconify (that is, minimize) the `xterm` window. After it is an icon, restore it back to the root window.

    Activate the `xterm` window. Three buttons appear to the right of the title bar. Left-click the iconify (or, if you prefer, the minimize) button, which is immediately to the right of the title bar. The `xterm` window apparently disappears, but an icon remains. The icon indicates that the process is still available for use; you need only reactivate it.

    Left-click the `xterm` icon. The icon disappears and the full-fledged `xterm` window reappears.

12. Now, maximize the `xterm` window. What happens?

    Look again at the three buttons to the right of the `xterm` window's title bar. Left-click the middle button, called the maximize (or, if you prefer, the restore) button.

13. After you maximize the `xterm` window, resize it to a smaller size.

    Note that three buttons appear to the right of maximized `xterm` window's title bar. Left-click the middle button again.

14. Use the root menu to open another `xterm` window.

    Move the mouse pointer to an open area of the root window (desktop) and click the left mouse button. The root menu appears. Left-click `New shell` or a similar command.

15. Start another different type of clock by using the root menu.

    Move the mouse pointer again to an open area of the desktop and click the right mouse button. The Programs menu appears. Move the mouse pointer down to `System` and left-click. The System submenu appears. Left-click `Time Tool`. A different type of clock appears.

16. The `xterm` command has many command-line options. Try viewing these options by using the `xterm -help` command. You need to pipe the output from the command to `more` or `less`. Note that some distributions of Linux/UNIX do not allow the use of the pipe with the `xterm` command.

    If an `xterm` window appears on the desktop, activate it. Otherwise, create an `xterm` window by left-clicking an open area of the desktop and selecting `New shell` or a similar command from the resulting root menu.

At the command line, enter

```
$ xterm -help | less<Enter>
```

17. Start another `xterm` window from the command line within an `xterm` window. Give the new window the following characteristics:
    - Background color: sky blue
    - Foreground color: green
    - Title: My New Window
    - Scrollbar: left side

```
$ xterm -bg skyblue -fg green -T "My New Window" -sb -leftbar<Enter>
```

18. Now, start an `xclock` from the command line within one of the windows. Have it run in the background. Give the clock the following characteristics:
    - Background color: white
    - Foreground color: red
    - Hands on the dial: blue
    - Second hand update: every second

```
$ xclock -bg white -fg red -hd blue -update 1 &<Enter>
```

19. For the next few exercises, make sure that you have created at least two `xterm` windows. (If you create an `xterm` window from another `xterm` window, make sure you add the `&` argument to the `xterm` command so that the second window runs in the background of the first and you can move easily from one window to the other.) In one `xterm` window, use the `vi` editor to create a file called *ex14file*. Add a few lines of text to this file, but do not exit from it:

```
$ vi ex14file<Enter>
i
Hi, how are you?
I am fine.
Today is not Friday.
I will transplant this line to ex14file.new.
I'll copy this line to ex14file.new, too.
<Esc>
:wq<Enter>
```

20. In the second `xterm` window, use the `vi` editor to create another file called *ex14file.new*. Go into Insert mode, but do not add any text to the file yet.

```
$ vi ex14file.new<Enter>
i
```

21. Copy a few lines of text from the *ex14file* in the first `xterm` window to the *ex14file.new* in the second window. Then, exit `vi` in both windows.
    - Move the mouse pointer back to the window in which you entered text in the file *ex14file*.
    - Place the mouse pointer at the beginning of the lines you want to move to *ex14file.new*.
    - Press the left mouse button and hold it while you move the mouse pointer across the lines you want to copy.
    - When you reach the end of the text you want to copy, release the left button.
    - Move the mouse pointer to the window with *ex14file.new* already begun and then point in the file where you want to insert the lines you are copying from *ex14file*.
    - If you have a three-button mouse, press the middle button to insert the text lines. If you have a two-button mouse, press both buttons at once to insert the lines.
    - Exit `vi` in both windows.

22. You have now completed the single-machine part of this exercise. Your choices now are to try the optional network-based exercises (proceed to Exercise 23) or to end your X session. If you choose to end your session, take the following actions:
    - Go to an open area of the desktop and left-click.
    - Select `Exit winmgr`.
    - From the resulting Really Quit winmgr submenu, select `Yes, Really Quit`.

23. As the root user, create a user named `remote1` on your client system and give that user the `remote1` password. Revert to your normal user identity and enter `xhost +<Enter>` in an `xterm` window to

enable all other users access to your X server. The shell should reply with a message such as `access control disabled, clients can connect from any host`.

```
# useradd remote1<Enter>
# passwd remote1<Enter>
Changing password for user remote1
New UNIX password: remote1
Re-type new UNIX password: remote1<Enter>
# logout<Enter>
login: teamxx<Enter>
Password: teamxx<Enter>
$ startx<Enter>
```

If an `xterm` window is created by default, activate it by left-clicking the mouse pointer on it. Then, at the command line, enter

```
$ xhost +<Enter>
```

24. Check to see whether the `DISPLAY` variable has been set.

```
$ echo $DISPLAY<Enter>
```

You have probably noticed that before you can set a value for `DISPLAY`, you need to know the TCP/IP name of your computer. This information can be obtained by typing the command `hostname` with no options or arguments.

```
$ hostname<Enter>
```

If the value for `DISPLAY` is already set to `hostname:0.0`, proceed directly to Exercise 25. If the `DISPLAY` value is not `hostname:0.0`, set it by entering the following:

```
$ DISPLAY=hostname:0.0<Enter>
```

25. After you have ensured that your `DISPLAY` variable is set correctly, stay at your own system but start an `xterm` session on someone else's computer and have it appear on your system. First, you need to know the other system's host name. When specifying a login ID, use `remote1`.

```
$ rlogin -l remote1 other_client_hostname<Enter>
Password: remote1<Enter>
$ export DISPLAY=your_client_hostname:0.0; xterm &<Enter>
```

Voila! A new window appears.

26. In the window you just started on the other system, use `hostname` to verify that the window is running on the remote system. Check the value of the `DISPLAY` variable on that system. It should indicate the name of your (remote) client/host system.

```
$ hostname<Enter>
$ echo $DISPLAY<Enter>
```

27. From the remote system's window, execute `xcalc &`.

```
$ xcalc &<Enter>
```

From which system is the calculator being executed?

You can verify your answer with the `ps` command. When you have completed this exercise, close the remote system's window.

```
$ ps<Enter>
$ exit<Enter>
```

28. Exit from your X Window environment.
    - Go to an open area of the desktop and left-click.
    - Select `Exit winmgr`.
    - If there is a resulting Really Quit winmgr submenu, select `Yes, Really Quit`.

See Appendix B for the answers.

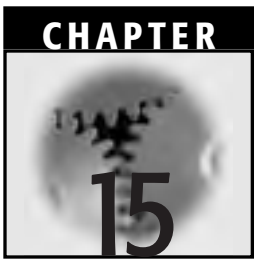
Quiz

1. Which of the following statements are true?
    - Connections between clients and servers in a Linux/UNIX environment can be based on any networking protocol.
    - A window manager is one of the basic components of an X Window System.

- In a Linux/UNIX environment, a server provides resources such as hard disk drives and printers.
- You can use only one window manager at a time.
- You can quit an X session with the Ctrl-Alt-Backspace key sequence, but it is not graceful.

2. Provide a brief definition of an X Server.
3. Which of the following commands is used to run a client on a different system?
   - `xhost`
   - `export`
   - `xterm`
   - `telnet`
   - `fvwm`
4. What do the following `xterm` options mean?
   - `-bg red`
   - `-fg black`
   - `-T ledger`
   - `-geometry 80x125+0+0`
   - `-n lgr`
5. When you are in X, how many ways are there to become a root user? Briefly, what are they?

See Appendix C for the answers.



Linux Documentation and Support

The Linux operating system consists of many subsystems and processes working together simultaneously. Thus, to monitor, measure, modify, and control your Linux system, you need information about commands, utilities, configuration, applications, and more.

In this book, we have introduced you to some basic Linux concepts, commands, utilities, and processes. To fully understand these concepts, however, you might have to refer to or have knowledge of things that are not discussed at length in this book. Because it is difficult to absorb, let alone commit to memory, all of the concepts, commands, and related information you need to make your Linux system perform the way you want now and in the future, it is important to be aware of all the sources of information available to you.

In this chapter, we list and discuss some sources of information and support that you will be able to call upon immediately and later, whether inside your own shop or elsewhere in the Linux world. Although we cannot possibly cover all available information sources, we do provide an introduction to some of the more popular and helpful ones.

NOTE **Throughout the book, when we suggest that you check other information sources, we are referring to the sources mentioned in this chapter.**

Distribution Package Documentation

You can obtain a Linux distribution in two basic ways: officially and unofficially. This section covers the information you can expect to find with both types of sources.

Official Linux Distributions

Every official Linux distribution ships with a large amount of documentation, and that is one reason why we recommend that you purchase an official distribution. Most distribution packages contain the following information, which is specific to the Linux software you have purchased:

Features of the enclosed Linux software and applications. You will learn which version of the Linux kernel you have, the hardware it supports, the applications you can install and use immediately, and what else is available and from what location.

Installation instruction and assistance. You will be told how to install Linux on your system (for example, how to install a graphical versus a text-based interface and options for basic and custom installations) and how to do basic fine-tuning.

Basic system administration instructions. These are important because you have to know how to create and administer the root and other users on your system just to get going.

Sources for troubleshooting and support after you register your software. Troubleshooting and support can be invaluable, especially if you are loading Linux for the first time or are updating to a version that is different from your current version. *Do not forget to register your software as soon as you unwrap or install it.*

A simplified history of UNIX and Linux. This information might aid your comprehension of the attributes and behavior of certain parts of your software, especially the actions of the shell (some of which we have discussed in earlier chapters).

Instructions for obtaining additional documentation. Most distribution materials are geared toward specific audiences (the majority are geared toward novices and non-experts, while some are geared toward business users, some toward programmers, and so on). But most packages contain references to their own hard-copy documentation, their own online documentation, and to other sources of information (books, magazines, Internet sites, and the like).

Downloaded Linux and Other Unofficial Distributions

Purchasing an official Linux distribution is not the only way to go. Some people and organizations acquire their copies of the Linux kernel with or without the simultaneous acquisition of applications by downloading Linux from the Internet or through other means.

If you take this route, you can find documentation in many venues. For example, recent years have seen an avalanche of books on Linux. You can find them at local bookstores, university or college bookstores, computer bookstores, and on the Internet. You can often find related applications or utilities and their respective documentation in the same places.

Many books and magazines also contain CD-ROMs of one of the Linux distributions, although it is usually one or two revisions behind the distribution or kernel package you can obtain directly from the manufacturer. The same book and CD-ROM combinations customarily contain some applications, as well. Meanwhile, new Linux-related magazines appear almost monthly. Later in this chapter, we will list a few of these books and magazines for you.

New Linux-oriented Internet sites crop up all the time. We list several of the more established sites later in the chapter, as well. Often, they are affiliated with educational institutions, Linux support or interest groups, or other organizations that have adopted Linux in their own shops or are otherwise promoting its use. The information and services offered by these sources are varied but almost always include installation instructions, some basic system administration help, troubleshooting advice, device drivers, fine-tuning tricks, and other types of support.

Linux support and user groups have emerged in almost every major urban center or region. Joining one of these groups is an economical and sociable way to learn more about Linux, to obtain free or inexpensive software, to see free demonstrations of hardware and software, and even to do

a good turn by helping others with their problems. To find these groups, check your local computer newspaper (usually free at software and hardware vendors, at major newsstands, and even at some convenience stores), go surfing with a good Internet search engine, or go straight to www.linux.tucows.com and drill down through the links found there (especially the "Software Library" links). Tucows even provides ratings of several Linux flavors.

Meanwhile, other Internet sites will link you to support groups, too (especially Linux Online! at www.linux.org).

Current Linux Distributors

The operative word in the title to this section is *current.* When we began to write this book, there were certainly fewer distributors of Linux than there were at the publication of this book. And by the time you read this book, there will no doubt be even more. If you are interested in investigating the various flavors of Linux, keep track of the distributions listed and described at the Linux Online! Web site at www.linux.org.

At last count, 40 full-featured distributions and 35 mini or specialty distributions were available in the English language, and more than 20 non-English distributions were available in Spanish, Italian, French, Portuguese, German, Chinese, Japanese, Thai, Ukrainian, Russian, Swedish, Finnish, Turkish, Croatian, and Korean. The Linux Online! site provides descriptions of the distributions as well as links to the respective download sites. The site is well worth checking out, even if only to see how Linux's popularity has spread throughout the world in a few short years. At the Web sites for the distributions listed at Linux Online!, you will find some variation of the following types of information and services:

- Company history and description
- Purchase information
- Names and addresses of resellers
- Patches, updates, or bug fixes for the products, including kernel updates
- Announcements regarding security issues pertinent to their products
- Press releases regarding their company or Linux in general
- FAQs (frequently asked questions) regarding their products or Linux in general

- Related or linked Internet sites (such as FTP sites or mirror sites)
- Support contact information (such as telephone and fax numbers and e-mail and surface mail addresses)

Some of these sources do not provide FTP sites to download their products but do provide sources for inexpensive CD-ROM copies.

We cannot leave the topic of Linux Online! without complimenting them on the "Getting Started with Linux" lesson found under the header "Linux 101" on the home page. Bravo! Hopefully there will be more courses like this one to come.

The Linux Documentation Project

The *Linux Documentation Project* (LDP) consists of guides, HOWTOs, FAQs, and projects. Perhaps the best way to describe it is to quote from its manifesto, which you can read at www.linuxdoc.org, the project's Web site. An excerpt follows:

> The Linux Documentation Project is working on developing free, high quality documentation for the GNU/Linux operating system. The overall goal of the LDP is to collaborate in all of the issues of Linux documentation. This includes the creation of "HOWTOs" and "Guides." We hope to establish a system of documentation for Linux that will be easy to use and search. This includes the integration of the manual pages, info docs, HOWTOs, and other documents.
>
> LDP's goal is to create the canonical set of free Linux documentation. While online (and downloadable) documentation can be frequently updated in order to stay on top of the many changes in the Linux world, we also like to see the same docs included on CDs and printed in books. If you are interested in publishing any of the LDP works, see the section "Publishing LDP Documents" (on this webpage).
>
> The LDP is essentially a loose team of volunteers with minimal central organization. Anyone who would like to help is welcome to join in this effort. We feel that working together informally and discussing projects on our mailing lists is the best way to go. When we disagree on things, we try to reason with each other until we reach an informed consensus.

LDP Guides

The LDP's guides are book-length documents that cover many aspects of Linux operations. To access them, drill down through the LDP Web site at www.linuxdoc.org or go directly to them at http://metalab unc.edu/pub/Linux/docs/LDP/.

The guides have been published by the LDP. Currently available and maintained guides include the following:

- Securing and Optimizing Linux: Red Hat Edition (Open Docs)
- Administering Linux: The Basics (Open Docs)
- Linux Complete (Sybex)
- Linux the Complete Reference (Linux System Labs)
- Linux Desktop Starter Kit (McGraw Hill)
- Linux Facile (An Italian Linux Manual)
- Linux Network Administrator's Guide, 2nd Edition (O. Kirch and T. Dawson)

HOWTOs

Linux HOWTOs are documents that explain specific topics rather than the broad subjects generally found in the guides. The following is a summary of available HOWTOs and their location:

226 HOWTOs at www.ibiblio.org/pub/Linux/docs/HOWTO/

4 unmaintained HOWTOs at
http://ibiblio.org/pub/Linux/docs/HOWTO/unmaintained/

131 mini-HOWTOs at
http://ibiblio.org/pub/Linux/docs/HOWTO/mini/

Remember that unmaintained documents are not updated and might now be invalid. But, as it states on the "unmaintained" Web page, " . . . sometimes old documentation is better than none at all . . . ."

If you want to create your own HOWTO or become the new custodian of an unmaintained HOWTO, contact the LDP through its Web site. See the "Projects" section later in this chapter.

Occasionally, HOWTOs are published in hard-copy format, such as in *Linux Undercover* (Red Hat, 1998).

Your distribution of Linux might include copies of HOWTOs. Look in directories named */usr/doc/HOWTO*, */usr/doc/HOWTO/mini*, or something similar. You will probably have to uncompress the HOWTO files before you can view them.

FAQs

As of this writing, the LDP has nine sets of *frequently asked questions* (FAQs) on its Internet site at www.linuxdoc.org/FAQ/. At one time, the LDP suggested that any new HOWTO documents also include a set of FAQs,

because readers of the documents often have questions about the documents themselves or about how the document instructions relate to their own systems. Although the LDP no longer states that explicitly, we think it is still a good idea.

Your Linux software distribution might have FAQs as well; look for them in the installation manual and in directories named */usr/doc/FAQ* or something similar. You will probably have to uncompress the FAQ files before you can view them.

Linux Man Pages

The LDP maintains the `man` pages for C programming and devices (sections 2, 3, 4, 5, 6, 7, and 9). But they recommend consulting the user command `man` pages (sections 1 and 8) on the respective distribution CDs.

Projects

The LDP always has documentation and other projects under development. The projects are organized into several categories, such as hardware ports; kernel, drivers, and file systems; papers; networking; organizations and groups; Linux and free or open software; research and science groups; distributions; benchmarks and standards; and miscellaneous.

To get involved, check out the advice in the "Current Projects and Getting Involved" section of the LDP manifesto at www.linuxdoc.org or contact the LDP coordinator. You can e-mail the LDP at feedback@linuxdoc.org.

Linux Books and Magazines

In the process of preparing course materials and writing this book, we were exposed to several sources of expertise and information, and we are grateful to them all. Here are some suggestions to help you find the ones you need:

Check out the references presented at the Linux Web site at www.linux.org/books. You will find a description and short evaluation of several (43, actually, as of late 2001) current and prominent reference books dealing with Linux, Linux applications, and the integration of Linux with other systems.

Use your favorite Internet search engine. Type an appropriate description, such as `Linux books`, click the Search button, and

watch your screen fill with Web site references. You will probably want to refine your search description to find something closer to your requirements. (Our search for `Linux reference books` and similar terms returned well over 100,000 responses. Overwhelming? Yes, but there is bound to be something out there to meet almost everyone's needs. All you have to do is refine your search.)

Check out the book reviews in Linux paper magazines or online magazines. For example, *Linux Weekly News* (a division of Tucows) at http://lwn.net or *Linux Planet* at www.linuxplanet.com.

Use the search features at the e-commerce sites operated by large booksellers. You can get descriptions and order them, as well.

Take the time to visit your local computer bookstore or "big-box" bookstore. This method is the best way to determine which books are best suited to your needs. Many establishments now enable you to peruse your potential purchase leisurely, even over an in-house beverage.

Although we are reluctant to recommend any particular Linux book out of the many available, we do suggest reading the following:

Operating Systems: Design and Implementation, 2nd ed., by Andrew S. Tanenbaum and Albert S. Woodhull (Prentice Hall). Tanenbaum also maintains a MINIX Web site at www.cs.vu.nl/~ast/minix.html.

The UNIX Programming Environment, by Brian Kernighan and Rob Pike (Prentice Hall Computer Books). These two Bell Labs researchers were heavily involved with the development of UNIX. This book, which presents the unique features of the UNIX design philosophy, is a classic.

Customizing and Upgrading Linux, *Second Edition*, by Linda McKinnon and Al McKinnon (John Wiley & Sons, 2002). Our companion book describes the installation of Linux on several platforms via various methods. It also tells you how to upgrade the Linux kernel. Please pardon our "tub-thumping" here, but we think it is pretty good stuff.

Several Linux journals are listed in Table 15.1. All are available as online magazines, and the first three are available in hard copy, too. You might already have heard of some of them, such as the *Linux Journal*, which has been publishing for five years or so. The others are similar in approach.

Table 15.1 Linux Online Magazines

TITLE	WEB SITE
Linux Journal	www2.linuxjournal.com
Linux Magazine	www.linux-mag.com
Linux Planet	www.linuxplanet.com
Linux Weekly News	http://lwn.net
Linux Today	http://linuxtoday.com
Linux Center	www.portalux.com
Linux Start	www.linuxstart.com
Linux Gazette	www.ssc.com/lg/ or www.linuxgazette.com
Linux Focus	www.linuxfocus.org
Penguin Magazine	www.penguinmagazine.com
ZDNet Linux Zone	http://linux.zdnet.com
Linux World	www.linuxworld.com
ITtoolbox Linux	http://linux.ittoolbox.com
Linux.com	www.linux.com
Linux Business Week.com	www.sys-con.com/linux
UNIXREVIEW.COM	www.unixreview.com

As we mentioned, almost every urban center or region has some type of free or inexpensive computer-oriented newspaper. They are a great source of up-to-date articles, contacts for support or user groups, and advertisements of all kinds. Support groups typically have e-mail-type newsletters, chat groups or newsgroups, and mailing lists as well.

More Linux Information Sites on the Internet

So many valuable Linux-related sites are on the Internet that it is difficult to decide where to begin and just as difficult to determine where to stop. We have already mentioned several Internet sites, and this section describes a few more that we hope you will find interesting and beneficial.

Linux Online

Linux Online!, the official Linux Web site (which we have mentioned a couple of times already), is at www.linux.org. Certainly, it has to be one of the more obvious places to start when you are looking for any Linux information. Everything you need to know about Linux in general is there, organized into categories such as General Information, Distributions, Applications, Support, Projects, Hardware, User Groups, Book Store, Vendors, Events, Services, Featured Books, The GNU General Public License, Documentation, Courses, News, People, and so on. You can link to most of the other significant sites through the categories. There is a search engine on the site, too. Hey, on the site you can even support Linux development by buying a Linux Online! coffee mug featuring Tux, the Linux penguin.

Tux, the Linux Penguin

Speaking of Tux, the official logo of Linux is a penguin (please see the end of the last section). To find out about him, try going to www.isc.tamu.edu/~lewing/linux, which is the site of Larry Ewing, his creator.

Linux Online in Canada

The www.linuxcanada.net site is recommended. Its major attraction is that it has many direct links to other excellent Internet sites. Its categories are Linux Books, Linux User Groups, Linux Facts, Linux Discussions, Linux News, Biz Press, and Linux Software.

Linux Online in Europe

Also recommended is www.linux.eu.org, designed to help European users find *Local UserGroups* (LUGs), information, and other Linux sites in their own language and/or country. There are "national flag" links from this site to sites in more than 30 European nations.

Software Sources

The following sites, which can be accessed from the two just-noted sites, bear special mention because they are reputable sources for Linux software and applications. All have a similar structure (articles, software, links to other sites, and so on):

- http://freshmeat.net
- http://slashdot.org (Its motto: "News for Nerds. Stuff that matters.")
- www.kernel.org ("The Linux Kernel Archives," the quintessential source for the latest Linux kernels.) If you want, you can also investigate www.ibiblio.org. Once you get there, click the "Enter the Navigator" button to find other Linux kernels and kernel information.

X Window System Sources

The following three sites deal with X Window System (explained in detail in Chapter 14, "The Linux X Window System").

The XFree86 Project, Inc.™ atwww.xfree86.org is the master Web site for X Window System information and downloads. On that site, you will learn what XFree86 is, what the latest version is, and what mirror sites you can access (if necessary). It will also tell you the latest X Window system news and provide documentation and other information. You can find an FTP download site at ftp://ftp.xfree86.org/pub/XFree86/ and an HTTP download site at http://ftp.XFree86.org/pub/XFree86/.

www.kde.org/ is the Web site for the *K Desktop Environment* (KDE), one of the more sophisticated X Window managers (mentioned in Chapter 14). The site is organized similarly to others: General Information, Worldwide Sites, Developer Information, Events, Supporting KDE, FAQs, Downloads, and so on.

www.gnome.org/ is the Web site for the *GNU Network Object Model Environment* (GNOME), considered by many to be *the* modern X Window manager. Based entirely on open-source (in other words, free) software, it has links to the GNOME software and Manifesto sites and is otherwise structured like the two previous sites.

Linux Newsgroups

USENET newsgroups are a worldwide system of computers that store, update, and exchange collections of discussion text files organized by category. Newsgroups are a means of public discussion. Their articles/messages look like e-mail, but they can potentially be read by millions of people all

over the world. Newsgroup articles are distributed via news servers, which contain databases of articles and are operated by *Internet service providers* (ISPs), schools, universities, and companies.

Newsreaders for Accessing Newsgroups

You will have to get newsreader software, such as PAN (http://pan.rebelbase.com/download.html), that also supports offline newsreading, multiple connections, and extra features (including those for alt.binary files). It is available in several formats (RPMs, tarballs, and so on) on the Web site, but it is also found packaged with the major Linux distributions. Documentation and other resources can be found on the Web site.

Once your newsreader is configured to read newsgroups, you should be able to click on the various newsgroup names to read them. If that does not work, you might have to contact your ISP for further advice on how to access USENET.

Here are a few Linux-oriented USENET newsgroups:

comp.os.linux.advocacy. The benefits of Linux compared to other OSs.

comp.os.linux.announce. All announcements that are relevant to the Linux community: new software, new documentation, warnings about bugs and security holes, notices about user group meetings, and so on.

comp.os.linux.answers. For posting Linux FAQs, HOWTOs, READMEs, and other documents.

comp.os.linux.development.apps. Questions and discussions regarding the writing of applications for Linux and the porting of applications to Linux.

comp.os.linux.alpha. Linux on Digital Alpha machines.

comp.os.linux.hardware. Questions and discussions specific to Linux visà-vis a particular piece of hardware.

comp.os.linux.m68k. Intended to further interest in, and development of, porting Linux to Motorola's 680x0 architecture.

comp.os.linux.powerpc. Linux running on PowerPC microprocessors.

comp.os.linux.networking. Questions and discussions relating to networking or communications, including Ethernet boards, serial communications, SLIP, and so on.

comp.os.linux.x. Questions and discussions relating to X Window System, version 11, compatible software including servers, clients,

libraries, and fonts running under Linux will be directed to this newsgroup.

comp.windows.x.i386unix. XFree86 issues that are not unique to Linux.

comp.os.linux.development.system. Questions and discussions regarding the Linux OS development: kernel, device drivers, and loadable modules.

comp.os.linux.development.apps. Writing Linux applications; porting applications to Linux.

comp.os.linux.setup. Questions and discussions relating to Linux installation and system administration.

comp.os.linux.misc. Questions and discussions relating to Linux but not covered by a more specific Linux newsgroup.

Other miscellaneous Linux newsgroups are as follows:

alt.uu.comp.os.linux.questions	han.sys.linux
aus.computers.linux	hannet.ml.linux.680x0
dc.org.linux-users	it.comp.linux.pluto
de.alt.sources.linux.patches	maus.os.linux
de.comp.os.linux.hardware	maus.os.linux68k
de.comp.os.linux.misc	no.linux
de.comp.os.linux.networking	okinawa.os.linux
de.comp.os.x	tn.linux
ed.linux	tw.bbs.comp.linux
fido.linux-ger	ucb.os.linux
fj.os.linux	uiuc.sw.linux
fr.comp.os.linux	umich.linux

Other newsgroups you might also try include the following:

- alt.linux presents alternative views of Linux (in addition to the following):
- alt.linux.redhat (alternative views of the Red Hat Linux distribution)
- alt.linux.stormix (alternative views of Storm Linux from Stormix Technologies)
- alt.linux.slakware (alternative views of the Slackware Linux distribution)

- alt.linux.sucks (alternative views of Linux)
- alt.linux.storage.* (one group; alternative views of Linux and various hdd storage issues)
- alt.linux.sux (more alternative views of Linux)
- linux.appletalk is the place for discussions regarding the use of Linux and Apple's LocalTalk networking protocol.

Browsing to Read Newsgroups

If you want to try to browse USENET for articles and you do not have a newsreader installed, then go to http://groups.google.com/ (or go to the Google search engine at www.google.com and click the "Google Groups" link). This site used to be called Deja News www.deja.com until Google acquired it and changed the Web site name to Google Groups. At the top of the site, you can Search Groups by providing a name or Browse Groups according to categories.

Linux mtools Sites

The `mtools` utilities enable easier manipulation of DOS floppy diskettes by using newer commands that are unlike the somewhat stilted, older-style UNIX commands (see Chapter 5, "Using Files in Linux"). The following sites provide information, software, and services regarding `mtools`:

- www.tux.org/pub/knaff/mtools/ (U.S. Web site)
- http://mtools.linux.lu/ (European Web site)
- ftp://www.tux.org/knaff/mtools/
- ftp://ibiblio.unc.edu/pub/Linux/utils/disk-management/

Exercises

1. Dial in to the Internet or activate the Internet browser on your system. Go to the Linux Web site at www.linux.org. Navigate around and become familiar with this site. When did Linus Torvalds release Version 1.0 of the Linux kernel? What is the current featured version,

and when was it released (this question is tricky; you might have to find a link to another site)?

2. Go to the Linux Documentation Project site at www.linuxdoc.org. What is the date of this master copy of the LDP Web site? What is the date of the mirror copy of the Web site located nearest to your location? (Hint: go up to LDP worldwide at the top-left corner of the home page and click the Mirrors link.)
3. Go to the Web site of the *Linux Journal* at www.linuxjournal.com and navigate it until you have familiarized yourself with the site. What is the journal's slogan?
4. Go to the freshmeat.net Web site at http://news.freshmeat.net. What is the name of the first software package listed at the top of their Web site, beneath the freshmeat search engine? What is the title of the first editorial in the right-hand side frame?
5. Go to http://groups.google.com/ (formerly www.deja.com, until Google acquired them and changed the Web site name to Google Groups). At the top of the site, under Search Groups, type `comp.os.linux.setup` in the search field and click Google Search. What is the date and title of the most recent message?

See Appendix B for answers.

Quiz

1. Provide the Internet addresses for the best-known Linux Web site and the best-known X Window System Web site.
2. What is the name of the official Linux penguin?
3. What are USENET newsgroups? What two ways are there to access them?
4. Who is working on developing free, high-quality documentation for the GNU/Linux operating system?

See Appendix C for answers.

Command Summary

Logging Off and Shutting Down

COMMAND	DESCRIPTION
`Ctrl-D`	Log off the Linux system (or the current shell)
`exit`	Log out from the Linux system (or the current shell)
`logout`	Log off the Linux system (or the current shell)
`shutdown`	Shut down the system by disabling all processes; requires the user to be a root user
	Use `-h` (halt) option in root user mode for total shutdown; cold boot required to restart system
	Use `-r` (reboot) option for (warm) reboot
	Use `now` or `two digit number` ("time in minutes") arguments with either option (for immediate or delayed action)

Directories

COMMAND	DESCRIPTION
`mkdir`	Make a new directory. Supply `directoryname` as an argument.
`cd`	Change to another directory; the default (no option/argument supplied) is the *$HOME* directory; otherwise, supply an existing `directoryname` as an argument.
`rmdir`	Remove/delete a directory (beware of files starting with dot); supply a `directoryname` as an argument.
`rm`	Remove a file; supply a `filename` as an argument. The `-R` option removes the directory and all files and subdirectories recursively.
`pwd`	"Print working directory" to the terminal screen (that is, tell user what directory they are in).
`ls`	List files; default (no options/arguments) displays names of all files and directories except hidden files. Use the `-a` option for all (including hidden) files. Use the `-i` option to display respective inode numbers. Use the -l option for long listings. Use the `-d` option for Directory information only. Use the `-r` option for Reverse alphabetic listing. Use the `-t` option to sort by "time when entry was changed." Use the `-C` option for Multicolumn format display. Use the `-R` option for Recursive listing. Use the `-F` option to place a `/` (backslash) after each directory name and an `*` (asterisk) after each executable file.

Basic File Management

COMMAND	DESCRIPTION
`cat`	List file contents (concatenate) to the terminal screen. Or, with the output redirection symbol (`>`), create a new file (for example, `cat> newfile<Enter>`). Then, use Ctrl-D to end input to the new file.
`chmod`	Change permission mode for files or directories. Use `r`, `w`, `x` symbolic permissions for read, write, and execute. Use `+` (plus), `-` (hyphen or minus), or `=` (equals sign) to grant, revoke, or specify exact permissions. Use `u`, `g`, `o`, `a` to give permissions to user, group, others, and all, respectively. Can also use octals (in other words, numerics): `4` = read, `2` = write, and `1` = execute.

NOTE **Linux sums permissions: the first is user, the next is group, and the last is other. For example,** `chmod 746 filename` **gives these permissions: user** `r`, `w`, `x`**; group** `r`**; and others** `r`, `w`.

COMMAND	DESCRIPTION
`chown`	Change the owner of a file (for example, `chown ownername` *filename*)
`cp`	Copy file; supply *filename* as argument
`del`	Delete files with prompting; `rm` (see above) deletes files with no prompting
`mv`	Move and/or rename file; supply *filename*(*s*) as argument(s)
`.`	(start at) Current directory
`..`	(start at) Parent directory
`/string`	Find string, proceeding in forward direction
`?string`	Find string, proceeding in backward direction
`- -`	Move backward - pages
`- ?`	Move forward - pages
`rm`	Remove (delete) files; supply *filename*(*s*) as argument(s) Use the `-r` option to remove the directory and all files and subdirectories.

Basic File Management, Continued

COMMAND	DESCRIPTION
`head`	Print first several lines of a file; supply *filename* as an argument
`tail`	Print last several lines of a file; supply *filename* as argument
`wc`	Report the number of specified elements in a file Use `-l` option to count and report number of lines Use `-w` option to count and report number of words Use `-c` option to count and report number of characters If you use no options, `wc` displays lines, words, *and* characters.
`su`	Switch user; supply `username` as argument; will likely have to supply password for authentication
`id`	Display user ID environment and how it is currently set
`tty`	Display the active device Useful in X Window System because you can create several `pts` devices, and it is handy to know which one is active. The `whoami` command provides the same function.

Advanced File Management

COMMAND	DESCRIPTION
`banner`	Display a banner Also redirect a banner to another terminal, `nn`, by specifying `>/dev/ttynn`
`cal`	Calendar command (`cal [month] year`)
`diff`	Show differences between two files

`find`	Find files anywhere on disk; specify location by path (searches all subdirectories under specified directory)
	Syntax: `find (path expression action)`
	Examples:
	`find / -name "*.txt" -print`
	`find / -name "*.txt" -exec li -l {} \;`
	Executes `li -l` where names found are substituted for `{}`.
	The `;` (semicolon) indicates end-of-command to be executed. The `\` (backslash) removes its usual interpretation as a command continuation character.

Advanced File Management, Continued

COMMAND	DESCRIPTION
`find`	Use `-name fl` option/argument to find filenames matching `fl` criteria
	Use `-user username` option/argument to find files owned by user `username`
	Use `-size +n` option/argument to find files larger or smaller than `n` blocks; can also use `-n` to find files smaller than `n` blocks
	Use `-perm num` option/argument to find files whose access permission modes match `num`
	Use `-exec commandname` option/argument to execute a command, using as input the results of the `find` command
	Use `-ok` option to execute a command, interactively using as input the results of the `find` command
	Use `-o` (logical OR) option to combine `find` with another command name
	Use `-print` option to display results of `find`; this command is usually included
`grep`	Search for a pattern within specified files (for example, `grep` *`pattern filenames`*); pattern can include regular expressions
	Use `-c` option to count lines with matches but do not list
	Use `-l` option to list files with matches
	Use `-n` option to list line numbers with matching lines
	Use `-v` option to find files without pattern
	Expression metacharacters
	Use square brackets `[ ]` to match any one character from set or range specified inside the brackets
	Use the Ctrl symbol (^) to match beginning of line when ^ begins the pattern
	Use the dollar sign `( $ )` to match end of line when `$` ends the pattern
	Use single dot (.) to match any single character (same as ? in shell)
	Use the asterisk (`* )` to match zero or more occurrences of a preceding character

continues

NOTE **.* (dot asterisk) is the same as * (asterisk) in the shell.**

Advanced File Management, Continued

COMMAND	DESCRIPTION
`sed`	Stream (text) editor; used with editing flat files
`sort`	Sort and merge files
	Use `-r` option to list in reverse order
	Use `-u` option to keep only unique lines

Editors

COMMAND	DESCRIPTION
`vi` (or `vim`)	On-screen text editor
`emacs`	On-screen text editor (GNU Emacs for X Window system)

Shells, File Descriptors, Redirection, Command Piping, etc.

COMMAND	DESCRIPTION
`<`	File Descriptor–Input Redirection
(read)	Example:
	`$ commandname < filename`
	"read input for `commandname` from *filename*"
`>`	File Descriptor–Output Redirection (Destructive)
(write)	Example:
	`$ commandname > filename`
	"write output for `commandname` to filename, overwriting the *filename*'s contents"
>>	File Descriptor–Output Redirection (Non-destructive)
(append)	Example:
	`$ command >> file`
	"append output from `commandname` to the end of *filename*"

Shells, File Descriptors, Redirection, Command Piping, etc., Continued

COMMAND	DESCRIPTION
`2>`	File Descriptor–Error Redirection (Destructive) Example: `$ commandname 2> filename` "write errors from `commandname` execution to *filename*"

`2>>`	File Descriptor–Error Redirection (Non-destructive)
	Example:
	`$ commandname 2>> filename`
	"append errors from `commandname` execution to end of *filename*"
	Other redirection examples:
	`$ commandname < infilename > outfilename 2> errorfilename`
	`$ commandname >> appendoutfilename 2>> appenderroroutfilename < infilename`

`;`	Command delimiter/terminator–used to string commands together on a single line
`|`	Pipe/pipeline output from one command to be input to the next command Example: `ls | cpio -o > /dev/fd0` "pass the results of the `ls` command to the `cpio` command; send final output to first floppy disk"
`\`	Continuation character–to continue commands on a second (or more) line; will be prompted with `>` (secondary prompt) for command continuation
`tee`	Reads standard input and sends standard output to both standard output *and* a file Example: `$ ls | tee ls.save | sort` `ls` output goes to `ls.save` and is also `piped` to the `sort` command

Metacharacters

COMMAND	DESCRIPTION
`?`	Match any single character
`[abc]`	Match any character from the list "abc"
`[a-c]`	Match any character from the list range a to c
`!`	Don't match to any of the following characters Example: `$ echo [!tn]*` "echo all filenames that do not begin with `t` or `n`"
`;`	Command delimiter/terminator; used to string commands together on a single line
`&`	Command(s) preceding the ampersand character are to be run in background mode
`#`	Comment character; put at left-hand side of comment lines
`\`	Remove special meaning of (in other words, do not interpret) the following character
`"`	Interpret only `$`, `'` (back quote), and `\` characters between the quotes
`` ` ``	Set value of variablename to the results of command execution Example: `` $ now = `date` `` "set the value of the variable `now` to the current results of the `date` command
`$`	Preceding variablename indicates the value of the variable

Variables

COMMAND	DESCRIPTION

=	Variable substitution to set the value of a variable
	Example:
	`$ d = "day"`
	"set the value of the variable `d` to the specified value day"
	Command substitution; set the value of the variable to be the results of a command
	Example:
	`$ now = 'date'`
	"set the value of the variable `now` to the current result of the `date` command"

HOME	Home directory
PATH	Directory paths to be checked by the shell when searching for commands
SHELL	Shell to be used by default
TERM	Terminal being used
PS1	Default primary prompt character, usually `$` or `#`
PS2	Secondary prompt character, usually `>`
$?	Return the status code of the last command executed
set	Display current local variable settings
export	Export variables so that they are inherited by child processes
env	Display inherited variables
echo	Display the following message back to the terminal screen Examples: `$ echo HI` or `$ echo $d` "display the word `HI`"; "display the value of the variable `d`"
\c	Turn off carriage returns (by placing `\c` at the end of the message)
\n	Print a blank line (by placing `\n` at the end of the message)

Tapes and Floppy Diskettes

COMMAND	DESCRIPTION
`fdformat`	Format a floppy diskette

`backup`	Back up individual files
	Use `-i` option to read file names from standard input
	Use `-v` (verbose) option to list files as they are backed up
	Example:
	`$ backup -iv -f/dev/rmt0 filename1, filename2`
	Use `-u` option to backup file system at the specified level
	Example:
	`$ backup -level -u filesystem`
	Use `|` (pipe) to list files to be backed up into command
	Example:
	`$ find . -print | backup -ivf/dev/rmt0`
	Where you are already in the directory that is to be backed up

`restore`	Restore from backup Use `-x` option to restore files created with `backup -i` Use `-v` option to list files as they are restored Use `-T` option to list files stored on tape or floppy diskette Use `-r` option to restore file system created with `backup -level -u` Example: `$ restore -xv -f/dev/rmt0`
`cpio`	Copy to and from an I/O device. Destroy all data previously on tape or floppy diskette. For input, must be able to place files in the same relative (or absolute) path name as when copied (can determine path names with the `-it` option). For input, if file exists, compare last modification date and keep most recent (can override with the `-u` option). `-o` Output `-i` Input `-t` Table of contents

Tapes and Floppy Diskettes, Continued

COMMAND	DESCRIPTION

	`-v` Verbose
	`-d` Create needed directory for relative path names
	`-u` Unconditional to override last modification date
	Example:
	`$ cpio -o > /dev/fd0`
	`filename1`
	`filename2`
	`<Ctrl>-d`
	or `$ cpio -iv filename1 < /dev/fd0`

`tar`	Alternative utility to back up and restore files

Transmitting

COMMAND	DESCRIPTION
`mail`	Send and receive mail. With `username` argument, send mail to that user. Without `username` argument, display your mail. When processing your mail, at the `?` prompt for each mail item, you can use the following subcommands, too: `d` Delete `s` Append `q` Quit `<Enter>` Skip `m` Forward
`uucp`	Copy file to other UNIX system (UNIX-to-UNIX copy)
`uux`	Execute on a remote system (UNIX-to-UNIX execute)

System Administration

COMMAND	DESCRIPTION
`kill PID`	Kill the batch process with the specified `PID` number
`kill-9 PID`	The "unconditional kill"
`mount`	Associate logical volume to a specified *directoryname* Example: `$ mount device directoryname`
`umount`	Disassociate file system from *directoryname*
`ps -ef`	Show status of process(es)

Miscellaneous

COMMAND	DESCRIPTION
`banner`	Display specified characters as a banner
`date`	Display current date and time
`nice`	Assign lower priority to the following specified `commandname` Example: `$ nice 'ps -f`
`passwd`	Modify current password (root user can supply `username` argument to change `username`'s password)
`sleep n`	System to remain dormant for specified number of seconds
`stty`	Show and/or set terminal settings
`touch`	Create zero length file; or update modification time
`startx`	Invoke X Window System manager
`wall`	Send message to all logged-in users
`who`	List users currently logged in (`whoami` identifies this user)
`man`	With `commandname` as argument, display manual pages for the specified `commandname`

System Files

COMMAND	DESCRIPTION
`/etc/group`	List of user groups
`/etc/motd`	Message of the day; usually displayed at login
`/etc/passwd`	List of users and default user information (for example, password, home directory, userid, and groupid) Can prvent password hacking by editing to `remove! (password)`. System-wide user profile executed at login
`/etc/profile`	Can override variables by resetting in the user's profile file

Shell Programming Summary

Shell Variables

COMMAND	DESCRIPTION
`Variablename=string`	Set variable value to be string; remember, no spaces between the `variablename`, equals sign, and the designated string Enclose spaces with double quotes
`Variablename=string`	Special characters in the string must be enclosed by single quotes to prevent substitution Piping (`|`), redirection (`<`, `>`, `>>`), and `&` symbols are not interpreted.
`$variablename`	Provide value of `variablename` to command sequence
`echo $variablename`	Display value of `variablename` to terminal screen Example: `$ echo $variablename`
`$HOME`	Home directory of user
`$MAIL`	Mail filename

continues

Shell Variables, Continued

COMMAND	DESCRIPTION
`$PS1`	Default primary prompt character, usually `$` or `#`
`$PS2`	Default secondary prompt character; usually `>` but can be changed by the user
`$PATH`	Default search path (when searching to invoke commands)
`$TERM`	Terminal type being used
`export`	Export variables to the environment
`env`	Display environment variables settings
`${variablename:-string}`	Give value of `variablename` in a command; if `variablename` is null, use `string` instead
`$1 $2 $3 . . .`	Positional parameters for `variablename` passed into the shell script
`$*`	Used for all arguments passed into the shell script
`$#`	Number of arguments passed into the shell script
`$0`	Name of the shell script
`$$`	Process ID (`PID`)
`$?`	Last return code from a command

Shell Commands

COMMAND	DESCRIPTION
`#`	Comment designator; use at left-hand side of line
`&&`	Logical `AND`. Means run the command following the `&&` *only* if the command preceding the `&&` succeeds (in other words, if the preceding command had a return code = 0).
`||`	Logical `OR`. Means run the command following the || only if the command preceding the || fails (in other words, if the preceding command had a return code <> 0).
`for loop`	Specify a command or a series of commands to be carried out as long as a variable or a filename meets certain value specifications. Example: `# Comment - Begin the loop`

Shell Commands, Continued

COMMAND	DESCRIPTION

	`for variable/file range`
	`do`
	`cp $a textdir/$a`
	`done`
	`#Comment - the loop is ended`
	Example:
	`# Comment - Begin the loop`
	`for a in *.doc`
	`do`
	`commandname(s)`
	`done`
	`#Comment - The loop is ended`

`if-then-else`	The user is allowed to select from alternatives based on the result of the execution of a command. The `else` portion is optional.
	Example:
	`# Comment - Begin the loop`
	`if`
	`commandname`
	`then`
	`commandname (alternative 1)`
	`else`
	`commandname (alternative 2)`
	`fi`
	`# Comment - The loop is ended`

`read`	Read from standard input
`shift`	Shift arguments 1–9 one position to the left and decrement number of arguments

continues

Shell Commands, Continued

COMMAND	DESCRIPTION

`test`	Used for conditional test; has two formats:
	1. `$ if test expression`
	Example:
	`$ if test $- -eq2`
	2. `$ If [expression]`
	Example:
	`$ if [ $# -eq 2 ]`
	Please Note: Spaces are required
	Operators:
	`-eq` =
	`-lt` <
	`-le` =<
	`-ne` < >
	`-gt` >
	`-ge` =>
	String operators:
	= Equal
	`!=` Not equal
	`-z` Zero length
	File status (for example, `-opt file 1`)
	`-f` Ordinary file
	`-r` Readable by this process
	`-w` Writable by this process
	`-x` Executable by this process
	`-s` Nonzero length

Shell Commands, Continued

COMMAND	DESCRIPTION

`while loop`	Enables the user to execute (or not) a command depending on the exit status (or results) of another command
	Example:
	`#Comment - Begin the loop`
	`while commandname(s)`
	`do`
	`commandname(s)`
	`done`
	`#Comment - The Loop is ended`

vi Editor

Entering vi

COMMAND	DESCRIPTION
`vi filename`	Display the specified file (in other words, bring it into the buffer) for editing
`vi filename1 filename2`	Edit the files consecutively (via `:n`)
`.exrc`	File that contains the `vi` profile
`wm=nn`	Set wrap margin to `nn` Enter at a point specified other than at the first line by adding: `+` (Enter at the last line) `+ n` (Enter at line `n`) `+ /pattern` (Enter at the first occurrence of `pattern`)
`:n`	Next file in stack
`:set all`	Show all options
`:set nu`	Display line numbers (off when `set nonu`)
`:set list`	Display control characters in file
`:set wm=n`	Set wrap margin to `n`
`:set showmode`	Set display of the word `INSERT` when in insert mode

Reading in, Writing in, and Exiting `vi`

COMMAND	DESCRIPTION
`:w`	Write buffer contents
`:w filename2`	Write buffer contents to *filename2*
`:w >> file2`	Write buffer contents to the end of *filename2*
`:q`	Quit this editing session (no writing of buffer contents if `q` is used alone)
`:q!`	Quit this editing session and discard any changes
`:r filename2`	Read *filename2* contents into buffer following the current cursor position
`:r! commandname`	Read the results of `commandname` shell command following the current cursor position
`:!`	Exit shell command (filter through command)
`:wq` or `ZZ`	Write buffer contents back to the file and then quit this editing session

Navigating in vi

COMMAND	DESCRIPTION
`h`, `l`	Move one character position to the left; one character position to the right
`k` or Ctrl-p	Move the cursor to the character position immediately above the present position
`j` or Ctrl-n	Move the cursor to the character position immediately below the present position
`^`, `$`	Move the cursor to the beginning of the current line; to the end of the current line
`w`, `b`	Move the cursor one word position to the right; one word position to the left
`Enter` or `+`	Move the cursor to the beginning of the next line
`-`	Move the cursor to the beginning of the previous line
`G`	Move the cursor to the beginning of the last line of the buffer contents

Cursor Movement in vi

Can precede cursor movement commands (including the cursor arrow) with the number of times to repeat; for example, `9 <right arrow>` moves right 9 characters.

COMMAND	DESCRIPTION
`0`	Move the cursor to the first character in the line
`$`	Move the cursor to the last character in the line
`^`	Move the cursor to the first nonblank character in the line
`fx`	Move the cursor to the right to character `x`
`Fx`	Move the cursor to the left to character `x`
`tx`	Move the cursor to the right, to the character preceding `x`
`Tx`	Move the cursor to the left, to the character preceding `x`
`w`	Tab one word (`nw` = `n` tab word) (punctuation marks are considered to be words)
`b`	Backtab one word (punctuation marks are considered to be words)
`B`	Backtab one word (ignore punctuation marks)
`e`	Tab to the last character of the next word (punctuation marks are considered to be words)
`E`	Tab to the last character of the next word (ignore punctuation marks)
`(`	Move the cursor to the beginning of the current sentence
`)`	Move the cursor to the beginning of the next sentence
`{`	Move the cursor to the beginning of the current paragraph
`}`	Move the cursor to the beginning of the next paragraph
`H`	Move the cursor to the first line on the screen
`M`	Move the cursor to the middle line on the screen
`L`	Move the cursor to the last line on the screen
`Ctrl-f`	Scroll forward one screen
`Ctrl-d`	Scroll forward half screen
`Ctrl-b`	Scroll backward one screen

continues

Cursor Movement in vi, Continued

COMMAND	DESCRIPTION
`Ctrl-u`	Scroll backward half screen
`G`	Go to last line in file
`nG`	Go to line `n`
`Ctrl-g`	Display the current line number

Searching and Replacing in vi

COMMAND	DESCRIPTION
`/pattern`	Search for the `pattern` in the forward direction
`?pattern`	Search for the `pattern` in the backward direction
`n`	Continue/repeat the search in the same direction
`N`	Continue/repeat the search in the opposite direction

Adding Text in vi

COMMAND	DESCRIPTION
`a`	Add text after the cursor (end with `<Esc>`)
`A`	Add text at the end of the current line (end with `<Esc>`)
`i`	Add text before the cursor (end with `<Esc>`)
`I`	Add text before first nonblank character in the current line
`o`	Add a line following the current line
`O`	Add a line before the current line
`<Esc>`	Return from insert mode to command mode

Deleting Text in vi

COMMAND	DESCRIPTION
`Ctrl-w`	Undo entry of current word
`@`	Kill the insert on this line
`x`	Delete the current character

Deleting Text in vi, Continued

COMMAND	DESCRIPTION
dw	Delete to the end of the current word (observe punctuation)
dW	Delete to the end of the current word (ignore punctuation)
dd	Delete the current line
D	Erase to the end of the same line (same as d$)
d)	Delete the current sentence
d}	Delete the current paragraph
dG	Delete from the current line through to the end of the buffer
d^	Delete to the beginning of the line
u	Undo last change
U	Restore the current line to state it was in before the modification

Replacing Text in vi

COMMAND	DESCRIPTION
ra	Replace the current character with an a
R	Replace all characters overtyped until <Esc>
s	Delete the current character and append text until <Esc>
s/s1/s2	Replace s1 with s2 (but only in the same line)
S	Delete all the characters in the line and append text
cc	Replace all the characters in the line (same as S)
ncx	Delete n text objects of type x, where x can be: w, b - words) - sentences } - paragraphs $ - to the end of the line ^ - to the beginning of the line <Enter> - append mode
C	Replace all the characters from the present cursor position to the end of the line

Moving Text in vi

COMMAND	DESCRIPTION
`p`	Paste the last text deleted after the present cursor position (`xp` transposes two characters)
`P`	Paste the last text deleted before the present cursor position
`nYx`	Yank `n` text objects of type `x`
`"ayy`	Use named registers for moving, copying; cut and paste with `"ayy` command for register `a` (use registers a-z)
	Then paste with the `"ap` command

Exercise Answers

Chapter 3

10. Yes. The name of this node is uname invocation, which is the same as what appeared adjacent to Next on the `date` page.
11. The name of the most-previous node is date invocation, which coincidentally is the first `date` page you encountered.
15. Linux's shell responds with a `usage` message, specifying options and other arguments for `mount`.
16. This time, Linux's shell responds with an invalid option diagnosis followed by the same usage message that appeared for Exercise 10 when you invoked `mount` with its `help` argument.
20. We hope you noticed something peculiar about September. The shortening of September 1752 was decreed by Pope Gregory to bring the calendar back into sync with Earth's rotation. The Pope's decision caused turmoil at the time, because many people believed he was trying to take away 11 days of their lives.

21. August 1999 and August 99 are not the same. The year 99 is taken literally as the second-to-last year of the first century AD, not a two-digit form of the year 1999. Remember to be specific about the century when asking about the calendar.
24. Chances are that `banner` did not work when invoked by itself. You will probably have to specify its directory path, too—something like `$ /usr/games/banner . . .` or so on.

Chapter 4

6. Using the `-a` option results in a listing of all files in the directory, including the hidden files that are not normally displayed when using the `ls` command alone. Using the `-R` option results in a recursive listing of all nonhidden files in the directory structure, from the current working directory to the bottom.
8. Because the two are fairly new directories and have had little or no activity, both are sized at 1,024 bytes (that is, two times the size of the basic 512-byte directory building block).
10. Because the two are new zero-length files, both are sized at 0 bytes. Regarding inode numbers, because both were created at nearly the same time in the same part of the directory structure, their numbers should be similar and close but still different. Despite their zero-byte size, they are still given inode numbers. Meanwhile, your answer might vary according to the workstation or terminal.
13. The Access time and date will be changed any time the directory or file is so much as viewed. The Modify and Change dates and times will only be changed when directory contents or inode structure are changed.
14. The `rmdir` command does not work and returns the following:

```
rmdir: mydir: Directory not empty.
```

Chapter 5

21. The file is difficult to read because it is longer than one screen and scrolls past the screen very quickly. When you use the `more` or `less` commands, the file is then listed one screen at a time and you can

move ahead (with `less`, ahead *and* back) at your own pace. Therefore, it is a lot easier to read.

25. You cannot remove the directory because it is not empty. After you remove the files from *goodstuff*, you can remove the directory. Note that if a directory contains only the `..` and `.` special files, it is considered empty, so removal is not impeded.

28. The name of the command is `date`, and the command is found in */usr/bin*.

29. At the top of the `date` command's first page, you read that the information has been taken from the *sh-utils.info* file and that the node name is date invocation. The name of the next node is uname invocation.

Chapter 6

3. Creating zero-length files.

4. There is no difference in the file information. Both correspond to the same physical file; it is just that the file has two different filenames. You can check this information by executing `ls -li` to get the inode number for each filename. You will see that each file has the same inode number.

   An ordinary file has a link count of 1. A file with a single link to it has a link count of 2. Both the original `mycal` file and its link have link counts of 2.

5. The output is identical to the output of `mycal`, which is a display of the calendar of the current month. You cannot execute the `mycal` command. You will get a reply from Linux of `bash: myscripts/mycal: Permission denied`. Changing the permissions to read-only on the linked filename is the same as changing permissions on the file to the (original) file itself.

6. No, removing the linked filename does not remove the original file to which it is linked. Removing *home_mycal* simply removes the directory entry in the *myscripts* directory that refers to *homemycal* and changes the link count from 2 to 1. The *./myscripts/mycal* file remains intact. If you execute an `ls -l` on the original file, however, you will see that the link count is reduced.

9. You cannot execute `mycat` against the *.bash_profile* file because you have removed the user's `x` permission. Linux responds with `bash:./mycat: Permission denied`.

13. The simple `$ ls myscripts` works because the user still has read permission on the *myscripts* directory, and that is all that `ls` needs. `$ ls -l myscripts`, however, which requires both read and execute to work, does not work in this case because the `x` permission has been removed from that directory. Instead, you get

    ```
    ls: myscripts/mycat: Permission denied
    ls: myscripts/mycat: Permission denied
    ```

14. `mycal` will not execute because the `x` permission is needed to access any files, including executables, in a directory.

15. Both `x` and `w` permissions are needed to remove something from a directory.

Chapter 7

1. The output of `ls` gives the names of the non-hidden files and subdirectories in only the current working directory. The output of `ls *` gives the names of the files and subdirectories in the current working directory and all subdirectories recursively down the directory tree from the current working directory. `ls *` does not display hidden files, but `ls -a` does. `ls -a` does not display the contents of subdirectories, but `ls *` does.

5. You might have, because the command is case sensitive. If you have files or directories beginning with upper-case letters from `C` through `T`, they are listed as well as all files and subdirectories recursively below them, no matter what they begin with and no matter whether they are uppercase or lowercase.

6. All three methods work because the shell always disables wildcards, no matter what quoting method is used.

9. The output of `echo '* $n `ls`'` is `* $n `ls`` because single quotes disable everything between them.

   The output of `echo "* $n `ls`"` is `* hello filea` because double quotes disable only variable and command substitution variables.

The output of `echo \* \$n \`ls\`` is `* $n` `ls` because a backslash disables the character following it. Note that we used a backslash in front of each back quote.

The output of `echo * $n` `ls` is `filea hello filea`.

The output of `echo * $n "ls"` is `filea hello ls`.

16. No. The word count, which is essentially the number of files found in the directory, is higher in Exercise 16 by one because by the time the count is taken in Exercise 16, the file created in Exercise 14 (*temp*) is included.
17. Yes, the answer is what you expect and is identical to the file count in Exercise 14. The file created in Exercise 15, *temp*, was removed before Exercise 17—so the word count and file count numbers should be the same.
18. Yes, the number is displayed on the screen and is the same number as found in Exercises 13 and 10. And yes, *junk2* has the right filename listing.
22. No, they do not. The output from one does not have to be pipelined to any other. A minor relation might be drawn between the output of `pwd` and `ls` because the files and directories listed by `ls` depend on the current working directory. But regardless of where in the structure you find yourself, there will be some output from `ls`. Consequently, it is considered a minor relationship.

Chapter 8

6. We anticipated that you listed more entries that began with `l` than files. That is because the `ls` also includes subdirectories (which, in turn, includes all their contents, not just those that begin with `l`).

Chapter 9

8. You probably noticed that the number of coded messages displayed by `diff` coincides with the number of changes you were asked to make to *file2*. That is not always exactly the case, especially if you combine some changes or press Enter once or twice in addition to the typed changes.

10. All the answers should be approximately, if not exactly, the same and should be in the 155KB range.
12. Again, unless the speed of compression is changed in individual cases, all the answers should be approximately, if not exactly, the same. Based on default parameters, the percentage of compression should be about 72 percent. The name of the file is *mymagic.gz* unless the original file name was modified beforehand. The size of the newly compressed file—again, if no changes have been made to the speed of compression—should be approximately 45KB. Finally, there should be agreement on the percentage of compression: the size of the compressed file, compared with the size of the original uncompressed file, should reflect approximately 72 percent compression by `gzip`.
15. Again, even if the default speed of compression settings have been changed for whatever reason, the size of the newly uncompressed *mymagic* file should be the same as the original *mymagic* file.

Chapter 11

7. You must use double quotes here because of the space between the words.
8. You have to use the quotes around the right angle bracket. Otherwise, the shell interprets the symbol as a redirection.
10. Your home directory will have reverted to the default */home/username* directory. Why? Changing the variable from the command line sets the value for only the length of the Linux session, which ends when you log out. And, after you log out, the changed variable is removed from your shell variables. When you log back in, the shell adopts the home directory defined in your profile.
16. All the answers are "yes." The message displays at login (using the dot does not require logging in, so you would not have seen the message), the primary prompt reflects whatever directory you were in, and `dir` returns the same information as `ls -l`.
17. This time, the answers are "no" to all the questions. The subshell, which is a child process to the login shell, does not inherit the variables, aliases, and other functions. In fact, you get a `command not found` error message to your attempt at invoking `dir`. That is

why we move along to Exercise 4 to set up some additional functions so that child processes are as customized as the login shell.

20. Now, the variables, functions, and aliases also apply to the subshell. The answers to the questions are "yes."

Chapter 12

2. The `PID` of the subshell is different from the `PID` of the login shell because a child `PID` is always different from its parent `PID`. Check this fact by comparing a child `PID` with its `PPID`. They should be different, and the child's `PPID` should be identical to the parent's `PID`.
4. You will be logging out of the system. It is handy to check where you are from time to time so that you can avoid accidental logouts.
7. In the subshell, the value of `x` is "null" and is not shown in the `set` command listing.
10. No, because the subshell has its own environment and its variable values are not passed back to its parent process.
12. You find yourself back in the directory in which you started. When `sc1` is invoked, it is invoked in a subshell, which has its own environment. In the subshell, then, you as the user are moved into the *root* directory. But then the command finishes, the subshell closes (thus ending the visit to the *root* directory), and you are returned to the login shell process, which is in your original directory (*/home/username*).
14. The values of both `var1` and `var2` are "null" (that is, they have no value). Script `sc2` runs in a subshell. When it finishes, control is passed back to the parent process. But variables and values set in the subshell are not passed back to the parent process. The exporting performed in the subshell benefits child processes to that subshell but *not* to the parent of the subshell.
15. This time, the value for `var1` is `hello` and the value for `var2` is your login name. Because the values for the variables were set in the current shell and therefore in the current process environment, they remain current and verifiable by the `echo` command. Setting them with `sc2` is just like setting them at the command line by typing them in manually.

Chapter 14

1. The answers to these questions vary with each Linux distribution. In fact, they vary from version to version of any single Linux distribution. Some distributions use `fvwm` or `fvwm95`. Some use `KDE` or `GNOME`. Typically, the window manager automatically displays an `xterm` window of some sort, and the window is usually called something similar to `xterm`.

8. Generally, no items appear grayed (meaning unavailable for the particular window) because this menu is, after all, a window operations menu. Occasionally, though, you might notice that some items are unavailable in a window, depending on the state of the window or the commands that have already been invoked in or on the window.

10. No, they do not work unless the window operations menu is displayed. Otherwise, they appear as characters typed on a command line.

12. The `xterm` window expands to cover the entire root window, covering any other windows that might have been displayed with the exception of any `xclocks`, which stays on top of the `xterms`.

27. You are still operating on the remote client/host, so `xcalc` is executing on that system.

Chapter 15

1. According to the Web site, Linus Torvalds released version 1.0 in 1994. The answer to the second question depends on when you access the Web site but will likely appear in the sentence following the version 1.0 statement.

2. The first answer is promptly displayed on the Web site; the answer to the second question is displayed on the page accessed by clicking the `here` hyperlink on the Web site. The answer depends on the date you access the Web site.

3. The magazine's slogan is "The Premiere Linux Magazine."

4. The names of those items depend on when the Web site is accessed; they change every day. The first software package is named and

described immediately after the `freshmeat.net` banner; the editorials are accessed by left-clicking the editorials' hyperlink to the right of the same banner.

5. Again, the dates and titles of the most recent message depend on when you access the newsgroup or discussion. They are updated frequently, though typically not on a daily basis.

Quiz Answers

Chapter 3

1. Two methods are available. The first is deliberately specifying the `- help` option after the command name you want to investigate. The second method is accidentally: the result of using a command incorrectly. Linux returns with a preliminary diagnosis of your problem, followed by a quick summary of the command's proper usage.
2. `locate`. The other three commands use the same basic `man` database. With `locate`, you search the file system or a database path that you can specify.
3. `$ man commandname | lpr<Enter>`. Although this question might seem trivial, someday you might find yourself on another system wanting to print some `man` information. Remembering basic command sequences like this one might prove handy.

4. K Desktop Environment: `"Help"`

   ASCII/command line: `info`

   Fvwm95: `xman`

   We disguised the `KDE Help` command a bit. Including its full name would have been a giveaway.

5. `mail -f newmail`

   Remember, the correct syntax is

   ```
   $ command (-option) [argument]
   ```

6. `$ mail username`
7. `talk`, `write`, or `wall`
8. The calendar for the year 8 AD.
9. `who` or `finger username`. Both provide the last login time.

Chapter 4

1. Relative path name: *dulcinea/pgms/suba*; absolute path name: */home/dulcinea/pgms/suba*.
2. The single dot (`.`) specifies "with respect to the current directory," and the double dot (`..`) specifies "with respect to the parent directory."
3. Moves you up two directories in the directory tree structure. It is as if you instructed the system to "go to the parent directory of the current directory; now go to the parent directory of the parent directory."
4. The directory to be removed must be empty, and your current working directory must be at least one directory level higher than the directory to be removed.
5. `-l` provides a long listing of files.

   `-a` lists hidden files.

   `-R` lists subdirectories and their contents.

   `-i` displays the inode number.

   `-d` displays information about a directory.

6. The following message appears: `mkdir: cannot make directory 'test' : File exists`. The system has seen that a directory named `test` already exists and reports that back to the user. No further action is taken by the system.
7. The first command creates a subdirectory named */dir1* in */home/quixoted* and then creates in */dir1* a subdirectory named */dir2*. After that, a subdirectory named */dir3* is created in the */dir2* subdirectory .

   The second command creates three subdirectories named */dir1*, */dir2*, and */dir3*, all in */home/panzasan*, but they are not within one another.

Chapter 5

1. The first command makes */home/quixoted* the current directory. The second command tells Linux to copy the *file1* file and name the copy *file2*. Thus, each copy of the file has a unique name and inode number. Changes made to one copy of the file will not affect the other copy.
2. The first command makes */home/quixoted* the current directory. The second command tells Linux to rename the file, originally called *file1*, to *newfile*. The newly named file has the attributes of the original file.
3. The first command makes */home/quixoted* the current directory. The second command tells Linux to allow the file called *newfile* to be known also as *myfile*. An `ls -l` on the */home/quixoted* directory shows two files, one by each name, although only one file is physically on the disk. An `ls -li` on that directory substantiates that by showing that both files have the same inode number. Any changes made to *newfile* also show up if you use or access *myfile*.
4. The commands are `cat`, which scrolls continuously until the entire file has been displayed on the screen; `more`, which displays continuous text one screen at a time; and `less`, which is the same as `more` but allows the flexibility of moving both forward and backward in the file while it is displayed on the terminal screen.

5. The command that the `man` pages automatically invoke is `less`. You know this information because you are given the ability to move forward and backward to examine the contents of a `man` page.
6. *!*

   `aBcDe`

   `my_file`

   `my.file`

   `.myfile`

7. Starting and/or refreshing the `lpd` daemon is mandatory at this point. Otherwise, the printing configuration is really not complete. If the `printtool` utility has already been closed, go to the command line and enter

   ```
   # printtool<Enter>
   ```

   The `printconf-gui` dialog box/window will appear. Click the `File` drop-down menu and select `Restart lpd`.
8. These procedures have updated the */etc/printcap* file. It is best to use the `printtool` utility to update that file. Unless you are experienced with it, changing the file manually might invite trouble.

Chapter 6

1. `755`
2. `$ chmod go-x reporta<Enter>`
3. `$ chmod 744 reporta<Enter>`
4. Yes. He is a member of the same group as Perez (that is, `knights2`). That group has execute permission on the `jobs` directory as well as write permission on the *joblog* file.
5. No. Although his group, `knights2`, has write permission on the *joblog* file, they do not have execute permission on the `work` directory. As a member of `others`, Nicholas might have execute permission on the `work` directory, but he does not have execute permission on the *joblog* file.

6. Yes. The `knights2` group and `others` have execute permission on the two directories, *jobs* and *work*. The group also has read permission on the *joblog* file. Nicholas presumably has write and execute permissions on his own home directory.

Chapter 7

1. Redirection, command/variable substitution, wildcard expansion, and command execution.
2. This command lists all the files beginning with any three characters. The fourth character must not be in the range `a` to `z`. Then, any number of characters can follow, after which the second-to-last character must be from 0 through 9. The final character must be `t`.
3. Home directory is */home/quixoted*.
4. Home directory is *$HOME*. (Note that this question used single quotes.)
5. Current directory is */home/quixoted/docs*. (Note that `pwd` is enclosed in back quotes.)
6. Files in this directory are *.
7. `aa bb cc /home/quixoted`
8. *
9. Standard input (0) is the keyboard; standard output (1) is the screen; standard error (2) is the screen.
10. Standard input (0) is a file named *letter*; standard output (1) is handled by the `mail` program; standard error (2) is the screen.
11. Standard input (0) is the keyboard; standard output (1) is a file named *newprofile*; standard error (2) is a file named `1` (or 1, if you would rather. It is sometimes difficult to tell with some of these fonts. At any rate, it was a bit of a trick question: you might have thought that we were clumsily trying to indicate a redirection to standard output and that the ampersand was missing. We were not; we deliberately left the ampersand out.).
12. `$ cat filea > fileb 2> filec`
13. `$ cat filea > fileb 2>&1`
14. `$ cat filea > fileb 2> /dev/null`

15. Output the text string `hello` on the screen of terminal `tty1` (provided that `tty1` has not set `mesg n`).

Chapter 8

1. `$ find / -name 'mis*' -type f<Enter>`
2. List all processes that begin with the `/sbin` string in their respective command paths and that belong to the root user. The `-w` option is a bit of a red herring (or, if you are an Alfred Hitchcock movie fan, a "McGuffin"). The `-w` option simply ensures that searches are performed on the whole word so that usernames or processes that contain the respective strings, but which form only part of longer strings, are not included in the output.
3. A recursive long listing is carried out from the */home/username* directory downward through the file system. We will be looking for listings of lines that end in `um` or `isc` or that contain `ync`. (We are using `egrep` because we need to do a sort of OR `grep`. Then, we take those long-listed file entries (as lines) and pipe them to the `sort` command. The `sort` command sorts the lines in reverse order based on the eighth field (that is, on the filenames themselves). Some but not all results are presented. The display begins with the second line, and only seven lines are displayed. As you can surmise, this complex command might require you to continue the command from the first command line, with the primary `PS1` prompt, to the second command line with its secondary `PS2` prompt.

Chapter 9

1. False. Unfortunately, `find` travels only recursively down through a file system structure.
2. `find` itself. This time, the quoting metacharacters are used to keep the interpretation away from the shell.
3. `file` is the command that reports to you on the type of files you choose to examine.
4. False. Everything is true except that the extension is `.gz`, not just `.z`.

5. Use `cat -etv` to display all possible nonprintable characters, including end-of-line indicators, tabs, spaces, and other invisible characters.
6. True. `diff` compares text files only, and it is really the best choice for doing so. Use `cmp` to compare all types of files except text files.

Chapter 10

1. Command mode and Insert mode.
2. Press the Esc key. Remember, however, that Esc does not toggle Command mode `on` and `off`. If you press the Esc key repeatedly, you remain in Command mode and eventually hear an annoying error notification "beep."
3. `a` and `i`. The other two commands delete text.
4. False. The `u` command will undo only the last command. The `undo` buffer contains only one entry.
5. True. Adding a single `g` at the beginning of a search and replace command enables you to do that.
6. Watch out. This question is a trick question. You cannot quit `vi` while in Insert mode. Thus, the answer is "none of the above." To exit `vi`, you first have to get into Command mode by pressing Esc. Then, you can use `:x`, `:wq`, Shift-zz, or `:q!`.

Chapter 11

1. False. A listing of shell variables would contain all terminal environment variables, not vice-versa.
2. False. You do not create built-in variables. Moreover, uppercase versus lowercase is only a convention.
3. Variable substitution and command substitution.
4. Although both files are read at login, only the *.bashrc* file is read every time a child process (such as a subshell) is invoked. The *.bash_profile* generally contains environment variables, and *.bashrc* contains aliases and other functions.

5. False. You are not obligated to have a *.bashrc* file. But you will get some unexpected and probably undesirable results if you put all your variables, aliases, and functions in only your *.bash_profile* file.
6. True. All Linux processes are spawned from the `init` process, which originates during system bootup.
7. The */etc/profile* file is called the global profile because it contains all the variables that will apply to all users on the system. If users want to override the */etc/profile* variables or create variables of their own, they must modify their respective *.bash_profile* files.
8. Both are environment variables. `HISTSIZE` defines the maximum number of previous commands that the `bash` shell displays when you enter the `history` or `fc` commands. If you do not specify a `HISTSIZE`, the default value is 17 (other sources say 16; try this task for yourself). `HISTFILE` is a variable that defines the name of the file in which you want the text of all your previous commands to be deposited after you log out. The default filename is *$HOME/.bash_history*.
9. You can undo any alias temporarily with `unalias`. If the `alias` has been defined in your *$HOME/.bashrc* file, it will be read and invoked every time a child process is spawned and every time you log in. Consequently, if you only want to knock out an `alias` in your current shell for your current session, use `unalias`. Meanwhile, if you want to permanently knock out an `alias`, you must modify the appropriate files.
10. If you are going to properly customize your environment, keeping these files straight is important. The sentence goes as follows: "In the `bash` shell, */etc/bashrc* is to */etc/profile* as *$HOME/.bashrc* is to *$HOME/.bash_profile*."

Chapter 12

1. You use the `export` command.
2. The answer is 5, the value of the original login shell. Changing the value in the child process (in this case, the subshell) has no effect on the value of the same variable in the parent process.

3. Strictly speaking, this sentence is false. Each command returns an exit (or return) code after any attempt at completion. That is, a return code is given back to the parent process even if the command has failed.
4. `$ echo $$<Enter>` and `$ ps ef<Enter>`

   The `ps` command provides more information, but one of the parameters it displays is the process ID (PID) number for the various processes that are currently running.
5. The variable and its value become part of the current process environment. The variable and its value are available for any child process but cannot be passed back to the parent process.
6. Use the `ps f` command. The `f` option (or flag) shows the family tree of each running process.
7. True. Only the root user or superuser can control the jobs of other users.
8. Signal `-9` (also known as `KILL` in the `tcsh` shell or `SIGKILL` in the `bash` shell) cannot be caught or ignored, so it terminates a process at the point of execution—not allowing for a graceful shutdown of the process. Use `-9` with caution.
9. (1) The job will not lock up the user's terminal while it is running. (2) If the user wants to or has to log out of the system, the job continues running. Upon the user's return, the status of the job can be checked.
10. They are called daemons. They are usually started when the system is started and stop only when the system shuts down.
11. False. The user is always in the original directory if the command was started in the normal fashion (that is, without using the dot [.] command). Because the command executes in a subshell, any environment changes made in the subshell are not passed back to the parent process.

Chapter 13

1. Whenever you are trying to change variable values in your current environment and you are using the `bash` shell.
2. The correct answers are the second (`r` but not necessarily `w` or `x`); fourth (`755`); and fifth (`-rw-r-xr-x`). These show that the file needs

only the read permission. The first answer (`drwsxrwxrwx`) might seem right at first, but it refers to a directory. The third answer (`x` and `r`) is incorrect because of the word *and*. The last two answers are red herrings.

3. The recommended solutions are a) and c). It is considered best practice to define the variables inside the script and export them to make them available to subshells.
4. At the beginning of the script file, use the `#!/bin/bash` statement. This action will ensure that the script will run the `bash` shell environment no matter what shell the user is in.
5. The results of this script are false because "foo" does not equal "bar."
6. The results are still false, because although you can set and even export the variables T1 and T2 at the command line when the script executes, it will set them right back to T1="foo" and T2="bar". This situation is a common error in scripting, which is why you should always check the variables before you implement them.

Chapter 14

1. Statements b), d), and e) are true. Regarding statement a), connections are based on TCP/IP. Regarding statement c), it is the definition of a server in non-Linux/UNIX environments.
2. The XFree86 program (as configured by using the *XF86Config* file) with other software that helps XFree86 to control other functions and attributes of the X environment is the X Server for your system(s).
3. `export`, along with another command and other options and arguments. A case could be made for including `telnet` as well because it too can be used to connect to other machines. Then, once connected, a user can execute clients on the remote system.
4. `-bg red`: the background color of the new `xterm` window is red.

   `-fg black`: the foreground characters are white.

   `-T ledger`: the title of the window is `ledger`.

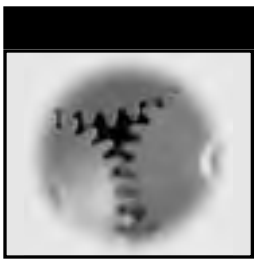
   `-geometry 80x125+0+0`: make the window 80 pixels wide and 125 pixels long and place it in the upper-left corner of the root window.

`-n lgr`: when the window is minimized, the title of its representative icon is `lgr`.

5. Two ways. (1) You can open another `xterm` window, and at the prompt type `su` and then supply the password. (2) You can left-click an open desktop area and then scroll down the resulting root menu and select `System Utilities`. On the drop-down menu called System Utilities Button Bar, scroll down and select `Root Shell`. An `xterm` window appears, titled Root_Window. Within this window is the root `Password:` prompt. Fill in the proper password, and you are given the `root # _` prompt.

Chapter 15

1. The best-known Linux Web site is Linux Online! at www.linux.org. The best-known X Window System Web site is www.XFree86.org.
2. This question is another "sort-of trick" question. It seems trivial, but his face is known throughout the world now. We mentioned it in the text and even provided the URL of his creator, Mr. Ewing's, Web site. And what is the penguin's name? Yes, his name is Tux.
3. The USENET newsgroups are a worldwide system of computers that store, update, and exchange collections of discussion text files. You can access them with specific software called newsreader software, or you can use an ordinary browser and go to Google Groups (at http://groups.google.com/googlegroups/deja_announcement.html) and use the search engines there. The site was formerly called Deja News (www.deja.com) until Google purchased it.
4. To quote from their manifesto, "The Linux Documentation Project is working on developing free, high-quality documentation for the GNU/Linux operating system."



Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z