CHAPTER 1 THE ESSENCE OF UNIX AND LINUX

UNIX and the UNIX look-alike system Linux both have something for everyone. For the everyday user, these are friendly systems that offer a huge array of commercial and free software, including a free office suite. For the programmer, UNIX and Linux are ideal for collaborative development of software because they offer powerful tools and utilities. For the system admin- istrator, UNIX and Linux contain time-tested and leading-edge tools for networking and multiuser management. In this book, you learn UNIX through the eyes of Linux. Linux is a modern operating system that has generated significant interest among all kinds of computer users—from general users to computer professionals. Also, it is a popular server system on the Web and in businesses. If you use Google to find something on the Internet, you are using a Linux Web server.

This chapter introduces you to operating systems in general and then explains the UNIX and Linux operating systems in particular. You also get an intro- duction to UNIX/Linux commands and command-line editing. As a variant of UNIX, Linux runs on PCs with Intel-type processors, but uses the same file

systems and commands as other UNIX versions. Linux can be run from an individual PC workstation or as a server operating system that is accessed through a network. When you access it through a network, you might use an old-fashioned UNIX terminal, a modern UNIX or Linux workstation, or a Windows-based workstation. Several versions of Linux are available, but this book uses Fedora, Red Hat Enterprise Linux, SUSE Linux, and Knoppix as examples (Knoppix examples are mentioned in the Hands-on Projects). These are among the most popular versions of Linux. Red Hat Enterprise Linux is a commercial version of Linux, and Fedora is a Red Hat-sponsored development project offering a free Linux version. SUSE Linux is sponsored by Novell in the commercial SUSE Linux Enterprise version and in the free openSUSE version. Knoppix is a free version of Linux that can be run from a CD/DVD and is well suited for educational use, but is also used in home and production environments. The commands and programming techniques you learn in this book can be applied to other UNIX and Linux versions.

You can learn more about Fedora at fedora.redhat.com. To find out more about Red Hat Enterprise Linux, go to www.redhat.com, and you can learn more about openSUSE Linux at en.opensuse.org/Welcome_to_openSUSE.org. For information about SUSE Linux Enterprise, visit www.novell.com.

UNDERSTANDING OPERATING SYSTEMS

An operating system (OS) is the most important program that runs on a computer. Operating systems enable you to store information, process raw data, use application software, and access all hardware attached to a computer, such as a printer or keyboard. In short, the operating system is the most fundamental computer program. It controls all the computer's resources and provides the base upon which application programs can be used or written. Figure 1-1 shows the relationship between an operating system and other parts of a computer system.

Different computer systems can have different operating systems. For example, the most common operating systems for desktop personal computers are Microsoft Windows, Mac OS, and Linux.Popular network andWeb server computer operating systems are MicrosoftWindows Server, UNIX/Linux, Novell NetWare, Novell Open Enterprise Server (which combines Netware and SUSE Linux Enterprise), and Mac OS X Server. Very large servers that are mainframe-class computers might use UNIX/Linux or the IBM z/OS operating system.

PC Operating Systems

A personal computer system, or PC, is usually a stand-alone machine, such as a desktop or laptop computer. A PC operating system conducts all the input, output, processing, and storage operations on a single computer. Figure 1-2 identifies some popular PC operating systems.

Server Operating Systems and Networks

Server operating systems are at the heart of a computer network. A computer network combines the convenience and familiarity of the personal computer with the ability to share files and other computer resources. With a network, you can share resources and exchange information with someone in the next room or on the other side of the world. Networked computers are connected by cables and through wireless communications.The Internet is one of the best examples of a network.

A server operating system controls the operations of a server computer, sometimes called a host computer, which accepts requests from user programs running on other machines, called clients. A server provides multiuser access to network resources, including shared files, hard disks, and printers. Figure 1-3 shows the relationship of a server and its clients on a network. Servers can be PC-type computers, clusters of PC-type computers working as one or several units, or mainframes. A mainframe is a large computer that has historically offered extensive processing, mass storage, and client access for industrial-strength computing. Mainframes are still in use today, but many have been replaced by PC-type computers that are designed as servers with powerful processing and disk storage capabilities—and cost considerably less than mainframes.

Server

Client Client Client Client

Figure 1-3 Relationship of a server and clients on a network

A server can be on a private or public network. For example, a server that stores legal files and accounting information in a law office is on a private network. A Web server,such as the one at www.redhat.com for Red Hat's Web site, is an example of a server on a public network.

In a centralized approach, the users' data and applications reside on the server. This type of network is called a server-based network. The system administrator secures all the information on the network by securing the server. The system administrator easily maintains the users' applications and performs backup operations directly on the server. If the server fails, however, clients cannot do their work until the server is returned to service.

Peer-to-peer networks, which are often used on small networks, are more distributed than server-based networks. In a peer-to-peer configuration, each system on the network is both a server and a client. There is no central server to manage user accounts; instead, each peer offers its own shared resources and controls access to those resources, such as through a workgroup of designated members or through accounts created on that peer workstation. Data and applications reside on the individual systems in the network. Software upgrades and backup operations must be performed locally at each computer. Security, which is implemented on each computer, is not uniform. Each user of the network is, to some degree, responsible for administering his or her own system. Despite the disadvantages a peer-to-peer network presents to the system administrator, the individual users do not depend on a central server. If one computer in the network fails, the other systems continue to operate.

INTRODUCING THE UNIX AND LINUX OPERATING SYSTEMS

UNIX and Linux are multiuser, multitasking operating systems with built-in networking functions. UNIX/Linux can be used on systems functioning as:

■ Dedicated servers in a server-based network ■ Client workstations connected to a server-based network ■ Client/server workstations connected to a peer-to-peer network ■ Stand-alone workstations not connected to a network

UNIX/Linux are multiuser systems, which let many people simultaneously access and share the resources of a server computer. Users must log in by typing their user name and a password before they are allowed to use a multiuser system. This validation procedure protects each user's privacy and safeguards the system against unauthorized use. UNIX and Linux are multitasking systems that allow one user to execute more than one program at a time. For example, you can update records in the foreground while your document prints in the background.

UNIX/Linux are also portable operating systems. Portability means these systems can be used in a variety of computing environments. In fact, they run on a wider variety of computers than any other operating system. They connect to the Internet, executing popular programs such as File Transfer Protocol (FTP), an Internet protocol used for sending files, and Telnet, an Internet terminal emulation program. A terminal emulation program is one that enables a PC to respond like a terminal (sometimes called a dumb terminal), which is a device that has a monitor and keyboard, but no CPU.

In addition to Telnet, most UNIX/Linux systems now employ Secure Shell (SSH), which is a form of authentication (a process of verifying that a user is authorized to access a computer) developed for UNIX/Linux systems to provide security for communications over a network, including FTP applications. You learn about SSH later in this chapter.

Many organizations choose UNIX and Linux because these operating systems: ■ Enable employees to work on a range of computers (portability) ■ Are stable, reliable, and versatile ■ Have thousands of applications written for them, both commercial and free ■ Offer many security options

■ Are well suited for networked environments (UNIX was one of the first server operating systems to be used on a network in the late 1960s.)

A Brief History of UNIX

A group of programmers at AT&T Bell Labs originally developed UNIX in the late 1960s and early 1970s. Bell Labs distributed UNIX in its source code form, so anyone who used

UNIX could customize it as needed. Attracted by its portability and low cost, universities began to modify the UNIX code to make it work on different machines. Eventually, two standard versions of UNIX evolved:AT&T Bell Labs produced SystemV (SysV),and the University of California at Berkeley developed Berkeley Software Distribution (BSD). Using features of both versions, Linux might be considered a more integrated version of UNIX than its predecessors. Currently, the Portable Operating System Interface for UNIX (POSIX) project,a joint effort of experts from industry,academia,and government, is working to standardize UNIX.

At this writing, Bell Labs is now part of Alcatel Lucent. For a review of the Bell Labs inventions that have had a profound impact on the world, including the UNIX operating system and the transistor, go to www.alcatel-lucent.com/wps/ portal/BellLabs/Top10Innovations. You can also learn more about Bell Labs and its discoveries at en.wikipedia.org/wiki/Bell_Labs.

For a more complete look at the history of UNIX, visit www.unix.org/what_is_ unix/history_timeline.html. You can also read an historic paper about UNIX by Dennis Ritchie at cm.bell-labs.com/cm/cs/who/dmr/hist.html. Dennis Ritchie has played key roles in the development of UNIX and the C programming language.

UNIX Concepts

UNIX pioneered concepts that have been applied to other operating systems. For example, Microsoft DOS (Microsoft's early PC operating system) and Microsoft Windows adopted original UNIX design concepts,such as the idea of a shell,which is an interface between the user and the operating system, and the hierarchical structure of directories and subdirectories.

The concept of layered components that make up an operating system also originated with UNIX. Layers of software surround the computer system's inner core to protect its vital hardware and software components and to manage the core system and its users. Figure 1-4 shows how the layers of a UNIX system form a pyramid structure.

At the bottom of the pyramid is the hardware. At the top are the users. The layers between them provide insulation, ensuring system security and user privacy. The kernel is the base operating system, and it interacts directly with the hardware, software services, application programs, and user-created scripts (which are files containing commands to execute). It is accessible only through Kernel mode, which is reserved for the system administrator. This prevents unauthorized commands from invading basic operating system code and hardware, resulting in actions that might hang or disrupt smooth operating system functions. User mode provides access to higher layers where all application software resides.

This layered approach, and all other UNIX features, were designed by programmers for use in complex software development. Because the programmers wrote UNIX in the C

Figure 1-4

Layers of a UNIX system

programming language, this operating system can be installed on any computer that has a C compiler. This portability, flexibility, and power make UNIX a logical choice for a network operating system. In addition, with the growth in popularity of Linux, more and more organizations are moving from UNIX and Windows to Linux.

Linux and UNIX

Linux is a UNIX-like operating system because it is not written from the traditional UNIX code. Instead, it is original code (the kernel) created to look and act like UNIX, but with enhancements that include the POSIX standards. Linus Torvalds, who released it to the public free of charge in 1991, originally created Linux. A number of organizations and companies now offer free and commercial distributions or versions of Linux. The following list is a sampling of Linux distributions:

■ Debian GNU/Linux (free, see www.debian.org) ■ Fedora (free, see fedoraproject.org) ■ Knoppix (free, see www.knoppix.org) ■ Mandriva (commercial and free versions, see www.mandriva.com) ■ Red Hat Enterprise Linux (commercial, see www.redhat.com)

■ openSUSE Linux (free, see en.opensuse.org/Welcome_to_openSUSE.org) ■ SUSE Linux Enterprise (commercial, see www.novell.com)

■ Turbo Linux (commercial and free versions, see www.turbolinux.com)

■ Ubuntu (free, see www.ubuntu.com)

Linux offers all the complexity of UNIX and can be obtained at no cost; or, for a relatively small amountofmoney,youcanpurchasecommercialversionsthathavespecializedtoolsandfeatures. With all the networking features of commercial UNIX versions, Linux is robust enough to handle large tasks. You can install Linux on your PC, where it can coexist with other operating systems, and test your UNIX skills. All these features make Linux an excellent way to learn UNIX, even when you have access to other computers running UNIX.

INTRODUCING UNIX/LINUX SHELLS

The shell is a UNIX/Linux program that interprets the commands you enter from the keyboard. UNIX/Linux provide several shells, including the Bourne shell, the Korn shell, and the C shell. Stephen Bourne at AT&T Bell Labs developed the Bourne shell as the first UNIX command processor. Another Bell employee, David Korn, developed the Korn shell. Compatible with the Bourne shell, the Korn shell includes many extensions, such as a history feature that lets you use a keyboard shortcut to retrieve commands you previously entered. The C shell is designed for C programmers' use. Linux uses the freeware Bash shell as its default command interpreter. Its name is an acronym for "Bourne Again Shell," and it includes the best features of the Korn and Bourne shells. No matter which shell you use, your initial communications with UNIX/Linux always take place through a shell interpreter. Figure 1-5 shows the role of the shell in UNIX/Linux.

If you use a graphical user interface (GUI) desktop (similar to Microsoft Windows with graphics and icons), which you learn about later in this chapter and in Chapter 11, then your communications occur through the GUI desktop. To use commands, you open a special window, called a terminal window, and your communications with the operating system occur through a shell interpreter within the terminal window. Most versions of UNIX and Linux that support using a GUI desktop offer a terminal window. This is a powerful feature because it is literally your window to using commands.

All of the commands that you learn in this book can be used in a terminal window or directly from the command line on a system that does not use a GUI desktop.

Choosing Your Shell

Before working with a UNIX/Linux system, you need to determine which shell to use as your command interpreter. Shells do much more than interpret commands: Each has extensive built-in commands that, in effect, turn the shells into first-class programming languages. (You pursue this subject in depth in Chapter 6, "Introduction to Shell Script Programming," and Chapter 7,"Advanced Shell Programming.") A default shell is associated with your account when it is created, but you have the option to switch to a different shell after you log in. Bash is the default shell in Linux, and it is the shell many users prefer. The following is a list of shells:

■ Bourne ■ Korn (ksh) ■ C shell (csh) ■ Bash ■ ash (a freeware shell derived from the Bourne and C shells) ■ tcsh (a freeware shell derived from the C shell) ■ zsh (a freeware shell derived from the Korn shell)

Switching from Shell to Shell

After you choose your shell, the system administrator stores your choice in your account record, and it becomes your assigned shell. UNIX/Linux use this shell any time you log in. However, you can switch from one shell to another by typing the shell's name (such as tcsh,

bash, or ash) on your command line. You work in that shell until you log in again or type another shell name on the command line. Users often use one shell for writing shell scripts (programs) and another for interacting with a program. (In Hands-on Project 1-8 later in this chapter, you learn how to switch shells, and in Chapter 7, you learn how to set your own default shell.)

CHOOSING USER NAMES AND PASSWORDS

Before you can work with UNIX/Linux, you must log in by providing a unique user name and password. Decide on a name you want to use to identify yourself to the UNIX/Linux system, such as "aquinn."This is the same name others on the UNIX/Linux system use to send you electronic mail. Some UNIX versions recognize only the first eight characters of a user name, but most versions of Linux, such as Fedora, Red Hat Enterprise Linux, and SUSE, recognize up to 32 characters.

You must also choose a password, which must contain six or more characters when using newer versions of UNIX/Linux, such as Fedora, Red Hat Enterprise Linux, and SUSE. The password should be easy for you to remember but difficult for others to guess, such as a concatenation of two or more words that have meaning to you—a combination of hobbies or favorite places, for example—written in a mix of uppercase and lowercase letters, numbers, and other characters. The password can contain letters, numbers, and punctuation symbols, but not control characters, such as Ctrl+x. (Control characters are codes that are a combination of the Control key and a letter, such as x, and that offer services to perform a specific action on a computer.)

The default minimum password length depends on your version of UNIX/Linux. Some earlier versions of Linux have a minimum length of five characters, but Fedora, Red Hat Enterprise Linux, and SUSE require a minimum length of six characters, which is the practice used in this book.

You can log in to any UNIX or Linux system as long as you have a user account and password on the workstation or host (server) computer. A UNIX/Linux system adminis- trator creates your account by adding your user name (also called a login name or user ID) and your password. You can change your password at any time by using the passwd command. You'll learn how to use the passwd command later in this chapter.

To use this book and the Hands-on Projects, you must have an account on a UNIX or Linux system along with some means to connect to that system. Some of the common ways to connect or to access a UNIX/Linux system are:

■ Through a Telnet or SSH connection to a remote computer, such as from another UNIX/Linux or aWindows-based operating system (Not all versions of Windows implement SSH, but you can obtain SSH from a third-party source, such as SSH Communications Security at www.ssh.com.)

■ Through client software on a UNIX/Linux client/server network ■ As a peer on a peer-to-peer, local area network in which each computer has the

UNIX/Linux operating system installed

■ On a stand-alone PC that has the UNIX/Linux operating system installed

■ Through a dumb terminal connected to a communications port on a UNIX/ Linux host

Appendix A, "How to Access a UNIX/Linux Operating System," describes several access methods, including how to set up and use Telnet or SSH. Also, see Appendix C, "How to Install Fedora and How to Use the Knoppix CD" for instructions on how to install the Fedora version of Linux on your computer and how to run Knoppix from the CD included with this book.

The steps you take to connect to a UNIX/Linux system vary according to the kind of connection you use. Connecting via a dumb terminal or accessing the OS through a stand-alone system are two of the easiest methods. In both cases, you need to log in to your account. Connecting by using client software for a client/server network might take special instructions or training from a network administrator.

If you connect on a peer-to-peer network, you can use Telnet or SSH. Connecting through Telnet or SSH are common methods and are described in the next section. You can use Telnet or SSH to access a UNIX/Linux peer or server computer over a local area network and through the Internet. Appendix A also discusses how to connect over a network using different methods.

CONNECTING TO UNIX/LINUX USING TELNET OR SSH

Telnet is a terminal emulation program. It runs on your computer and connects your PC to a server, or host, on the network. The PC from which you connect can be running UNIX, Linux, a Windows-based operating system, or Mac OS. You can then log in to a UNIX/ Linux host and begin working. Most UNIX/Linux versions include Telnet, as do most versions of Microsoft Windows and later versions of Mac OS.

Each computer on the Internet has an Internet Protocol (IP) address. An IP address is a set of four numbers (in the commonly used IP version 4) separated by periods, such as 172.16.1.61. Most systems on the Internet also have a domain name, which is a name that identifies a grouping of computer resources on a network. Internet-based domain names consist of three parts: a top-level domain (such as a country or organization type), a subdomain name (such as a business or college name), and a host name (such as the name of a host computer). An example using the three-part identification is research.campus.edu, in which “research” is the host name, “campus” is the subdomain name, and “edu” is the top-level domain. Both the IP address and the domain name identify a system on the network. Programs such as Telnet use IP addresses or domain names to access remote systems.

The general steps used to access a UNIX/Linux host via Telnet are: 1. Determine the remote host’s IP address or domain name. 2. Connect to your network or the Internet.

3. Start your Telnet program, and connect to the UNIX/Linux system. For example, to start Telnet in Windows XP or Vista, open a Command Prompt window, type telnet, and press Enter. To connect to Telnet in Fedora, Red Hat Enterprise Linux, or SUSE, open a terminal window, type telnet, and press Enter.

4. FollowtheinstructionsinyourTelnetprogramtoconnecttoaremotehost. Usually, you must provide the host name or IP address to connect to a UNIX/Linux system. For example, after the command prompt in a Windows 2000/XP/Server 2003/Vista Command Prompt window or at the UNIX/ Linux command line to access the system lunar.campus.edu, you can type the following command:

```
telnet lunar.campus.edu
```

Press Enter after you type the command.

5. Provide a user name and password to log in to the remote UNIX/Linux computer.

Secure Shell (SSH) was developed for UNIX/Linux systems to provide authentication security for TCP/IP applications, such as FTP and Telnet. Historically, the authentication for these applications has largely consisted of providing an unencrypted account and password, making both extremely vulnerable. SSH applies modern security techniques to ensure the authentication of a communications session. SSH can encrypt communications as they go across a network or the Internet.

In Fedora, Red Hat Enterprise Linux, SUSE, and other versions of UNIX/Linux, the ssh command can be used instead of telnet to establish a secure connection to a remote computer also running UNIX/Linux and that is compatible with openSSH. openSSH is a version of SSH that includes protocols and software intended for free distribution and which can be used on many UNIX/Linux systems.

To use ssh, you open a terminal window (or access the command line) and enter ssh -l on the command line along with the user account name and the name of the host computer. Two other options are to enter ssh with user@hostname or ssh with the IP address.

Hands-on Project 1-1 shows you how to useTelnet inWindows 2000/XP/Vista,and Hands-on Project 1-2 shows you how to access a terminal window in Fedora, Red Hat Enterprise Linux, or SUSE and use ssh to access a remote computer.

To use Telnet or SSH, you need to enable them on your system. See the note with Hands-on Project 1-1 to learn how to enable Telnet or SSH in different UNIX/Linux systems.

Logging In to UNIX/Linux

After you boot or connect to a UNIX/Linux system, you must log in by specifying your user name and password. You should see either a command line or a login dialog box if you are using a graphical user interface (GUI). For security, the password does not appear on the screen as you type it.

When you connect through the network or a dumb terminal, you log in and execute commands using a command-line screen. If you are on a stand-alone PC, the system might be configured to use only the command-line (text) mode, or it might be configured using a GUI. In UNIX/Linux, the foundation of a GUI is called the X Window interface. The X Window interface can have a different look and feel depending on what desktop environ- ment is used with it. The Fedora examples and figures in this book use the popular (and free) GNU Network Object Model Environment (GNOME) desktop. GNU stands for "Gnu's Not Unix," which was an endeavor started in 1983 to develop a free, open-standards, UNIX-like operating system (and additional operating system utilities). In the beginning, they were typically written in the C language.

To learn more about the GNOME Project, visit the Web site at www.gnome.org. Also, you learn more about the X Window interface and GNOME desktop in Chapter 11, "The X Window System."

You cannot log in without an authorized user account. If your password fails, or if you wait too long before entering your user name and password, contact your system administrator for help.

After you log in, you are ready to begin using the system. If you access UNIX/Linux through a network or a dumb terminal—or if your stand-alone system is configured for the command-line text mode—you can immediately enter commands at the command prompt. However, if you are using a stand-alone computer and an X Window desktop such as GNOME, you must open a terminal window (see Figure 1-6) to access the command prompt. Hands-on Project 1-2 demonstrates how to access a terminal window in Fedora, Red Hat Enterprise Linux, or SUSE (to execute the ssh command).

USING COMMANDS

To interact with UNIX/Linux, you enter a command, which is text you type after the command prompt. When you finish typing the command, press Enter. UNIX/Linux are case sensitive; that is, they distinguish between uppercase and lowercase letters, so John differs from john. You type most UNIX/Linux commands in lowercase.

Commands are divided into two categories: user-level commands that you type to perform tasks, such as retrieve information or communicate with other users, and system- administration commands, which the system administrator uses to manage the system.

You must know a command's syntax to enter it properly. Syntax refers to a command's format and wording, as well as the options and arguments you can use to extend and modify its functions. Most commands are single words, such as the command clear. If you enter a command using correct syntax, UNIX/Linux execute the command. Otherwise, you receive a message that UNIX/Linux cannot interpret your command.

Appendix B, "Syntax Guide to UNIX/Linux Commands," alphabetically lists all the commands in this book and tells you how to enter each command and use its options.

The place on the screen where you type the command is called the command line (refer to Figure 1-6). Commands use the following syntax:

Syntax command_name [-option] [argument]

Dissection

- The command_name specifies what operation to perform. In the syntax illustrations in this book, command names appear in boldface. (In regular text, command names appear in italic.)

- Command options are ways to request that UNIX/Linux carry out a command in a specific style or variation. Options follow command names, separated by a space. They usually begin with a hyphen (-). Options are also case sensitive. For example, -R differs from -r. You do not need to type an option after every command; however, some commands do not work unless you specify an option. The syntax illustrations in this book list options in square brackets ([ ]) when the command does not require them.

■ Command arguments follow command options, separated by white space (blank space). Command arguments are usually file and directory names. In the syntax illustrations in this book, arguments appear in italic. Square brackets surround arguments if the command does not require them.

In the following sections, you start your journey learning UNIX/Linux with an introduc- tion to the following basic commands:

■ date ■ cal ■ who ■ clear ■ man ■ whatis

You also learn command-line editing techniques, how to enter multiple commands, and how to recall commands you've used previously. And, you learn how to log out of an active session.

The date Command

The UNIX/Linux date command displays the system date, which the system administrator maintains (see Figure 1-7). Because the date and time on a multiuser system are critical for applications, only the system administrator can change the date. For example, the Accounting Department might need to associate a date with a specific file used for reporting tax information, or the Publications Department might have to date stamp a document to ensure a specific copyright date.

The date command has an option,-u,which displays the time in Greenwich MeanTime (GMT).

GMT is also known as Greenwich Meridian Time and Coordinated Universal Time (UTC). UTC is considered the international time standard. To learn more about UTC, visit NASA's Web page at www.ghcc.msfc.nasa.gov/utc.html.

Hands-on Project 1-3 enables you to use the date command.

Syntax date [-option]

Dissection

■ Displays the system date and time ■ Commonly used options include:

-u view Greenwich Mean Time -s to reset the date or time

The cal Command

Use the cal command to show the system calendar. This command can be useful for scheduling events or determining a specific date of a project you completed in the past or

intend to complete in the future. The cal command can also be used to determine the Julian date, by using cal -j at the command line. The Julian date is the number of the date from the beginning of the year, and is a value between 1 and 366 (including leap year). Programmers sometimes use the Julian date for specific programming functions, such as determining the number of days an employee has worked in an organization for the current year. Figure 1-8 shows the results of the command cal -j 2009, showing the Julian date for monthly calendars in 2009. Hands-on Project 1-4 enables you to use the cal command.

Figure 1-8 Using the cal command to determine the Julian date

Syntax cal [-option]

Dissection

■ ■

Generates a calendar for the current year or for a year specified by the user Commonly used options include:

-j for Julian date -s to show Sunday as the first day in the week -m to show Monday as the first day in the week -y to show all of the months for the current year

The who Command

To determine information about who is logged in, use the who command. In a multiuser system, knowing who is logged in to the system is important for the administrator, so the administrator can periodically verify authorized users and levels of use.

Knowing who is logged in is also valuable for ordinary users, who can use that information to judge how busy the system is at a given time or who might want to contact another user.

Syntax who [-option]

Dissection

■ ■

Provides a listing of those logged in to the operating system Commonly used options include:

am i (type who in front, as in who am i) for information about your own session whoami (type whoami as all one word) to see what account you are using

-H to show column headings -u to show idle time for each user (the older -i is being retired from the who options) -q for a quick list and total of users logged in -b used by system administrators and others to verify when the system was last booted

Try Hands-on Project 1-5 to learn how to use the who command.

The clear Command

As you continue to enter commands, your screen might become cluttered. Unless you need to refer to commands you previously entered and to their output, you can use the clear command to clear your screen. It has no options or arguments.

You use the clear command in Hands-on Project 1-6.

Syntax clear

■ Clears the terminal screen, display, or terminal window

The man Program

For reference, UNIX/Linux include an online manual that contains all commands, including their options and arguments. The man program in UNIX/Linux displays this online manual, called the man pages, for command-line assistance. Although the man pages for some commands contain more information than others, most man pages list the following items:

■ Name—The name of the command and a short statement describing its purpose

Synopsis—A syntax diagram showing the usage of the command ■ Description—A more detailed description of the command than the name item

gives as well as a list of command options and their descriptions

■ Author—The name or names of the author or authors who developed the command or program (if available)

■ Reporting Bugs—The information about how to report bugs or problems ■ History—The information that is sometimes included to show where the com-

mand originated

■ Other Versions—The information that is sometimes included to indicate there are other versions of the command available

■ See Also—The other commands or man pages that provide related information

The man program usually accepts only one argument—the name of the command about which you want more information. The online manual shows the valid command formats that your system accepts. To close the online manual, type q.

Syntax man [-option] argument

Dissection

■ ■

■

Shows information from the online documentation Example options include:

-d to print information for debugging -f displays a short description of a command (produces the same information as using the whatis command described in the next section) -K to find a certain string by searching through all of the man information

The argument is to supply the name of the command or program you want to learn more about, such as man who

As an example, consider the man pages for the cal command, as shown in Figures 1-9 and 1-10. In this example, the top line shows the name of the command, which is cal, and a brief description, which is "displays a calendar." Next, the Synopsis section provides information about the way in which the command is used, showing that it can be used with options, such as any of -smjyl3, as well as specifying the month or year as arguments. The Description section provides more information about the purpose of the cal command and explains the default usage, such as that the current month is displayed if there are no arguments used. The Description section also shows the options, such as -s to display the calendar starting with Sunday.

In Figure 1-10, more information about the use of the cal command appears at the end of the Description section. The History section shows that this command appeared inVersion 6 AT&T UNIX. Finally, the OtherVersions section contains information about other versions of cal that you can obtain and use,and at the end a date shows when this man page was written or updated.

Many systems also offer info pages in addition to the man pages. Sometimes the info pages provide more information about commands, and sometimes the man pages do. Further, a new command might only be covered in man or info. If you need help with a particular command, consider checking both sources. For example, to find out about the who command, enter info who.

The whatis Command

Sometimes you find that the man pages contain more information than you want to see. To display a brief summary of a command, use the whatis command (see Figure 1-11). The whatis command shows only the name and brief description that appears near the top of a command's man page.

Figure 1-11 Using whatis for a quick summary of the cal command

The whatis command relies on information stored in a database. On some UNIX/Linux systems, the administrator must execute the whatis command, which creates the database, before the whatis command operates properly. In Fedora and Red Hat Enterprise Linux, log in to the root account and type /usr/sbin/makewhatis to create the database—although recent versions of these operating systems and of SUSE Linux already come with the database. (SUSE Linux does not include makewhatis in the /usr/sbin directory.)

Syntax whatis argument

Dissection

■ Displays the short descriptions of commands as obtained from a whatis database

■ In many UNIX/Linux versions, including Fedora and Red Hat Enterprise Linux, the whatis command only takes an argument (the name of a command or program) and there are no options. In SUSE, whatis offers several options. One important option is -w to search the whatis database using a wildcard in the spelling, such as * and ? when using the Bash shell (for example, enter whatis -w ma? when searching for man). Another useful option is -m system to enable SUSE to search the whatis database on a different UNIX/Linux system on the same network, such as on a BSD UNIX computer.

Hands-on Project 1-7 enables you to use the man and whatis commands.

Command-line Editing

Shells support certain keystrokes for performing command-line editing. For example, Bash (which is the default Linux shell) supports the left and right arrow keys, which move the cursor on the command line. For instance, if you misspell a command or argument, you can use the left and right arrows to move around on the active command line to the misspelling, correct it, and then execute the command—all without retyping it. Other keys, used in combination with the Alt or Ctrl key, are used for other editing operations, and the Del key is used to delete a character. Table 1-1 illustrates common Alt, Ctrl, and Del key combinations you can use for command- line editing. Also, try Hands-on Project 1-8 to practice editing on the command line.

Table 1-1 Common Alt, Ctrl, and Del key combinations for command-line editing

Key Combination	Description
Ctrl+b	Moves the cursor to the previous letter
Alt+d	Deletes a word or consecutive characters
Alt+l	Moves the cursor to the position just before the first character of the next word
Ctrl+a	Moves the cursor to the beginning of the command line
Ctrl+k	Deletes the content of the command line from the current cursor position to the end of the command line
Del	Deletes a character

Not all shells support command-line editing in the same manner. Table 1-1 applies to the Bash shell in Linux.

Hands-on Project 1-8 enables you to practice command-line editing, and to determine whether you are employing the Bash shell.

Multiple Command Entries

You can type more than one command on the command line by separating commands with a semicolon (;). When you press Enter, the commands execute in the order in which you entered them. For example, if you type date ; cal, you see today's date and then the calendar for the current month. As you learn in later chapters, this is an important capability for completing several operations at a time, such as working on the data in a file and then printing or displaying specific data. Try Hands-on Project 1-9 to execute multiple com- mands from a single command-line entry.

The Command-line History

Often, you find yourself entering the same command several times within a short period of time. Most shells keep a list of your recently used commands and allow you to recall a command without retyping it. You can access the command history with the up and down arrow keys. Pressing the up arrow key once recalls the most recently used command. Pressing the up arrow key twice recalls the second most recently used command. Each time you press the up arrow key, you recall an older command. Each time you press the down arrow key, you scroll forward in the command history. When you locate the command you want to execute, press Enter. This capability can save time and frustration when you need to enter the same or similar commands in one session. Hands-on Project 1-10 enables you to use the command-line history capability.

Logging Out of UNIX/Linux

When you finish your day's work or leave your computer or terminal for any reason, log out of the UNIX/Linux system to ensure security. Logging out ends your current process and indicates to the OS you are finished. How you log out depends on the shell you are using. For the Bourne, Korn, or Bash shells, enter exit on the command line or press Ctrl+d. In the C shell, enter logout on the command line. These commands log you out of your system, if you are not using a desktop environment.

If you are working in an X Window desktop environment, such as GNOME, typing exit and pressing Enter or using Ctrl+d only closes the terminal window. To log out, use the Log Out option for the desktop. For example, if you are using GNOME in Fedora or Red Hat Enterprise Linux, click the System menu in the Panel at the top of the screen, click Log Out username, and click Log Out to verify that is what you want to do. In openSUSE version 10.2 and higher, click the Computer menu in the Panel at the bottom of the screen, click Log Out, click Log out on the next menu, and click OK. In SUSE versions up through 10.0 with the GNOME desktop, click the Desktop menu in the Panel at the top of the screen, click Log Out, click Log out on the next menu, and click OK.

UNDERSTANDING THE ROLE OF THE UNIX/LINUX SYSTEM ADMINISTRATOR

There are two types of users on a UNIX/Linux system: system administrators and ordinary users. As the name suggests, a system administrator manages the system by adding new users, deleting old accounts, and ensuring that the system performs services well and efficiently for all users. Ordinary users are all other users. The system administrator is also called the superuser, because the system administrator has unlimited permission to alter the system. UNIX/Linux grant this permission when the operating system is initially installed. The system administrator grants privileges and permissions to ordinary users.

The system administrator has a unique user name: root. This account has complete access to a UNIX/Linux system. The password for the root account is confidential; only the system administrator and a backup person know it. If the root's password is lost or forgotten, the system administrator uses an emergency rescue procedure to reset the password.

The System Administrator's Command Line

Although ordinary users type their commands after the $ (dollar sign) command prompt, the system administrator's prompt is the # (pound) symbol. The UNIX/Linux system generates a default setting for the command prompt for the system administrator in the following format:

[root@hostname root]#

In the prompt, hostname is the name of the computer the system administrator logged in to. On some computers the hostname is simply localhost to refer to the local computer, or localhost is used when the computer does not have a name. Besides the reference to the root account, there are two other meanings of root which you learn more about later in this book. One meaning is the base level for all directories and another is the default home directory for the root account.

When you use the GNOME terminal window in SUSE, the user's prompt is a right-pointing arrow with the account name and computer name (or operating system name) appearing before the arrow, such as mpalmer@aspen: → (where mpalmer is the user name and aspen is the computer name). Also, the system administrator's prompt in SUSE consists of the computer name, a colon, a tilde (for the current directory), and the pound sign, such as aspen: ~ #.

The Ordinary User's Command Line

The $ (dollar sign) (or the right-pointing arrow in SUSE) is traditionally associated with ordinary users. The UNIX/Linux system generates a default setting for the command prompt for ordinary users. The following formats are common on Linux systems:

[username@hostname username] $ [username@hostname ~] $ username@hostname: →

In the prompt, username is the user's login name, such as jean, and hostname is the name of the computer to which the user is logged in (or localhost is used). In the first example of the

prompt shown above, the second instance of username refers to the name of the user's home directory (which by default has the same name as the user name). When a tilde (~) is used, this also refers to the user's home directory (in this instance); and if you change directories, the name of the directory you change to is shown instead of the tilde.

CHANGING PASSWORDS

Your user name, or login name, identifies you to the system. You can choose your own user name and give it to the system administrator, who then adds you as a new user. As mentioned earlier, some UNIX/Linux versions recognize up to eight characters, while other versions, such as Fedora, Red Hat Enterprise Linux, SUSE, and Knoppix, recognize up to 32 characters in your user name—which is often your first initial and last name, your last name and first initial, your last name, or sometimes a nickname.

A user name is unique but not confidential, and can be provided to other users. The password, on the other hand, is confidential and secures your work on the system. You can change your password, if necessary, by using the passwd command, but you must know your current password to change it. If your account does not have a password, use the passwd command to create one.

Syntax passwd [-option] [argument]

Dissection

Used by an account owner or system administrator to change a password.

-e used by the system administrator to expire a password so the user has to create a new password the next time she logs in

-l locks an account and is used by the system administrator -S typically used by a system administrator to view the password status of an account, such

as to ensure that an account has a password

UNIX/Linux system administrators can apply rules for passwords, such as they must have a minimum number of characters, contain a combination of letters, numbers, and other characters, or cannot be the same as recent passwords you have used. Hands-on Project 1-11 enables you to change your password.

After changing your password, you should log out and log in again to make certain UNIX/Linux recognizes your new password.

Remember your password! You need your password every time you log in to UNIX/Linux. For more information about the passwd command and for tips about keeping your password secure, enter the man passwd command.

VIEWING FILES USING THE CAT, MORE, LESS, HEAD, AND TAIL COMMANDS

Three UNIX/Linux commands allow you to view the contents of files: cat, more, and less. The more and less commands display a file one screen at a time. The more command scrolls only down, whereas less enables you to scroll down and up. The cat command displays the whole file at one time. Two other commands, head and tail, allow you to view the first few

or last few lines of a file (ten lines by default).

The cat command gets its name from the word concatenate, which means to link. You can display multiple files by entering their file names after the cat command and separating them with spaces. UNIX/Linux then display the files' contents in the order in which you entered them.

Try Hands-on Projects 1-12, 1-13, and 1-14 to use the cat command, the more and less commands, and, finally, the head and tail commands.

REDIRECTING OUTPUT

In UNIX/Linux, the greater-than sign (>) is called an output redirection operator.You can use this redirection operator to create a new file or overwrite an existing file by attaching it to a command that produces output. In effect, you redirect the output to a disk file instead of to the monitor. For example, if you type who > current_users and press Enter, this creates a file called current_users that contains the information from the who command.

Redirecting output is useful in many circumstances. For example, when you monitor a system, you might redirect output to a file that you can examine later. Or, you might have a program or report-generating utility that manipulates data so that you can redirect the results to a file instead of to the screen. You learn about other redirection operators later in this book, but, for now, the > operator is a good starting point.

Try Hands-on Projects 1-15 and 1-16 to use the > output redirection operator.

You can also use the cat command combined with the output redirection operator to create files from information you type at the keyboard. Type cat > filename after the command prompt, where filename is the name of the file you are creating. Enter the data in the file, and then press Ctrl+d to end data entry from the keyboard. Hands-on Project 1-17 enables you to use the cat command with the > redirection operator.

Use the output redirection operator (>) to send output to a file that already exists only if you want to overwrite the current file. To append output to an existing file, use two redirection operators (>>). This adds information to the end of an existing file without overwriting that file.

CHAPTER SUMMARY

The operating system is the most fundamental computer program. It controls all computer resources and provides the base upon which application programs can be used or written.

In a centralized server-based network, all the users' data and applications reside on the server, which is secured, maintained, and backed up by the system administrator. Each computer in a server-based network relies on the server. All systems in a peer-to-peer network function as both server and client. The security and maintenance of the network are distributed to each system. If one of the systems in a peer-to-peer network fails, the other systems continue to function.

UNIX/Linux operating systems are multiuser systems, enabling many people to access and share the computer simultaneously. These are also multitasking operating systems, which means they can perform more than one task at one time.

UNIX/Linux systems can be configured as dedicated servers in a server-based network, client workstations in a server-based network, client/server workstations in a peer-to- peer network, or stand-alone workstations not connected to a network.

The concept of the layered components that make up an operating system originated with UNIX. Layers of software surrounding the computer system's inner core protect the vital hardware and software components and manage the core system for users.

Linux is a UNIX-like operating system that you install on your PC. It can run alone or it can coexist with other operating systems such as Windows. Like UNIX, Linux is portable, which means it runs on computers from PCs to mainframes.

In UNIX/Linux, you communicate with the operating system programs through an interpreter called the shell, which interprets the commands you enter from the keyboard. UNIX/Linux provide several shell programs, including the Bourne, Korn, and C shells. The Bash shell provides enhanced features from the Bourne and the Korn shells. It is the most popular shell on the Linux system.

In UNIX/Linux, the system administrator sets up accounts for ordinary users. To set up your account and to protect the privacy and security of the system, you select and give the system administrator your user name and password. You can log in to any UNIX or Linux system anywhere as long as you have a user account and password on the host (server) computer. You can also use UNIX/Linux, Microsoft Windows, and Mac OS Telnet or SSH programs to log in to a remote UNIX/Linux system.

The commands you type to work with UNIX/Linux have a strict syntax that you can learn by referring to the online manual called the man pages. Use the man program to display the syntax rules for a command. Use the whatis command to see a brief description of a command. Use the who command to list who is logged in and where they are located. Use the cal command to display the system calendar for all or selected months. Use the passwd command to change your account's password. To log out when you decide to stop using UNIX/Linux, use the exit or logout command from a system that does not use a GUI. Or, on a GUI-based system, use the Log Out option that you access from the menus.

Most shells provide basic command-line editing capabilities and keep a history of your most recently used commands. Use the up and down arrow keys to scroll backward and forward through the list of recently used commands. You can enter multiple commands on a single command line by separating them with a semicolon. UNIX/Linux execute the commands in the order in which you enter them.

You can use the "view" commands to examine the contents of files. Use the cat command to create a file by typing information from the keyboard. Use the less and more commands to display multipage documents. Use the head and tail commands to view the first or last few lines of a file.

COMMAND SUMMARY: REVIEW OF CHAPTER 1 COMMANDS

cal

Shows the system calendar

-j displays the Julian date format. -s shows Sunday as the first day in the week. -m shows Monday as the first day in the week. -y shows all of the months for the current year.

Cat Displays multiple files

-n displays line numbers.

Clear Clears the screen

date Displays the system date.

-u displays the time in Greenwich Mean Time. -s resets the date and time.

exit or logout Exits UNIX/Linux when a GUI is not used

head Displays the first few lines of a file

-n displays the first n lines of the specified file.

less Displays a long file one screen at a time, and you can scroll up and down

man Displays the online manual for the specified command

-d prints information for debugging. -f gives a short description of the command (same as using the whatis command)

-K finds a certain string by searching through all of the man information.

more Displays a long file one screen at a time, and you can scroll down

passwd Changes your UNIX/Linux password

-e expires a password causing the user to have to re-create it -l locks an account -S displays the password status of an account

tail Displays the last few lines of -n displays the last n lines of the a file specified file.

Whatis Displays a brief description of a command

who Allows you to see who is logged in (also whoami shows the

account currently logged in and who am i displays information about the account session)

-H displays column headings. -u displays session idle times. -q displays a quick list of users. -b verifies when the system was last booted.

KEY TERMS

argument —Text that provides UNIX/Linux with additional information for executing a command. On the command line, an argument name follows an option name, and a space separates the two. Examples of arguments are file and directory names. authentication — The process of verifying that a user is authorized to access a particular computer, server, network, or network resource, such as Telnet or FTP.

Bash shell — A UNIX/Linux command interpreter (and the default Linux shell). Incorporates the best features of the Bourne shell and the Korn shell. Its name is an acronym for "Bourne Again Shell." Berkeley Software Distribution (BSD) — A distribution of UNIX developed through the University of California at Berkeley, which first distributed the BSD UNIX version in 1975. Bourne shell — The first UNIX/Linux command interpreter, developed at AT&T Bell Labs by Stephen Bourne.

C shell — A UNIX/Linux command interpreter designed for C programmers.

case sensitive — A property that distinguishes uppercase letters from lowercase letters—for example, John differs from john. UNIX is case sensitive.

client — A computer on a network running programs or accessing files from a mainframe, network server, or host computer.

command —Text typed after the command-line prompt which requests that the computer take a specific action.

command line — The onscreen location for typing commands.

domain name — A name that identifies a grouping of computer resources on a network. Internet-based domain names consist of three parts: a top-level domain (such as a country or organization type), a subdomain name (such as a business or college name), and a host name (such as the name of a host computer).

File Transfer Protocol (FTP) — An Internet protocol for sending and receiving files.

graphical user interface (GUI) — Software that transforms bitmaps into an infinite variety of images, so that when you use an operating system you see graphical images.

host — See server.

Internet Protocol (IP) address — A set of four numbers (for the commonly used IP version 4) separated by periods—for example, 172.16.1.61—and used to identify and access.

kernel — The basic operating system, which interacts directly with the hardware and services user programs.

Kernel mode — A means of accessing the kernel. Its use is limited to the system administrator to prevent unauthorized actions from interfering with the hardware that supports the entire UNIX/Linux structure.

Korn shell — A UNIX/Linux command interpreter that offers more features than the original Bourne shell. Developed by David Korn at AT&T Bell Laboratories.

log in — A process that protects privacy and safeguards a multiuser system by requiring each user to type a user name and password before using the system.

mainframe — A large computer that has historically offered extensive processing, mass storage, and client access for industrial-strength computing. Mainframes are still in use today, but many have been replaced by PC-type computers that are designed as servers with powerful processing and disk storage capabilities.

man pages — The online manual pages for UNIX/Linux commands and programs that can be accessed by entering man plus the name of the command or program.

multitasking system — An operating system that enables a computer to run two or more programs at the same time.

multiuser system — A system in which many people can simultaneously access and share a server computer's resources. To protect privacy and safeguard the system, each user must type a user name and password in order to use, or log in to, the system. UNIX and Linux are multiuser systems.

network — A group of computers connected by network cable or wireless communications to allow many users to share computer resources and files. It combines the convenience and familiarity of the personal computer with the processing power of a mainframe.

operating system (OS) — The most fundamental computer program, it controls all the computer's resources and provides the base upon which application programs can be used or written.

options — The additional capabilities you can use with a UNIX/Linux command.

ordinary user — Any person who uses the system, except the system administrator or superuser.

output redirection operator —The greater-than sign (>) is one example of a redirection operator. Typing > after a command that produces output creates a new file or overwrites an existing file and then sends output to a disk file, rather than to the monitor.

peer-to-peer network — A networking configuration in which each computer system on the network is both a client and a server. Data and programs reside on individual systems, so users do not depend on a central server. The advantage of a peer-to-peer network is that if one computer fails, the others continue to operate.

personal computer (PC) — A single, stand-alone machine, such as a desktop or laptop computer, that performs all input, output, processing, and storage operations.

portability — A characteristic of an operating system that allows the system to be used in a number of different environments, particularly on different types of computers. UNIX and Linux are portable operating systems.

Portable Operating System Interface for UNIX (POSIX) — Standards developed by experts from industry, academia, and government through the Institute of Electrical and Electronics Engineers (IEEE) for the portability of applications, including the standardiza- tion of UNIX features.

root — The system administrator's unique user name; a reference to the system adminis- trator's ownership of the root account and unlimited system privileges. Also, root has two other meanings: (1) the basis of the treelike structure of the UNIX/Linux file system and the name of the file (root directory) located at this level and (2) the home directory for the root account.

Secure Shell (SSH) — A form of authentication developed for UNIX/Linux systems to provide authentication security for TCP/IP applications, including FTP and Telnet.

server — The computer that has a network operating system and, as a result, can accept and respond to requests from user programs running on other computers (called clients) in the network. Also called a host.

server-based network — A centralized approach to networking, in which client com- puters' data and programs reside on the server.

server operating system — An operating system that controls the operations of a server or host computer, which accepts and responds to requests from user programs running on other computers (called clients) on the network.

shell — An interface between the user and the operating system. superuser — See system administrator.

System V (SysV) — A version of UNIX originating from AT&T Bell Labs and first released as System 3 in the early 1980s as a commercial version of UNIX. Today, commercial and free versions based on SystemV are available.

syntax — A command's format, wording, options, and arguments.

system administrator — A user who has an account that can manage the system by adding new users, deleting old accounts, and ensuring that the system performs services well and efficiently for all users.

Telnet — An Internet terminal emulation program.

terminal — A device that connects to a server or host, but consists only of a monitor and keyboard and has no CPU. Sometimes called a dumb terminal.

terminal window — A special window that is opened from a UNIX or Linux GUI desktop and that enables you to enter commands using a shell, such as the Bash shell.

User mode — A means of accessing the areas of a system where program software resides.

CHAPTER 2 EXPLORING THE UNIX/LINUX

FILE SYSTEMS AND FILE SECURITY

An essential reason for deploying UNIX/Linux is to store and use information. A file system enables you to create and manage information, run programs, and save information to use later. Through a file system, you can protect information and programs to ensure that only specific users have access. You can also copy information from one location to another, and you can delete information you are no longer using.

In this chapter, you explore UNIX/Linux file systems, including the basic concepts of directories and files and their organization in a hierarchical tree structure. You learn to navigate the file system, and then you practice what you've learned by creating directories and files and copying files from one directory to another. You also have the opportunity to set directory and file permissions, which is vital for security in a UNIX/Linux multiuser system.

UNDERSTANDING UNIX/LINUX FILE SYSTEMS

In UNIX/Linux, a file is the basic component for data storage. UNIX/Linux consider everything with which they interact a file, even attached devices such as the monitor, keyboard, and printer. A file system is the UNIX/Linux system's way of organizing files on storage devices, such as hard disks and CDs or DVDs. A physical file system is a section of the hard disk that has been formatted to hold files. UNIX/Linux consist of multiple file systems that form virtual storage space for multiple users. Virtual storage in this sense is storage that can be allocated using different disks or file systems (or both), but that is transparently accessible as storage to the operating system and users. The file system's organization is a hierarchical structure similar to an inverted tree; that is, it is a branching structure in which top-level files contain other files, which in turn contain other files. Figure 2-1 illustrates a typical UNIX/Linux hierarchical structure.

One reason why UNIX and Linux systems are so versatile is that they support many different file systems. Some file systems are native to UNIX/Linux and others provide compatibility with different operating systems, such as Windows.

Most versions of UNIX and Linux support the UNIX file system (ufs), which is the original native UNIX file system. ufs is a hierarchical (tree structure) file system that is expandable, supports large amounts of storage, provides excellent security, and is reliable. In fact, many qualities of other file systems are modeled after ufs. ufs supports journaling, so that if a system crashes unexpectedly, it is possible to reconstruct files or to roll back recent changes for minimal or no damage to the integrity of the files or data. Journaling means that the file system keeps a log (journal) of its own activities. If the operating system crashes or is not properly shut down, such as during a power failure, the operating system reads the journal file when it is restarted. The information in the journal file enables files to be brought back to their previous or stable state before the crash. This is particularly important for files that were being updated before the crash and that did not have time to finish writing the updates to disk. ufs also supports hot fixes, which automatically move data on damaged portions of disks to areas that are not damaged.

In Linux, the native file system is called the extended file system (ext or ext fs), which is installed by default. ext is modeled after ufs, but the first version contained some bugs, supported files up to only 2 GB, and did not offer journaling. However, in Linux, ext provides an advantage over all other file systems because it enables the use of the full range of built-in Linux commands, file manipulation, and security. Newer versions of Linux use either the second (ext2), third (ext3), or fourth (ext4) versions of the extended file system. ext2 is a reliable file system that handles large disk storage. ext3 has the enhancements of ext2, with the addition of journaling.

Appearing in October 2006, ext4 is the newest version of ext. ext4 allows a single volume to hold up to 1 exabyte (1,152,921,504,606,846,976 bytes, which is over 1.1 quintillion bytes) of data and it enables the use of extents. An extent is used to reduce file fragmen- tation, because a block of contiguous disk storage can be reserved for a file. For example, consider a file of names and addresses that continuously grows as you add more people. Each time you add more people to the file, the new data is stored right next to the old in the extent of contiguous disk space reserved for that file. This is an improvement over other file systems in which the new data may be stored in a different location on the disk. Over time the data in a file (without the use of an extent) may be spread all over the disk, resulting in more time to locate data and more disk wear.

ext3 and ext4 are compatible unless extents are used in ext4. ext3 is also compatible with ext2. This means that many disk utilities that work with ext2 are likely to work with ext3 and ext4 (without the use of extents in ext4). Also note that a gigabyte is 230, whereas an exabyte is significantly larger at 260 in binary, or 109 compared to 1018 in decimal.

Table 2-1 summarizes ufs, ext, and other file systems typically supported by UNIX/Linux. Also,Table 2-2 compares FAT, NTFS, ext4, and ufs to give you a taste of what to consider when using a file system. As you consider a file system, keep in mind that the actual capabilities of that file system are also contingent on what is supported by the UNIX/Linux operating system you use, and even the version of the kernel in that particular operating system.

Table 2-1 Typical file systems supported by UNIX/Linux

File System - Description

Extended file system (ext or ext fs) and the newer versions: second extended file system (ext2 or ext2 fs), third extended file system (ext3 or ext3 fs), and fourth extended file system (ext4 or ext4 fs)

Comes with Linux by default (compatible with Linux and FreeBSD); ext3 offers journaling, which is important for reliability and recovery when a system goes down unexpectedly; ext4 adds larger volume sizes plus extents

High-performance file system (HPFS) Developed for use with the OS/2 operating system

International Organization for Standardization (ISO) Standard Operating System 9660 (iso9660 in Linux, hsfs in Solaris, cd9660 in FreeBSD) Developed for CD and DVD use; does not sup- port long file names

Journaled File System (JFS) Modeled after IBM's JFS; offers mature journaling features, fast performance for processing larger files, dynamic inode allocation for better use of free space, and specialized approaches for orga- nizing either small or large directory structures

msdos Offers compatibility with FAT12 and FAT16 (does not support long file names); typically installed to enable UNIX to read floppy disks made in MS-DOS or Windows

Network file system (NFS) Developed by Sun Microsystems for UNIX sys- tems to support network access and sharing of files (such as uploading and downloading); sup- ported on virtually all UNIX/Linux versions as well as many other operating systems

NT file system (NTFS) Used by Windows NT, Windows 2000, Windows XP, Windows Vista, and Windows Server systems

Proc file system Presents information about the kernel status and the use of memory (not truly a physical file sys- tem, but a logical file system)

ReiserFS Developed by Hans Reiser and similar to ext3 and ext4, with journaling capabilities; designed to be faster than ext3 and ext4 (up to 15 times) for handling small files; intended to encourage pro- grammers to create efficient code through use of smaller files

Swap file system File system for the swap space—that is, disk space used exclusively to store spillover informa- tion from memory when memory is full (called virtual memory) and used by virtually all UNIX/ Linux systems; on newer UNIX/Linux systems, the swap file system is encrypted for improved security

Universal Disk Format (UDF) Developed for CD and DVD use and broadly replacing iso9660. UDF read capability is sup- ported in Windows, UNIX/Linux, and Mac OS systems prior to 2006; read/write capability is supported in Windows Vista, UNIX/Linux ver- sions after 2005, and Mac OS Tiger and the newer Leopard.

uMS/DOS Compatible with extended FAT16 as used by Windows NT, 2000, XP, Vista, and Server, but also supports security permissions, file ownership, and long file names

UNIX file system (ufs; also called Berkely Fast File System) Original file system for UNIX; compatible with the Berkeley Fast File System) virtually all UNIX systems and most Linux systems

vfat Compatible with FAT32 and supports long file names

XFS Silicon Graphics' file system for the Irix version of UNIX; offers many types of journaling features and is targeted for use with large disk farms (mul- tiple disk storage devices available through high- speed connections)

Feature	Total volume or partition size
FAT	2 GB to 2 TB
NTFS	2 TB
ext4	1 exabyte in Linux depend- ing on the ker- nel version*
ufs	1 TB in Linux; 4 GB to 2 TB in UNIX depending on the version

TABLE 2-2 COMPARISON OF TYPICAL FILE SYSTEMS SUPPORTED BY UNIX/LINUX (CONTINUED)

FEATURE	**MAXIMUM SIZE**	**SECURITY**	**RELIABILITY THROUGH FILE ACTIVITY TRACK- ING OR JOURNALING**	**POSIX SUPPORT**	**RELIABILITY THROUGH HOT FIX CAPABILITY**	**SUPPORT FOR EXTENTS**
FAT	2 GB for file size FAT16; 4 GB for FAT32	Limited secu- rity based on attributes and shares	None	None (FAT16); limited (FAT32)	Limited	NO
NTFS	Potentially 16 TB, but limited by the volume size (up to 2 TB)	Extensive secu- rity through permissions, groups, and auditing options	Journaling	Yes	Supported	Yes, when pre- allocated via a program
ext4	16 GB to 2 TB in Linux depending on the kernel version*	Extensive secu- rity through permissions and groups	Journaling	Yes	Supported	Yes, when enabled
ufs	2 GB in Linux; 2 GB to 16 TB in UNIX depending on the version	Extensive secu- rity through permissions and groups	Journaling	Yes	Supported	NO

*These maximums are limited by the kernel version and are based on Linux kernel version 2.6.19.

Understanding the Standard Tree Structure

The treelike structure for UNIX/Linux file systems starts at the root file system level. Root is the name of the file at this basic level, and it is denoted by the slash character (/). The slash represents the root file system directory. Notice in Figure 2-1 that there is also a directory that is used to store files for the root account, but this is designated as /root and is under the main root file system directory (/).

A directory is a special kind of file that can contain other files and directories. Regular files store information, such as records of employee names and addresses or payroll information, while directory files store the names of regular files and the names of other directories, which are called subdirectories. The subdirectory is considered the child of the parent directory because the child directory is created within the parent directory. In Figure 2-1, the root system directory (/) is the parent of all the other directories, such as /bin, /boot, /dev, /etc, /home, and so on. The /home directory is the parent of the /jean, /tricia, and /joseph subdirectories; /usr is the parent of the /bin, /dict, and /etc subdirectories.

USING UNIX/LINUX PARTITIONS

The section of the disk that holds a file system is called a partition. One disk might have many partitions, each separated from the others so that it remains unaffected by external disturbances such as structural file problems associated with another partition. When you install UNIX/Linux on your computer, one of your first tasks is deciding how to partition your hard drive (or hard drives, if you have more than one).

UNIX/Linux partitions are identified with names; for example, Linux uses “hda1” and “hda2” for some types of disks. In this case, the first two letters tell Linux the device type; “hd,” for instance, identifies the commonly used IDE type of hard disk. The third letter,“a” in this case, indicates whether the disk is the primary or secondary disk (a=primary, b=secondary).

Partitions on a disk are numbered starting with 1. The name “hda1” tells Linux that this is the first partition on the disk, and the name “hda2” indicates it is the second partition on the same disk. If you have a second hard disk with two partitions, the partitions are identified as “hdb1” and “hdb2.”

Computer storage devices such as hard disks are called peripheral devices. Computer peripherals connect to the computer through electronic interfaces. The two most popular hard disk interfaces are Integrated Drive Electronics (IDE) and Small Computer System Interface (SCSI). Enhanced IDE (EIDE), which is IDE with built-in speed improvements, is now used more commonly than the original IDE technology, but often appears in computer system information as IDE.

On PCs used by individuals, IDE/EIDE hard disk drives (identified as hdx) are more common than SCSI (pronounced "scuzzy"). SCSI is faster and more reliable, so it is often used on servers. If you have a primary SCSI hard disk with two partitions, the two partitions are named "sda1" and "sda2." Figure 2-2 shows two partition tables: one with an IDE drive and the other with a SCSI drive.

IDE is sometimes referred to as Integrated Device Electronics. The American National Standards Institute (ANSI) standard for IDE is actually named Advanced Technology Attachment (ATA). ATA and SCSI are standards devel- oped by the ANSI-sponsored T10 and T13 committees; you can find out more about them by visiting www.T10.org and http://www.t13.org/.

Modern computers come with one or more Universal Serial Bus (USB) connec- tions for connecting keyboards, pointing devices, printers, and external hard drives. An external hard drive (that you plug into a USB port on your computer) may be referred to as a Serial ATA, SATA, or eSATA drive. SATA/eSATA is a newer technology than ATA, and the bottom line for the user is that it is generally faster than ATA (when used with a USB 2.0 port). Also, internal SATA drives are in many new PCs.

Figure 2-2 Sample Linux partition tables

Note that the first table in Figure 2-2 identifies "hda" as the device, which indicates an IDE drive. The second table identifies "sda" as the device, which indicates a SCSI drive.

Fedora, Red Hat Enterprise Linux, and SUSE have an Automatic Partitioning option that you can select as you are installing these systems. This tool auto- matically allocates space to create the swap, /boot, and root partitions described in the next section of this book.

Setting Up Hard Disk Partitions

Partitioning your hard disk provides organized space to contain your file systems. If one file system fails, you can work with another. This section provides general guidelines on how to partition hard disks. These recommendations are suggestions only. How you partition your hard disk might vary depending on your system's configuration, number of users, and planned use. Partition size is measured in megabytes (MB, about a million characters) or gigabytes (GB, about a billion characters). Some UNIX/Linux vendors recommend at least three partitions: root, swap, and /boot.

You can begin the process by setting up a partition for the root file system, which holds the root file system directory (remember that this is referred to as "/"). A partition must be mounted before it becomes part of the file system. The kernel mounts the root file system when the system starts.

References to the root file system directory (/) and to the directory used by the root account (/root) can get confusing. Some UNIX/Linux users refer to the root file system directory as "slash" and to /root as the "root directory" to help avoid confusion.

The size of the root partition depends on the type of installation you are performing. For example, in Fedora, Red Hat Enterprise Linux, or SUSE, the root partition should be a minimum of 1.2 GB to load the basic operating system required for a workstation or portable computer installation. A 1.2 GB partition for a basic system does not include enough space to load the GNOME desktop or many software packages. If you are setting up a server or loading the full complement of software packages that come with Fedora, Red Hat Enterprise Linux, or SUSE, use a partition of 5-10 GB or larger. Besides loading the software packages, this allows space for a desktop, such as GNOME, KDE, or a combination of both. (You learn more about desktops in Chapter 11, "The X Window System.")

After creating the root partition, you should set up the swap partition. The swap partition acts like an extension of memory, so that UNIX/Linux have more room to run large programs. As a general rule, the swap partition should be the same size as the amount of RAM in your computer. For instance, if you have 256 MB of RAM, make your swap space 256 MB. If you have a large amount of RAM, such as 1 GB, but your disk space is limited, you can make the swap space smaller than 1 GB. However, before configuring the swap partition, check the documentation for your version of UNIX/Linux. For example, for Fedora and Red Hat Enterprise Linux, Red Hat suggests that your swap space be a minimum of 256 MB, or two times the size of the RAM in the computer (use the larger figure). For SUSE, consider a swap partition of about 500 MB or a little larger. However, the swap space should not be too large, such as over 2 GB, because then you begin sacrificing speed because disk access is slower than direct RAM access. For instance, if you have 256 MB of RAM in a Fedora system, it doesn't make sense to allocate 2 GB of disk for the swap partition. From the standpoint of performance, it makes more sense to install 256 MB more RAM for a total of 512 MB and then allocate 1 GB for the swap partition.

A swap partition enables virtual memory. Virtual memory means you have what seem to be unlimited memory resources. Swap partitions accomplish this by providing swap space on a disk and treating it like an extension of memory (RAM). It is called swap space because the system can use it to swap information between disk and RAM. Setting up swap space makes your computer run faster and more efficiently.

You can create and use more than one swap partition in Linux. Having multiple swap partitions spread across several hard disks can sometimes improve appli- cation performance on busy systems. Also note that you can often improve the speed of a server by installing higher amounts of RAM, such as 1 GB or more.

The /boot partition is used to store the operating system files that compose the kernel. The size of this partition depends on how much space is needed for the operating system

files in your version of UNIX/Linux. Generally, this is a relatively small partition. For example, if you are installing Fedora, Red Hat Enterprise Linux, or SUSE, consider creating a /boot partition that is about 100 to 200 MB in size.

If you plan to have multiple users accessing your system, consider having a /usr partition in which to store some or all of the nonkernel operating system programs that are accessed by users. These programs include software development packages that support computer programming, networking, Internet access, graphical screens (including desktop software), and the large number of UNIX/Linux utilities. Utilities are programs that perform operations such as copying files, listing directories, and communicating with other users. The /usr partition should be large enough—such as 10 GB or more—to accommodate all of the software that you install.

Also, if you plan to have multiple users access the system, you can create a /home partition, which is the home directory for all users' directories. Having separate /usr and /home partitions makes many system administration tasks, such as backing up only software or only data, much easier.

The /home partition is the storage space for all users' work. If the root partition (/)—or any other partition—crashes, having a /home partition ensures that you do not lose all the users' information. Although regular user accounts are restricted from reading information in other partitions, you own and can access most files in your home directory. You can grant or deny access to your files as you choose. See "Configuring File Permissions for Security" later in this chapter for more information on file ownership.

Finally, you can create a /var partition to hold files that are created temporarily, such as files used for printing documents (spool files) and files used to record monitoring and adminis- tration data, often called log files. Fedora, Red Hat Enterprise Linux, and SUSE also use the /var partition for files used to update the operating system. Plan on using a /var partition that is over 5 GB.

Consider a small geological research company that plans to set up a Red Hat Enterprise Linux server that has 512 MB of RAM. The company might set up a swap partition at 1024 MB. The /boot partition would be 150 MB. They plan to install 20 GB of nonkernel programs for users, so there might be a 25 to 50 GB /usr partition (allowing for programs not currently anticipated). Each of the 15 users will be allocated 10 GB of space in their home directories,which means the /home partition must be at least 150 GB (adding another 20 GB or more would provide some margin for extra needs). Finally, a 10 GB /var partition should be created because all of the users print large documents, often at the same time.

Setting up partitions might seem like a big task when you are learning UNIX/ Linux. Fortunately, many operating systems, including Fedora, Red Hat Enter- prise Linux, and SUSE, offer tools to automatically set up partitions during installation (see Appendix C, "How to Install Fedora and How to Use the Knoppix CD"). Consider using these tools until you have more experience with UNIX/Linux.

Using Inodes

Partitions containing directories and files in the ufs and ext file systems are built on the concept of information nodes, or inodes. Each directory or file has an inode and is identified by an inode number. Inode 0 contains the root of the directory structure (/) and is the jumping-off point for all other inodes.

An inode contains (1) the name of a directory or file, (2) general information about that directory/file, and (3) information (a pointer) about how to locate the directory/file on a disk partition. In terms of general information, each inode indicates the user and group ownership, the access mode (read, write, and execute security permissions, discussed later in this chapter), the size and type of the file, the date the file was created, and the date the file was last modified and read.

The pointer information is based on logical blocks. Each disk is divided into logical blocks ranging in size (depending on the version of UNIX/Linux) from 512 to 8,192 bytes or more (blocks can also be divided into multiple subblocks or fractions as needed by the file system). The inode for a file contains a pointer (number) that tells the operating system how to locate the first in a set of one or more logical blocks that contain the specific file contents (or, it specifies the number of blocks or links to the first block used by the directory or file). In short, the inode tells the operating system where to find a file on the hard disk.

Everything in the UNIX/Linux file system is tied to inodes. Space is allocated one block, or fraction of a block, at a time. Directories are really simple files that have been marked with a directory flag in their inodes. The file system itself is identified by the superblock. The superblock contains information about the layout of blocks on a specific partition. This information is the key to finding anything on the file system, and it should never change. Without the superblock, the file system cannot be accessed. For this reason, many copies of the superblock are written into the file system at the time the file system is created through partitioning and formatting. If the superblock is destroyed, you can copy one of the superblock copies over the original, damaged superblock to restore access to the file system.

You can display inode information for directories and files by using the ls –i command, which you will learn later in this chapter.

EXPLORING THE ROOT HIERARCHY

The root (/) file system is mounted by the kernel when the system starts. To mount a file system is to connect it to the directory tree structure. The system administrator uses the mount command to mount a file system.

UNIX/Linux must mount a file system before any programs can access files on that file system. After mounting, the root file system is accessible for reading only during the initial

systems check and boot-up sequence—after that, it is remounted as read and write. The root file system contains all essential programs for file system repair: restoring from a backup, starting the system, and initializing all devices and operating resources. It also contains the information for mounting all other file systems. Nothing beyond these essentials should reside in the root partition.

You can restore a crashed root partition using rescue files stored on disks, CDs, DVDs, tapes, or other removable media. The installation media that comes with Fedora, Red Hat Enterprise Linux, and SUSE can be used to create rescue discs, or your installation CDs/DVDs can double as rescue discs.

The following sections describe commonly used directories under the root file system.

The /bin Directory

The /bin directory contains binaries, or executables, which are the programs needed to start the system and perform other essential system tasks. This directory holds many programs that all users need to work with UNIX/Linux.

The /boot Directory

The /boot directory normally contains the files needed by the bootstrap loader (the utility that starts the operating system); it also contains the kernel (operating system) images.

The /dev Directory

Files in /dev reference system devices. They access system devices and resources, such as the hard disks, mice, printers, consoles, modems, memory, and CD/DVD drives. UNIX/Linux versions include many device files in the /dev directory to accommodate separate vendor devices that can be attached to the computer.

UNIX/Linux devices are managed through the use of device special files, which contain information about I/O devices that are used by the operating system kernel when a device is accessed. In many UNIX/Linux systems, two types of device special files exist:

- Block special files (also called block device files) are used to manage random access devices that involve handling blocks of data, including CD/DVD drives, hard disk drives, tape drives, and other storage devices.

- Character special files (also called character device files) handle byte-by-byte streams of data, such as through serial or universal serial bus (USB) connections, including terminals, printers, and network communications. USB is a relatively high-speed I/O port found on most modern computers. It is used to interface mice, keyboards, monitors, digital sound cards, disk drives, and other external computer hardware, such as printers and digital cameras.

Another method for managing devices is the use of a named pipe, which offers a method for handling internal communications, such as redirecting file output to a monitor. You learn about piping in Chapter 5.

When you install a UNIX/Linux operating system, device special files are created for the devices already installed on the system. Table 2-3 shows a sampling of device special files.

Table 2-3 UNIX/Linux device special files

File	**Description**
/dev/console	For the console components, such as the monitor and key- board attached to the computer (/dev/tty0 is also used at the same time on many systems)
/dev/fdn	For floppy disk drives, where n is the number of the drive, such as fd0 for the first floppy disk drive
/dev/hdxn	For IDE and EIDE hard drives, where x represents the disk and the n represents the partition number, such as hda1 for the first disk and partition
/dev/modem	For a modem, a symbolic link to the device special file (typi- cally linked to /dev/ttys1), where a symbolic link enables one file or directory to point to another (in later versions of Fedora/Red Hat Enterprise Linux, the modem file may be in /usr/share/applications, and in SUSE this file may be under /usr/share/applications/YaST2 because it is managed using the YaST management tool)
/dev/mouse	For a mouse or other pointing device, a symbolic link to the device special file (typically linked to /dev/ttys0)—in Fedora/ Red Hat Enterprise Linux, the mouse file may be under /usr/share/applications, and in SUSE it may be under /opt/gnome/ share/applications
/dev/sdxn	For a hard drive connected to a SCSI interface, where x repre- sents the disk and the n represents the partition, such as sda1 for the first SCSI drive and first partition on that drive
/dev/stn	For a SCSI tape drive, where n represents the number of the drive, such as st0 for the first tape drive
/dev/ttyn	For serial terminals connected to the computer
/dev/ttysn	For a serial device connected to the computer, such as ttys0 for the mouse

If you need to create a device special file for a new device, you can do so by using the mknod command as in the following general steps: 1. Log in to the root account. 2. Access a terminal window or the command prompt. 3. Type cd /dev and press Enter to switch to the /dev folder. 4. Use the mknod command plus the device special file name, such as ttys42, and the type of file, such as character (c) or block (b), and a major and minor node value used by the kernel (check with the device manufacturer for these values). For example, you might type mknod ttys20 c 8 68, and press Enter for a new serial device.

You can see the list of device files by typing ls -l /dev and pressing Enter after the command prompt. (See "Listing Directory Contents" later in this chapter for more information on the ls command.) The far-left character in the list tells you whether the file is a character device (c) or a block device (b), as shown in Figure 2-3. Try Hands-on Project 2-8 to view the contents of the /dev directory on your computer. Hands-on Project 2-8 also teaches you to use the ls command.

Explanations of the kinds of items you will see in the /dev directory are as follows:

■ console refers to the system's console, which is the monitor connected directly to your system.

■ ttyS1 and cua1 are devices used to access serial ports.For example,/dev/ttyS1 refers to COM2, the communication port on your PC.

■ cua devices are callout devices used in conjunction with a modem.

■ Device names beginning with hd access IDE hard drives.

■ Device names beginning with sd are SCSI drives.

■ Device names beginning with lp access parallel ports. The lp0 device refers to LPT1, the line printer.

■ null is a "black hole"; any data sent to this device is gone forever. Use this device when you want to suppress the output of a command appearing on your screen. Chapters 6 and 7 ("Introduction to Shell Script Programming" and "Advanced Shell Programming," respectively) discuss this technique.

■ Device names beginning with tty refer to terminals or consoles. Several "virtual consoles" are available on your Linux system. (You access them by pressing (Ctrl+Alt+F1, Ctrl+Alt+F2, and so on.) Device names beginning with pty are "pseudoterminals."They are used to provide a terminal to remote login sessions. For example, if your machine is on a network, incoming remote logins would use one of the pty devices in /dev.

The /etc Directory

The /etc directory contains configuration files that the system uses when the computer starts. Most of this directory is reserved for the system administrator, and it contains system-critical information stored in the following files:

■ fstab—The mapping information about file systems to devices (such as hard disks and CDs/DVDs)

Figure 2-3 Some block devices in /dev

File type - Meaning	
-	Normal
d	Subdirectory
b	Block device
c	Character device

■ group—The user group information file ■ inittab—The configuration file for the init program, which performs essential

chores when the system starts

■ login.defs—The configuration file for the login command

■ motd—The message-of-the-day file

■ passwd—The user information file

■ printcap—The printer-capability information file

■ profile and bashrc—The files executed at login that let the system administrator set global defaults for all users

■ rc—The scripts or directories of scripts to run when the system starts ■ termcap—The terminal capability information file

---

To get a taste of what is in the /etc directory, try viewing the contents of the fstab file. Enter more /etc/fstab and you'll see default setup information for file systems and devices such as CD/DVD drives.

The /home Directory

The /home directory is often on the /home partition and is used to offer disk space for users, such as on a system that has multiple user accounts. In Figure 2-1, for example, three home directories exist for three user accounts: /home/jean, /home/tricia, and /home/joseph.

The /lib Directory

This directory houses kernel modules, security information, and the shared library images, which are files that programmers generally use to share code in the libraries rather than creating copies of this code in their programs. This makes the programs smaller and, in some cases, they can run faster using this structure. Many files in this directory are symbolic links to other library files. A symbolic link is a name, file name, or directory name that contains a pointer to a file or directory in the same directory or in another directory on your system. Another related use of a symbolic link is to create a name or shortcut notation for accessing a directory. In a directory's long listing, l in the far-left position identifies files that are symbolic links.

One way to save typing time is to create a symbolic link to a directory that has a long path. For example, assume that you store many files in the /data/ manufacturing/inventory/parts subdirectory. Each time you want to perform a listing of that subdirectory, you must type ls /data/manufacturing/inventory/ parts. If you enter ln -s /data/manufacturing/inventory/parts to create a sym- bolic link (ln is the command to create a link and -s is the option for a symbolic link) to that directory; in the future you only have to type ls parts to see the contents. To learn more about the ln command, type man ln or info ln and press Enter.

The /mnt Directory

Mount points for temporary mounts by the system administrator reside in the /mnt directory. A temporary mount is used to mount a removable storage medium, such as a CD/DVD or USB/flash storage so that it can be easily unmounted for quick removal. For example, you might mount a CD to burn a disc and then quickly unmount it to give the disc to an office associate. The /mnt directory is often divided into subdirectories, such as /mnt/cdrom, to clearly specify device types.

The /media Directory

In newer distributions of UNIX/Linux, mount points for removable storage are in the /media directory, which is a relatively new recommendation of the Filesystem Hierarchy Standard (FHS). Modern Linux distributions include both /mnt and /media directories, but automated software to detect insertion of a CD/DVD typically uses /media. Linux users and programmers are often encouraged to use /media instead of /mnt as a way to follow the newer FHS recommendation.

Guidelines for the file hierarchy in UNIX/Linux are provided by the Filesystem Hierarchy Standard (FHS). You can learn more about FHS at the official Web site: http://www.pathname.com/fhs. Also, Wikipedia offers an introduction to FHS at en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard.

The /proc Directory

The /proc directory occupies no space on the disk; it is a virtual file system allocated in memory only.Files in /proc refer to various processes running on the system as well as details about the operating system kernel.

The /root Directory

The /root directory is the home directory for the root user—the system administrator.

The /sbin Directory

The /sbin directory is reserved for the system administrator. Programs that start the system, programs needed for file system repair, and essential network programs are stored here.

The /tmp Directory

Many programs need a temporary place to store data during processing cycles. The traditional location for these files is the /tmp directory. For example, a payroll program might create several temporary data files as it processes a payroll for 5,000 people. The temporary files might hold data briefly needed for calculating withholdings for taxes and retirement and then be deleted after the withholding information is written to tape or CD to send to federal and state agencies.

The /usr Directory

Frequently on the /usr partition, this directory houses software offered to users. The software might be accounting programs, manufacturing programs, programs for research applications, or office software.

The /var Directory

Located on the /var partition, the /var directory holds subdirectories that often change in size. These subdirectories contain files such as error logs and other system performance logs that are useful to the system administrator. The /var/spool/mail subdirectory can contain incoming mail from the network, for example. Another example is the /var/spool/lpd subdirectory, which is the default directory for holding print files until they are fully transmitted to a printer.

Try viewing the contents of a log to get an idea of what is in the /var directory. For example, log in as root and enter more /var/log/boot.log to see how information about booting the system is retained for informational purposes and troubleshooting. (If the boot.log file is empty, try entering a log version number, such as boot.log.2.)

USING THE MOUNT COMMAND

As you learned, UNIX/Linux use the mount command to connect the file system partitions to the directory tree when the system starts. Users can access virtually any file system that has been mounted and to which they have been granted permission. Additional file systems can be mounted at any time using the mount command. The CD and DVD drives are the file system devices beyond the hard disk that are most commonly mounted. The syntax for the mount command is as follows:

Syntax mount [-option] [device-name mount-point]

Dissection

■■■

Use the -t option to specify a file system to mount. device-name identifies the device to mount. mount-point identifies the directory in which you want to mount the file system.

To ensure security on the system, only the root user can normally use the mount command. Ordinary users can sometimes mount and unmount file systems located on floppy disks and CDs/DVDs, but some operating systems require the root account to mount one or both.

Suppose you want to access files on a CD for your organization. You or the system administrator can mount a CD by inserting a disk in the CD drive, and then using one of the following mount commands (depending on whether your UNIX/Linux distribution and version use /mnt or /media):

```
mount -t iso9660 /dev/cdrom /mnt/cdrom
```

or

```
mount -t iso9660 /dev/cdrom /media/cdrom
```

This command mounts the CD on a device called “cdrom” located in the /dev directory. The actual mount point in UNIX/Linux is /mnt/cdrom or /media/cdrom, a directory that references the CD device. After the CD is mounted, you can access its files through the /mnt/cdrom or /media/cdrom directory.

UNIX/Linux support several different types of file systems. The type of file system is specified with the -t option. CDs are classified as iso9660 or udf devices, so the system administrator types -t, followed by the argument, such as iso9660, to specify the file system for CDs. On some newer versions of Linux, CDs/DVDs are mounted automatically through program software. The contents of CDs/DVDs can be viewed by double-clicking the CD's/DVD's icon on the desktop or as a subdirectory in the /media directory under the root (/).

After a CD is mounted, you can view the device paths, file system, and permissions by typing the mount command without options or arguments.

Some systems still include legacy floppy disk drives. To mount a floppy disk, first insert it and then use one of the following commands (where filesystem is the floppy disk file system, such as vfat for porting a floppy to a Windows system): mount -t filesystem /dev/fd0 /mnt/floppy

or

mount -t filesystem /dev/fd0 /media/floppy

After accessing manually mounted file systems, the system administrator unmounts them using the umount command before removing the storage media, as in the following example:

umount /mnt/cdrom (or umount /media/cdrom)

Syntax umount mount-point

Dissection

■ mount-point identifies the directory to unmount.

Notice that the command is umount, not unmount; there is only one "n" in umount.

Try Hands-on Project 2-1 to use the mount command to view the file systems you can mount in your version of UNIX/Linux and to view what file systems are currently mounted. Also, Hands-on Project 2-2 enables you to mount or load a CD and view the files on the CD. See Appendix B, "Syntax Guide to UNIX/Linux Commands," for a brief description of the mount and umount commands.

USING PATHS, PATHNAMES, AND PROMPTS

As you've learned, all UNIX/Linux files are stored in directories in the file system, starting from the root file system directory. To specify a file or directory, use its pathname, which follows the branches of the file system to the desired file. A forward slash (/) separates each directory name. For example, suppose you want to specify the location of the file named phones.502. You know that it resides in the source directory in Jean's home directory, /home/jean/source, as illustrated in Figure 2-1. You can specify this file's location as /home/jean/source/phones.502.

Using and Configuring Your Command-Line Prompt

The UNIX/Linux command prompt can be configured to show your directory location within the file system. For example, in Fedora the prompt [jean@localhost ~]$ is the default prompt that the system generated when the system administrator first created the user account called "jean." The prompt [jean@localhost ~]$ means that "jean" is the user working on the host machine called "localhost" in her home directory, which is signified by the tilde (~). The ~ is shorthand for the home directory, which typically has the same name as the user's account name. The account jean would typically have a home directory also called jean that is located at /home/ jean. When Jean changes her location to /home/jean/source, her prompt looks like:

[jean@localhost source]$

When the system is initially installed, the default root prompt looks like this: [root@localhost root]#. To simplify the meaning of the command prompts in this book, the steps use $ to represent the ordinary user's command prompt and # to represent the system administrator's command prompt.

Your command prompt is configured automatically when you log in. An environment variable, PS1, contains special formatting characters that determine your prompt's configuration. An environment variable is a value in a storage area that is read by UNIX/Linux when you log in. Environment variables can be used to create and store default settings, such as the shell that you use or the command prompt format you prefer. You learn more about environment variables and how to configure them in Chapter 6. Figure 2-4 illustrates how the PS1 variable is configured by default for a user account in Fedora.

Figure 2-4 Viewing the contents of the PS1 variable

In Figure 2-4, the PS1 variable contains: [\u@\h \W]\$. Characters that begin with \ are special Bash shell formatting characters. \u prints the username, \h prints the system host name, and \W prints the name of the working (current) directory. The characters \$ print either a # or a $, depending on the type of user logged in. The brackets, [and], and the space that separates \h and \W are not special characters, so they are printed just as they appear. When Jean is logged into the system localhost and working in her home directory, her prompt appears as [jean@localhost ~]$ in the format shown previously.

Table 2-4 shows other formatting characters for configuring your Bash shell prompt.

Formatting Character - Purpose

\a	Sounds an alarm
\d	Displays the date
\e	Uses an escape character
\h	Displays the host name
\j	Shows the number of background jobs
\n	Displays a new line
\nnn	Displays the ASCII character that corresponds to the octal number nnn
\r	Places a carriage return in the prompt
\s	Displays the shell name
\t	Displays the time
\u	Displays the username
\v	Displays the Bash version and release number
\w	Displays the path of the working directory
\A	Displays the time in 24-hour format
\D(format)	Displays the time in a specific format
\H	Has the same effect as \h
\T	Displays the time in 12-hour format
\V	Displays the Bash version, release number, and patch level
\W	Displays the name of the working directory without any other path information
\!	Displays the number of the current command in the com- mand history
\#	Displays the number of the command in the current session

\$	Displays a # if root is the user, otherwise displays a $
\@	Displays the time in 12-hour format
$PWD	Displays the path of the current working directory
\[	Marks the beginning of a sequence of nonprinting charac- ters, such as a control sequence
\]	Marks the end of a sequence of nonprinting characters
\\	Displays a \ character

Hands-On Project 2-3 gives you the opportunity to view the contents of the PS1 environment variable for your user account and then to configure your Bash shell prompt.

The pwd Command

If you have configured your prompt so that it does not show your working directory,you can use the pwd command (pwd stands for print working directory) to verify in what directory you are located, along with the directory path. This command can be important for several reasons. One is that you can list the contents of a directory and not find the files you are expecting, and so it can help to ensure that you are in the right directory before doing anything else (such as restoring the files). In other situations, you might have created a script or program that you can only run from a specific directory. If the script or program is not working, first use pwd to verify you are in the right directory. Hands-on Project 2-4 enables you to use pwd.

Syntax pwd

Dissection

■ ■

Use pwd to determine your current working directory. Typically, there are no options with this command.

pwd is a simple but important command for all users. Often, users, program- mers, and administrators alike experience errors or misplace a file because they are working in the wrong directory. Periodically using pwd can be invaluable for making certain you are in the right place.

NAVIGATING THE FILE SYSTEM

To navigate the UNIX/Linux directory structure, use the cd (change directory) command. Its syntax is:

Syntax cd [directory]

Dissection

■ directory is the name of the directory to which you want to change. The directory name is expressed as a path to the destination, with slashes (/) separating subdirectory names.

When you log in, you begin in your home directory, which is under the /home main directory. When you change directories and then want to return to your home directory, type cd, and press Enter. (Some shells also use the tilde character (~) to denote the user’s home directory.) Try Hands-on Project 2-5 to use the cd command.

In UNIX/Linux, you can refer to a path as either an absolute path or a relative path. An absolute path begins at the root level and lists all subdirectories to the destination file. For example, assume that Becky has a directory named lists located under her home directory. In the lists directory, she has a file called todo. The absolute path to the todo file is /home/becky/lists/todo. This pathname shows each directory that lies in the path to the todo file.

Any time the / symbol is the first character in a path, it stands for the root file system directory. All other / symbols in a path serve to separate the other names.

A relative path takes a shorter journey. You can enter the relative path to begin at your current working directory and proceed from there. In Figure 2-1, Jean,Tricia, and Joseph each have subdirectories located in their home directories. Each has a subdirectory called "source." Because Jean is working in her home directory, she can change to her source directory by typing the following command and pressing Enter:

```
cd source
```

In this example, which is called relative path addressing, Jean is changing to her source directory directly from her home directory, /home/jean. Her source directory is one level away from her current location, /home/jean. As soon as she enters the change directory command, cd source, the system takes her to /home/jean/source because it is relative to her current location.

If Tricia, who is in the /home/tricia directory, enters the command cd source, the system takes her to the /home/tricia/source directory.ForTricia to change to Jean's source directory,she can enter:

```
cd /home/jean/source
```

This example uses absolute path addressing because Tricia starts from the root file system directory and works through all intervening directories. (Tricia, of course, needs permission to access Jean's source directory, which is discussed later in this chapter.)

Hands-on Project 2-6 enables you to practice absolute and relative path addressing.

Using Dot and Dot Dot Addressing Techniques

UNIX/Linux interpret a single dot character to mean the current working directory, and dot dot (two consecutive dots) to mean the parent directory. Entering the following command keeps you in the current directory:

```
cd .
```

When you use UNIX/Linux commands, always pay close attention to spaces. For example, in cd . and cd .. there is one space between cd and the single or double dot characters. Remember, you must always use a space after a UNIX/ Linux command before including options or arguments with that command.

If you use two dots, you move back to the parent directory. Do not type a space between the two dots. The next example shows how the user jean, who is currently in the /home/jean/ source directory, returns to her home directory, which is /home/jean:

```
cd ..
```

Assume you are Jean in her home directory and want to go to Tricia's source directory. Use the following command:

```
cd ../tricia/source
```

In the preceding example, the dot dot tells the operating system to go to the parent directory, which is /home. The first / separator followed by the directory name tells the operating system to go forward to the tricia subdirectory. The second / separator followed by the directory name tells UNIX/Linux to go forward to the source subdirectory, the final destination. If no name precedes or follows the slash character, UNIX/Linux treat it as the root file system directory.

Otherwise, / separates one directory from another. Hands-on Project 2-7 gives you practice using the dot and dot dot conventions.

Listing Directory Contents

Use the ls (list) command to display a directory's contents, including files and other directories. When you use the ls command with no options or arguments, it displays the names of regular files and directories in your current working directory. You can provide an argument to the ls command to see the listing for a specific file or to see the contents of a specific directory, such as ls myfile or ls /etc.

Syntax ls [-option] [directory or filename]

Dissection

■ Common arguments include a directory name (including the path to the directory) or a file name.

■ Useful options include: -l to view detailed information about files and directories -S to sort by size of the file or directory -X to sort by extension -r to sort in reverse order -t to sort by the time when the file or directory was last modified -a to show hidden files -i to view the inode value associated with a directory or file

Remember, when you log in, you begin in your home directory, which is /home/username.

You can also use options to display specific information or more information than the command alone provides. The -l option for the ls command generates a long directory listing, which includes more information about each file, as shown in Figure 2-5.

Figure 2-5 Using ls -l to view the root file system directory contents

Notice the first line in Figure 2-5 for the /bin directory:

drwxr-xr-x 2 root root 4096 Mar 2 2007 bin

If you look in the far-right column, you see bin, the name of a file. All of the columns to its left contain information about the file bin. Here is a description of the information in each column, from left to right.

■ File type and access permissions—The first column of information shown is the following set of characters:

drwxr-xr-x

The first character in the list, d, indicates that the file is actually a directory. If bin were an ordinary file, a hyphen (-) would appear instead. The rest of the characters indicate the file's access permissions. You learn more about these later in this chapter, in the section "Configuring File Permissions for Security."

■ Number of links—The second column is the number of files that are hard-linked to this file.(You learn more about links in Chapter 5,"Advanced File Processing.") If the file is a directory, this is the number of subdirectories it contains. The listing for bin shows it contains two (2) entries. (A directory always contains at least two entries: dot and dot dot.)

■ Owner—The third column is the owner of the file. The root user owns the /bin directory.

■ Group—The fourth column is the group that owns the file. The root group owns the /bin directory.

■ Size—The fifth column shows the size of the file in bytes, which is 4096 for the /bin directory.

■ Date and time—The sixth and seventh columns show the date when a directory or file was created, or if information has been changed for a directory or file, it shows the date and time of the last modification.

■ Name—The eighth column shows the directory or file name.

You can also use the -a option with the ls command to list hidden files. Hidden files appear with a dot at the beginning of the file name. The operating system normally uses hidden files to keep configuration information, among other purposes. To view the inode value for a directory or file, use the -i option.

Try Hands-on Project 2-8 to use the ls command. Also, see Appendix B for a brief description of the ls command.

Using Wildcards

A wildcard is a special character that can stand for any other character or, in some cases, a group of characters. Wildcards are useful when you want to work with several files whose names are similar or with a file whose exact name you cannot remember. UNIX/Linux support several wildcard characters. In this section, you learn about two: * and ?.

The * wildcard represents any group of characters in a file name. For example, assume Becky has these 10 files in her home directory:

friends instructions.txt list1 list2 list2b memo_to_fred memo_to_jill minutes.txt notes readme

If she enters ls *.txt and presses Enter, she sees the following output: instructions.txt minutes.txt

The argument *.txt causes ls to display the names of all files that end with .txt. If she enters ls memo*, she sees the following output:

memo_to_fred memo_to_jill

If she enters the command ls *s and presses Enter, ls displays all file names that end with “s”. She sees the output:

friends notes

The ? wildcard takes the place of only a single character. For example, if Becky types ls list? and presses Enter, ls displays all files whose names start with “list” followed by a single character. She sees the output:

list1 list2

She does not see the listing for the file list2b, because two characters follow the word “list” in its name. To see the list2b file in the listing, Becky could use two wildcard characters as in ls list??. Further, she can combine wildcard characters. For instance, if she wants to include the readme file in a listing, she might enter ls ??a* .

You work again with wildcard characters in Chapter 6. In this chapter, Hands-on Project 2-9 enables you to use wildcards with the ls command.

Wildcards are connected to the shell that you are using. The* and ? wildcards are available in the Bash shell. You can determine what wildcards are supported by a shell by reading the man documentation for that shell. For example, to read the documentation about the Bash shell, enter man bash at the command line.

CREATING AND REMOVING DIRECTORIES

You sometimes need to organize information by creating one or more new directories, such as under your home directory. System administrators also create new directories to hold programs, data, utilities, and other information. The mkdir command is used to create a new directory.

Syntax mkdir [-option] directory

Dissection

■ The argument used with mkdir is a new directory name. ■ There are only a few options used with mkdir. One option is to use -v to display a message that verifies the directory has been made.

Hands-on Project 2-10 enables you to make a directory and begin a set of projects in which you create a telephone database.

You can delete empty directories by using the remove directory command, rmdir. First, use the cd command to change to the parent directory of the subdirectory you want to delete. For example, if you want to delete the old directory in /home/old, first change to the home directory. Then type rmdir old and press Enter. In many versions of UNIX/Linux, including Fedora, Red Hat Enterprise Linux, and SUSE, rmdir will not delete a directory that contains files and you must delete, or move and delete, the files before you can delete the directory. Also, the rm -r command can be used to delete a directory that is not empty. You learn more about deleting directories in Chapter 4,"UNIX/Linux File Processing."

Syntax rmdir [-option] directory

Dissection

■ The argument used with rmdir is a directory. ■ As is true for mkdir, rmdir has only a few options. Consider using the -v option to display a message that verifies the directory has been removed.

COPYING AND DELETING FILES

The UNIX/Linux copy command is cp, which is used to copy files from one directory to another. The -i option provides valuable insurance because it warns you that the cp command overwrites the destination file, if a file of the same name already exists. You can also use the dot notation (current directory) as shorthand to specify the destination of a cp command. Try Hands-on Project 2-11 to use the cp command.

Syntax cp [-option] source destination

Dissection

■ The argument consists of the source and destination directories and files, such as cp /home/myaccount/myfile /home/youraccount.

■

Common options include:

-b makes a backup of the destination file if the copy will overwrite a file

-i provides a warning when you are about to overwrite a file

-u specifies to only overwrite if the file you are copying is newer than the one you are overwriting

To delete files you do not need, use the remove command, rm. First, use the cd command to change to the directory containing the file you want to delete. Then type rm filename. For example,to delete the file"old"in the current working directory,type rm old.Depending on your version of UNIX/Linux, you might or might not receive a warning before the file is deleted. However, you can have the operating system prompt to make certain you want to perform the deletion by using the -i option. The best insurance, though, is to be certain you want to remove a file permanently before using this command. You learn more about the rm command in Chapter 4.

Syntax rm [-option] filename

Dissection

■ The argument consists of the name of the file to delete. ■ The -i option causes the operating system to prompt to make certain you want to delete

the file before it is actually deleted.

CONFIGURING FILE PERMISSIONS FOR SECURITY

Early in computing, people didn’t worry much about security. Stolen files and intrusions were less of a concern, in part because networks were rare and there was no Internet. As you have probably learned through the media, friends, and school, times are different and you need to protect your files. Security is important on UNIX/Linux systems because they can house multiple users and are connected to networks and the Internet, all potential sources of intrusion.

Users can set permissions for files (including directories) they own so as to establish security. System administrators also set permissions to protect system and shared files. Permissions manage who can read, write, or execute files.

The original owner of a file is the account that created it; however, file ownership can be transferred to another account. The permissions the owner sets are listed as part of the file description. Figure 2-6 shows directory listings that describe file types.

Notice the long listing of the two directories. (Remember that the directory is just another file.) An earlier section of this chapter,“Listing Directory Contents,” describes the informa- tion presented in a long listing.Now,you can look closer at the file permissions.For the first file described, the column on the far left shows the string of letters drwxr-xr-x. You already know the first character indicates the file type, such as - for a normal file and d for a directory/subdirectory. The characters that follow are divided into three sections of file permission specifiers, as illustrated in Figure 2-7.

The first section of file permission specifiers indicates the owner’s permissions. The owner, like all users, belongs to a group of users. The second section indicates the group’s permissions. This specification applies to all users (other than the owner) who are members of the owner’s group. The third section indicates all others’ permissions. This specification applies to all users who are not the owner and not in the owner’s group. In each section, the first character indicates read permissions. If an “r” appears there, that category of users has permission to read the file. The second character indicates write permission. If a “w” appears there, that category of users has permission to write to the file. The third character indicates the execute permission. If an “x” appears there, that category of users has permission to execute the file, such as a program file. If a dash (-) appears in any of these character positions, that type of permission is denied.

If a user is granted read permission for a directory, the user can see a list of its contents. Write permission for a directory means the user can rename,delete,and create files in the directory. Execute permission for a directory means the user can make the directory the current working directory.

File type	Meaning
-	Normal file
d	Subdirectory
l	Symbolic link
b	Block device file
c	Character device file

From left to right, the letters rwxr-xr-x mean:

r — File's owner has read permission

w — File's owner has write permission

x — File's owner has execute permission (can run the file as a program) r — Group has read permission - — Group does not have write permission x — Group has execute permission

r — Others have read permission - — Others do not have write permission x — Others have execute permission

You can change the pattern of permission settings by using the chmod command. For example, setting others' permissions to --- removes all permissions for others. They cannot read, write, or execute the file. In the first line of Figure 2-6, notice that the owner has read, write, and execute (rwx) permissions for the subdirectory X11. The first character is the file type, in this case, a "d" for a subdirectory. The rwx gives the owner read, write, and execute permissions. The next r-x indicates that the group of users that shares the same group id as the owner has only read and execute permissions; the final r-x gives read and execute permissions to others.

The system administrator assigns group ids when he or she adds a new user account. A group id (GID) gives a group of users equal access to files that they all share. Others are all other users who are not associated with the owner's group by a group id, but who have read and execute permissions.

In many UNIX/Linux distributions, when the system administrator creates an account, a group with the same name as the user account is created. The system administrator can choose to suppress the creation of the group with the same name as the account and instead assign new accounts to a general group called users or to another group (or groups) the administrator has previously created. Groups are simply a tool the administrator uses to manage security.

Syntax chmod [-option] mode filename

Dissection

- The argument can include the mode (permissions) and must include the file name. You can also use a wildcard to set the permissions on multiple files.

- Permissions are applied to owner (u), group (g), and others (o). The permissions are read (r), write (w), and execute (x). Use a plus sign (+) before the permissions to allow them or a hyphen (-) to disallow permissions. Octal permissions are assigned by a numeric value for each owner, group, and others.

Use the UNIX/Linux chmod command to set file permissions. In its simplest form, the chmod command takes as arguments a symbolic string (individual characters that are abbreviations

for permissions) followed by one or more file names. The symbolic string specifies permissions that should be granted or denied to categories of users. Here is an example: ugo+rwx. In the string, the characters ugo stand for user (same as owner), group, and others. These categories of users are affected by the chmod command. The next character, the + sign, indicates that permissions are being granted. The last set of characters, in this case rwx, indicates the permissions being granted. The symbolic string ugo+rwx indicates that read, write, and execute permissions are being granted to the owner, group, and others. The following is an example of how the symbolic string is used in a command to modify the access permissions of myfile:

```
chmod ugo+rwx myfile
```

The following command grants group read permission to the file customers:

```
chmod g+r customers
```

It is also possible to deny permissions with a symbolic string. The following command denies the group and others write and execute permissions for the file account_info.

```
chmod go-wx account_info
```

From your home directory, you can create any subdirectory and set permissions for it. However, you cannot create subdirectories outside your home directory unless the system administrator makes a special provision.

The octal permission format is another way to assign permissions; it assigns a number on the basis of the type of permission and on the basis of owner, group, and other. The type of permission is a number. For example, execute permission is assigned 1, write is 2, and read is 4. These permission numbers are added together for a value between 0 and 7. For instance, a read and write permission is a 6 (4 + 2) and read and execute is a 5 (4 + 1),as shown in the following list:

■ 0 is no permissions. ■ 1 is execute (same as x). ■ 2 is write (same as w). ■ 3 is write and execute (same as wx). ■ 4 is read (same as r). ■ 5 is read and execute (same as rx). ■ 6 is read and write (same as rw). ■ 7 is read, write, and execute (same as rwx).

One of these numbers is associated with each of three numeric positions (xxx) after the chmod command. The first position gives the permission number of the owner, the second position gives the permission number of the group, and the final position gives the

permission number of other. For example, the command chmod 755 myfile assigns read, write, and execute permissions to owner (7) for myfile; it assigns read and execute permissions to both group and other (5 in both positions). Here are some other examples:

■ chmod 711 data—For the file data, this command assigns read, write, and execute to owner; execute to group; and execute to other (programmers often use this for programs they write, enabling users to execute those programs).

■ chmod 642 data—For the file data, this command assigns read and write to owner; read to group; and write to other.

■ chmod 777 data—For the file data, this command assigns read, write, and execute to owner, group, and other.

■ chmod 755 data—For the file data, this command assigns read, write, and execute to owner; read and execute to group; and read and execute to other (another permission often used by programmers).

■ chmod 504 data—For the file data, this command assigns read and execute to owner; no permissions to group; and read permission to other.

If you want to set security on a directory to ensure that users must know the exact path to a file in that directory—so they can execute a program, but not snoop—configure the directory to have 711 permissions. This gives all permissions to the owner (you) and only the execute permission to group and others.

Some versions of UNIX/Linux include the umask command, which enables you to set permissions on multiple files at one time. This command is more complex than using chmod octal commands, but can save time for system administrators. For example, umask 022 grants rwx permissions for all users. However, you can also grant permissions on multiple files by using the wildcard asterisk (*) with chmod. For example, chmod 777 * grants full permission on all files in the current directory to all users and groups.

Now that you've learned about permissions, check out Table 2-5 for suggestions about setting permissions. Also, try Hands-on Project 2-12 to configure permissions. See Appendix B for a brief description of the chmod command.

Table 2-5 Suggestions for setting permissions

Type of File or Directory - Permissions Suggestion

System directories such as /bin, /dev,/etc, /sbin, /sys, and /usr Give all permissions to root (the owner), rx to /boot, group and others—chmod 755.

/root directory for the root account Give all permissions to root (the owner), rx to group, and no permissions to others—chmod 750.

Your home directory Give all permissions to owner (your account), x or no permissions to group, and no permissions to others—chmod 710 or chmod 700. (If you are a student and need to give your instructor access to your home directory, consider using chmod 705 so your instructor has rx permissions.)

A subdirectory under your home directory that you want to share with others so they can access and create files Give all permissions to owner (your account), group, and others—chmod 777.

A file in your home directory that people to be able to view, but not change Give all permissions to owner (your account), rx you want to group, and rx to others—chmod 755.

A file that should only be accessed by you Give all permissions to owner and no permissions to group and others—chmod 700.

An archived file in your home directory that should not be changed (just preserved) and that only you should be able to view Give rx permissions to owner and no permissions to group and others—chmod 500.

There are three advanced permissions that deserve brief mention so that you are aware of them: sticky bit, set user id (SUID) bit, and set group ID (SGID) bit. All three permissions are typically used by a system administrator for special purposes.

On older UNIX and Linux distributions, the sticky bit has been used to cause an executable program (a file you run as a program) to stay resident in memory after it is exited. This action ensures that the program is immediately ready to use the next time around or that it stays ready for multiple users on a server.In current operating systems,the sticky bit is used instead to enable a file to be executed, but only the file's owner or root have permission to delete or rename it. The symbol for the sticky bit is t (used in place of x), such as when you view permissions using ls -l. For example, when the sticky bit is set on a file, the permissions might look like: -rwxr-xr-t.

The SUID bit is generally used on programs and files used by programs. SUID gives the current user (user ID) temporary permissions to execute program-related files as though they are the owner. For example, programs on a multiuser system or server are usually installed by root. However, an ordinary user may need capabilities to execute and possibly modify files to run those programs as though they are the root account. Setting the SUID bit gives them the access they need to use the programs—temporarily treating the user as root (the owner).Even though someone is using the program with the SUID bit permission, root still retains actual ownership.

The SGID bit works similarly to SUID, but it applies to groups. For example, your company might have a group of people who use accounting files on a computer. The system administrator can create a group called accounting and, through the SGID bit, give temporary access as an owner to each member of the group while she or he is using the accounting programs and files. The symbol for SUID or SGID is an s. For example, when both SUID and SGID are set, the permissions on a file might look like -rwsr-sr-x (notice that the x permission is replaced with s for the owner and group).

CHAPTER SUMMARY

In UNIX/Linux, a file is the basic component for data storage. UNIX/Linux consider everything to be a file, even attached devices such as the monitor, keyboard, and printer. Even a directory,which can contain both files and subdirectories,is really just a special file in UNIX/Linux.

A file system is the UNIX/Linux systems' way of organizing files on storage devices such as hard disks and removable media such as CDs/DVDs. Files are stored in a file system, which is a hierarchical, treelike structure in which top-level directories contain subdirec- tories, which in turn can contain other subdirectories. Every file can be located by using a pathname—a listing of names of directories leading to a particular file.

The standard tree structure starts with the root (/) file system directory, which serves as the foundation for a hierarchical group of other directories and subdirectories.

The section of the disk that holds a file system is called a partition. One disk might have many partitions, each separated from the others so that it remains unaffected by external disturbances such as structural file problems associated with another partition. The UNIX/Linux file system is designed to allow access to multiple partitions after they are mounted in the tree structure.

A path, as defined in UNIX/Linux, serves as a map to access any file on the system. An absolute path is one that always starts at the root level. A relative path is one that starts at your current location.

You can customize your command prompt to display the current working directory name, the date, the time, and several other items.

The ls command displays the names of files and directories contained in a directory. The ls -l command, or long listing, displays detailed file information. The ls -a command shows hidden files.

Wildcard characters can be used in a command, such as ls, and take the place of other characters in a file name. In the Bash shell, the * wildcard can take the place of any string of characters, and the ? wildcard can take the place of any single character. The specific wildcards you can use are related to the shell.

You can use the mkdir command to create a new directory as long as you own the parent directory. A file's original owner is the person who creates it, and he becomes the one who controls access to it (although the root account also can control access).

Use the cp command to copy a source file to a destination file. UNIX/Linux might overwrite the destination file without warning unless you use the -i option. The dot notation (current directory) is a shorthand way to specify the destination in a cp command.

You can use the chmod command to set permissions for files that you own. The basic permission settings are rwx, which mean read, write, and execute, respectively. File permissions are set to control file access by three types of users: the owner (u), the group (g), and others (o). You must remember to change permission settings on any directories you own if you want others to access information in those directories. Also, to run a program file, the intended users (owner, group, others) must have execute permissions.

COMMAND SUMMARY: REVIEW OF CHAPTER 2 COMMANDS

cd Changes directories (with no options,. Changes to the current working cd goes to your home directory) directory.

.. Changes to the parent directory.

chmod Sets file permissions for specified files + assigns permissions. -removes permissions.

cp Copies files from one directory to another

-b makes a backup of the desti- nation file, if an original one already exists (so you have a backup if overwriting a file).

-i prevents overwriting of the destination file without warning. -u overwrites an existing file only if the source is newer than the file in the current destination.

ls Displays a directory's contents, includ- ing its files and subdirectories

-a lists the hidden files. -l (lowercase L) generates a long listing of the directory. -r sorts the listing in reverse order. -S sorts the listing by file size. -t sorts by the time when the file or directory was last modified. -X sorts by extension.

mkdir Makes a new directory -v verifies that the directory is made.

mount Connects the file system partitions to the directory tree when the system starts, and mounts additional devices, such as the CD/DVD drive

-t specifies the type of file system to mount.

pwdDisplays your current path

rm Removes a file-i prompts before you delete the file.

rmdir Removes an empty directory

-v provides a message to verify the directory is removed.

umask Sets file permissions for multiple files

umount Disconnects the file system partitions from the directory tree

---

KEY TERMS

/boot partition — A partition that is used to store the operating system files that compose the kernel. /home partition — A partition that is on the home directory and provides storage space for all users' directories. A separate section of the hard disk, it protects and insulates users' personal files from the UNIX/Linux operating system software.

/usr partition — A partition in which to store some or all of the nonkernel operating system programs that will be accessed by users. /var partition — A partition that holds temporarily created files, such as files used for printing documents and log files used to record monitoring and administration data. absolute path — A pathname that begins at the root file system directory and lists all subdirectories to the destination file.

binaries — The programs residing in the /bin directory and elsewhere that are needed to start the system and perform other essential tasks. See also executables.

block special file — In UNIX/Linux, a file used to manage random access devices that involve handling blocks of data, including CD/DVD drives, hard disk drives, tape drives, and other storage devices. Also called a block device file.

bootstrap loader — A utility residing in the /boot directory that starts the operating system.

character special file — A UNIX/Linux I/O management file used to handle byte-by- byte streams of data, such as through serial or USB connections, including terminals, printers, and network communications. Also called a character device file.

child — A subdirectory created and stored within a (parent) directory.

device special file — A file used in UNIX/Linux for managing I/O devices. It can be one of two types: block special file or character special file.

directory — A special type of file that can contain other files and directories. Directory files store the names of regular files and other directories, called subdirectories. Enhanced IDE (EIDE) — An improved version of IDE that offers faster data transfer speeds and is commonly used in modern computers. See also Integrated Drive Electronics. executables — The programs residing in the /bin directory that are needed to start the system and perform other essential tasks. See also binaries. extended file system (ext or ext fs) — The file system designed for Linux that is installed, by default, in Linux operating systems. It enables the use of the full range of built-in Linux commands, file manipulation, and security. Released in 1992, ext had some bugs and supported only files of up to 2 GB. In 1993, the second extended file system (ext2 or ext2 fs) was designed to fix the bugs in ext, and supported files up to 4 TB. In 2001, ext3 (or ext3 fs) was introduced to enable journaling for file and data recovery. ext4 was introduced in 2006, enabling a single volume to hold up to 1 exabyte of data and supporting the use of extents. ext, ext2, ext3, and ext4 support file names up to 255 characters.

extent — A portion of a disk, such as a block or series of blocks, that is reserved for a file and that represents contiguous space, so that as the file grows, all of it remains in the same location on disk. The use of extents reduces file fragmentation on a disk, which reduces disk wear and the time it takes to retrieve information.

file — The basic component for data storage. file system — An operating system's way of organizing files on mass storage devices, such as hard and floppy disks. The organization is hierarchical and resembles an inverted tree. In the branching structure, top-level files (or folders or directories) contain other files, which in turn contain other files. group id (GID) — A number used to identify a group of users. hidden file — A file that the operating system uses to keep configuration information, among other purposes. The name of a hidden file begins with a dot. hot fixes — The ability to automatically move data on damaged portions of disks to areas that are not damaged. information node, or inode — A system for storing essential information about direc- tories and files. Inode information includes (1) the name of a directory or file, (2) general information about that directory/file, and (3) information (a pointer) about how to locate the directory/file on a disk partition. Integrated Drive Electronics (IDE) — Sometimes called Integrated Device Electronics, the most popular electronic hard disk interface for personal computers. This is the same as the ANSI Advanced Technology Attachment (ATA) standard. journaling — The process of keeping chronological records of data or transactions so that if a system crashes without warning, the data or transactions can be reconstructed or backed out to avoid data loss or information that is not properly synchronized. mount — The process of connecting a file system to the directory tree structure, making that directory accessible. parent — The directory in which a subdirectory (child) is created and stored. partition — A separate section of a disk that holds a file system and that is created so activity and problems occurring in other partitions do not affect it. pathname — A means of specifying a file or directory that includes the names of directories and subdirectories on the branches of the tree structure. A forward slash (/) separates each directory name. For example, the pathname of the file phones (the destination file) in the source directory of Jean's directory within the /home directory is /home/jean/source/phones. peripherals —The equipment connected to a computer via electronic interfaces.Examples include hard and floppy disk drives, printers, and keyboards. permission — A specific privilege to access and manipulate a directory or file, for example, the privilege to read a file.

physical file system — A section of the hard disk that has been formatted to hold files. relative path — A pathname that begins at the current working directory and lists all subdirectories to the destination file. root file system directory — The main or parent directory (/) for all other directories (the highest level of the file system); also can refer to the directory in which the system administrator's files are stored (/root).

set group ID (SGID) bit — Enables the owner of a program to keep full ownership, but also gives members of a group temporary ownership while executing that program. set user ID (SUID) bit — Enables the owner of a program to retain full ownership, but also gives an ordinary user temporary ownership while executing that program.

shared library images — The files residing in the /lib directory that programmers use to share code, rather than copying this code into their programs. Doing so makes their programs smaller and faster. Small Computer System Interface (SCSI) — Pronounced "scuzzy," a popular and fast electronic hard disk interface commonly used on network servers. SCSI is actually a set of standards that defines various aspects of fast communications with a hard disk.

sticky bit — An executable permission that either causes a program to stay resident in memory (on older UNIX/Linux systems) or ensures that only root or the owner can delete or rename a file (on newer systems). subdirectory — A directory under a higher or parent directory.

superblock — A special data block on a partition that contains information about the layout of blocks. This information is the key to finding anything on the file system, and it should never change. swap partition — A section of the hard disk separated from other sections so that it functions as an extension of memory, which means it supports virtual memory. A computer system can use the space in this partition to swap information between disk and RAM so the computer runs faster and more efficiently.

symbolic link — A name or file name that points to and lets you access a file using a different name in the same directory or a file using the same or a different name in a different directory. UNIX file system (ufs) — A hierarchical (tree structure) file system supported in most versions of UNIX/Linux. It is expandable, supports large storage, provides excellent security, is reliable, and employs information nodes (inodes).

utility — A program that performs useful operations such as copying files, listing directories, and communicating with other users. Unlike other operating system programs, a utility is an add-on and not part of the UNIX/Linux shell, nor a component of the kernel. virtual file system — A system that occupies no disk space, such as the /proc directory. The virtual file system references and lets you obtain information about which programs and processes are running on a computer.

virtual memory — A memory resource supported by the swap partition, in which the system can swap information between disk and RAM, allowing the computer to run faster and more efficiently.

virtual storage — The storage that might be allocated via different disks or file systems (or both), but that is transparently accessible as storage to the operating system and users. wildcard — A special character that can stand for any other character or, in some cases, a group of characters and is often used in an argument, such as ls file.* .

CHAPTER 3 MASTERING EDITORS

After reading this chapter and completing the exercises, you will be able to:

♦ Explain the basics of UNIX/Linux files, including ASCII, binary, and executable files

♦ Understand the types of editors ♦ Create and edit files using the vi editor ♦ Create and edit files using the Emacs editor

The ability to create and modify the contents of files is a fundamental skill not only in producing documents such as memos, reports, and letters, but also in writing programs and customizing system configuration files. All operating systems, including UNIX/Linux, provide one or more editors that enable you to work with the contents of files.

This chapter introduces two important UNIX/Linux editors, which you use throughout the rest of this book. The vi editor provides basic editing functions and is often preferred by UNIX/Linux administrators and programmers for its simplicity. The Emacs editor offers more sophisticated editing capabilities for writing all kinds of documents as well as programs. Both of these popular editors can be started from the command line and are included in most versions of UNIX/Linux.

UNDERSTANDING UNIX/LINUX FILES

Almost everything you create in UNIX/Linux is stored in a file. All information stored in files is in the form of binary digits. A binary digit, called a bit for short, is in one of two states. The states are 1 (on) and 0 (off). They can indicate, for example, the presence or absence of voltage in an electronic circuit. Because the computer consists of electronic circuits that are either in an on or off state, binary numbers are perfectly suited to report these states. The exclusive use of 0s and 1s as a way to communicate with the computer is known as machine language. The earliest programmers had to write their programs using machine language, a tedious and time-consuming process.

ASCII TEXT FILES

To make information stored in files accessible, computer designers established a standard method for translating binary numbers into plain English. This standard uses a string of eight binary digits, called a byte, which is the abbreviation for binary term. A byte can be configured into fixed patterns of bits, and these patterns can be interpreted as an alphabetic character, decimal number, punctuation mark, or a special character, such as &, *, or @. Each byte, or code, has been standardized into a set of bit patterns known as ASCII. ASCII stands for the American Standard Code for Information Interchange. Computer files containing nothing but printable characters are called text files, and files that contain nonprintable characters, such as machine instructions, are called binary files. The ASCII character set represents 256 characters. Figure 3-1 lists the printable and nonprintable ASCII characters.

Many nonprintable ASCII characters are available. Some examples are charac- ters used to control printers, such as the escape (ESC) character to show the start of a printing command, a form feed (FF) character, and a line feed (LF) character. If you use Microsoft Word or OpenOffice.org Writer, you are familiar with other nonprinting characters, such as the paragraph symbol. You can view nonprinting characters in Microsoft Word or OpenOffice.org Writer by clicking the Show/Hide ¶ or Nonprinting Characters button (paragraph symbol) on the Standard toolbar.

Some operating systems also support Unicode. Unicode offers up to 65,536 characters, although not all of the possible characters are currently defined. Unicode was developed because the 256 characters in ASCII are not enough for some languages, such as Chinese, that use more than 256 characters. Visit www.unicode.org to learn more about Unicode.

BINARY FILES

Computers are not limited to processing ASCII codes. To work with graphic information, such as icons, illustrations, and other images, binary files can include strings of bits representing white and black dots, in which each black dot represents a 1 and each white dot

PRINTING CHARACTERS (PUNCTUATION CHARACTERS)

Dec Octal

32 040 33 041 34 042 35 043 36 044 37 045 38 046 39 047 40 050 41 051 42 052 43 053 44 054 45 055 46 056 47 057

Hex ASCII

20 (Space) 21 ! 22 “ 23 # 24 $ 25 % 26 & 27 ‘

28 (29) 2A * 2B + 2C , 2D - 2E . 2F /

Printing Characters (Alphabet—Uppercase)

Dec Octal

65 101 66 102 67 103 68 104 69 105 70 106 71 107 72 110 73 111 74 112 75 113 76 114 77 115 78 116 79 117 80 120 81 121 82 122 83 123 84 124 85 125 86 126 87 127 88 130 89 131 90 132

Hex ASCII

41 A 42 B 43 C 44 D 45 E 46 F 47 G 48 H 49 I 4A J 4B K 4C L 4D M 4E N 4F O 50 P 51 Q 52 R 53 S 54 T 55 U 56 V 57 W 58 X 59 Y 5A Z

Printing Characters (Alphabet—Lowercase) Dec Octal Hex ASCII

97 141 61 a 98 142 62 b 99 143 63 c

100 144 64 d 101 145 65 e 102 146 66 f 103 147 67 g 104 150 68 h 105 151 69 i 106 152 6A j 107 153 6B k 108 154 6C l 109 155 6D m 110 156 6E n 111 157 6F o 112 160 70 p 113 161 71 q 114 162 72 r 115 163 73 s 116 164 74 t 117 165 75 u 118 166 76 v 119 167 77 w 120 170 78 x 121 171 79 y 122 172 7A z

(Decimal Numbers—Print)

Dec Octal

48 060 49 061 50 062 51 063 52 064 53 065 54 066 55 067 56 070 57 071

Hex ASCII

30 0 31 1 32 2 33 3 34 4 35 5 36 6 37 7 38 8 39 9

(Special Characters—Print)

Dec Octal

58 072 59 073 60 074 61 075 62 076 63 077 64 080

Hex ASCII

3A : 3B ; 3C < 3D = 3E > 3F ? 40 @

Nonprinting Characters (Abridged)

Control Characters Dec Octal Hex

0 000 00 7 007 07 8 010 08 9 011 09 10 012 0A

11 013 0B 12 014 0C 13 015 0D

ASCII

^@ (Null) Bell Backspace Tab

Line Feed, Newline Vertical tab Form feed Carriage return

Figure 3-1 ASCII characters

represents a 0. Graphics files include bit patterns—rows and columns of dots called a bitmap—that must be translated by graphics software, commonly called a graphics viewer, which transforms a complex array of bits into an image.

Executable Program Files

Many programmers develop source code for their programs by writing text files; then, they compile these files to convert them into executable program files. Compiling is a process of translating a program file into machine-readable language. Programmers and users also develop scripts, which are files containing commands. Scripts are typically not compiled into machine code prior to running, but are executed through an interpreter. At the time the script is run, the interpreter looks at each line and converts the commands on each line into actions taken by the computer. Scripts are interpreted program files that are executable. You learn more about writing program code, scripts, compilers, and interpreters later in this book.

Compiled and interpreted files that can be run are called executable program files (or sometimes just executables). These files can be run from the command line.

USING EDITORS

An editor is a program for creating and modifying files containing source code, text, data, memos, reports, and other information. A text editor is like a simplified word-processing program; you can use a text editor to create and edit documents, but many text editors do not allow you to format text using boldface text, centered text, or other text-enhancing features.

Editors let you create and edit ASCII files. UNIX/Linux normally include the two editors vi and Emacs. They are screen editors, because they display the text you are editing one screen at a time and let you move around the screen to change and add text. Both are text editors as well, because they work like simple word processors. You can also use a line editor to edit text files. A line editor lets you work with only one line or group of lines at a time. Although line editors do not let you see the context of your editing, they are useful for general tasks, such as searching, replacing, and copying blocks of text. In UNIX/Linux, however, most users prefer vi or Emacs to using a simple line editor, which is another reason why vi and Emacs are included with UNIX/Linux systems.

The vi and Emacs editors do not offer the same functionality as GUI-based editors such as Microsoft Word (although Emacs now has lots of functionality and “snap-ins” for extra functions like editing Web documents). Also, UNIX/ Linux systems can use more sophisticated GUI editors, such as OpenOffice.org Writer, gedit in the GNOME desktop, and KEdit in the KDE desktop. However, both vi and Emacs are typically preferred for system, configuration, and programming activities, because they are quickly initiated from the command line and offer a simple, direct way to perform critical editing tasks.

USING THE VI EDITOR

The vi editor is so called because it is visual—it immediately displays on screen the changes you make to text. It is also a modal editor; that is, it works in three modes: insert mode, command mode, and extended (ex) command set mode.

Insert mode, which lets you enter text, is accessed by typing the letter “i” after the vi editor is started. Command mode, which is started by pressing Esc, lets you enter commands to perform editing tasks, such as moving through the file and deleting text. Ex mode employs an extended set of commands that were initially used in an early UNIX editor called ex. You can access this mode by pressing Esc to enter command mode, and then typing a colon (:) to enter extended commands at the bottom of the screen.

You can simulate a line editor using vi by starting the vi editor with the -e option (vi -e filename), which places vi exclusively in ex mode. Also, when you open vi, it is set up by default to edit a text file. You can edit a binary file by using the -b option with vi.

To use the vi editor, it is important to master the following tasks: ■ Creating a file

■ Inserting, editing, and deleting text ■ Searching and replacing text ■ Adding text from other files ■ Copying, cutting, and pasting text ■ Printing a file

■ Saving a file ■ Exiting a file

Different versions of the vi editor are included in different versions of UNIX/Linux. The commands described in this chapter generally apply to most UNIX/Linux vi editor versions. However, they particularly apply to the vi editor in Fedora, Red Hat Enterprise Linux, and SUSE, which is technically called the vim (vi improved) editor.

CREATING A NEW FILE IN THE VI EDITOR

To create a new file in the vi editor: 1. Access the command line.

2. Enter vi plus the name of the file you want to create, such as vi data.

These steps open the vi editor and enable you to begin entering text in the file you specify. Remember, though, at this point the file is in memory and is not permanently saved to the disk until you issue the command to save it.

As you enter text, the line containing the cursor is the current line. Lines containing a tilde (~) are not part of the file; they indicate lines on the screen only, not lines of text in the file. (See Figure 3-2.)

Figure 3-2 Creating a new file in the vi editor Hands-on Project 3-1 enables you to create a new file using the vi editor.

Sometimes you might open the vi editor without specifying the file name with the vi command on the command line. You can save the file and specify a file name at any time by pressing Esc, typing :w filename, and pressing Enter.

INSERTING TEXT

When you start the vi editor, you’re in command mode. This means that the editor interprets anything you type on the keyboard as a command. Before you can insert text in your new file, you must use the i (insert) command. In insert mode, every character you type appears on the screen. You can return to command mode at any time by pressing the Esc key.

Try Hands-on Project 3-2 to insert text in the vi editor.

REPEATING A CHANGE

The vi editor offers features that can save you time. One such feature is the ability to replicate any changes you make. When you are in command mode, you can use a period (.) to repeat the most recent change you made. This is called the repeat command and it can save you time when typing the same or similar text. Hands-on Project 3-3 uses the repeat command.

MOVING THE CURSOR

When you want to move the cursor to a different line or to a specific position on the same line, use command mode (press Esc). In command mode, you can move forward or back one word, move up or down a line, go to the beginning of the file, and so on. Table 3-1 summarizes useful cursor-movement commands. You can practice moving the cursor in Hands-on Project 3-4.

KEY	MOVEMENT
h or left arrow	Left one character position
l or right arrow	Right one character position
k or up arrow	Up one line
j or down arrow	Down one line
H	Upper-left corner of the screen
L	Last line on the screen
G	Beginning of the last line
nG	The line specified by a number, n
W	Forward one word
b	Back one word
0 (zero)	Beginning of the current line
$	End of the current line
Ctrl+u	Up one-half screen
Ctrl+d	Down one-half screen
Ctrl+f or Page Down	Forward one screen
Ctrl+b or Page Up	Back one screen

Remember that the Ctrl key combinations and the letter keys shown in Table 3-1 are designed to work in command mode. The arrow keys, which are used for moving around text, work in both command and insert mode. Try Hands-on Project 3-4 to practice using vi in command mode to move around in a file.

Using the letter keys to move the cursor can be traced to the time when UNIX/Linux used teletype terminals that had no arrow keys. Designers of vi chose the letter keys because of their relative position on the keyboard.

DELETING TEXT

The vi editor employs several commands for deleting text when you are in command mode. For example, to delete the text at the cursor, type x. Use dd in command mode to delete the current line. Use dw to delete a word or to delete from the middle of a word to the end of the word. To delete more than one character, combine the delete commands with the cursor movement commands you learned in the preceding section. Table 3-2 summarizes the most common delete commands.

Table 3-2 vi editor's delete commands

COMMAND	PURPOSE
X	Delete the character at the cursor.
dd	Delete the current line (putting it in a buffer so it can also be pasted back into the file).
dw	Delete the word starting at the cursor. If the cursor is in the middle of the word, delete from the cursor to the end of the word.
d$	Delete from the cursor to the end of the line.
d0	Delete from the cursor to the start of the line.

The command to delete a line, dd, actually places deleted lines in a buffer. You can then use the command p to paste deleted (cut) lines elsewhere in the text. (Position the cursor where you want to paste the information.) To copy and paste text, use the "yank" command, yy, to copy the lines. After yanking the lines you want to paste elsewhere, move the cursor, and type p to paste the text in the current location. You learn more about the p and yy commands in later sections of this chapter.

Hands-on Project 3-5 gives you the opportunity to practice using delete commands.

Undoing a Command

If you complete a command and then realize you want to reverse its effects, you can use the undo (u) command. For example, if you delete a few lines from a file by mistake, type u to restore the text.

Searching for a Pattern

You can search forward for a pattern of characters by typing a forward slash (/), typing the pattern you are seeking, and then pressing Enter. Programmers often call this a "string search." For example, suppose you want to know how many times you used the word "insure" in a file. First, go to the top of the file, type /insure, and press Enter to find the first instance of insure. To find more instances, type n while you are in command mode.

When placed after the forward slash, several special characters are supported by vi when searching for a pattern. For example, the special characters \> are used to search for the next word that ends with a specific string. If you enter /te\> you find the next word that ends with "te," such as "write" or "byte."The characters \< search for the next word that begins with a specific string, such as using /\<top to find the next word that begins with "top," which might be "topology," for example. The ^ special character searches for the next line that begins with a specific pattern. For instance, /^However finds the next line that starts with "However." Use a period as a wildcard to match characters. For example, /m.re finds "more" and "mere," and /s..n finds "seen," "soon," and "sign." Also, use brackets [ ] to find any of the characters between the brackets, such as /theat[er] to find "theater" or "theatre," and /pas[st] to find "pass" or "past." Finally, use $ to find a line that ends with a specific character. For instance /!$ finds a line that ends with an exclamation point "!". You can type n after searching with any of these special characters to find the next pattern that matches. Table 3-3 summarizes the special characters used to match a pattern.

Special Character	**Purpose**
\>	Searches for the next word that ends with a specific string.
\<	Searches for the next word that begins with a specific string.
.	Acts as a wildcard for one character.
[]	Finds the characters between the brackets.
$	Searches for the line that ends with a specific character.
*All of these special characters must be preceded with a slash (/) from the command mode.	

Hands-on Project 3-6 provides experience in pattern matching.

If you are in an editing session and want to review information about the file status, press Ctrl+g or Ctrl+G (you can use uppercase G or lowercase g). The status line at the bottom of the screen displays information, including line- oriented commands and error messages.

Searching and Replacing

Suppose you want to change all occurrences of "insure" in the file you are editing to "ensure." Instead of searching for "insure," and then deleting it and inserting "ensure," you can search and replace with one command. The commands you learned so far are screen-oriented. Commands that can perform more than one action (searching and replac- ing) are line-oriented commands and they operate in ex mode.

Screen-oriented commands execute at the location of the cursor. You do not need to tell the computer where to perform the operation because it takes place relative to the cursor. Line-oriented commands, on the other hand, require you to specify an exact location (an address) for the operation. Screen-oriented commands are easy to type, and their changes appear on the screen. Typing line-oriented commands is more complicated, but they can execute independently of the cursor and in more than one place in a file, saving you time and keystrokes.

A colon (:) precedes all line-oriented commands. It acts as a prompt on the status line, which is the bottom line on the screen in the vi editor. You enter line-oriented commands on the status line, and press Enter when you complete the command.

In this chapter, all instructions for line-oriented commands include the colon as part of the command.

For example, to replace all occurrences of "insure" with "ensure," you first enter command mode (press Esc), type :1,$s/insure/ensure/g, and press Enter. This command means access the ex mode (:),beginning with the first line (1) to the end of the file ($),search for"insure," and replace it with "ensure" (s/insure/ensure/) everywhere it occurs on each line (g).

Try Hands-on Project 3-7 to use a line-oriented command for searching and replacing.

SAVING A FILE AND EXITING VI

As you edit a file, periodically saving your changes is a good idea. This is especially true if your computer is not on an uninterruptible power supply (UPS). A UPS is a device that provides immediate battery power to equipment during a power failure or brownout.

You can save a file in several ways. One way is to enter command mode and type :w to save the file without exiting. (See Figure 3-3.) If you are involved in a relatively long editing session, consider using this command every 10 minutes or so to periodically save your work. If you want to save your changes and exit right away, use :wq or ZZ from command mode. You can also use :x to save and exit. You should always save the file before you exit vi; otherwise, you will lose your changes. Hands-on Project 3-8 enables you to save your changes and exit an editing session.

Figure 3-3 Saving without exiting

ADDING TEXT FROM ANOTHER FILE

Sometimes, the text you want to include in one file is already part of another file. For example, suppose you already have a text file that lists customer accounts and you have another file called customerinfo that contains customer information. You want to copy the customer accounts text into the customerinfo file and make further changes to the customerinfo file. It is much easier to use the vi command to copy the text from the accounts file into the customerinfo file than it is to retype all of the text. To copy the entire contents of one file into another file: (1) use the vi editor to edit the file you want to copy into; and (2) use the command :r filename, where filename is the name of the file that contains the information you want to copy. Hands-on Project 3-9 enables you to copy from one file into another.

LEAVING VI TEMPORARILY

If you want to execute other UNIX/Linux commands while you work with vi, you can launch a shell or execute other commands from within vi. For example, suppose you're working on a text or program file and you want to leave to check the calendar for the current month. To view the calendar from command mode, type :!cal (a colon, an exclamation point, and the command) and press Enter. This action executes the cal command, and when you press Enter again, you go back into your vi editing session. Using :! plus a command-line command enables you to start a new shell, run the command, and then go back into the vi editor.

When you want to run several command-line commands in a different shell without first closing your vi session, use the Ctrl+z option to display the command line. (See Figure 3-4.) When you finish executing commands, type fg to go back into your vi editing session.

Using Ctrl+z in this context is really a function of the Bash shell, which in this example leaves the vi editor running in the background and takes you to the shell command line. When you enter fg, this is a shell command that brings the job you left (the vi editing session) back to the foreground.

Hands-on Project 3-10 enables you to use the :! and Ctrl+z commands from a vi editing session.

You can set up a script file (a file of commands) that automatically runs when you launch vi. The file is called .exrc and is a hidden file located in your home directory. This file can be used to automatically set up your vi environment. Programmers, for example, who want to view line numbers in every editing session might create an .exrc file and include the set number command in the file. To learn more about scripts, see Chapters 6 and 7 ("Introduction to Shell Script Programming" and "Advanced Shell Programming").

Changing Your Display While Editing

Besides using the vi editing commands, you can also set options in vi to control editing parameters, such as using a line number display. Turn on line numbering when you want to work with a range of lines, for example, when you're deleting or cutting and pasting blocks of text. Then, you can refer to the line numbers to specify the text.

Figure 3-4 Accessing a shell command line from the vi editor

To turn on line numbering, use the :set number command. Then, if you want to delete lines 4 through 6, for example, it is easy to determine that these are the lines you intend to delete, and you simply use the :4,6d command to delete them. Try Hands-on Project 3-11 to turn on line numbering and then refer to it for deleting text.

Copying or Cutting and Pasting

You can use the yy command in vi to copy a specified number of lines from a file and place them on the clipboard. To cut the lines from the file and store them on the clipboard, use the dd command. After you use the dd or yy commands, you can use the p command to paste the contents in the clipboard to another location in the file. These commands are handy if you want to copy text you already typed and paste it in another location. Hands-on Project 3-12 enables you to cut text and paste it in another location within the same file.

Printing Text Files

Sometimes you want to print a file before you exit the vi editor. You can use the lpr (line print) shell command to print a file from vi. Type !lpr and then type the name of the file you want to print. Hands-on Project 3-13 enables you to print a file on which you are working in the vi editor.

You can also specify which printer you want to use via the -P printer option, where printer is the name of the printer you want to use. For example, you might have two printers, lp1 and lp2. To print the file accounts to lp2, enter :!lpr -P lp2 accounts and press Enter.

Syntax lpr [-option] [filename]

Dissection

■ ■

The argument consists of the name of the file to print. Options include:

-P specifies the destination printer; you include the name of the printer just after the option.

-# specifies the number of copies to print (up to 100 copies). -r deletes a print file after it is printed (saving disk space).

Canceling an Editing Session

If necessary, you can cancel an editing session and discard all the changes you made in case you change your mind. Another option is to save only the changes you made since last using the :w command to save a file without exiting vi. In Hands-on Project 3-14, you exit a file without saving your last change.

GETTING HELP IN VI

You can get help in using vi at any time you are in this editor. To access the online help documentation while you are editing a file, use the help command. You can access help documentation after you start the vi editor by pressing Esc, then typing a colon (:), and then help. You access the help documentation in Hands-on Project 3-14.

You can also view documentation about vi using the man vi command. While you are in vi, press Esc, type :!man vi, and press Enter (type q and press Enter to go back into your editing session). Or, when you are at the shell command line and not in a vi session, type man vi and press Enter.

USING THE EMACS EDITOR

Emacs is another popular UNIX/Linux text editor. Unlike vi, Emacs is not modal. It does not switch from command mode to insert mode. This means that you can type a command without verifying that you are in the proper mode. Although Emacs is more complex than vi, it is more consistent. For example, you can enter most commands by pressing Alt or Ctrl key combinations.

Emacs also supports a sophisticated macro language. A macro is a set of commands that automates a complex task. Think of a macro as a "superinstruction." Emacs has a powerful command syntax and is extensible. Its packaged set of customized macros lets you read electronic mail and news and edit the contents of directories. You can start learning Emacs by learning its common keyboard commands. Although Emacs also supports many conventional mouse-based menu options, it is worth your time to learn the keyboard commands. When you know these, you can often navigate and use Emacs keyboard features even faster than going through menus to find the same options. Table 3-4 lists the Emacs keyboard commands.

TABLE 3-4 COMMON EMACS COMMANDS

COMMAND	PURPOSE
Alt+<	Move the cursor to the beginning of the file.
Alt+>	Move the cursor to the end of the file.
Alt+b	Move the cursor back one word.
Alt+d	Delete the current word.
Alt+f	Move the cursor forward one word (moving space to space between words).
Alt+q	Reformat current paragraph using word wrap so that lines are full.
Alt+t	If the cursor is under the first character of the word, transpose the word with the preceding word; if the cursor is not under the first character, transpose the word with the following word.
Alt+u	Capitalize all letters from the cursor position in a word to the end of that word.
Alt+w	Scroll up one screen.
Alt+x doctor	Enter doctor mode to play a game in which Emacs responds to your statements with questions. Save your work first. Not all versions support this mode.
Ctrl+@	Mark the cursor location. After moving the cursor, you can move or copy text to the mark.
Ctrl+a	Move the cursor to the beginning of the line.
Ctrl+b	Move the cursor back one character.

Ctrl+e	Move the cursor to the end of the line.
Ctrl+f	Move the cursor forward one character.
Ctrl+g	Cancel the current command.
Ctrl+h	Use online help.
Ctrl+k	Delete text to the end of the line.
Ctrl+n	Move the cursor to the next line.
Ctrl+p	Move the cursor to the preceding line.
Ctrl+t	Transpose the character before the cursor and the character under the cursor.
Ctrl+v	Scroll down one screen.
Ctrl+w	Delete the marked text. Press Ctrl+y to restore deleted text.
Ctrl+y	Insert text from the file buffer, and place it after the cursor.
Ctrl+h, t	Run a tutorial about Emacs.
Ctrl+x, Ctrl+c	Exit Emacs.
Ctrl+x, Ctrl+s	Save the file.
Ctrl+x, u	Undo the last change.
Ctrl+Del	Delete text from the current cursor location to the end of the current word.

In most instances, Ctrl and Alt commands in Emacs are not case sensitive, so Alt+B and Alt+b are the same command.

If you are using Emacs through a remote connection, such as through SSH (see Chapter 1, “The Essence of UNIX and Linux”), try pressing Ctrl+F10 or just F10 to access the menu bar mode. The menu choices appear in a buffer window displayed under your main editing window. This is very useful when you cannot use the common Emacs commands through a remote connection.

CREATING A NEW FILE IN EMACS

Start Emacs by entering the emacs command in the terminal window or at a command line in UNIX/Linux. If you type a file name after this command, Emacs creates a new, blank file with that name or opens an existing file with that name. If you type emacs with no file name, Emacs automatically displays several introductory screens, beginning with the one shown in Figure 3-5 for Fedora. The Emacs window runs under the X Window desktop you have configured, such as GNOME.

Figure 3-5 Emacs opening screen (without a file name) in Fedora with GNOME

When you start Emacs without specifying a file to open, the introductory screen display ends on a screen on which you can type notes in a buffer you do not plan to save. You can open an existing file to edit by typing Ctrl+x then Ctrl+f and entering the path and name of the file.

If you have installed a desktop, such as GNOME, there is likely to be a menu option for starting Emacs. At this writing, in Fedora and Red Hat Enterprise Linux with the GNOME desktop you can open Emacs by clicking Applications, pointing to Programming, and clicking Emacs Text Editor. In SUSE, click the Computer menu, click More Applications, click Utilities in the left pane, and click Emacs.

As Figure 3-5 illustrates, there is a menu bar at the top of the Emacs screen. When you click one of the items, such as File, a menu appears. The default menu bar has the following categories:

-
-

■

■

■

■

You also

■ ■ ■ ■ ■ ■ ■ ■

File—Provides options for operations such as opening a file, opening a directory, saving information in a buffer, inserting information from another file, going into the split window mode, and closing the currently open file (buffer)

Edit—Offers text-editing functions, such as undoing a change, cutting or copying text, pasting text, and so on

Options—Provides all kinds of special options, such as syntax highlighting, region highlighting, word wrap modes, file decompression/compression, debugger options, and "mule" options that are used to set the language environment, fonts, and input method

Buffers—Enables you to open any of the editor's storage buffers that currently hold information, including the text that is already in the file

Tools—Provides options for compiling a program file, executing a shell command, checking the spelling of text, comparing or merging files, installing Emacs patches, reading and sending e-mail, and searching a directory

Help—Provides assistance through access to manuals, a tutorial, Emacs FAQs, and the Emacs psychiatrist, which lets you ask Emacs questions

see an icon bar under the menu bar that provides options for the following: Reading a file into Emacs Reading a directory to access its files Exiting and not saving the current buffer

Saving the contents of the current buffer to a file Copying the current buffer contents to a different file Undoing the most recent task you performed Cutting text

Copying text

■ Pasting text ■ Searching ahead for a word pattern or string ■ Printing the contents of the current buffer ■ Configuring your editing preferences ■ Viewing the Help menu

NAVIGATING IN EMACS

To create a new file in Emacs, type emacs plus the file name at the command line, such as emacs research. After you start Emacs, to navigate in the file, you can use either the cursor movement keys—such as the arrow keys, Page Down, Page Up, Home, and End—or Ctrl/Alt key combinations, such as Alt+f to move the cursor forward one word (see Table 3-4). When you want to save your work: (1) use the File menu; (2) use the icon to save the current buffer to the file; or (3) press Ctrl+x, Ctrl+s. Also, to exit Emacs, use the File menu, Exit Emacs option, or enter Ctrl+x, Ctrl+c.

In Hands-on Project 3-15, you create a file and practice saving and exiting. In Hands-on Project 3-16, you practice navigating in a file.

DELETING INFORMATION

You can use the Del or Backspace keys to delete individual characters in Emacs. Also, use Ctrl+k to delete to the end of a line. If you decide to undo a deletion, use Ctrl+x, u (do not press Ctrl with the u) to repeatedly undo each deletion. Hands-on Project 3-17 enables you to delete text and then undo your deletion.

COPYING, CUTTING, AND PASTING TEXT

In Emacs, you can insert text simply by typing. You can also insert text by copying and pasting, or by cutting and pasting. Before you copy or cut text, you first need to mark the text with which to work. When you use command keys, navigate to the beginning of the text you want to replicate and press Ctrl+Spacebar. Next, navigate to the end of the text you want to include and press Alt+w to copy the text, or press Ctrl+w to cut the text. Next, move the cursor where you want to place the copied or cut text and press Ctrl+y (the yank command). This might sound confusing at first; the best way to learn the process is by doing it. Hands-on Project 3-18 enables you to copy and paste text in Emacs.

SEARCHING IN EMACS

Like the vi editor, Emacs lets you search for specific text. One way to search is by pressing Ctrl+s, entering on the status line the string of characters you want to find, and pressing Ctrl+s repeatedly to find each occurrence. You can also use Ctrl+r to search backward.

Another way to search for a string is to use the Search forward for a string icon or to click the Edit menu, point to Search, and click Search. In both cases, you then type the search string on the status line and press Enter. Try Hands-on Project 3-19 to search for a string.

REFORMATTING A FILE

Often, as you create a document, you want to set it up so that the lines automatically wrap around from one line to the next. Use the Alt+q command to turn on the word wrap feature in Emacs. Hands-on Project 3-20 teaches you to use word wrap.

GETTING HELP IN EMACS

Emacs comes with extensive documentation and a tutorial. The Emacs tutorial is a good way to get up to speed quickly. Click the Help menu and click Emacs Tutorial; or, in most versions of Emacs, type Ctrl+h and then type t. You can also view general Emacs docu- mentation by entering Ctrl+h (press this one or two times) while you are in Emacs or type man emacs at the command line.

CHAPTER SUMMARY

Bits represent digital 1s and 0s. Bytes are computer characters (a series of bits) stored using numeric codes. A set of standardized codes known as ASCII codes is often used to represent characters. ASCII stands for the American Standard Code for Information Interchange. Computer files that contain only ASCII characters (bytes) are called text files.

The vi editor is a popular choice among UNIX/Linux users. Standard editors process text files. Text files are also called ASCII files. The vi editor is a modal editor, because it works in three modes: insert, command, and ex mode. Insert mode (press i) lets you enter text, whereas command mode (press Esc) lets you navigate the file and modify the text. Ex mode (type : in command mode) is used to access an extended set of commands, including the commands to save and exit a file.

In the vi editor’s insert mode, characters you type are inserted in the file. They are not interpreted as vi commands. To exit insert mode and reenter command mode, press Esc.

With vi, you initially edit a copy of the file placed in the computer's memory. You do not alter the file itself until you save it on disk.

To get help for the vi editor, press Esc and enter :help or view the man documentation from the command line by entering man vi.

The Emacs editor is a popular alternative to the vi editor and, along with vi, is included with most UNIX/Linux systems.

Unlike vi, Emacs is not modal—it does not switch between modes. Emacs has a powerful command syntax, is extensible, and supports a sophisticated language of macro commands. A macro is a set of commands designed to simplify a complex task. Emacs' packaged set of customized macros lets you read electronic mail and news, and edit the contents of directories.

You can start Emacs by typing emacs at the command line with or without a file name. If you enter this command and then type a file name, Emacs creates a new, blank file with that name, or opens an existing file with that name. If you type emacs with no file name, Emacs displays an introductory screen. You can then use a command to open an existing file or create a new file.

You can use either the cursor movement keys—such as the arrow keys, Page Down, Page Up, Home, and End—or Ctrl/Alt key combinations to navigate an Emacs file.

In Emacs, as well as in vi, you can undo your editing changes in sequence, even after you've made many changes.

In Emacs, you can insert text simply by typing. You can also insert text by copying and pasting, or by cutting and pasting. Like the vi editor, Emacs lets you search for specific text. Emacs also has an automatic word wrap capability.

For Emacs help, type Ctrl+h (press t for a tutorial) while in Emacs or use the man emacs command from the command line.

COMMAND SUMMARY: REVIEW OF CHAPTER 3 COMMANDS

COMMAND	**PURPOSE**
vi commands:	
. (repeat)	Repeat your most recent change.
/	Search forward for a pattern of characters.
:!	Leave vi temporarily.
:q	Cancel an editing session.
:r	Read text from one file and add it to another.
:set	Turn on certain options, such as line numbering.
:w	Save a file and continue working.
:wq	Write changes to disk and exit vi.
:x	Save changes and exit vi.
:!lpr filename	Print a file.
I	Switch to insert mode.
P	Paste text from the buffer.
U	Undo your most recent change.
vi	Start the vi editor.
yy	Copy (yank) text to the clipboard.
ZZ	In command mode, save changes and exit vi.
Ctrl+z	Use this shell-based command (not truly a vi command) to leave vi to temporarily access the command line—use the fg command to return to vi.

COMMAND	PURPOSE
UNIX/Linux commands:	
lpr	Print a file. -P prints to a specific printer. -# prints a specific number of copies. -r deletes the print file from disk storage.
Emacs commands: See Table 3-4	

KEY TERMS

ASCII — An acronym for American Standard Code for Information Interchange; a standard set of bit patterns organized and interpreted as alphabetic characters, decimal numbers, punctuation marks, and special characters. The code is used to translate binary numbers into ordinary language, and, therefore, makes information stored in files accessible. ASCII can represent up to 256 characters (bit patterns).

binary file — A file containing non-ASCII characters (such as machine instructions). bit — The abbreviation for binary digit; a number composed of one of two numbers, 0 and 1. UNIX/Linux store all data in the form of binary digits. Because the computer consists of electronic circuits in either an on or off state, binary digits are perfect for representing these states. bitmap — The rows and columns of dots or bit patterns that graphics software transforms into an infinite variety of images. byte — The abbreviation for binary term; a string of eight binary digits or bits. These digits can be configured into patterns of bits, which, in turn, can be interpreted as alphabetic characters, decimal numbers, punctuation marks, and special characters. This is the basis for ASCII code. command mode — A feature of a modal editor that lets you enter commands to perform editing tasks, such as moving through the file and deleting text. The UNIX/Linux vi editor is a modal editor. compiling — A process of translating a program file into machine-readable language. editor — A program for creating and modifying computer documents, such as program and data files. ex mode — A text-editing command mode, currently used in the vi editor, that employs an extended set of commands initially used in an early UNIX editor called ex. executable program file — Also called an executable; a compiled file (from a program- ming language) or an interpreted file (from a script) that can be run on the computer. insert mode — A feature of a modal editor that lets you enter text. The UNIX/Linux vi editor is a modal editor. line editor — An editor that lets you work with only one line or a group of lines at once. Although you cannot see the context of your file, you might find a line editor useful for tasks such as searching, replacing, and copying blocks of text.

line-oriented command — A command that can perform more than one action, such as searching and replacing, in more than one place in a file. When using a line-oriented command, you must specify the exact location where the action is to occur. These commands differ from screen-oriented commands, which execute relative to the location of the cursor.

machine language — The exclusive use of 0s (which mean off) and 1s (which mean on) to communicate with the computer. Years ago, programmers had to write programs in machine language, a tedious and time-consuming process. macro — A set of commands that automates a complex task. A macro is sometimes called a superinstruction.

modal editor — A text editor that enables you to work in different modes. For example, the vi editor has three modes: insert, command, and ex. screen editor — An editor supplied by the operating system that displays text one screen at a time and lets you move around the screen to add and change text. UNIX/Linux have two screen editors: vi and Emacs.

screen-oriented command — A command that executes relative to the position of the cursor. Screen-oriented commands are easy to type, and you can readily see their result on the screen. These commands differ from line-oriented commands, which execute indepen- dently of the location of the cursor.

text editor — A simplified word processor used to create and edit documents but that has no formatting features to boldface or center text, for example.

text file — A computer file composed entirely of ASCII characters.

Unicode — A set of bit patterns that supports up to 65,536 characters and was developed to offer more characters than ASCII for a broader range of languages, such as Chinese.

CHAPTER 4 UNIX/LINUX FILE PROCESSING

The power of UNIX/Linux is based on its storage and handling of files. You've already learned about file systems, security, and UNIX/Linux editors. Now it's time to put your knowledge to work by manipulating files and their contents. To give you some background, this chapter starts with a short discussion of UNIX/Linux file types and file structures. Next, you learn more about using redirection operators, including how to use them to store error messages. You go on to learn file manipulation tools that you will use over and over again, either as a UNIX/Linux administrator or as an everyday user. These essential tools enable you to create, delete, copy, and move files. Other tools enable you to extract information from files to combine fields and to sort a file's contents. Finally, you learn how to assemble the information you extract from files, such as for creating reports. You also create your first script to automate a series of commands, link files with a common field, and get a first taste of the versatile awk command to format output.

UNIX AND LINUX FILE PROCESSING

UNIX/Linux file processing is based on the idea that files should be treated as nothing more than character sequences. This concept of a file as a series of characters offers a lot of flexibility.Because you can directly access each character,you can perform a range of editing tasks, such as correcting spelling errors and organizing information to meet your needs.

Reviewing UNIX/Linux File Types

Operating systems support several types of files. UNIX and Linux, like other operating systems, have text files, binary files, directories, and special files. As discussed in Chapter 3, "Mastering Editors," text files contain printable ASCII characters. Some users also call these regular, ordinary, or ASCII files. Text files often contain information you create and manipulate, such as a document or program source code. Binary files, also discussed in Chapter 3, contain nonprint- able characters, including machine language code created from compiling a program. In UNIX/Linux, text files and binary files are considered to be regular files, and you will sometimes see this terminology when working with files at the command line.

Chapter 2, "Exploring the UNIX/Linux File Systems and File Security," explained that directories are system files for maintaining the structure of the file system. In Chapter 2, you also learned about device special files. Character special files are used by input/output devices for communicating one character at a time, providing what is called raw data. The first character in the file access permissions is "c," which represents the file type, a character special file. Block special files are also related to devices, such as disks, and send information using blocks of data. The first character in these files is "b." For comparison, as you learned in Chapter 2, the first character for a directory is "d," and for a normal file—not a device special file—the first character is a dash "-."

Character special and block special files might also be called character device and block device files or character-special device and block-special device files.

Understanding File Structures

Files can be structured in several ways. For example, UNIX/Linux store data, such as letters, product records, or vendor reports, in flat ASCII files. The internal structure of a file depends on the kind of data it stores. A user structures a letter, for instance, using words, paragraphs, and sentences. A programmer can structure a file containing employee records using characters and words grouped together, with each individual employee record on a separate line in a file. Information about an employee in each separate line or record can be divided by separator characters or delimiters, such as colons. This type of record is called a variable-length record, because the length of each field bounded by colons can vary. The following is a simple example of an employee telephone record that might be stored in a flat

ASCII file and used by a human resources program. The first three fields, separated by colons, are the employee's home telephone number, consisting of the area code, prefix, and number. A human resources professional or boss might display some or all of this informa- tion in a program or report to be able to call the employee at home.

```
219:432:4567:Harrison:Joel:M:4540:Accountant:09-12-1985
```

Another way to create records is to have them start and stop in particular columns. For example, the area code in the previous example might start in column 1 and end in column 3. The prefix in the telephone number might go from column 5 to column 7, the last four digits in the telephone number would be in columns 9 through 12, and so on. Figure 4-1 illustrates this type of record, which is called a fixed-length record.

1–3 5–7 9–12 14–25 27–36 38 40–4345–60 62–71 219 432 4567 Harrison Joel M 4540 Accountant 09–12–1985

Figure 4-1 Fixed-length record Three simple kinds of text files are unstructured ASCII characters, records, and trees. Figure

4-2 illustrates these three kinds of text files.

Figure 4-2(a) shows a file that is an unstructured sequence of bytes and is a typical example of a simple text file. This file structure gives you the most flexibility in data entry, because you can store any kind of data in any order, such as the vendor name Triumph Motors and other information related toTriumph Motors,which might be a unique vendor number,the vendor's address,and so on.However,you can only retrieve the data in the same order,which might limit its overall usefulness. For example, suppose you list the vendors (product suppliers) used by a hotel in an unstructured ASCII text file. In this format, if you want to view only vendor names or vendor numbers, you really don't have that option. You most likely will have to print the entire file contents, including address and other information for all vendors.

Figure 4-2(b) shows data as a sequence of fixed-length records, each having some internal structure. In a UNIX/Linux file, a record is a line of data, corresponding to a row. For example, in a file of names, the first line or row might contain information about a single individual, such as last name, first name, middle initial, address, and phone number. The second row would contain the same kind of data about a different person, and so on. In this structure, UNIX/Linux read the data as fixed-length records. Although you must enter data as records, you can also manipulate and retrieve the data as records. For example, you can select only certain personnel or vendor records to retrieve from the file.

The third kind of file, illustrated in Figure 4-2(c), is structured as a tree of records that can be organized as fixed-length or variable-length records. In Figure 4-2(c), each record contains a key field, such as a record number, in a specific position in the record. The key

T

r

i

u

m

p

h

M

o

t

o

r

s

T r i u m p h

M o t o r s

ASCII characters RecordsVendor file (bytes)

1300

1400

1500

1100

1200

1203

1210

1214

Triumph

B&S Engineers

Novus Furniture

Bridger Elevators

Privett Industries

Monarch Interiors

Belmont Gas

(c) Tree

W e s t

P l u

m b i n g

(a) Byte sequence

(b) Record sequence

Figure 4-2 Three kinds of text files field sorts the tree, so you can quickly search for a record with a particular key. For example,

you can quickly find the record for Triumph Motors by searching for record #1203.

You will practice creating and manipulating different kinds of files and records in this and later chapters.

PROCESSING FILES

When performing commands, UNIX/Linux process data by receiving input from the standard input device—your keyboard, for example—and then sending it to the standard output: the monitor or console. System administrators and programmers refer to standard input as stdin. They refer to standard output as stdout. The third standard device, or file, is called standard error, or stderr. When UNIX/Linux detect errors in processing system tasks and user programs, they direct the errors to stderr, which, by default, is the screen.

stdin, stdout, and stderr are defined in IEEE Std 1003.1, “Standard for Informa- tion Technology—Portable Operating System Interface (POSIX),” and the ISO 9899:1999 C language standard (for C programming). The Institute of Electrical and Electronics Engineers (IEEE) and the International Organization for Stan- dardization (ISO) set computer-based and other standards.

In Chapter 1,“The Essence of UNIX and Linux,” you learned about the > and >> output redirection operators. You can use these and other redirection operators to save the output of a command or program in a file or use a file as an input to a process. The redirection operators are a tool to help you process files.

Using Input and Error Redirection

You can use redirection operators (>, >>, 2>, <, and <<) to retrieve input from something other than the standard input device and to send output to something other than the standard output device.

You already used the output redirection operators in Chapter 1, when you created a new file by redirecting the output of several commands to files. Redirect output when you want to store the output of a command or program in a file. For example, recall that you can use the ls command to list the files in a directory, such as /home. The ls command sends output to stdout, which, by default, is the screen. To redirect the list to a file called homedir.list, use the redirection symbol by entering ls > hom edir.list.

You can also redirect the input to a program or command with the < operator. For instance, a program that accepts keyboard input can be redirected to read information from a file instead. In Hands-On Project 4-1, you create a file from which the vi editor reads its commands, instead of reading them from the keyboard.

You can also use the 2> operator to redirect commands or program error messages from the screen to a file. For example, if you try to list a file or directory that does not exist, you see the following error message: No such file or directory. Assume that Fellowese is not a file or directory. If you enter ls Fellowese 2> errors, this places the No such file or directory error message in the errors file. Try this redirection technique in Hands-On Project 4-2.

MANIPULATING FILES

When you manipulate files, you work with the files themselves as well as their contents. This section explains how to complete the following tasks:

■ ■ ■ ■

Create files Delete files Remove directories Copy files

■ Move files ■ Find files ■ Combine files ■ Combine files through pasting ■ Extract fields in files through cutting ■ Sort files

Creating Files

You can create a new fileby using the output redirection operator (>). You learned how to do this to redirect the cat command’s output in Chapters 1–3. You can also use the redirection operator without a command to create an empty file by entering > and the name of the file. For example, the following command:

```
> accountsfile
```

creates an empty file called accountsfiles. Hands-On Project 4-3 enables you to create a file using the > redirection symbol.

You can also use the touch command to create empty files. For example, the following command creates the file accountsfile2, if the file does not already exist.

```
touch accountsfile2
```

Syntax touch [-options] filename(s)

Dissection

■ ■

Intended to change the time stamp on a file, but can also be used to create an empty file Useful options include:

-a updates the access time only -m updates the last time the file was modified -c prevents the touch command from creating the file, if it does not already exist

To view time stamp information in full, use the --full-time option with the ls command, such as ls --full-time myfile.

The primary purpose of the touch command is to change a file's time stamp and date stamp. UNIX/Linux maintain the following date and time information for every file:

■ Change date and time—The date and time the file's inode was last changed

■ ■

Access date and time—The date and time the file was last accessed Modification date and time—The date and time the file was last modified

Recall from Chapter 2 that an inode is a system for storing key information about files. Inode information includes the inode number, the owner of the file, the file group, the file size, the change date of the inode, the file creation date, the date the file was last modified and last read, the number of links to this inode, and the information regarding the location of the blocks in the file system in which the file is stored.

Although the touch command cannot alter a file's inode changed date and time, it can alter the file's access and modification dates and times. By default, it uses the current date and time for the new values. Hands-On Project 4-4 gives you experience using the touch command.

Deleting Files

When you no longer need a file, you can delete it using the rm (remove) command. If you use rm without options, UNIX/Linux delete the specified file without warning. Use the -i (interactive) option to have UNIX/Linux warn you before deleting the file. You can delete several files with similar names by using the asterisk wildcard. (See Chapter 2.) For example, if you have 10 files that all begin with the letters "test," enter rm test* to delete all of them at one time. Hands-On Project 4-5 enables you to use the rm command.

Syntax rm [-options] filename or directoryname

Dissection

■ Used to delete files or directories ■ Useful options include:

-i displays a warning prompt before deleting the file (or directory) -r when deleting a directory, recursively deletes its files and subdirectories (to delete a

directory that is empty or that contains entries, use the -r option with rm)

Removing Directories

When you no longer need a directory, you can use the commands rm or rmdir to remove it. For example,if the directory is already empty,use rm -r or rmdir.If the directory contains files or subdirectories, use rm -r to delete them all. The rm command with the -r option removes a directory and everything it contains. It even removes subdirectories of subdirectories. This operation is known as recursive removal. Note that if you use rm alone, in many versions of UNIX/Linux, including Fedora, Red Hat Enterprise Linux, and SUSE, it does not delete a directory.

Hands-On Project 4-6 enables you to use rmdir to delete an empty directory and rm -r to delete a directory that is not empty.

Syntax rmdir [-options] directoryname

Dissection

■ ■

Used to delete directories A directory must be empty to delete it with the rmdir command.

Use rm -r with great care by first making certain you have examined all of the directory's contents and intend to delete them along with the directory. If you are just deleting an empty directory, it is safer to use the rmdir command in case you make a typo when you enter the name of the directory. Also, when you use rm with the -r option, consider using the -i option as well to prompt you before you delete. Additional precautions employed by some users are to (1) use pwd to make certain you are in the proper working directory before you delete another directory and (2) use the full path to the directory you plan to delete, because, if you mistype a name, the deletion is likely to fail rather than delete the wrong directory.

Copying Files

In Chapter 2, you were introduced to the cp command for copying files, which we explore further here. Its general form is as follows:

Syntax cp [-options] source destination

Dissection

■ ■

Used to copy files or directories Useful options include:

-i provides a warning before cp writes over an existing file with the same name -s creates a symbolic link or name at the destination rather than a physical file (a symbolic

name is a pointer to the original file, which you learn about in Chapter 6)

-u prevents cp from copying over an existing file if the existing file is newer than the source file

The cp command copies the file or files specified by the source path to the location specified by the destination path. You can copy files into another directory, with the copies keeping the same names as the originals. You can also copy files into another directory, with the copies taking new names, or copy files into the same directory as the originals, with the copies taking new names.

For example, assume Tom is in his home directory (/home/tom). In this directory, he has the file reminder. Under his home directory, he has another directory, duplicates (/home/tom/ duplicates). He copies the reminder file to the duplicates directory with the following command:

```
cp reminder duplicates
```

After he executes the command, a file named reminder is in the duplicates directory. It is a duplicate of the reminder file in the /home/tom directory. Tom also has the file class_of_88 in his home directory. He copies it to a file named classmates in the duplicates directory with the following command:

```
cp class_of_88 duplicates/classmates
```

After he executes the command, the file classmates is stored in the duplicates directory. Although it has a different name, it is a copy of the class_of_88 file. Tom also has a file named memo_to_boss in his home directory. He wants to make a copy of it and keep the copy in his home directory. He enters the following command:

```
cp memo_to_boss memo.safe
```

After he executes this command, the file memo.safe is stored in Tom's home directory. It is a copy of his memo_to_boss file.

You can specify multiple source files as arguments to the cp command. For example, Tom wants to copy the files project1, project2, and project3 to his duplicates directory. He enters the following command:

```
cp project1 project2 project3 duplicates
```

The final entry in a multiple copy (cp) or move (mv) is a directory, as in the preceding example. You learn about the move command in the next section.

After he executes the command, copies of the three files are stored in the duplicates directory.

You can also use wildcard characters with the cp command. For example,Tom has a directory named designs under his home directory (/home/tom/designs). He wants to copy all files in the designs directory to the duplicates directory. He enters the following command:

```
cp designs/* duplicates
```

After he executes this command, the duplicates directory contains a copy of every file in the designs directory. As this example illustrates, the cp command is useful not only for copying but also for preventing lost data by making backup copies of files. You use the cp command in Hands-On Project 4-7 to make copies in a backup directory.

Moving Files

Moving files is similar to copying them, except you remove them from one directory and store them in another. However, as insurance, a file is copied before it is moved. To move a file, use the mv (move) command along with the source file name and destination name. You can also use the mv command to rename a file by moving one file into another file with a different name.

Moving and renaming a file are essentially the same operation.

When you are moving files, using the -i option with the mv command can be a good idea so that you don't unexpectedly overwrite a destination file with the same name.

Syntax mv [-options] source destination

Dissection

■ Used to move and to rename files ■ Useful options include:

-i displays a warning prompt before overwriting a file with the same name -u overwrites a destination file with the same name,if the source file is newer than the one

in the destination

Hands-On Project 4-8 enables you to use the mv command.

Finding Files

Sometimes, you might not remember the specific location of a file you want to access. The find command searches for files that have a specified name. Use the find command to locate files that have the same name or to find a file in any directory.

Syntax find [pathname] [-name filename]

Dissection

■ ■

Used to locate files in a directory and in subdirectories Useful options include:

pathname is the path name of the directory you want to search. The find command searches recursively; that is, it starts in the named directory and searches down through all files and subdirectories under the directory specified by pathname.

-name indicates that you are searching for files with a specific filename. You can use wildcard characters in the file name. For example, you can use phone* to search for all file names that begin with "phone."

-iname works like -name, but ignores case. For example, if you search for phone* as the search name, you'll find all files that begin with "phone," "Phone," "PHONE," or any combination of upper and lowercase letters.

-mmin n displays files that have been changed within the last n minutes.

-mtime n displays files that have been changed within the last n days.

-size n displays files of size n, where the default measure for n is in 512-byte blocks (you can also use nc, nk, nM, or nG for bytes kilobytes, megabytes, or gigabytes, such as find -size 2M to find files that are 2 megabytes). For other search conditions you can use with find, refer to Appendix B, "Syntax Guide to UNIX/Linux Commands."

When you are using the find command, you can only search areas for which you have adequate permissions. As the search progresses, the find command might enter protected directories; you receive a "Permission denied" message each time you attempt to enter a directory for which you do not have adequate permissions. Also, when you use find, it is useful to note that some UNIX versions require the -print option after the file name to display the names of files.

Try Hands-On Project 4-9 to use the find command.

Combining Files

In addition to viewing and creating files, you can use the cat command to combine files. You combine files by using a redirection operator, but in a somewhat different format than you use to create a file. As you already know, if you enter cat > janes_research,you can then type information into the file and end the session by typing Ctrl+d, creating the file janes_ research. Assume that Jane has created such a file containing research results about bighorn

sheep. Now assume that there is also the file marks_research, which contains Mark's research on the same topic. You can use the cat command to combine the contents of both files into the total_research file by entering the following:

```
cat janes_research marks_research > total_research
```

Hands-On Project 4-10 enables you to use this technique for combining files.

Combining Files with the paste Command

The paste command combines files side by side, whereas the cat command combines files end to end. When you use paste to combine two files into a third file, the first line of the output contains the first line of the first file followed by the first line of the second file. For example, consider a simple file, called vegetables, containing the following four lines:

Carrots Spinach Lettuce Beans

Also, the bread file contains the following four lines:

Whole wheat White bread Sourdough Pumpernickel

If you execute the command paste vegetables bread > food, the vegetables and bread files are combined, line by line, into the file food. The food file's contents are shown in Figure 4-3.

The paste command normally sends its output to stdout (the screen). To capture it in a file, use the redirection operator.

Figure 4-3 Using the paste command to merge files

Syntax paste [-options] source files [> destination file]

Dissection

■ ■ ■

Combines the contents of one or more files to output to the screen or to another file By default, the pasted results appear in columns separated by tabs Useful options include:

-d enables you to specify a different separator (other than a tab) between columns -s causes files to be pasted one after the other instead of in parallel

As you can see, the paste command is most useful when you combine files that contain columns of information. When paste combines items into a single line, it separates them with a tab. For example, look at the first line of the food file:

Carrots Whole wheat

When paste combined "Carrots" and "Whole wheat," it inserted a tab between them. You can use the -d option to specify another character as a delimiter. For example, to insert a comma between the output fields instead of a tab, you would enter:

paste -d',' vegetables bread > food

After this command executes, the food file's contents are:

Carrots,Whole wheat Spinach,White bread

Lettuce,Sourdough Beans,Pumpernickel

Try Hands-On Project 4-11 to begin learning to use the paste command.

Extracting Fields Using the cut Command

You have learned that files can consist of records, fields, and characters. In some instances, you might want to retrieve some, but not all, fields in a file. Use the cut command to remove specific columns or fields from a file. For example, in your organization, you might have a vendors file of businesses from which you purchase supplies. The file contains a record for each vendor, and each record contains the vendor's name, street address, city, state, zip code, and telephone number; for example, Office Supplies: 2405 S.E. 17th Street: Boulder: Colorado: 80302:303-442-8800. You can use the cut command to quickly list only the names of vendors, such as Office Supplies, in this file. The syntax of the cut command is as follows:

Syntax cut [-f list] [-d char] [file1 file2 . . .] or cut [-c list] [file1 file2 . . .]

Dissection

■ ■

Removes specific columns or fields from a file Useful options include:

-f specifies that you are referring to fields

list is a comma-separated list or a hyphen-separated range of integers that specifies the field. For example, -f 1 indicates field 1, -f 1,14 indicates fields 1 and 14, and -f 1-14 indicates fields 1 through 14.

-d indicates that a specific character separates the fields char is the character used as the field separator (delimiter), for example, a comma. The

default field delimiter is the tab character. file1, file2 are the files from which you want to cut columns or fields

-c references character positions. For example, -c 1 specifies the first character and -c 1,14 specifies characters 1 and 14.

Recall the example vegetables and bread files discussed in the preceding section. Assume that you also have the file meats. When you use the command paste vegetables bread meats > food, the contents of the food file are now as follows:

Carrots Whole wheat Spinach White bread Lettuce Sourdough Beans Pumpernickel Ham

Turkey Chicken Beef

Next, assume that you want to extract the second column of information (the bread list) from the file and display it on the screen. You enter the following command:

```
cut -f2 food
```

The option -f2 tells the cut command to extract the second field from each line. Tab delimiters separate the fields, so cut knows where to find the fields. The result of the cut -f2 food command is output to the screen, as shown in Figure 4-4.

Figure 4-4 Using the cut command Another option is to extract the first and third columns from the file using the following

command:

```
cut -f1,3 food
```

The result of the command is:

Carrots Turkey Spinach Chicken Lettuce Beef Beans Ham

Hands-On Project 4-12 enables you to practice using the cut command.

Sorting Files

Use the sort command to sort a file's contents alphabetically or numerically. UNIX/Linux display the sorted file on the screen by default, but you can specify that you want to store the sorted data in a particular file by using a redirection operator.

Syntax sort [-options] [filename]

Dissection

■ ■

Sorts the contents of files by individual lines Useful options include:

-k n sorts on the key field specified by n -t indicates that a specified character separates the fields -m merges input files that have been previously sorted (does not perform a sort) -o redirects output to the specified file -d sorts in alphanumeric or dictionary order -g sorts by numeric (general) order -r sorts in reverse order

The sort command offers many options, which Appendix B also describes. The following is an example of its use:

```
sort file1 > file2
```

In this example, the contents of file1 are sorted and the results are stored in file2. (If the output is not redirected, sort displays its results on the screen.) If you are sorting a file of records and specify no options, for instance, the values in the first field of each record are sorted alphanumerically. A more complex example is as follows:

```
sort -k 3 food > sortedfood
```

This command specifies a sorting key. A sorting key is a field position within each line. The sort command sorts the lines based on the sorting key. The -k is the key field within the file. For instance, -k 3 in the preceding example sorts on the third field in the food file, which is the listing of meats, and writes the results of the sort to the file sortedfood (see Figure 4-5). Notice in Figure 4-5 that the third field in the first record is Beef (and all of the records are sorted by the third field).

Hands-On Project 4-13 enables you to use the sort command. Also, try Hands-On Project 4-14 to use the cat, cut, paste, and sort commands in a project that puts together in one place what you have learned so far.

CREATING SCRIPT FILES

As you have seen, command-line entries can become long, depending on the number of options you need to use. You can use the shell's command-line history retrieval feature to recall and reexecute past commands. This feature can work well for you if you need to repeat commands shortly after executing them, but it is a problem if you need to perform

Figure 4-5 Results of sorting on the third field in the food file

the same task a few days later. Also, there might be other people, such as an assistant or supervisor, who need to execute your stored commands and cannot access them from their own user accounts. MS-DOS and Windows users resolve this problem by creating batch files, which are files of commands that are executed when the batch file is run. UNIX/Linux users do the same: They create shell script files to contain command-line entries. Like MS-DOS/Windows batch files, script files contain commands that can be run sequentially as a set. For example, if you often create a specific report using a combination of the cut, paste, and sort commands,you can create a script file containing these commands.Instead of having to remember the exact commands and sequence each time you want to create the report, you instead execute the script file. Creating script files in this way can save you a significant amount of time and aggravation.

After you determine the exact commands and command sequence, use the vi or Emacs editor to create the script file.(See Figure 4-6.) Next,make the script file executable by using the chmod command with the x argument, as you learned in Chapter 2. Finally, use the ./ command to run a script, such as typing ./myscript and pressing Enter to run the script file myscript.

Script files can range from the simple to the complex. In Hands-On Project 4-15, you get a basic introduction to using these files. Chapters 6 and 7, “Introduction to Shell Script Programming” and “Advanced Shell Programming,” give you much more experience with script files (or scripts for short).

Figure 4-6 Sample script file

USING THE JOIN COMMAND ON TWO FILES

Sometimes, it is useful to know how to link the information in two files. You can use the join command to associate lines in two files on the basis of a common field in both files. If you want the results sorted,you can either sort the files on a common field before you join the information or sort on a specific field after you join the information from the files. For example, suppose you have a file that contains the employee’s last name in one field, the employee’s company ID in another field, and the employee’s salary in the final field, as follows:

Brown:82:53,000 Anders:110:32,000 Caplan:174:41,000 Crow:95:36,000

Also, you have another file that contains each employee’s last name, first name, middle initial, department, telephone number, and other information, but that file does not contain salary information, as follows:

Brown:LaVerne:F:Accounting Department:444-7508: . . . Anders:Carol:M:Sales Department:444-2130: . . . Caplan:Jason:R:Payroll Department:444-5609: . . . Crow:Lorretta:L:Shipping Department:444-8901: . . .

You want to create a new third file to use for budgeting salaries that contains only the employee’s last name, first name, department, and salary. To do this, you could use the join command to create a file with the following contents:

Brown:LaVerne:Accounting Department:53,000 Anders:Carol:Sales Department:32,000 Caplan:Jason:Payroll Department:41,000 Crow:Lorretta:Shipping Department:36,000

In this simple example, the common field for the two original files is the employee’s last name. Note that in this context, the common field provides a key for accessing and joining the information to create a report or to create another file with the joined information.(Also refer back to Figure 4-2(c) for an example of a key-based file structure.)

The join command is also associated with linking information in complex databases. The use of these databases, such as relational databases, is beyond the scope of this book. However, learning the join command to manipulate data in flat files, as used in this book, can be useful. It is used here as another file manipulation tool to complement your knowledge of the paste, cut, and sort commands.

Syntax join [-options] file1 file2

Dissection

- Used to associate information in two different files on the basis of a common field or key in those files

■ file1, file2 are two input files that must be sorted on the join field—the field you want to use to join the files. The join field is also called a key. You must sort the files before you can join them. When you issue the join command, UNIX/Linux compare the two fields. Each output line contains the common field followed by a line from file1 and then a line from file2. You can modify output using the options described next. If records with duplicate keys are in the same file, UNIX/Linux join on all of them. You can create output records for unpairable lines, for example, to append data from one file to another without losing records.

■ Useful options include: -1 fieldnum specifies the common field in file1 on which to join -2 fieldnum specifies the common field in file 2 on which to join

-o specifies a list of fields to output. The list contains blank-separated field specifiers in the form m.n, where m is the file number and n is the position of the field in the file. Thus, -o 1.2 means "output the second field in the first file."

-t specifies the field separator character. By default this is a blank, tab, or new line character. Multiple blanks and tabs count as one field separator.

-a filenum produces a line for each unpairable line in the file filenum. (In this case, filenum is a 1 for file1 or a 2 for file2.)

-e str replaces the empty fields for the unpairable line in the string specified by str. The string is usually a code or message to indicate the condition, for example, -e "No Vendor Record."

Hands-On Project 4-16 gives you an opportunity to use the join command.

A BRIEF INTRODUCTION TO THE AWK PROGRAM

Awk, a pattern-scanning and processing language, helps to produce reports that look professional. Although you can use the cat and more commands to display the output file that you create with your join command, the awk command (which starts the Awk program when you enter it on the command line) lets you do the same thing more quickly and easily.

The name Awk is formed from the initials of its inventors, (Alfred) Aho, (Peter) Weinberger, and (Brian) Kernighan. They have provided a rich and powerful programming environment in UNIX/Linux that is well worth the effort to learn, because it can perform actions on files that range from the simple to the complex—and can be difficult to duplicate using a combination of other commands.

In Fedora, Red Hat Enterprise Linux, SUSE and some other versions of UNIX and Linux, you actually use gawk, which includes enhancements to awk and was developed for the GNU Project by Paul Rubin and Jay Fenlason. When you type awk at the command line, you really execute gawk—or you can just type gawk.

Syntax awk [- Fsep] ['pattern {action} ..'] filenames

Dissection

■ awk checks to see if the input records in the specified files satisfy the pattern and, if they do, awk executes the action associated with it. If no pattern is specified, the action affects every input record.

■ -F: means the field separator is a colon

The awk command is used to look for patterns in files. After it identifies a pattern, it performs an action that you specify. One reason to learn awk is to have a tool at your fingertips that lets you manipulate data files very efficiently. For example, you can often do the same thing in awk that would take many separate commands using a combination of paste, cut, sort, and join. Another reason for learning awk is that you might have a project you simply can't complete using a combination of paste, cut, sort, and join, but you can complete it using awk.

Some of the tasks you can do with awk include: ■ Manipulate fields and records in a data file. ■ Use variables. (You learn more about variables in Chapter 6.)

■ Use arithmetic, string, and logical operators. (You learn more about these types of operators in Chapters 6 and 7.)

■ Execute commands from a shell script. ■ Use classic programming logic, such as loops. (You learn more looping logic in

Chapter 6.) ■ Process and organize data into well-formatted reports.

Consider a basic example in which you want to print text to the screen. The following is a simple awk command-line sequence that illustrates the syntax:

```
awk 'BEGIN { print "This is an awk print line." }'
```

When you type this at the command line, the following appears on the screen:

```
This is an awk print line.
```

The awk command-line sequence to produce this output does the following things: 1. awk starts the Awk program to process the command-line actions. 2. The pattern is signaled by BEGIN. 3. The pattern and the action are enclosed in single quotation marks.

4. The action in the curly brackets { } is processed by the Awk program.

5. The Awk print command is executed to print the string inside the double quotation marks (input from the keyboard or stdin) so that it appears on the screen (stdout).

Using a more advanced example, you can use awk to process input from a data file and display a report as output. Consider the following sample awk command-line sequence:

```
awk -F: '{printf "%s\t %s\n", $1, $2}' datafile
```

In this example, the following happens:

1. awk -F: starts the Awk program and tells Awk that the field separator between records in the input file (datafile) is a colon.

2. The pattern and action are enclosed within the single quotation marks.

3. printf is a command used in the Awk program to print and format the output. (You learn more about printf in Chapter 5,“Advanced File Processing”.) In this case, the output goes to the screen (stdout).

4. $1 and $2 signify that the fields to print and format are the first ($1) and sec- ond ($2) fields in the specified input file, which is datafile.

5. datafile is the name of the input file that contains records divided into fields. Try Hands-On Projects 4-17 and 4-18 for a further introduction to awk.

awk is presented here to give you a first, experiential taste of this powerful tool. There is a lot to learn about using awk, and you learn more in later chapters. For now, consider this brief introduction of awk as a natural follow-on to your introduction to the join command—like a musician sight-reading new music as a rudimentary step to learning more about it. For more information about awk, type man awk to read the online documentation.

CHAPTER SUMMARY

UNIX/Linux support regular files, directories, character special files, and block special files. Regular files contain user information. Directories are system files for maintaining the file system's structure. Character special files are related to serial input/output devices, such as printers. Block special files are related to devices, such as disks.

Files can be structured in several ways. UNIX/Linux store data, such as letters, product records, or vendor reports, in flat ASCII files. File structures depend on the kind of data being stored. Three kinds of regular files are unstructured ASCII characters, records, and trees.

Often, flat ASCII data files contain records and fields. They typically use one of two formats: variable-length records and fixed-length records. Variable-length records usually have fields that are separated by a delimiter, such as a colon. Fixed-length records have fields that are in specific locations, such as a column range, within a record.

When performing commands, UNIX/Linux process data—they receive input from the standard input device and then send output to the standard output device. UNIX/Linux refer to the standard devices for input and output as stdin and stdout, respectively. By default, stdin is the keyboard and stdout is the monitor. Another standard device, stderr, refers to the error file that defaults to the monitor. Output from a command can be redirected from stdout to a disk file. Input to a command can be redirected from stdin to a disk file. The error output of a command can be redirected from stderr to a disk file.

The touch command updates a file's time stamp and date stamp and creates empty files.

The rmdir command removes an empty directory. Also, the rm command can be used to delete a file, and the rm command with the -r option can be used to delete a directory that contains files and subdirectories.

The cut command extracts specific columns or fields from a file. Select the fields you want to cut by specifying their positions and separator character, or you can cut by character positions, depending on the data's organization.

To combine two or more files, use the paste command. Where cat appends data to the end of the file, the paste command combines files side by side. You can also use paste to combine fields from two or more files.

Use the sort command to sort a file's contents alphabetically or numerically. UNIX/Linux display the sorted file on the screen by default, but you can also specify that you want to store the sorted data in a particular file.

To automate command processing, include commands in a script file that you can later execute as a program. Use the vi editor to create the script file, and use the chmod command to make it executable.

Use the join command to extract information from two files sharing a common field. You can use this common field to join the two files. You must sort the two files on the join field—the one you want to use to join the files. The join field is also called a key. You must sort the files before you can join them.

Awk is a pattern-scanning and processing language useful for creating a formatted report with a professional look. You can enter the Awk language instructions in a program file using the vi editor and call it using the awk command.

COMMAND SUMMARY: REVIEW OF CHAPTER 4 COMMANDS

Command	Purpose	Options (This chapter)
awk	Starts the awk program to format output.	-F identifies the field separator. format output

		-f indicates code is coming from a disk file, not the keyboard.
cat	Views the contents of a file, creates a file, merges the contents of files	
cp	Copies one or more files	-i provides a warning before cp writes over an existing file with the same name. -s creates a symbolic link or name at the destination rather than a physical file. -u prevents cp from copying over an existing file, if the existing file is newer than the source file.
cut	Extracts specified columns or fields from a file	-c refers to character positions. -d indicates that a specified character sepa- rates the fields. -f refers to fields.
find	Finds files	-iname specifies the name of the files you want to locate, but the search is not case sensitive. -name specifies the name of the files you want to locate, but the search is case sensitive. -mmin n displays files that have been changed within the last n minutes. -mtime n displays files that have been changed within the last n days. -size n displays files of size n.
join	Combines files having a common field	-a n produces a line for each unpairable line in file n. -e str replaces the empty fields for an unpairable file with the specified string. -1 and -2 with the field number are used to specify common fields when joining. -o outputs a specified list of fields. -t indicates that a specified character separates the fields.
mv	Moves one or more files	-i displays a warning prompt before over- writing a file with the same name. -u overwrites a destination file with the same name, if the source file is newer than the one in the destination.
paste	Combines fields from two or more files	-d enables you to specify a different separator (other than a tab) between columns. -s causes files to be pasted one after the other instead of in parallel.
rm	Removes one or more files	-i specifies that UNIX/Linux should request confirmation of file deletion before removing the files. -r specifies that directories should be recursively removed.

rmdir	Removes an empty directory	
sort	Sorts the file's contents	-k n sorts on the key field specified by n. -t indicates that a specified character separates the fields. -m means to merge files before sorting. -o redirects output to the specified file. -d sorts in alphanumeric or dictionary order. -g sorts by numeric (general) order. -r sorts in reverse order.
touch	Updates an existing file's time stamp and date stamp or creates empty new files	-a specifies that only the access date and time are to be updated. -m specifies that only the modification date and time are to be updated. -c specifies that no files are to be created.

KEY TERMS

fixed-length record — A record structure in a file in which each record has a specified length, as does each field in a record. flat ASCII file — A file that you can create, manipulate, and use to store data, such as letters, product reports, or vendor records. Its organization as an unstructured sequence of bytes is typical of a text file and lends flexibility in data entry, because it can store any kind of data in any order. Any operating system can read this file type. However, because you can retrieve data only in the order you entered it, this file type's usefulness is limited. Also called an ordinary file or regular file.

key — A common field in every file record shared by each of one or more files. The common field, or key, enables you to link or join information among the files, such as for creating a report. regular file — A UNIX/Linux reference to ASCII/text files and binary files. Also called an ordinary file.

relational database — A database that contains files that UNIX/Linux treat as tables, records that are treated as rows, and fields that are treated as columns and that can be joined to create new records. For example, using the join command, you can extract information from two files in a relational database that share a common field.

shell script — A text file that contains sequences of UNIX/Linux commands that do not need to be converted into machine language by a compiler. sorting key — A field position within each line of a file that is used to sort the lines. For instance, in the command sort -k 2 myfile, myfile is sorted by the second field in that file. The sort command sorts the lines based on the sorting key.

stderr — An acronym used by programmers for standard error. When UNIX/Linux detect errors in programs and program tasks, the error messages and analyses are directed to stderr, which is often the screen (part of the IEEE Std 1003.1 specification). stdin — An acronym used by programmers for standard input and used in programming to read input (part of the IEEE Std 1003.1 specification).

stdout — An acronym used by programmers for standard output and used in programming to write output (part of the IEEE Std 1003.1 specification).

variable-length record — A record structure in a data file in which the records can have variable lengths and are typically separated by a delimiter, such as a colon.

CHAPTER 5 ADVANCED FILE PROCESSING

With file-processing commands, you can manage files through sorting, cutting, formatting, translating, comparing, and using other processing techniques.Your UNIX/Linux abilities are strengthened from knowing the versatility of these commands. In this chapter, you learn many new file- processing, selection, manipulation, and transformation commands to put into your expanding toolbox.

You begin by learning new file-processing commands and progress to using more complex manipulation and format commands. In the second portion of the chapter, you put your knowledge to work by designing an application in a step-by-step process. The beginning steps involve designing a file structure. Next, you use commands to determine ways to extract information from the files.Then, you build small shell scripts of commands and test each one.After you have the small scripts individually built, you combine them into a larger application that you run.

After reading this chapter and completing the exercises, you will be able to:

♦ Use the pipe operator to redirect the output of one command to another command

♦ Use the grep command to search for a specified pattern in a file

♦ Use the uniq command to remove duplicate lines from a file

♦ Use the comm and diff commands to compare two files

♦ Use the wc command to count words, characters, and lines in a file

♦ Use manipulation and transformation commands, which include sed, tr, and pr

♦ Design a new file-processing application by creating, testing, and running shell scripts

Some of the capabilities that you discover in this chapter can also be performed using graphical utilities from a GUI-based desktop, such as GNOME or KDE. By learning how to use these capabilities through commands instead of GUI tools, you often have the advantage of being able to do more—and do it faster. Also, you can often perform actions from the command line that you cannot perform with the same versatility from a GUI desktop tool.

ADVANCING YOUR FILE-PROCESSING SKILLS

In Chapter 4, “UNIX/Linux File Processing,” you learned to use several UNIX/Linux commands to extract and organize information from existing files and transform that information into a useful format. Now you build on those skills and learn to use new file-processing commands and operators.The commands you use for file processing can be organized into two categories: selection commands and manipulation and transformation commands.

Selection commands focus on extracting specific information from files, such as using the comm command to compare file contents.Table 5-1 lists the selection commands you have already mastered plus new commands you learn in this chapter.

Table 5-1 Selection commands

Command	Purpose
comm	Compares sorted files and shows differences
cut	Selects columns (fields)
diff	Compares and selects differences in two files
grep	Selects lines or rows
head	Selects lines from the beginning of a file
tail	Selects lines from the end of a file
uniq	Selects unique lines or rows (typically preceded by a sort)

wc	Counts characters, words, or lines in a file

Manipulation and transformation commands alter and transform extracted informa- tion into useful and appealing formats.Table 5-2 lists these commands.

Table 5-2 Manipulation and transformation commands

Command	**Purpose**
awk	Invokes Awk, a processing and pattern-scanning language
cat	Concatenates files
chmod	Changes the security mode of a file or directory
join	Joins two files, matching row by row
paste	Pastes multiple files, column by column
pr	Formats and prints
sed	Edits data streams
sort	Sorts and merges multiple files
tr	Translates and deletes character by character

USING THE SELECTION COMMANDS

You used the head and tail commands in Chapter 1,"The Essence of UNIX and Linux," and the cut command in Chapter 4. You also learned to use redirection operators. Now you learn a new redirection operator, called a pipe, and work with the grep, diff, uniq, comm, and wc commands for processing files.

See Appendix B, "Syntax Guide to UNIX/Linux Commands," for additional information about these commands.

Using the Pipe Operator

As you have learned, most UNIX/Linux commands take their input from stdin (the standard input device) and send their output to stdout (the standard output device).You have also used the > operator to redirect a command's output from the screen to a file, and you have used the < operator to redirect a command's input from the keyboard to a file.The pipe operator (|) redirects the output of one command to the input of another command.The pipe operator is used in the following way:

```
first_command | second_command
```

The pipe operator connects the output of the first command with the input of the second command. For example, when you list the contents of a large directory, such as /etc or /sbin using the ls -l command, the output races across the screen and you really see only the end of the listing. If you are using a terminal window, you might be able to use the scroll bar to go backward through the listing, but this might not be as convenient as other ways to view the output, or the terminal window or command-line access on your system might not support fully scrolling back.An alternative is to pipe output of the ls -l command to use as input of the more command. For example, when you enter the following command using the pipe operator, you can view the contents of the /sbin directory one screen at a time and use the spacebar to advance to the next screen.

```
ls -l /sbin | more
```

Hands-on Project 5-1 enables you to use the pipe operator for a directory listing.

You can also use the less command with a directory to view its contents one screen at a time, such as less /sbin.

The pipe operator can connect several commands on the same command line, in the following manner:

```
first_command | second_command | third_command ...
```

This technique can be useful, for example, when you want to list and then sort in reverse order the contents of a large directory and display the result one screen at a time, as shown in Figure 5-1 for the contents of the /etc directory. Try Hands-on Project 5-2 to use the pipe operator to combine commands on one line.

Figure 5-1 Combining commands using the pipe operator Using the grep Command

Use the grep command to search for a specified pattern in a file, such as a particular word or phrase. UNIX/Linux find and then display the line containing the pattern you specify.

Syntax grep [-options] pattern [filename]

Dissection

■ ■ ■ ■

Finds and displays lines containing a particular search pattern Can be used on text and binary regular files Can search multiple files in one command Useful options include:

-i ignores case -l lists only file names -c counts the number of lines instead of showing them -r searches through files under all subdirectories -n includes the line number for each line found -v displays only lines that don’t contain the search pattern

Three typical meanings are associated with grep: Global Regular Expression Print, Global Regular Expression Parser, and Get Regular Expression Processing.

Consider a situation in which you have written a document for a company in which you refer multiple times to the Computer Resources Committee. Further, your company’s management is contemplating broadening the focus of the committee and calling it the Computer and Telecommunications Resources Committee. When company management asks you to determine how often existing company documentation in the /documentation directory refers to the Computer Resources Committee, you can use the grep command to find out. Here is an example of what you would enter:

```
grep –r Computer Resources Committee /documentation
```

In some cases, when you use grep, it is helpful to enter the character pattern you are trying to find in single or double quotes. For example, this is true when you are looking for two or more words, so that the words can be distinguished from a file, as in the command: grep ‘red hat’ operating_system. In this example,‘red hat’ is the character pattern and operating_system is the file you are searching. If you enter red hat without quotes you are likely to get an error because grep interprets hat as a file name. In Hands-on Project 5-3, you use the grep command to search for and extract specific text.

Many UNIX/Linux systems offer a combination of four grep-type commands: grep, egrep, fgrep, and zgrep. For example, besides grep there is egrep (also executed as grep -E), which is used for “extended” or more complex expressions. fgrep (or grep -F on most systems) searches for fixed or text strings only and not expressions. zgrep is used to perform searches on files that are compressed or zipped.

Using the uniq Command

The uniq command removes duplicate lines from a file. Because it compares only consecu- tive lines, the uniq command requires sorted input. The syntax of the uniq command is as follows:

Syntax uniq [-options] [file1 > file2]

Dissection

■ Removes consecutive duplicate lines from one file and writes the result to another file ■ Useful options include:

-u outputs only the lines of the source file that are not duplicated -d outputs one copy of each line that has a duplicate, and does not show unique lines -i ignores case -c starts each line by showing the number of each instance

In its simplest form, the uniq command removes successive identical lines or rows from a file. For example, consider a simple file called parts that contains the following entries:

```
muffler muffler shocks alternator battery battery radiator radiator coil

spark plugs spark plugs coil
```

You can use the uniq command to create an output file called inventory that removes all the successive duplicates.The command to use is as follows (see Figure 5-2):

```
uniq parts > inventory
```

Figure 5-2 Using uniq to remove duplicate entries and create a new output file

Notice in Figure 5-2 that coil is still listed twice. This is because in the original parts file, the two occurrences of coil are not successive. In the parts file, the first instance of coil is just before the first listing for spark plugs, and the second instance of coil is after the second instance of spark plugs.

The -u option instructs uniq to generate as output only the lines of the source file that are not duplicated successively. (If a line is repeated successively, it is not generated as output.) Here is an example (see Figure 5-3):

```
uniq -u parts > single_items
```

In Figure 5-3, coil is also listed twice because in the original parts file, the two occurrences of coil are not successive.

The -d option instructs uniq to generate as output one copy of each line that has a successive duplicate line. Unduplicated lines are not generated as output. Here is an example:

```
uniq -d parts > multi_items
```

Hands-on Project 5-4 enables you to use the uniq command.

Figure 5-3 Creating a file containing only lines not duplicated

Using the comm Command

Like the uniq command, the comm command identifies duplicate lines. Unlike the uniq command, it doesn't delete duplicates, and it works with two files rather than one.The comm command locates identical lines within two identically sorted files. It compares lines common to file1 and file2, and produces three-column output:

■ ■ ■

The first column contains lines found only in file1. The second column contains lines found only in file2. The third column contains lines found in both file1 and file2.

The syntax of comm is as follows:

Syntax comm [-options] file1 file2

Dissection

■ Compares two sorted files for common lines and generates three columns of output to show which lines are unique to each file and which are common to both files

■ Useful options include: -1 do not display lines that are only in file1 -2 do not display lines that are only in file2 -3 do not display lines appearing in both file1 and file2

Hands-on Project 5-5 uses the comm command.

Using the diff Command

The diff command shows lines that differ between two files and is commonly used to determine the minimal set of changes needed to convert file1 to file2.The command's output displays the line(s) that differ. Differing text in file1 is preceded by the less-than symbol (<), and for file2 is preceded by the greater-than symbol (>).

Syntax diff [-options] file1 file2

Dissection

■ ■

Shows lines that differ between two files Useful options include:

-b ignores blanks that repeat -B does not compare for blank lines -i ignores case -c shows lines surrounding the line that differs (for context) -y display the differences side-by-side in columns

Consider a comparison of two files, zoo1 and zoo2, that contain variable-length records of food supplies for zoo animals. You create these files in Hands-on Project 5-4, and they contain the following lines. File zoo1 contains:

Monkeys:Bananas:2000:850.00 Lions:Raw Meat:4000:1245.50 Lions:Raw Meat:4000:1245.50 Camels:Vegetables:2300:564.75 Elephants:Hay:120000:1105.75 Elephants:Hay:120000:1105.75

File zoo2 contains:

Monkeys:Bananas:2000:850.00 Lions:Raw Meat:4000:1245.50 Camels:Vegetables:2300:564.75 Elephants:Hay:120000:1105.75

When you enter diff zoo1 zoo2, the first lines of output are as follows: 3d2

< Lions:Raw Meat:4000:1245.50

In this example line of output, the code 3d2 indicates that to make the files the same, you need to delete the third line in file1, so file1 matches file2.The d means delete, the 3 means the third line from file1, and the 2 means that file1 and file2 will be the same up to, but not including, line 2.

In another example, assume that you reverse the order of the files in the comparison by entering diff zoo2 zoo1 and the first lines of output are as follows:

2a3 > Lions:Raw Meat:4000:1245.50

The code 2a3 indicates you need to add a line to file1, so file1 matches file2.The a means to add a line or lines to file1.The 3 means line 3 is to be added from file1 to file2.The 2 indicates that the line must be added in file2 following line 2.

The diff command is an example of a command that is easier to understand after you use it. Try Hands-on Project 5-6 to further explore how this command works.

Using the wc Command

Use the wc command to count the number of lines (option -l), words (option -w), and bytes or characters (option -c) in text files.You can specify all three options in the command line, such as -lwc or any other combination. If you enter the command without options, you see counts of lines, words, and characters in that order. (See Figure 5-4.)

Syntax wc [-options] [files]

Dissection

■ Calculates the line, word, and byte count of the specified file(s) ■ Useful options include:

-c shows byte count -l shows line count -w shows word count

Hands-on Project 5-7 gives you experience using the wc command.

Figure 5-4 Using wc to count lines, words, and bytes in a file

USING MANIPULATION AND TRANSFORMATION COMMANDS

In addition to the commands that you learned in Chapter 4 that are used to manipulate and format data, you can also use the sed, tr, and pr commands to edit and transform data's appearance before you display or print it.

Introducing the sed Command

When you want to make global changes to large files, you need a different kind of tool than an interactive editor, such as vi and Emacs. Another UNIX/Linux editor, sed, is designed specifically for this purpose, and is sometimes called a stream editor because input to sed is rendered in standard output (to display on the screen).The minimum requirements to run sed are an input file and a command that lets sed know what actions to apply to the file. sed commands have two general forms: (1) provided as part of the command line and (2) provided as input from a script file.

Syntax sed [-options] [command] [file(s)] sed [-options] [-f scriptfile] [file(s)]

Dissection

■ sed is a stream editor that can be used on one or more files, and is particularly useful for making global changes on large files.

■ The first form lets you specify an editing command on the command line. ■ The second form lets you specify a script file containing sed commands. ■ Useful options include:

d deletes lines specified by the -n option (no hyphen in front of the d option) p prints to output the lines specified by the -n option (no hyphen in front of the p option) s substitutes specified text (no hyphen in front of this s option) a\ appends text (no hyphen in front of this option) -e specifies multiple commands on a command line -n specifies line numbers on which to work

For example, you can use sed to work with a new file, to display only certain lines—such as only lines 3 and 4—and then to work on or replace only those lines.You learn to use sed in this way by working through Hands-on Project 5-8, entering the edit commands from the command line.

To append new lines in sed, you must use the a\ command.This command appends lines after the specified line number. Like all other sed commands, it operates on all lines in the file if you do not specify a line number. In Hands-on Project 5-9, you create and manipulate a document's contents by using the a\ command from a script.

Translating Characters Using the tr Command

The tr or translate command (also called the translate characters command) copies data from the standard input to the standard output, substituting or deleting characters specified by options and patterns.The patterns are strings and the strings are sets of characters.

Syntax tr [-options] [“string1”“string2”]

Dissection

■ In its simplest form, tr translates each character in string1 into the character in the corresponding position in string2.The strings typically need to be “quoted” with either single or double quotation marks.

■ Useful options include: -d deletes characters -s substitutes or replaces characters

A popular use of tr is to convert lowercase characters to uppercase characters. For example, you can translate the contents of a file from lowercase to uppercase characters by using [a-z] to specify the lowercase characters and [A-Z] to specify the uppercase characters. Other commonly used applications are to use the -d option to delete characters and -s to replace or substitute characters.

When translating characters, you often need to use either single quotation marks or double quotation marks in the command line around the characters you intend to translate. Consider the following examples:

```
tr "c" " " < constants
```

and

```
tr 'c' ' ' < constants
```

Both of these commands accomplish the same thing.They replace all occurrences of the letter “c” with one blank space in the file constants (the input file), and display the translated result to the screen.

You use the tr command in Hands-on Project 5-10.

Using the pr Command to Format Your Output

The pr command prints the specified files on the standard output in paginated form. If you do not specify any files or you specify a file name of “-”, pr reads the standard input.

By default, pr formats the specified files into single-column pages of 66 lines. Each page has a five-line header, which, by default, contains the current file’s name, its last modification date, the current page, and a five-line trailer consisting of blank lines.

Syntax pr [-options] [file ...]

Dissection

■ Formats one or more files by providing pagination, columns, and column heads ■ Common options include:

-h (header format) lets you customize your header lines -d double-spaces output -l n sets the number of lines per page

Hands-on Project 5-11 enables you to use the pr command.

DESIGNING A NEW FILE-PROCESSING APPLICATION

One reason for learning UNIX/Linux selection, manipulation, and transformation com- mands is to develop an application.Whether you are creating an application for yourself or for others, the most important phase in developing a new application is creating a design. The design defines the information an application needs to produce.The design also defines how to organize this information into files, records, and fields, which are called logical structures because each represents a logical entity, such as a payroll file, an employee pay record, or a field for an employee Social Security number. Files consist of records, and records consist of fields.You learned about records and fields in Chapter 4.

This chapter gives you a preliminary look at developing an application by starting with record design considerations. How you set up records in a file can influence what you can do with an application. It also affects the ways in which you can use selection, manipulation, and transformation commands. If you pack the file with more information or fields than are needed, you make accessing data for a specific purpose inefficient or difficult. If you fail to include data that is needed, the application has limited value to the user, and the versatility of data-handling commands is underused.

Another consideration when you design records is how specific fields in one file might have particular importance for data handling. As you learned in Chapter 4, some fields can be used as key fields.These fields are important for enabling useful sorts and for linking the contents of two or more files through the join command. For example, by placing an employee's last name in a separate field without the first name and middle initial, you can sort on the last name or use the last name as a common field between two files you want to link.

Some organizations also give employees or students a special ID that can be used in records, such as in human resources or student information records.This ID can be a valuable key field for sorting, selecting, joining, and handling all types of information. For example, in a four-character ID, the first two characters might represent a department and the second two might represent the individual employee in that department. A user or programmer can use this field to sort employee records by department in a report. Another option is to use the grep command to create a specialized report for a particular department. A last name or ID field can also make it easier to evaluate duplicate records using the uniq and comm commands.

In this portion of the book, you begin by considering record design and key fields. Next, you apply this information and you use the tools you have learned to create an example Programmer Activity Status Report, such as might be developed for a company or organization. The report will show programmers' names and the number of projects on which each programmer is working. As you work your way through the next sections, be certain to stop and perform the Hands-on Projects referenced in each section before you move on to the next section.

In the following sections, you start by learning file, record, and field design, and then use the selection, manipulation, and transformation commands to select, manipulate, and format

information—all in preparation for formulating the Programmer Activity Status Report. You also learn more about shell scripts—how to use them in an application and how to run them using alternate methods. Finally, you create and test scripts that implement your knowledge and culminate in the Programmer Activity Status Report.

Designing Records

The first task in the record design phase is to define the fields in the records. These definitions take the form of a record layout that identifies each field by name and data type (such as numeric or nonnumeric). Design the file record to store only those fields relevant to the record's primary purpose. For example, to design your Programmer Activity Status Report in the Hands-on Projects, you need two files: one for programmer information and another for project information.You include a field for the programmer's name in the programmer file record and a field for the project description in the project file record. However, you do not store a programmer's name in a project file, even though the programmer might be assigned to the project. Also, you do not store project names in the programmer files.This structure is intended to give you an idea of how actual data files might be set up, so that each type of file can be used for a different purpose in a larger system of files and programs.

Allocating the space needed for only the necessary fields of the records keeps records brief and to the point. Short records, like short sentences, are easier to understand. Likewise, the simpler you make your application, the better it performs. However, you also want to be certain to include a field that uniquely identifies each record in the file. For instance, the programmer file record in this example includes a programmer number field to separate programmers who might have the same name.

The programmer number field in the programmer file record should be numeric. Numeric fields are preferable to nonnumeric fields for uniquely identifying records because the computer interprets numbers faster than nonnumeric data in the fields. The project record can use a nonnumeric project code to uniquely identify each project record, such as EA-100.

Linking Files with Keys

As you learned in Chapter 4, multiple files can be joined by a key—a common field shared by each of the linked files. Another important task in the design phase is to plan a way to join files, if necessary. For example, the programmer-project application uses the programmer's number to link the programmer to the project file. In Hands-on Project 5-12, you create two data files, the programmer and the project files, that both contain the programmer's number field for use as a key on which to manipulate and join data.

Before you begin to consider the process of creating files for the Programmer Activity Status Report project, review the record layouts for the programmer and project files illustrated in Figure 5-5.

Figure 5-5 Programmer and project file record layouts A sampling of records for the programmer file is as follows:

101:Johnson:John:K:39000 102:King:Mary:K:39800 103:Brown:Gretchen:K:35000 104:Adams:Betty:C:42000 ...

In this record design, the first field contains the programmer number, such as 101 in the first record.The programmer number is included as a key field to allow all kinds of data handling, such as using the sort, comm, and join commands—as you do in Hands-on Project 5-15, for example.The next three fields include the programmer's last name, first name, and middle initial. Dividing the full name into three fields opens the way for many data-handling

techniques, including sorting by last name or using all three fields for identifying duplicate records.The final field contains the employee's salary, which can be useful for printing salary information reports related to employee evaluations and raises.

A sampling of the records for the project file created in Hands-on Project 5-12 is as follows:

EA-100:1:Reservation Plus:110 EA-100:1:Reservation Plus:103 EA-100:1:Reservation Plus:107 EA-100:1:Reservation Plus:109 ...

The first field in the record is a code to identify a specific project, such as EA-100. An organization might use such a code to not only identify the project, but also to identify the persons, project team, department, division, or subsidiary who requests the project. In this example, the first two letters (EA) represent the department and the last three digits (100) represent the unique project number for that department.The second field contains the project status:

■ 1=Unscheduled ■ 2=Started ■ 3=Completed ■ 4=Cancelled

The third field contains the name of the project, such as "Reservation Plus." The fourth field is the programmer number to show which programmer is working on that project. One reason the programmer number is important to the records in the project file is that it can be used as a key field to link specific information in the project file with information in the programmer file.

Creating the Programmer and Project Files

Now that you have reviewed the basic elements of designing and linking records, you can begin the steps to implement your application design. Recall from Chapters 2 and 3 (“Exploring the UNIX/Linux File Systems and File Security” and “Mastering Editors”) that UNIX/Linux file processing can use flat files.Working with these files is easy, because you can create and manipulate them with text editors, such as vi and Emacs.The flowchart in Figure 5-6 provides an overview and analysis of programmer project assignments as derived from the programmer and project files used in this example.

A first step in this process is to create the programmer and project files and fill them with records,which you do in Hands-on Project 5-12.The files use a variable-record format,with a colon between each field as the delimiter.

As you read these sections, plan to complete each Hands-on Project as it is mentioned before reading further.

Formatting Output

Chapter 4 introduced the awk command and Awk programming language, which simplify preparation of formatted output.You get another introductory lesson in using awk here, because the printf capability in awk can be very powerful for creating a polished report for an application—and specifically for the Programmer Activity Status Report you are developing in the Hands-on Projects. As you have learned, Awk is a full-featured programming language and could have a chapter unto itself.The limited presentation in this chapter gives you another glimpse of Awk by introducing the use of the printf function within the awk command, which formats output.The printf function has the following syntax:

Syntax printf (format, $expr1, $expr2, $expr3)

Dissection

■ format is always required. It is an expression with a string value that contains literal text and specifications of how to format expressions in the argument list. Each specification begins with a percentage character (%), which identifies the code that follows as a modifier (- to left-justify; width to set size; .prec to set maximum string width or digits to the right of the decimal point; s for an array of characters (string); d for a decimal integer; f for a floating-point decimal number). Enclosed in double quotation marks (" "), format is often referred to as a mask that overlays the data fields going into it.

■ $expr1, $expr2, $expr3 represent data fields.These expressions typically take the form $1, $2, $3, and so on. In the programmer file, the expression $1 indicates the programmer number (the first field), $2 indicates the programmer’s last name (the second field), and $3 indicates the programmer’s first name (the third field).

You can use the awk command and printf function to print the following information from the programmer file: programmer number, programmer last name, and programmer first name, all left-justified.The command line to accomplish this is:

```
awk -F: '{printf "%d %-12.12s %-10.10s\n", $1, $2, $3}' programmer
```

Hands-on Project 5-13 enables you to use this command on the programmer file. Each % symbol in the format string corresponds with a $ field, as follows:

■ %d indicates that field $1 (programmer number) is to appear in decimal digits.

■ %-12.12s indicates that field $2 (programmer name) is to appear as a string.The hyphen (-) specifies the string is to be left-justified.The 12.12 indicates the string should appear in a field padded to 12 spaces, with a maximum size of 12 spaces.

■ %-10.10s indicates that field $3 (programmer salary) is to appear as a string.The hyphen (-) specifies the string is to be left-justified.The 10.10 indicates the string should appear in a field padded to 10 spaces, with a maximum size of 10 spaces.

The spaces that appear in the format string are printed exactly where they appear in relation to the awk and printf parameters—a space is between each % parameter (that is, after %d), for example, but spaces after each field designator, such as after $1, are, in this one case, optional. The trailing \n tells awk to skip a line after displaying the three fields. See Figure 5-7 for an example of the output of the report from this awk command.

Figure 5-7 awk report using printf to display the three fields

The awk command provides a shortcut when compared to other UNIX/Linux file- processing commands, when you need to extract and format data fields for output. For example, although it takes a few lines of code, you can use the cut, paste, and cat commands to extract and display the programmers' last names and salaries.As an alternative, you can do the same thing using a one-line awk command. Hands-on Project 5-14 enables you to compare using both techniques.Also, try Hands-on Project 5-15 for more experience using the cut, sort, uniq, comm, and join commands.

Using a Shell Script to Implement the Application

The report-generating application you are developing in the Hands-on Projects consists of many separate commands that must run in a certain order. As you recall from Chapter 4, you can create a script file to simplify the application.You store commands in a script file, which Designing a New File-Processing Application 233

in effect becomes a program.When you develop an application, you should usually test and debug each command before you place it in your script file.You can use the vi or Emacs editor to create script files. (Chapters 6 and 7,"Introduction to Shell Script Programming" and "Advanced Shell Programming," cover shell script programming in more detail.)

A shell script should contain not only the commands to execute, but also comments to identify and explain the shell script so that users or programmers other than the script's author can understand how it works. Comments also enable the original author to remember the logic of the script over time. Use the pound (#) character in script files to mark comments. This tells the shell that the words following # are a comment, not a UNIX/Linux command. Hands-on Project 5-16 enables you to build a shell script for a set of commands that create a temporary file showing information about the number of programs on which programmers are currently working.This script and temporary file will become a part of the process used to produce the Programmer Activity Status Report.

Running a Shell Script

You can run a shell script in virtually any shell that you have on your system. In this book, you use the Bourne Again Shell, or Bash, which is commonly used in Linux systems. In different shells, some incompatibilities often exist in terms of the exact use, syntax, and options associated with commands. One advantage to using the Bash shell is that it accepts more variations in command structures than the original Bourne shell; Bash is a freeware derivative of the Bourne and Korn shells.

When you create a shell script to run in Bash, you can immediately run the script by typing sh (to call the Bash shell interpreter) and then the name of the script, as follows:

```
sh testscript
```

Another advantage of using sh is that you can accompany it with several debugging options to help you troubleshoot problems with your script. (You learn more about these debugging options in Chapter 6.) For your beginning experiences with shell scripts in this chapter, you use sh simply to run your scripts.

In UNIX systems, sh calls the shell command interpreter for the shell that is the default to the particular UNIX system. In Linux, including Fedora, Red Hat Enterprise Linux, and SUSE, you can use either sh or bash to run a shell script and call the Bash shell interpreter. In these systems, sh is actually a link to the Bash shell.

Another way to run a shell script, which you learn more about in Chapter 6 (and already got a glimpse of in Hands-on Project 4-15 in Chapter 4), is to make it executable by using the x permission and then typing ./ prior to the script name when you run the script itself. In addition, when you write a script, it is advisable to specify with what shell the script is intended to be used.You do this by including a command—such as #!/bin/bash for the Bash 234 Chapter 5 Advanced File Processing

shell—on the first line of the script. Chapter 7 shows you how to implement this practice as your shell scripts become more advanced.

In Hands-on Project 5-17, you use the sh command to run the script created in Hands-on Project 5-16. In Hands-on Projects 5-18 and 5-19, you create and run scripts that are the next steps in creating the final Programmer Activity Status Report.

Putting It All Together to Produce the Report

An effective way to develop applications is to combine small scripts into a larger script file. In this way, it is easier to complete a large task by dividing it into a series of smaller ones—a basic programming rule. Also, through this approach, you can test each small script to ensure it works. In Hands-on Projects 5-16 through 5-19, you create, execute, and test individual small scripts in the process of preparing to create a Programmer Activity Status Report.After the scripts are tested, you can place the contents of each smaller script into a larger script file in the proper sequence to produce the final Programmer Activity Status Report. Hands-on Project 5-20 pulls together your smaller projects into one large task to generate the report.

CHAPTER SUMMARY

The UNIX/Linux file-processing commands can be organized into two categories: (1) selection commands and (2) manipulation and transformation commands. Selection commands extract information. Manipulation and transformation commands alter and transform extracted information into useful and appealing formats.

The grep command searches for a specific pattern in a file. The uniq command removes duplicate lines from a file.You must sort the file because uniq

compares only consecutive lines.

The comm command compares lines common to two different files, file1 and file2, and produces three-column output that reports variances between the files.

The diff command attempts to determine the minimum set of changes needed to convert the contents of one file to match the contents of another file.

When you want to know the byte, word, or line count in a file, use the wc command.

The sed command is a stream editor designed to make global changes to large files. Minimum requirements to run sed are an input file and a command that tells sed what actions to apply to the file. Input to the sed action can be from the command line or through a script file.

The tr command copies data read from the standard input to the standard output, substituting or deleting the characters specified by options and patterns.

The pr command prints the standard output in pages.

The design of a file-processing application reflects what the application needs to produce. The design also defines how to organize information into files, records, and fields, which are also called logical structures.

Use a record layout to identify each field by name and data type (numeric or nonnumeric). Design file records to store only those fields relevant to each record's primary purpose.

Shell scripts should contain commands to execute and comments to identify and explain the script.The pound (#) character is used in script files for comments.

Write shell scripts in stages so that you can test each part before combining them into one script. Using small shell scripts and combining them in a final shell script file is an effective way to develop applications.

COMMAND SUMMARY: REVIEW OF CHAPTER 5 COMMAND

mand	Purpose	Option
omm	Compares and outputs lines common to two files	-1 do not display lines that are only in file1 -2 do not display lines that are only in file2 -3 do not display lines appearing in both file1 and file2
diff	Compares two files and determines which lines differ	-b ignores blanks that repeat -B does not compare for blank lines -i ignores case -c shows lines surrounding the line that differs (for context) -y displays the differences side-by-side in columns
rep	Selects lines or rows	-i ignores case -l lists only file names -c only counts the number of lines matching the pattern instead of showing them -r searches through files under all subdirectories -n includes the line number for each line found -v displays only lines that don't contain the search pattern
pr	Formats a specified file	-d double-spaces the output -h customizes the header line -l n sets the number of lines per page
rintf	Tells the Awk program what action to take for formatting and printing information	
sed	Specifies an editing command or a script file containing sed commands	a\ appends text after a line p displays lines d deletes specified text s substitutes specified text -e specifies multiple commands on one line -n indicates line numbers on which to work
sh	Executes a shell script	
tr	Translates characters	-d deletes input characters found in string1 from the output

		-s checks for sequences of string1 repeated consecutive times
niq	Removes duplicate lines to create unique output	-u outputs only the lines of the source file that are not duplicated -d outputs one copy of each line that has a duplicate, and does not show unique lines -i ignores case -c starts each line by showing the number of each instance
wc	Counts the number of lines, bytes, or words in a file	-c counts the number of bytes or characters -l counts the number of lines -w counts the number of words

KEY TERMS

logical structure — The organization of information in files, records, and fields, each of which represents a logical entity, such as a payroll file, an employee's pay record, or an employee's Social Security number. manipulation and transformation commands — A group of commands that alter and format extracted information so that it's useful and can be presented in a way that is appealing and easy to understand.

pipe operator (|) — The operator that redirects the output of one command to the input of another command. record layout — A program and data file design step that identifies the fields, types of records, and data types to be used in data files.

selection commands — The file-processing commands that are used to extract information.

CHAPTER 6

INTRODUCTION TO SHELL SCRIPT PROGRAMMING

After reading this chapter and completing the

exercises, you will be able to:

- Understand the program development cycle
- Compare UNIX/Linux shells for creating scripts
- Use shell variables, operators, and wildcard character

PREVIEWING THE APPLICATION

As you learned in Chapters 4 and 5 ("UNIX/Linux File Processing" and "Advanced File

Processing"), commands such as grep, cut, paste, and awk are powerful commands for

manipulating data. Although these commands are powerful, they can be difficult for nontechnical users, in part because they often must be combined in long sequences to achieve the results you want. Repeatedly executing these command sequences can be cumbersome,even for experienced technical users. You've discovered in earlier chapters that shell scripts can help eliminate these problems.

One advantage of shell scripts is that you can create them to present user-friendly screens—for example, screens that automatically issue commands such as grep and awk to extract, format, and display information. This gives nontechnical users access to powerful features of UNIX/Linux. For your own use, shell scripts save time by automating long command sequences that you must perform often.

The shell script application you develop in this chapter and enhance in Chapter 7, "Advanced Shell Programming,"is a simulated employee information system that stores and displays employee data—such as you might commonly find in a human resources system in an organization. It presents a menu of operations from which the user can choose. Among other tasks,these operations automate the process of inputting,searching for,formatting,and displaying employee records. For preliminary testing, you create and use a data file that contains a sampling of employee records, similar to one that an experienced shell programmer might use for testing.

As you learn the tools needed to develop your application in this chapter, you gain experience with the following scripting and programming features of the UNIX/
Linux shell:

■ Shell variables—Your scripts often need to keep values in memory for later use.
Shell variables temporarily store values in memory for use by a shell script. They use symbolic names that can access the values stored in memory. In this case, a symbolic name is a name consisting of letters, numbers, or characters and is used to reference the contents of a variable; often the name reflects a variable's purpose or contents.

■ Shell script operators—Shell scripts support many shell script operators, including those for assigning the contents of a shell variable, for evaluating information, for performing mathematical operations, and for piping or redirection of input/
output.

■ Logic or control structures—Shell scripts support logic structures (also called control structures), including sequential logic (for performing a series of commands), decision logic (for branching from one point in a script to a different
point), looping logic (for repeating a command several times), and case logic (for choosing an action from several possible alternatives).

In addition, you learn special commands for formatting screen output and positioning the cursor. Before you begin writing your application, it is important to understand more about the program development cycle and the basic elements of programming.

THE PROGRAM DEVELOPMENT CYCLE

The process of developing an application is known as the program development cycle. The steps involved in the cycle are the same whether you are writing shell scripts or high-level language programs.

The process begins by creating program specifications—the requirements the application must meet. The specifications determine what data the application takes as input, the processes that must be performed on the data, and the correct output.

After you determine the specifications, the design process begins. During this process, programmers create file formats,screen layouts,and algorithms. An algorithm is a sequence of procedures,programming code,or commands that result in a program or that can be used as part of a program.Programmers use a variety of tools to design complex applications. You learn about some of the tools in this chapter and about additional tools in Chapter 7.

After the design process is complete,programmers begin writing the actual code,which they must then test and debug. Debugging is the process of going through program code to locate errors and then fix them. When programmers find errors, they correct them and begin the testing process again. This procedure continues until the application performs satisfactorily.

Figure 6-1 illustrates the program development cycle.

Using High-Level Languages

Computer programs are instructions often written using a high-level language, such as COBOL,Visual Basic, C, or C++. A high-level language is a computer language that uses English-like expressions. For example, the following COBOL statement instructs the computer to add 1 to the variable COUNTER:

```
ADD 1 TO COUNTER.
```

Here is a similar statement, written in C++:

```
counter = counter + 1;
```

A program's high-level language statements are stored in a file called the source file. This is the file that the programmer creates with an editor such as vi or Emacs. The source file cannot execute, however, because the computer can only process instructions written in low-level machine language. As you recall from Chapter 3,"Mastering Editors," machinelanguage instructions are cryptic codes expressed in binary numbers. Therefore, the highlevel

source file must be converted into a low-level machine language file,as described next.

The source file is converted into an executable machine--language file by a program called a compiler. The compiler reads the lines of code that the programmer wrote in the source file and converts them to the appropriate machine-language instructions. For example, the Linux C and C++ compilers are named gcc and g++. The following command illustrates how to compile the C++ source code file, datecalc.C, so that you can run it as the program datecalc: g++ datecalc.C -o datecalc In this sample command, the -o option followed by datecalc instructs the compiler to create an executable file,datecalc. The source file is datecalc.C. The command causes the compiler to translate the C++ program datecalc.C into an executable machine-language program, which is stored in the file datecalc. You learn more about C and C++ programming in

Chapter 10,"Developing UNIX/Linux Applications in C and C++."

As you learn in Chapter 10, some important differences exist between

C and C++ source code, and therefore it is necessary to use the correct compiler

(gcc versus g++). Remember, when you invoke one of these compilers in Linux,

the gcc compiler expects C files to have the .c extension, whereas the g++

compiler expects C++ files to have the .C extension.

If a source file contains syntax errors (grammatical mistakes in program language use), it

cannot be converted into an executable file. The compiler locates and reports any syntax

errors, which the programmer must correct.

After compiling, the executable program might still contain fatal run-time errors

or logic errors. Fatal run-time errors cause the program to abort, for example,

due to an invalid memory location specified in the program code. Logic errors

cause the program to produce invalid results because of problems such as

flawed mathematical statements.

Another way to accomplish programming tasks is to develop UNIX/Linux shell scripts,

which you learn in this chapter.

Using UNIX/Linux Shell Scripts

First introduced in Chapter 4,UNIX/Linux shell scripts are text files that contain sequences

of UNIX/Linux commands. Like high-level source files, a programmer creates shell scripts

with a text editor. Unlike high-level language programs, shell scripts do not have to be

converted into machine language by a compiler. This is because the UNIX/Linux shell acts

as an interpreter when reading script files. As this interpreter reads the statements in a script

file, it immediately translates them into executable instructions, and causes them to run. No

executable file is produced because the interpreter translates and executes the scripted

statements in one step.If a syntax error is encountered,the execution of the shell script halts.

After you create a shell script, you tell the operating system that the file can be executed.

This is accomplished by using the chmod ("change mode") command that you learned in Chapters 2 and 4 ("Exploring the UNIX/Linux File Systems and File Security" and "UNIX/Linux File Processing,") to change the file's mode. The mode determines how the file can be used. Recall that modes can be denoted by single-letter codes: r (read), w (write), and x (execute). Further, the chmod command tells the computer who is allowed to use the file: the user or owner (u), the group (g), or all other users (o). For a description of the chmod command, see Appendix B,"Syntax Guide to UNIX/Linux Commands."

Recall from Chapter 4 that you can change the mode of a file so that UNIX/Linux recognize it as an executable program (mode x) that everyone (user, group, and others) can use. In the following example, the user is the owner of the file:

```
$ chmod ugo+x filename <Enter>
```

Alternatively, you can make a file executable for all users by entering either:

```
$ chmod a+x filename <Enter>
```

or

```
$ chmod 755 filename <Enter>
```

In chmod a+x, the a stands for all and is the same as ugo. Also, remember from Chapter 2 that chmod 755 gives owner (in the first position) read, write, and execute permissions (7). It also gives group (in the second position) read and execute permissions (5), and gives others (in the third position) read and execute permissions (5).

After you make the file executable, you can run it in one of several ways:

■ You can simply type the name of the script at the system command prompt. However, before this method can work, you must modify your default directory path to include the directory in which the script resides. The directory might be the source or bin directory under your home directory. If you use this method, before any script or program can be run it must be retrieved from a path identified in the PATH variable, which provides a list of directory locations where UNIX or Linux looks to find executable scripts or programs. You learn how to temporarily modify the PATH variable in the"Variables"section later in this chapter; you learn how to permanently modify the PATH variable in Chapter 7.

■ If the script resides in your current directory, which is not in the PATH variable, you can run the script by preceding the name with a dot slash (./) to tell UNIX/Linux to look in the current directory to find it, as follows:

```
$ ./filename <Enter>
```

■ If the script does not reside in your current directory and is not in the PATH variable, you can run it by specifying the absolute path to the script. For example, if the script is in the data directory under your home directory,you can type either of the following (using Tom's home directory as an example):

```
$ /home/tom/data/filename <Enter>
```

or

```
$ ~/data/filename <Enter>
```

Shell scripts run less quickly than compiled programs because the shell must interpret each UNIX/Linux command inside the executable script file before it is executed. Whether a programmer uses a script or a compiled program (such as a C++ program) is often related to several factors:

■ Whether the programmer is more proficient in writing scripts than source code for a compiler

■ Whether there is a need for the script or program to execute as quickly as possible, such as to reduce the load on the computer's resources when there are multiple users

■ Whether the job is relatively complex; if so, a compiled program might offer more flexible options or features

Prototyping an Application

A prototype is a running model of your application, which lets you review the final results before committing to the design. Using a shell script to create a prototype is often the quickest and most efficient method because prototyping logic and design capabilities reside within UNIX/Linux.

After the working prototype is approved, the script can be rewritten to run faster using a compiled language such as C++. If the shell script performs well, however, you might not need to convert it to a compiled program.

Using Comments

In Chapter 5, you were introduced to using comments to provide documentation about a script. Plan to use comments in all of your scripts and programs, so that later it is easier to remember how they work.

Comment lines begin with a pound (#) symbol, such as in the following example from the pact script you created in Hands-on Project 5-16 in Chapter 5:

```
# =====================================================
# Script Name: pact
# By: Your initials
# Date: November 2009
# Purpose: Create temporary file, pnum, to hold the
# count of the number of projects each
# programmer is working on. The pnum file
# consists of:
# prog_num and count fields
# ==========================================================
cut -d: -f4 project | sort | uniq -c | awk '{printf "%s:
%s\n",$2,$1}' > pnum
# cut prog_num, pipe output to sort to remove duplicates
# and get count for prog/projects.
# output file with prog_number followed by count
```

In this example, comment lines appear at the beginning of the script and after the cut command. You can place comment lines anywhere in a script to provide documentation. For example, in the Hands-on Projects for this chapter, you typically place comments at the beginning of a script to show the script name, the script's author, the date the script was written, and the script's purpose. As you write code in this and later chapters, insert any

additional comment lines that you believe might be helpful for later reference. Some examples of what you might comment include:

■ Script name, author(s), creation date, and purpose

■ Modification date(s) and the purpose of each modification

■ The purpose and types of variables used (You learn about variables in this chapter.)

■ Files that are accessed, created, or modified

■ How logic structures work (You create logic structures in this chapter.)

■ The purpose of shell functions (You create shell functions in Chapter 7.)

■ How complex lines of code work

■ The reasons for including specific commands

Although writing comments might take a little extra time,in the long run the comments can save you much more time when you need to modify that script or incorporate it in an application with other scripts.

THE PROGRAMMING SHELL

Before you create a script, choose the shell in which to run the script. As you learned in Chapter 1, UNIX/Linux versions support different shells and each shell has different capabilities. Also, recall that all Linux versions use the Bash shell (Bourne Again Shell) as the default shell. Table 6-1 lists the three shells that come with most Linux distributions, their derivations, and distinguishing features in relation to shell programming.

ame	**Original shell from which derived**	**Description in terms of Shell Programming**
	Bourne and Korn shells	Offers strong scripting and programming language features, such as shell variables, logic structures, and math/logic expressions; combines the best features of the Bourne and Korn shells
csh	C shell	Conforms to a scripting and programming language format; shell expressions use operators similar to those found in the C programming language
sh	Korn shell	Is similar to the Bash shell in many respects, but also has syntax similar to that of C programming; useful if you are familiar with older Korn shell scripts

The Bash shell offers improved features over the older Bourne and Korn shells and is fully backward compatible with the Bourne shell. In addition, the Bash shell, when compared to the other shells, has a more powerful programming interface. For these reasons, you use the Bash shell for shell scripts in this book.

The manual pages in Fedora, Red Hat Enterprise Linux, and SUSE contain a generous amount of documentation about the Bash shell. Just enter man bash to access the documentation.

Now that you have selected the shell,it is important to learn about several basic features used by shell scripts, including variables, shell operators, and special characters.

VARIABLES

Variables use symbolic names that represent values stored in memory. The three types of variables discussed in this section are configuration variables, environment variables, and shell variables. Configuration variables are used to store information about the setup of the operating system, and after they are set up, you typically do not change them.

You can set up environment variables with initial values that you can change as needed. These variables, which UNIX/Linux read when you log in, determine many characteristics of your login session. For example, in Chapter 2 you learned about the PS1 environment variable, which determines the way your prompt appears. In addition, UNIX/Linux use environment variables to determine such things as where it should look for programs,which shell to use, and the path of your home directory.

Shell variables (defined earlier) are those you create at the command line or in a shell script. They are very useful in shell scripts for temporarily storing information.

Environment and Configuration Variables

Environment and configuration variables bear standard names, such as PS1, HOME, PATH, SHELL,USERNAME,and PWD.(Configuration and environment variables are capitalized to distinguish them from user variables.) A script file in your home directory sets the initial values of environment variables. You can use these variables to set up and personalize your login sessions. For example, you can set your PATH variable to search for the location of shell scripts that other users have created, so you can more easily execute those scripts. Table 6-2 lists standard Bash shell environment and configuration variables.

You can, at any time, use the printenv command to view a list of your current environment and configuration variables, which you should typically do before you change any. (See Figure 6-2.) Hands-on Project 6-1 enables you to view your environment variables.

Syntax printenv [-options] [variable name]

Dissection

■ Prints a listing of environment and configuration variables

■ Specifies one or more variables as arguments to view information only about those variables

Besides the printenv command, consider using the set command (discussed later in this chapter) with no arguments to view your current Bash shell environment, including environment variables, shell script variables, and shell functions. (You learn about shell functions in Chapter 7.) To learn more about the environment and configuration variables used on your system, type man bash at the command line. Scroll to the section, Shell Variables.

Table 6-2 Standard Bash shell environment and configuration variables

Name	**Variable Contents**	**Determined by**
HOME	Identifies the path name for user's home directory	System
LOGNAME	Holds the account name of the user currently logged in	System
PPID	Refers to the parent ID of the shell	System
TZ	Holds the time zone set for use by the system	System
IFS	Enables the user to specify a default delimiter for use in working with files	Redefinable
LINEND	Holds the current line number of a function or script	Redefinable
MAIL	Identifies the name of the mail file checked by the mail utility for received messages	Redefinable
MAILCHECK	Identifies the interval for checking and received mail (example: 60)	Redefinable
PATH	Holds the list of path names for directories searched for executable commands	Redefinable
PS1	Holds the primary shell prompt	Redefinable

PS2	Contains the secondary shell prompt	Redefinable
PS3 and PS4	Holds prompts used by the set and select commands	Redefinable
SHELL	Holds the path name of the program for the type of shell you are using	Redefinable
BASH	Contains the absolute path to the Bash shell, such as /bin/bash	User defined
BASH_VERSION	Holds the version number of Bash	User defined
CDPATH	Identifies the path names for directories searched by the cd command for subdirectories	User defined
ENV	Contains the file name containing commands to initialize the shell, as in .bashrc or .tcshrc	User defined
EUID	Holds the user identification number (UID) of the currently logged in user	User defined
EXINIT	Contains the initialization commands for the vi editor	User defined
FCEDIT	Enables you to access a range of commands in the command history file; FCEDIT is a Bash shell utility and is the variable used to specify which editor (vi by default) is used when you invoke the FC command	User defined
FIGNORE	Specifies file name suffixes to ignore when working with certain files	User defined
FUNCNAME	Contains the name of the function that is running, or is empty if there is no shell function running	User defined
GROUPS	Identifies the current user's group memberships	User defined
HISTCMD	Contains the sequence number that the currently active command is assigned in the history index of commands that already have been used	User defined
HISTFILE	Identifies the file in which the history of the previously executed commands is stored	User defined
HISTFILESIZE	Sets the upward limit of command lines that can be stored in the file specified by the HISTFILE variable	User defined
HISTSIZE	Establishes the upward limit of commands that the Bash shell can recall	User defined
HOSTFILE	Holds the name of the file that provides the Bash shell with information about its network host name (such as localhost.localdomain) and IP address (such as 129.0.0.24); if the HOSTFILE variable is empty, the system uses the file /etc/hosts by default	User defined
HOSTTYPE	Contains information about the type of computer that is hosting the Bash shell, such as i386 for an Intel-based processor	User defined
INPUTRC	Identifies the file name for the Readline start-up file overriding the default of /etc/inputrc	User defined
MACHTYPE	Identifies the type of system, including CPU, operating system, and desktop	User defined
MAILPATH	Contains a list of mail files to be checked by mail for received messages	User defined
MAILWARNING	Enables (when set) the user to determine if she has already read the mail currently in the mail file	User defined
OLDPWD	Identifies the directory accessed just before the current directory	User defined
OPTIND	Shows the index number of the argument to be processed next, when a command is run using one or more option arguments	User defined
OPTARG	Contains the last option specified when a command is run using one or more option arguments	User defined

OPTERR	Enables Bash to display error messages associated with command-option arguments, if set to 1 (which is the default established each time the Bash shell is invoked)	User defined
OSTYPE	Identifies the type of operating system on which Bash is running, such as linux-gnu	User defined
PROMPT_COMMAND	Holds the command to be executed prior to displaying a primary prompt	User defined
PWD	Holds the name of the directory that is currently accessed	User defined
RANDOM	Yields a random integer each time it is called, but you must first assign a value to the RANDOM variable to properly initialize random number generation	User defined
REPLY	Specifies the line to read as input, when there is no input argument passed to the built-in shell command, which is read	User defined
SHLVL	Contains the number of times Bash is invoked plus one, such as the value 3 when there are two Bash (terminal) sessions currently running	User defined
TERM	Contains the name of the terminal type in use by the Bash shell	User defined
TIMEFORMAT	Contains the timing for pipelines	User defined
TMOUT	Enables Bash to stop or close due to inactivity at the command prompt, after waiting the number of seconds specified in the TMOUT variable (TMOUT is empty by default so that Bash does not automatically stop due to inactivity.)	User defined
UID	Holds the user identification number of the currently logged in user	User defined

SHELL VARIABLES

Shell variables are variables that you can define and manipulate for use with program commands that you employ in a shell. These are variables that are temporarily stored in memory and that you can display on the screen or use to perform specific actions in a shell script. For example, you might define the shell variableTODAY to store today's date so you can later recall it and then print it on a report generated from a shell script.

When you work with shell variables, keep in mind guidelines for handling them and for naming them. Some basic guidelines for handling shell variables are:

■ Omit spaces when you assign a variable without using single or double quotation marks around its value, such as when assigning a numerical value—use x=5 and not x=5. (This type of assignment also enables you to perform mathematical operations on the assigned value.)

■ To assign a variable that must contain spaces, such as a string variable, enclose the value in double or single quotation marks—use fname="Thomas F. Berentino" and not fname=Thomas F. Berentino. A string variable is a nonnumeric field of information treated simply as a group of characters. Numbers in a string are considered characters rather than digits.

■ To reference a variable, use a dollar sign ($) in front of it or enclose it in curly brackets ({ }).

■ If the variable consists of an array (a set of values), use square brackets ([ ]) to refer to a specific position of a value in an array—use myarray[0]=value1 for the first value in the array, for example.

■ Export a shell variable to make the variable available to other shell scripts (as discussed in the following section).

■ After you create a shell variable, you can configure it so that it cannot be changed by entering the readonly command with the variable name as the argument,such as readonly fname.

Sample guidelines for naming shell variables are:

■ Avoid using the dollar sign in a variable name, because this can create confusion with using the dollar sign to reference the shell variable.

■ Use names that are descriptive of the contents or purpose of the shell variable—use lname for a variable to contain a person's last name, instead of var or x, for example.

■ Use capitalization appropriately and consistently—for instance, if you are defining address information, use variable names such as city, state, zip and not City, STATE, zip. Note that some programmers like to use all lowercase letters or all uppercase letters for variable names. For example, many script, C, and C++ programmers prefer using all lowercase letters when possible.

■ If a variable name is to consist of two or more words, use underscores between the words—use last_name and not last name, for example.

In the next section, you learn about shell operators, which are used to define and evaluate variables, such as environment and shell variables.

SHELL OPERATORS

Bash shell operators are divided into four groups:

■ Defining operators

■ Evaluating operators

■ Arithmetic and relational operators

■ Redirection operators

You learn about each of these groups of operators in the following sections. You also learn how to use the export command to make a variable you have defined available to a shell script. Finally, you look at how to modify the PATH environment variable to make it easier to run shell scripts.

DEFINING OPERATORS

Defining operators are used to assign a value to a variable. Evaluating operators are used for actions such as determining the contents of a variable. The equal sign (=) is one of the most common operators used to define a variable. For example, assume that you want to create a variable called NAME and assign Becky as the value to be contained in the variable. You would set the variable as follows:

```
NAME=Becky
```

The variable names or values that appear to the left and right of an operator are its operands. The name of the variable you are setting must appear to the left of the = operator. The value of the variable you are setting must appear to the right.

Notice there are no spaces between the = operator and its operands.

Sometimes, it is necessary to assign to a variable's contents a string of characters that contain spaces, such as Becky J. Zubrow. To make this kind of assignment, you surround the variable contents with double quotation marks as follows:

```
NAME="Becky J. Zubrow"
```

Another way to assign a value to a shell variable is by using the back quote (‘) operator.(The back quote is not the same as the apostrophe or single quotation mark; see the following Tip.) This operator is used to tell the shell to execute the command inside the back quotes and then store the result in the variable. For example, in the following:

```
LIST=‘ls‘
```

The ls command is executed and a listing of the current working directory is stored in the LIST variable.

SHELL OPERATORS 285 6

On many standard keyboards, the key for the back quote operator is located in the upper-left corner under the Esc key and is combined with the tilde (~) on that key.

EVALUATING OPERATORS

When you assign a value to a variable, you might want to evaluate it by displaying its contents via an evaluating operator. You can use the dollar sign ($) in front of the variable along with the echo command to view the contents. For example, if you enter:

```
echo $NAME
```

you see the contents of the NAME variable you created earlier. You can also use the format echo "$NAME" to view the variable's contents. However, if you enter:

echo '$NAME'

using single quotation marks, the contents of NAME are suppressed and all you see is

$NAME echoed on the screen.

Try Hands-on Project 6-2 to use the defining and evaluation operators.

ARITHMETIC AND RELATIONAL OPERATORS

Arithmetic operators consist of the familiar plus (+) for addition, minus (-) for subtraction, asterisk (*) for multiplication, and slash (/) for division. Relational operators compare the relationship between two values or arguments,such as greater than (>),less than (<), equal to (=), and others.Table 6-3 explains the arithmetic and relational operators. For a complete listing of arithmetic operators enter man bash and go to the section, ARITHMETIC EVALUATION.

TABLE 6-3 EXAMPLES OF THE SHELL'S ARITHMETIC AND RELATIONAL OPERATORS

OPERATOR	DESCRIPTION	EXAMPLE
-, +	Unary minus and plus	+R (denotes positive R) -R (denotes negative R)
!, ~	Logical and bitwise negation	!Y (returns 0 if Y is nonzero, returns 1 if Y is zero) ~X (reverses the bits in X)
***, /, %**	Multiplication, division, and remainder	A*B (returns A times B) A/B (returns A divided by B) A%B (returns the remainder of A divided by B)
+, -	Addition, subtraction	X+Y (returns X plus Y) X-Y (returns X minus Y)
>, <	Greater than and less than	M>N (Is M greater than N?) M<N (Is M less than N?)
=, !=	Equality and inequality	Q=R (Is Q equal to R?) Q != R (Is Q not equal to R?)

When using arithmetic operators, the usual mathematical precedence rules apply: Multiplication and division are performed before addition and subtraction. For example, the value of the expression 6+4 * 2 is 14, not 20. Precedence can be overridden, however, by using parentheses. For example, the value of the expression (6 + 4) * 2 is 20, not 14. Other mathematical rules also apply; for example, division by zero is treated as an error.

To store arithmetic values in a variable, use the let statement. For example, the following command stores 14 in the variable X (See Figure 6-3 using the echo command to show the contents of X after using the let command.):

```
let X=6+4*2
```

Notice in the preceding example that there is one space between let and the expression that

follows it. Also, there are no spaces in the arithmetic equation following a let statement. In

this example, there are no spaces on either side of the equal (=), plus (+), and multiplication

(*) operators.

You can use shell variables as operands to arithmetic operators. Assuming the variable X has

the value 14, the following command stores 18 in the variableY:

let Y=X+4

Figure 6-3 Using let to set the contents of a shell variable

Shell Operators 287 6

SYNTAX LET EXPRESSION WITH OPERATORS

Dissection

■ Performs a given action on numbers that is specified by operators and stores the result in

a shell variable

■ Parentheses are used around specific expressions if you want to alter the mathematical

precedence rules or to simply ensure the result is what you intend.

let is a built-in command for the Bash shell. For documanetation about let, enter man

bash, and scroll down to the section SHELL BUILTIN COMMANDS. Try Hands-on

Project 6-3 to learn how to use the let command.

REDIRECTION OPERATORS

Recall that the > redirection operator overwrites an existing file. For example, in cat file1

> file2, the contents of file1 overwrite the contents of file2. If you write a shell script that

uses the > operator to create a file,you might want to prevent it from overwriting important

information. You can use the set command with the -o noclobber option to prevent a file from

being overwritten, as in the following example:

$ set -o noclobber <Enter>

SYNTAX SET [-OPTIONS] [ARGUMENTS]

Dissection

■ With no options,displays the current listing of Bash environment and shell script variables

■ Useful options include:

-a exports all variables after they are defined

-n takes commands without executing them, so you can debug errors without affecting

data (Also see the sh -n command later in this chapter.)

-o sets a particular shell mode—when used with noclobber as the argument,it prevents files

from being overwritten by use of the > operator

-u shows an error when there is an attempt to use an undefined variable

-v displays command lines as they are executed

set is another built-in command for the Bash shell. For documentation about set, enter man

bash, and scroll down to the section SHELL BUILTIN COMMANDS.

If you want to save time and automatically export all shell script variables you

have defined, use set with the -a option.

However,you can choose to overwrite a file anyway by placing a pipe character (|) after the

redirection operator:

```
$ set -o noclobber <Enter>
$ cat new_file > old_file <Enter>
bash: old_file: cannot overwrite existing file
$ cat new_file >| old_file <Enter>
```

Avoid employing the -o noclobber option if you are using the Bash shell in the

X Window interface with the KDE desktop. On some distributions, using the option in this manner can unexpectedly terminate the command-line session.

EXPORTING SHELL VARIABLES TO THE ENVIRONMENT

Shell scripts cannot automatically access variables created and assigned on the command line

or by other shell scripts. To make a variable available to a shell script,you must use the export

command to give it a global meaning so that it is viewed by the shell as an environment

variable.

SYNTAX EXPORT [-OPTIONS] [VARIABLE NAMES]

Dissection

■ Makes a shell variable global so that it can be accessed by other shell scripts or programs,

such as shell scripts or programs called within a shell script

■ Useful options include:

-n undoes the export, so the variable is no longer global

-p lists exported variables

export is a built-in Bash shell command, which means you can find help documentation by

entering man bash and scrolling to the SHELL BUILTIN COMMANDS section. Try

Hands-on Project 6-4 to use the export command in the Bash shell.

MODIFYING THE PATH VARIABLE

Just as shell variables are not universally recognized until you export them, the same is true

for executing a shell script. Up to this point, you have used ./ to run a shell script. This is because the shell looks for programs in the directories specified by the PATH variable.If you are developing a shell script in a directory that is not specified in your PATH environment variable, you must type ./ in front of the shell name. If you just type the name of the shell by itself, the script doesn't run because it is not in your currently defined path—which means that the shell interpreter cannot find it to run. You need to type ./ to tell the shell interpreter to look in your current working directory.

For example,in some UNIX/Linux systems,such as Fedora,Red Hat Enterprise Linux,and SUSE, your home directory is not automatically defined in your current path. You can verify this by typing the following to see what directories are in your path:

```
echo $PATH
```

In Fedora or Red Hat Enterprise Linux, for example, you will likely see a path such as the following:

```
/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/
home/username/bin
```

Notice that each directory in the path is separated by a colon. Your home directory, as represented by /home/username is not in the path by default, but another directory, /home/username/bin, in which programs might be stored, is in the path by default—although the actual directory, /home/username/bin, might not be created by default on your system even though it is in the path.

Because new shell scripts are most often kept in the current directory while they are being tested, you should add the current working directory to the PATH variable. Here is an example command for how you can do this quickly:

```
PATH=$PATH:.
```

Remember, the shell interprets $PATH as the contents of the PATH variable. This sample command sets the PATH variable to its current contents. The colon and dot (.) add the current directory to the search path so that the shell program can locate the new program. After you type this command, you can execute new shell scripts by simply using the name of the script without prefacing it with the ./ characters. Hands-on Project 6-5 enables you to try this.

When you type PATH=$PATH:., the current working directory is temporarily stored as part of your path only for the duration of your login session.

Configuring to have the current directory set in your path does involve some risk if a hacker gains access to your account while you are logged in. For example, a hacker might gain access through an open port (communication path in a network protocol). If you choose to put your current working directory in the PATH variable, be certain you have secured access to your account, such as through closing unused ports. For more Information about operating system security, including for UNIX/Linux systems, see Guide to Operating Systems Security (Course Technology, ISBN 0-619-16040-3).

MORE ABOUT WILDCARD CHARACTERS

Shell scripts frequently use the asterisk (*) and other wildcard characters (such as ? and [ ]), which help to locate information containing only a portion of a matching pattern. For example, to list all program files with names that contain a .c extension, use the following command:

```
ls *.c
```

Wildcard characters are also known as glob characters. If an unquoted argument contains one or more glob characters,the shell processes the argument for file name generation.Glob characters are part of glob patterns, which are intended to match file names and words. Special constructions that might appear in glob patterns are:

- The question mark (?) matches exactly one character, except for the backslash and period.
- The asterisk (*) matches zero or more characters in a file name.
- [chars] defines a class of characters. The glob pattern matches any single character in the class. A class can contain a range of characters, as in [a-z].

For example, assume the working directory contains files chap1, chap2, and chap3. The following command displays the contents of all three files:

```
more chap[1-3] <Enter>
```

The commands and variables used in shell scripts are organized into different logic structures. In the next sections, you learn how to use logic structures for effective script handling.

SHELL LOGIC STRUCTURES

Logic structures are techniques for structuring program code and affect the order in which the code is executed or how it is executed, such as looping back through the code from a particular point or jumping from one point in the code to another. The four basic logic structures needed for program development are:

- Sequential logic
- Decision logic
- Looping logic

■ Case logic

Each of these logic structures is discussed in the following sections.

SEQUENTIAL LOGIC

Sequential logic works so that commands are executed in the order in which they appear in the script or program.

For example, consider a sales manager in a company who begins each week by tallying sales information.First,she runs the tally_all program to compute the year’s gross sales statistics up to the current date.Then she runs the profit_totals program to tally the profit statistics.Next, she runs the sales_breakdown report program that shows the sales performance of the 40 salespeople she manages. Finally, she runs the management_statistics report program to provide the sales statistics needed by her management.The sales manager can automate her work by creating a script that uses sequential logic, first running the tally_all program, then the profit_totals, the sales_breakdown, and the management_statistics programs. Her shell script with sequential logic would have the following sequence of commands (note that her system is set up so that all she needs to do is to enter the programs names at the commandline):

```
tally_all
profit_totals
sales_breakdown
management_statistics
```

The only break in sequence logic comes when a branch instruction changes the flow of execution by redirecting to another location in the script or program. A branch instruction is one that tells the program to go to a different section of code.

Many scripts are simple, straightforward command sequences. An example is the Programmer Activity Status Report script you wrote in Chapter 5, and is listed next. The shell executes the script’s commands in the order they appear in the file. You use sequential logic to write this type of application.

```
#===============================================================
# Script Name: practivity
# By: MP
# Date: November 2009
# Purpose: Generate Programmer Activity Status Report
#===============================================================
cut -d: -f4 project | sort | uniq -c | awk '{printf "%s:
```

```
%s \n",$2,$1}' > pnum
cut -d: -f1-4 programmer | sort -t: +0 -1 | uniq > pnn
join -t: -a1 -j1 1 -j2 1 pnn pnum > pactrep
```

```
# Print the report
awk '
BEGIN {
{ FS = ":" }
{ print "\tProgrammer Activity Status Report\n" }
{ "date" | getline d }
{ printf "\t %s\n",d }
{ print "Prog# \t*--Name--* Projects\n" }
{ print "============================================\n"}
}
{ printf "%-s\t%-12.12s %-12.12s %s\t%d\n",
$1, $2, $3, $4, $5 } ' pactrep
# remove all the temporary files
rm pnum pnn pactrep
```

Hands-on Project 6-6 enables you to build a short script to demonstrate sequential logic as well as practice the let command, and to build expressions using constants, variables, and arithmetic operators.

DECISION LOGIC

Decision logic enables your script or program to execute a statement or series of statements only if a certain condition exists. In this usage, a statement is another name for a line of code that performs an action in your program. The if statement is a primary decision-making logic structure in this type of logic.

In decision logic, the script is programmed to make decisions as it runs. By using the if statement,you can specify conditions for the script to evaluate before it makes a decision.For example, if a occurs, then the script does b; but if x occurs instead, then the script does y.

Consider a situation in which a magazine publisher gives the reader two subscription choices based on price. If the reader sends in $25, then the magazine publisher gives him 12 weeks of the publication, but if the reader sends in $50 dollars, then the magazine publisher gives him 24 weeks of the publication.

Another way to use a decision structure is as a simple yes or no situation. For example, you might use a portion of a script to enable the user to continue updating a file or to close the file.When the script asks:“Do you want to continue updating (y or n)?”, if you answer y then the script gives you a blank screen form in which to enter more information to put in the file. If instead you answer n, then the script closes the file and stops.

Consider, for example, the following lines of code. In this shell script, the user is asked to enter a favorite vegetable. If the user enters “broccoli,” the decision logic of the program displays “Broccoli is a healthy choice.” If any other vegetable is entered, the decision logic displays the line “Don’t forget to eat your broccoli also.”

```
echo -n "What is your favorite vegetable? "
read veg_name
if [ "$veg_name" = "broccoli" ]
then
echo "Broccoli is a healthy choice."
else
echo "Don’t forget to eat your broccoli also."
fi
```

Throughout the sample scripts, variables are always enclosed in double quotation marks, as in “$veg_name”, “$choice”, “$looptest”, “$yesno”, “$guess”, and “$myfavorite”, because of how the shell interprets variables. All shell variables, unless declared otherwise, are strings, which are arrays of alphanumeric characters. If you do not enter data in the string variables, the variables are treated as blank strings, which result in an invalid test. The enclosing double quotation marks, therefore, maintain the validity of strings, with or without data, and the test is carried out without producing an error condition.

You create and run this script in Hands-on Project 6-7. However, before you attempt the project, let’s examine the contents of the script. The first statement uses the echo command to display a message on the screen. The -n option suppresses the line feed that normally appears after the message. The second statement uses the read command,which waits for the user to type a line of keyboard input. The input is stored in the variable specified as the read command’s argument. The line in the script reads the user’s input into the veg_name variable.

The next line begins an if statement. The word “if” is followed by an expression inside a set of brackets ([ ]). (The spaces on either side that separate the [and] characters from the enclosed expression are necessary.) The expression, which is tested to determine if it is true or false, compares the contents of the veg_name variable with the string broccoli. (When you use the = operator in an if statement's test expression, it tests its two operands for equality. In this case, the operands are the variable $veg_name and the string broccoli. If the operands are equal, the expression is true—otherwise, it is false. If the contents of the $veg_name variable are equal to broccoli, the statement that follows the word "then" is executed. In this script, it is the echo statement,"Broccoli is a healthy choice."

If the if statement's expression is false (if the contents of the $veg_name variable do not equal broccoli), the statement that follows the word "else" is executed. That statement reads, "Don't forget to eat your broccoli also." In this script, it is a different echo statement.

Notice the last statement, which consists of the characters "fi." fi ("if" spelled backward) always marks the end of an if or an if...else statement.

When you evaluate the contents of a variable using a logic structure, such as the if statement, you need to define the variable first, such as through a read statement.

You can nest a control structure, such as an if statement, inside another control structure. To nest means that you layer statements at two or more levels under an original statement

structure. For example, a script can have an if statement inside another if statement. The frst if statement controls when the second if statement is executed, as in the following code sample:

```
echo -n "What is your favorite vegetable? "
read veg_name
if [ "$veg_name" = "broccoli" ]
then
echo "Broccoli is a healthy choice."
else
if [ "$veg_name" = "carrots" ]
then
echo "Carrots are great for you."
else
echo "Don't forget to eat your broccoli also."
fi
fi
```

As you can see, the second if statement is located in the first if statement's else section. It is only executed when the first if statement's expression is false.

Decision logic structures, such as the if statement, are used in applications in which different courses of action are required, depending on the result of a command or comparison.

Looping Logic

In looping logic, a control structure (or loop) repeats until a specific condition exists or some action occurs. The basic idea of looping logic is to keep repeating an action until some condition is met. For example, the logic might keep printing a list of people's names until it reaches the final name on the list, prints the final name, and stops. In another example, a script might open an inventory file of mountain bike models, print the model and quantity for the first bike, do the same for the second bike, and so on until there are no more bikes listed in the file.

You learn two looping mechanisms in the sections that follow: the for and the while loop.

The For Loop

Use the for command to loop through a range of values. It causes a variable to take on each value in a specified set, one at a time, and perform some action while the variable contains each individual value. The loop stops after the variable has taken on the last value in the set and has performed the specified action with that value.

An example of a for loop is as follows:

```
for USERS in john ellen tom becky eli jill
do
echo $USERS
done
```

In this for loop structure, the first line specifies the values that will be assigned, one at a time, to the USERS shell variable.Because six values are in the set,the loop repeats six times.Each

time it repeats, USERS contains a different value from the set, and the statement between the do and done statements is executed. The first time around the loop,"john" is assigned to the USERS variable and is then displayed on the screen via the echo $USERS command. Next,"ellen" is assigned to the USERS variable and displayed. The loop continues through "tom,""becky,""eli,"and"jill."After"jill"is assigned to USERS and displayed on the screen, the looping comes to an end and the done command is executed to end the looping logic.

Hands-on Project 6-8 enables you to use a for loop in a shell script.

Executing Control Structures at the Command Line

Most shell script control structures, such as the if and for statements, must be written across several lines. This does not prevent you from executing them directly on the command line, however. For example, you can enter the for statement at the command prompt, enter the variable name and elements to use for the variable, and press Enter. You go into a command-line processor to execute the remaining statements in the loop, pressing Enter after each statement. The shell knows more code comes after you type the first line. It displays the > prompt, indicating it is ready for the control structure's continuation. The shell reads further input lines until you type the word "done," which marks the end of the for loop.

In the following lines, each of the elements tennis, swimming, movies, and travel are displayed one line at a time until the loop ends after displaying travel. (See Figure 6-4.)

```
$ for myhobbies in tennis swimming movies travel <Enter>
> do <Enter>
> echo $myhobbies <Enter>
> done <Enter>
```

Hands-on Project 6-8 enables you to compare using the command line to using a shell script for executing a simple for loop.

Using Wildcard Characters in a Loop

The [ ] wildcard characters can be very useful in looping logic. For example, consider that you have four files that all start with the same four characters: chap1, chap2, chap3, and chap4. You can use the wildcard notation chap[1234] to output the contents of all four file names to the screen using the following statement:

```
for file in chap[1234]; do
more $file
done
```

Notice that in the first line, two commands are combined by using the semicolon (;) character to run each on one line. Try Hands-on Project 6-9 to use brackets as wildcards.

The While Loop

A second approach to looping logic is the while statement. The while statement continues to loop and execute commands or statements as long as a given condition or set of conditions is true. As soon as the condition or conditions are false, the loop is exited at the done statement. The following is an example of a simple shell script that uses a while statement:

```
echo -n "Try to guess my favorite color: "
read guess
while [ "$guess" != "red" ]; do
echo "No, not that one. Try again. "; read guess
done
```

In this example, the first line asks the user to “Try to guess my favorite color: ” and the user’s response is read into the variable guess.

The while loop tests an expression in a manner similar to the if statement. As long as the statement inside the brackets is true, the statements inside the do and done statements repeat. In this example, the expression inside the brackets is “$guess” != “red”, which tests to see if the guess variable is not equal to the string,“red”. Note that != is the not-equal (inequality) operator. (Refer to Table 6-3.) The while statement tests the two operands on either side of the != operator and returns true if they are not equal. Otherwise, it returns false. In this example, the echo and read statements inside the loop repeat until the user enters red, which makes the expression “$guess” != “red” false.

Figure 6-4 Using the for loop from the command line

Shell Logic Structures 297

6

Hands-on Project 6-10 gives you the opportunity to program the sample code described here and then to program a more complex example such as might be used to input name and address information in a file.

Use looping logic in the form of for and while statements in applications in which code must be repeated a determined or undetermined number of times.

CASE LOGIC

The case logic structure simplifies the selection of a match when you have a list of choices. It allows your script to perform one of many actions, depending on the value of a variable. Consider a fund raiser in which contributors receive different premiums. If someone gives $20 they get a free pen. If they give $30 they get aT-shirt and if they give $50 they get a free mug. People who give $100 get a free CD. In case logic, you can take the four types of contributions and associate each one with a different action. For example, when the user types in 30 in answer to the question,“How much did you contribute?”, then the screen displays “Your free gift is a T-shirt.” If the user types in 100, then the screen displays,“Your free gift is a CD.”The advantage of case logic in this example is that it simplifies your work by using fewer lines—you use one case statement instead of several if statements,for example.

One common application of the case logic structure is in creating menu selections on a computer screen. A menu is a screen display that offers several choices. Consider, for example, a menu used in a human resources application in which there is a menu option to enter information for a new employee, another menu option to view employee name and address information,another menu option to view salary information,and so on.Each menu option branches to a different program. For example, if you select the option to enter information for a new employee, this starts the employee data entry program. Or, if you select to view employee salary information, a salary report program runs.

The following is a basic example of how the case statement works in case logic:

```
echo –n "Enter your favorite color: "; read color
case "$color" in
"blue") echo "As in My Blue Heaven.";;
"yellow") echo "As in the Yellow Sunset.";;
"red") echo "As in Red Rover, Red Rover.";;
"orange") echo "As in Autumn has shades of Orange.";;
*) echo "Sorry, I do not know that color.";;
esac
```

In this sample script, the case structure examines the contents of the color variable, and searches for a match among the values listed. When a match is found, the statement that immediately follows the case value is executed. For example, if the color variable contains orange,the echo statement that appears after“orange”) is executed: “As inAutumn has shades

of Orange.” If the contents of the color variable do not match any of the values listed, the statement that appears after *) is executed: “Sorry, I do not know that color.”

Note the use of two semicolons (;;) that terminate the action(s) taken after the case structure matches what is being tested. Also notice that the case structure is terminated by the word “esac,” which is “case” spelled backward.

As you can see,case logic is designed to pick one course of action from a list of many,depending on the contents of a variable. This is why case logic is ideal for menus,in which the user chooses one of several values. Try Hands-on Project 6-11 to program using case logic.

USING SHELL SCRIPTING TO CREATE A MENU

When you create an application that consists of several shell scripts,it is often useful to create a menu with options that branch to specific scripts. You create menus in the Hands-on Projects section of this chapter. In preparation for creating a menu, you need to have one more command under your belt, the tput command. This is one of the less-publicized

UNIX/Linux commands, but is important to know for developing an appealing and

user-friendly menu presentation.

SYNTAX TPUT [-OPTIONS] ARGUMENTS

Dissection

■ Can be used to initialize the terminal or terminal window display, position text, and

position the cursor

■ Useful options include:

bold=`tput smso` offbold=`tput rmso` enables/disables boldfaced type

clear clears the screen

cols prints the number of columns

cup positions the cursor and text on the screen

The tput command enables you to initialize the terminal display or terminal window, to

place text and prompts in desired locations, and to respond to what the user selects from the

menu. Some examples of what you can do with the tput command are:

■ tput cup 0 0 moves the cursor to row 0, column 0, which is the upper-left corner

of the screen.

■ tput clear clears the screen.

■ tput cols prints the number of columns for the current terminal display.

■ bold=`tput smso` offbold=`tput rmso` sets boldfaced type by setting the bold shell

variable for stand-out mode sequence and by setting the offbold shell variable to

turn off stand-out mode sequence.

Hands-on Project 6-12 enables you to gain experience using the tput command. Hands-on

Project 6-15 uses the tput command so you can begin building a menu to use with an actual

application.

DEBUGGING A SHELL SCRIPT

As you have probably discovered by this point, sometimes a shell script does not execute

because there is an error in one or more commands within the script. For example, perhaps

you entered a colon instead of a semicolon or left out a bracket. Another common problem

is leaving out a space or not putting a space in the correct location.

Now that you have some experience developing shell scripts and have possibly encountered

some problems, you can appreciate why it is important to have a tool to help you troubleshoot errors. The sh command that is used to run a shell script includes several

options for debugging.

SYNTAX SH [-OPTIONS] [SHELL SCRIPT]

Dissection

■ In UNIX and Linux,it calls the command interpreter for shell scripts;and in Linux,it uses

the Bash shell with the command interpreter

■ Useful options include:

-n checks the syntax of a shell script, but does not execute command lines

-v displays the lines of code while executing a shell script

-x displays the command and arguments as a shell script is run

Two of the most commonly used sh options are -v and -x. The -v option displays the lines

of code in the script as they are read by the interpreter. The -x option shows somewhat

different information by displaying the command and accompanying arguments line by line

as they are run.

By using the sh command with these options, you can view the shell script line by line as it

is running and determine the location and nature of an error on a line when a script fails.

Try Hands-on Project 6-13 to compare sh -v and sh -x to debug a shell script.

Further, sometimes you want to test a script that updates a file, but you want to give the

script a dry run without actually updating the file—particularly so that data in the file is not

altered if the script fails at some point.Use the -n option for this purpose because it reads and

checks the syntax of commands in a script, but does not execute them. For example, if you

are testing a script that is designed to add new information to a file, when you run it with

the sh -n command, the script does not actually process the information or add it to the file.

If a syntax error exists, you see an error message so you know that you must fix the script

before using it on live data.

Now that you have an idea of how to create a menu script, it is helpful to learn some

additional shell features and commands before creating your application. You first learn

more about how to customize your personal environment and how to use the trap command

to clean up unnecessary files in your environment.

CUSTOMIZING YOUR PERSONAL ENVIRONMENT

When your work requirements center on computer programming and shell scripting,

consider customizing your environment by modifying the initial settings in the login scripts.

A login script is a script that runs just after you log in to your account. For example, many programmers set up a personal bin directory in which they can store and test their new programs without interfering with ongoing operations.The traditional UNIX/Linux name for directories that hold executable files is bin.

A useful tool for customizing the command environment is the alias. An alias is a name that represents another command. You can use aliases to simplify and automate commands you use frequently. For example, the following command sets up an alias for the rm command.

```
alias rm="rm -i"
```

This command causes the rm -i command to execute any time the user enters the rm command. This is a commonly used alias because it ensures that users are always prompted before the rm command deletes a file. The following are two other common aliases that help safeguard files:

```
alias mv="mv -i"
alias cp="cp -i"
```

SYNTAX ALIAS [-OPTIONS] [NAME =“COMMAND”]

Dissection

■ Creates an alternate name for a command

■ Useful options include:

-p prints a list of all aliases

alias is a built-in Bash shell command.You can learn more about alias by entering man bash and find alias under the SHELL BUILTIN COMMANDS section. Hands-on Project 6-14 enables you to set aliases.

The .bashrc file that resides in your home directory as a hidden file (enter ls -a to view hidden files) can be used to establish customizations that take effect for each login session. The .bashrc script is executed each time you generate a shell, such as when you run a shell script. Any time a subshell is created, .bashrc is reexecuted. The following .bashrc file is commented to explain how you can make your own changes.

```
# .bashrc
# Source global definitions
if [ -f /etc/bashrc ]; then
. /etc/bashrc # if any global definitions are defined
# run them first
alias rm='rm -i' # make sure user is prompted before
```

# removing files

alias mv='mv -i' # make sure user is prompted before

# overlaying files

set -o ignoreeof # Do not allow Ctrl-d to log out

set -o noclobber # Force user to enter >| to write

# over existing files

PS1="\w \$" # Set prompt to show working directory

In addition to knowing how to customize your work environment, you should also be familiar with the trap command to clean your storage of temporary files.

THE TRAP COMMAND

trap is a command in the Bash shell that is used to execute another command when a specific signal is received. For example, you might use trap to start a new program after it detects through an operating system signal that a different program has terminated. Another example is using trap to end a program when trap receives a specific signal, such as trapping when the user types Ctrl-c and gracefully ending the currently running program.

The trap command is useful when you want your shell program to automatically remove any temporary files that are created when the shell script runs. The trap command specifies that a command, listed as the argument to trap, is read and executed when the shell receives a specified system signal.

SYNTAX TRAP [COMMAND] [SIGNAL NUMBER]

Dissection

■ When a signal is received from the operating system, the argument included with trap is executed.

■ Common signals used with trap include:

0 The completion of a shell script has occurred

1 A hang up or logout signal has been issued

2 An interrupt has been received, such as Ctrl+c

302 Chapter 6 Introduction to Shell Script Programming

3 A quit signal has been issued

4 An illegal instruction has been received

9 A termination signal has been issued

15 A program has been ended, such as through a kill command

19 A process has been stopped

20 A process has been suspended

■ Useful options include:

-l displays a listing of signal numbers and their associated signal designations

Here is an example of a use for the trap command:

```
trap "rm ~/tmp/* 2> /dev/null; exit" 0
```

This command has two arguments:a command to be executed and a signal number from the operating system. The command rm ~/tmp/* 2> /dev/null; exit deletes everything in the user’s tmp directory, redirects the error output of the rm command to the null device (so it does not appear on the screen),and issues an exit command to terminate the shell. The signal specified is 0,which is the operating system signal generated when a shell script is exited.So, if this sample command is part of a script file, it causes the specified rm command to execute when signal 0 is sent by the operating system.

The programmer often sets up ~/tmp (a subdirectory of the user’s home directory) to store temporary files. When the script file exits, any files placed in ~/tmp can be removed. This is called “good housekeeping” on the part of the programmer.

The trap command is another example of a built-in Bash shell command. You can learn more about trap by entering man bash and finding the trap command listed in the SHELL BUILTIN COMMANDS section.

PUTTING IT ALL TOGETHER IN AN APPLICATION

In this chapter,you learned all of the pieces necessary to create a multifunctional application.

You learned how to:

■ Assign and manage variables

■ Use shell operators

■ Employ shell logic structures

■ Use additional wildcard characters

■ Use tput for managing screen initialization and screen text placement

■ Use the trap command to clean up temporary files used by an application

In Hands-on Projects 6-15 through 6-20, you use the skills and knowledge you have acquired in this and previous chapters to build a multipurpose application. The application

you build simulates one that an organization might use to track telephone numbers and other information about its employees. This application enables the user to input new telephone number and employee information, to print a list of telephone numbers, and to search for a specific telephone number. You build the application entirely from shell scripts. Also,to make this undertaking manageable (and be consistent with programming practices), you build the application through creating small pieces that you prototype,test,and later link together into one menu-based application.

CHAPTER SUMMARY

A high-level language (such as C, C++, or COBOL) is a language that uses English-like expressions. A high-level language must be converted into a low-level (machine) language before the computer can execute it. Programmers use a compiler to convert the high-level language to machine language.

An interpreter reads commands or a programming language and interprets each line into an action. A shell,such as the Bash shell,interprets UNIX/Linux shell scripts.Shell scripts do not need to be converted to machine language because the UNIX/Linux shell interprets the lines in shell scripts.

UNIX/Linux shell scripts, created with the vi or Emacs editor, contain instructions that do not need to be written from scratch, but can be selectively chosen from the operating system's inventory of executable commands.

Linux shells are derived from the UNIX Bourne, Korn, and C shells. The three typical Linux shells are Bash,csh/tcsh,and ksh/zsh;Bash is the most commonly used Linux shell.

UNIX/Linux employ three types of variables: configuration, environment, and shell. Configuration variables contain setup information for the operating system.Environment variables hold information about your login session. Shell variables are created in a shell script or at the command line. The export command is used to make a shell variable an environment variable.

The shell supports many operators, including ones that perform arithmetic operations.

You can use wildcard characters in shell scripts, including the bracket ([ ]) characters. Brackets surround a set of values that can match an individual character in a name or string.

The logic structures supported by the shell are sequential, decision, looping, and case.

You can use the tput command to manage cursor placement.

You can customize the .bashrc file that resides in your home directory to suit particular needs, such as setting alias and default shell conditions.

You can create aliases and enter them into .bashrc to simplify commonly used commands, such as ls -l and rm -i.

Use the trap command inside a script file to remove temporary files after the script file has been run (exited).

COMMAND SUMMARY: REVIEW OF CHAPTER 6 COMMANDS

COMMAND	PURPOSE	OPTIONS COVERED IN THIS CHAPTER
alias	Establishes an alias	-p prints all aliases.
case. . .in. . .esac	Allows one action from a set of possible actions to be performed, depending on the value of a variable	
export	Makes a shell variable an environment variable	-n can be used to undo the export. -p lists the exported variables
for: do. . .done	Causes a variable to take on each value in a set of values; an action is performed for each value	
if. . .then. . . else. . .fi	Causes one of two actions to be performed, depending on the condition	
let	Stores arithmetic values in a variable	
printenv	Prints a list of environment variables	
set	Displays currently set shell variables; when options are used, sets the shell environment	-a exports all shell variables after they are assigned. -n takes commands without executing them, so you can debug errors. -o sets a particular shell mode—when used with noclobber as the argument, it prevents files from being overwritten by use of the > operator. -u yields an error message when

		there is an attempt to use an undefined variable. -v displays command lines as they are executed.
sh	Calls the command interpreter for shell scripts	-n checks the syntax of a shell script, but does not execute command lines. -v displays the lines of code while executing a shell script. -x displays the command and arguments as a shell script is run.
tput cup	Moves the screen cursor to a specified row and column	
tput clear	Clears the screen	
tput cols	Prints the number of columns on the current termina	
tput smso	Enables boldfaced output	
tput rmso	Disables boldfaced output	
trap	Executes a command when a specified signal is received from the operating system	-l displays a listing of signal numbers and their signal designations.
while: do. . .done	Repeats an action while a condition exists	

KEY TERMS

.bashrc file — A file in your home directory that you can use to customize your work environment and specify what occurs each time you log in. Each time you start a shell, that shell executes the commands in .bashrc.

algorithm — A sequence of instructions, programming code, or commands that results in a program or that can be used as part of a program.

alias — A name that represents a command. Aliases are helpful in simplifying and automating frequently used commands.

arithmetic operator — A character that represents a mathematical activity. Arithmetic operators include + (addition), - (subtraction), * (multiplication), and / (division).

branch instruction — An instruction that tells a program to go to a different section of code.

case logic — One of the four basic shell logic structures employed in program development. Using case logic, a program can perform one of many actions, depending on the value of a variable and matching results to a test. It is often used when there is a list of several choices.

compiler — A program that reads the lines of code in a source file, converts them to machine-language instructions or calls the assembler to convert them into object code, and creates a machine-language file.

configuration variable — A variable that stores information about the operating system and does not change the value.

control structures — See logic structures.

debugging — The process of going through program code to locate errors and then fixing them.

decision logic — One of the four basic shell logic structures used in program development. In decision logic, commands execute only if a certain condition exists. The if statement is an example of a coded statement that sets the condition(s) for execution.

defining operator — Used to assign a value to a variable.

environment variable — A value in a storage area that is read by UNIX/Linux when you log in.Environment variables can be used to create and store default settings,such as the shell that you use or the command prompt format you prefer.

evaluating operator — Enables you to evaluate the contents of a variable, such as by displaying the contents.

glob —A character used to find or match file names; similar to a wildcard. Glob characters are part of glob patterns.

glob pattern — A combination of glob characters used to find or match multiple file names.

high-level language — A computer language that uses English-like expressions. COBOL, Visual Basic (VB), C, and C++ are high-level languages.

interpreter — A UNIX/Linux shell feature that reads statements in a program file, immediately translates them into executable instructions, and then runs the instructions. Unlike a compiler, an interpreter does not produce a binary (an executable file) because it translates the instructions and runs them in a single step.

logic structures — The techniques for structuring program code that affect the order in which the code is executed or how it is executed, such as looping back through the code from a particular point or jumping from one point in the code to another. Also called control structures or control logic.

login script — A script that runs just after you log in to your account.

looping logic — One of the four basic shell logic structures used in program development.

In looping logic, a control structure (or loop) repeats until some specific condition exists or some action occurs.

nest —When creating program code,a practice of layering statements at two or more levels under an original statement structure.

operand — The variable name that appears to the left of an operator or the variable value that appears to the right of an operator. For example, in NAME=Becky, NAME is the variable name,= is the operator,and Becky is the variable value.Note that no spaces separate the operator and operands.

PATH variable — A path identifier that provides a list of directory locations where UNIX/Linux look for executable programs.

program development cycle —The process of developing a program,which includes (1) creating program specifications, (2) the design process, (3) writing code, (4) testing, (5) debugging, and (6) correcting errors.

prototype — A running model, which lets programmers review a program before committing to its design.

redirection operator — An operator or symbol that changes the input or output data stream from its default direction, such as using > to redirect output to a file instead of to the screen.

relational operator — Compares the relationship between two values or arguments, such as greater than (>), less than (<), equal to (=), and others.

sequential logic — One of four basic logic structures used for program development. In sequential logic, commands execute in the order they appear in the program, except when a branch instruction changes the flow of execution.

shell script operator — The symbols used with shell scripts that define and evaluate information, that perform arithmetic actions, and that perform redirection or piping operations.

shell variable — A variable you create at the command line or in a shell script.It is valuable for use in shell scripts for storing information temporarily.

source file — A file used for storing a program’s high-level language statements (code) and created by an editor such as vi or Emacs,To execute, a source file must be converted to a low-level machine language file consisting of object code.

statement — A reference to a line of code that performs an action in a program.

string — A nonnumeric field of information treated simply as a group of characters.

Numbers in a string are considered characters rather than digits.

symbolic name — A name used for a variable that consists of letters, numbers, or characters, that is used to reference the contents of a variable, and that often reflects the variable’s purpose or contents.

syntax error — A grammatical mistake in a source file or script. Such mistakes prevent a compiler or interpreter from converting the file into an executable file or from running the commands in the file.