variables, that establish it as a powerful programming language in its own right. These features are used to design **shell scripts**—programs that run UNIX commands in a batch.

Many of the system's functions can be controlled and automated by using these shell scripts. If you intend taking up system administration as a career, then you'll have to know the shell's programming features very well. Proficient UNIX programmers seldom refer to any other language (except **perl**) for text manipulation problems. Shell programming is taken up in Chapter 13.

1.11.7 Documentation

UNIX documentation is no longer the sore point it once was. Even though it's sometimes uneven, usually the treatment is quite lucid. The principal online help facility available is the **man** command, which remains the most important reference for commands and their configuration files. Today there's no feature of UNIX on which a separate textbook is not available. UNIX documentation and the man facility are discussed in Chapter 2.

Apart from the online documentation, there's a vast ocean of UNIX resources available on the Internet. There are several newsgroups on UNIX where you can post your queries in case you are stranded with a problem. The FAQ (Frequently Asked Questions)—a document that addresses common problems—is also widely available on the Net. Then there are numerous articles published in magazines and journals and lecture notes made available by universities on their Web sites.

With the goal of building a comfortable relationship with the machine, Thomson and Ritchie designed a system for their own use rather than for others. They could afford to do this because UNIX wasn't initially developed as a commercial product, and the project didn't have any predefined objective. They acknowledge this fact too: "We have not been faced with the need to satisfy someone else's requirements, and for this freedom we are grateful."

SUMMARY

A computer needs an *operating system* (OS) to allocate memory, schedule programs, and control devices. The UNIX system also provides a host of applications for the use of programmers and users.

Multiprogramming systems like UNIX allow multiple programs to reside in memory. Even though a program may run for the duration of the time slice allocated for it, it may prematurely leave the CPU during a *blocking* operation (like reading a file) that keeps the CPU idle.

You enter a UNIX system by entering a user-id and a password. You can terminate a session by using the **exit** or **logout** command or pressing *[Ctrl-d]*.

UNIX commands are generally in lowercase. **date** displays the system date and time. **who** displays the list of users logged on to the system. **ps** lists all processes running at a terminal. It always shows the shell process running.

You can display a file with **cat**, copy it with **cp**, rename it with **mv**, and remove it with **rm**.

mkdir creates a directory, **pwd** displays the pathname of the current directory, and **cd** changes the current directory. **rmdir** removes an empty directory.

UNIX was developed at AT&T Bell Laboratories by Ken Thompson and Dennis Ritchie. It was finally written in C. Notable work was also done at Berkeley. AT&T introduced System V Release 4 (SVR4) to merge their own version, Berkeley, and other variants.

Linux is a UNIX implementation that is constantly growing with contributions from the Free Software Foundation (formerly, GNU).

Modifications to the system made by vendors led to both enhancement and fragmentation of UNIX. Two merged standards, *POSIX* and the *Single UNIX Specification,* are today used as guidance for development work on UNIX.

All work is shared by the *kernel* and *shell.* The kernel manages the hardware, and the shell interacts with the user. The shell and applications communicate with the kernel using *system calls,* which are special routines built into the kernel.

The *file* and *process* are the two basic entities that support the UNIX system. UNIX considers everything as a file. A process represents a program (a file) in execution.

UNIX is a *multiuser* and *multitasking* system. Several users can use the system together, and a single user can also run multiple jobs concurrently.

UNIX uses a building-block approach in the design of some of its tools and lets you develop complex command routines by connecting these tools.

The UNIX **man** command is the primary online help facility available.

SELF-TEST

1.1 The _____ interacts with the hardware, and the _____ interacts with the user.
1.2 A *program* is synonymous with a *process*. True or false?
1.3 Every character has a number associated with it. What is it called?
1.4 If you see a prompt like `mailhost login:`, what do you think `mailhost` represents?
1.5 If the system echoes `Login incorrect`, does it mean that your user-id is incorrect?
1.6 Name the commands you used in this chapter to display (i) filenames, (ii) processes, (iii) users.
1.7 Run **ps** and note the PID of your shell. Log out and log in again, and run **ps** again. What do you observe?
1.8 Create two files, `fool` and `foo2`, with the **echo** command, and then use **cat fool foo2**. What do you observe?
1.9 Now run the command **cat foo[12]**, and note your observations.
1.10 Enter the command **echo SHELL**. What mistake did you make?
1.11 Create a file `foo` containing the words `hello dolly`. Now create a directory `bar`, and then run **mv foo bar**. What do you observe when you run both **ls** and **ls bar**?
1.12 Who are the principal architects of the UNIX operating system?
1.13 Why did AT&T virtually give away UNIX to the world?
1.14 Where did BSD UNIX originate? Name some features of UNIX that were first found in BSD UNIX.
1.15 Which flavor of UNIX is available for free and runs on the PC?
1.16 Identify the companies associated with the following brands: (i) Solaris, (ii) AIX, (iii) Tru64 UNIX.
1.17 What does X/OPEN represent? Who owns the UNIX trademark today?

1.18 Who are the two brains behind Linux?
1.19 What is the distinctive characteristic about the GNU General Public License?
1.20 Why is UNIX more portable than other operating systems?
1.21 Can you divide UNIX into two major schools? To which school does Sun's UNIX belong?
1.22 Why do UNIX tools perform simple jobs rather than complex ones?
1.23 What is the windowing system of UNIX known as?
1.24 Name some interpretive languages available on UNIX systems.
1.25 Name three notable Linux flavors.

EXERCISES

1.1 Operating systems like UNIX provide services both for programs and users. Explain.
1.2 What does a program do when it needs to read a file?
1.3 Does a program always complete its time quantum before it makes way for another program?
1.4 Explain the significance of the terms *multiprogramming, multiuser,* and *multitasking*.
1.5 Why are UNIX commands noninteractive, and why is their output not usually preceded by header information?
1.6 What are *system calls,* and what role do they play in the system? How is C programming so different and powerful in the UNIX environment compared to Windows?
1.7 Two UNIX systems may use the same system calls. True or false?
1.8 Name the three commands that you would try in sequence to log yourself out of the system. Which one of them will always work?
1.9 Run the following commands, and then invoke **ls.** What do you conclude?

```
echo > README[Enter]
echo > readme[Enter]
```

1.10 Enter the following commands, and note your observations: (i) **who** and **tty**, (ii) **tput clear**, (iii) **id**, (iv) **ps** and **echo $$**.
1.11 When you log in, a program starts executing at your terminal. What is this program known as? Name four types of this program that are available on a system.
1.12 What is the significance of your user-id? Where in the system is the name used?
1.13 What are the two schools of UNIX that initially guided its development? Mention the outcome of the standardization efforts that are currently in force today.
1.14 Create a directory, and change to that directory. Next, create another directory in the new directory, and then change to that directory too. Now, run **cd** without any arguments followed by **pwd**. What do you conclude?
1.15 Why is the shell called a *command interpreter*?
1.16 What is the one thing that is common to directories, devices, terminals, and printers?

CHAPTER 2

Becoming Familiar with UNIX Commands

A major part of the job of learning UNIX is to master the essential command set. UNIX has a vast repertoire of commands that can solve many tasks either by working singly or in combination. In this chapter, we'll examine the generalized UNIX command syntax and come to understand the significance of its options and arguments. The complete picture of command usage is available in the man pages, and we'll learn to look up this documentation with the **man** command.

We'll next try out some of the general-purpose utilities of the system. We'll change the password and get comfortable with email using a command-line tool. We'll learn about other tools that tell us the date, the users of the system, and some specifics of the operating system. At times we need to consider situations where the output of these commands can be processed further. Finally, we take a look at the common traps that befall the user and how the **stty** command can change many keyboard settings.

Objectives

- Understand the breakup of the *command line* into *arguments* and *options*.
- Learn how the shell uses the PATH variable to locate commands.
- Learn how commands can be used singly or in combination.
- Use the **man** command to browse the UNIX documentation.
- Understand the organization of the documentation.
- Display messages with **echo**, and understand why **printf** is superior.
- Save all keystrokes and command output in a file with **script**.
- Understand email basics and why you need a command-line email program like **mailx**.
- Use **passwd** to change your own password.
- Know your machine's name and operating system with **uname**.
- Find out the users of the system with **who**.
- Display the system date in various formats with **date**.
- Know what can go wrong, and use **stty** to change keyboard settings.
- Get introduced to the X Window system.

2.1 Command Basics

UNIX commands are generally implemented as disk files representing executable programs. They are mainly written in C, but UNIX supports programs written in any

You can have several terminal emulators (apart from several programs) on your desktop, and you can invoke a separate application in each one of them. You can also switch from one application to another without quitting any of them.


Tip

If you have difficulty in copy-paste operations using the technique described above, you can use the window menu which also offers options to do the same work. Often, the keys are the same ones used in Microsoft Windows—*[Ctrl-c]* for copying and *[Ctrl-v]* for pasting.

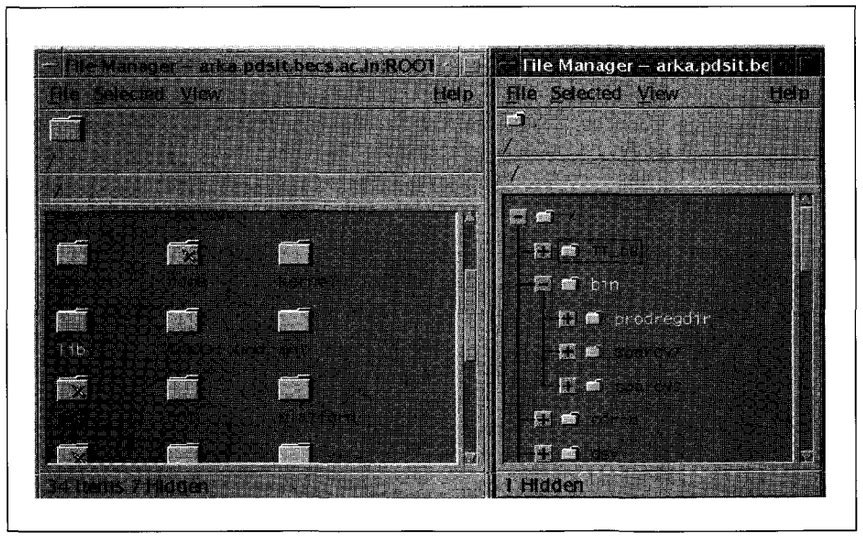
2.15.2 The File Manager

We use files all the time, copying, moving, and deleting them several times a day. Every X implementation offers a file management program that can perform these tasks. A file manager can also be used to view file contents and execute programs. Windows offers a similar application—Windows Explorer. The file manager on the CDE is **dtfile**, which is shown in Fig. 2.5. Linux users may use Konqueror instead. However, your system may contain other file managers.

Using menu options, you can create and remove directories. Try creating some. To copy or move files from one directory to another, you need to work with two windows of the same program. Look up the menu option that splits a window. Every file is represented by an icon, and you select it by clicking it with the mouse. You can also select multiple files by pressing *[Ctrl]* and then clicking on each icon. To select all files, use the option offered by the menu; it often is *[Ctrl-a]*. You can now drag the files by keeping the left mouse button pressed and drop them to their new location by releasing the button. Files can thus be copied and moved in this way.

In Chapter 3, you'll use the **mkdir**, **rmdir**, **cp**, and **mv** commands for file and directory handling. You'll see how effortlessly you can work with groups of files and

FIGURE 2.5 *Two Views of the* **dtfile** *File Manager*



directories using these commands. However, it's good to know the X techniques now because that will help you appreciate the power of the UNIX command line interface later. The limitations of the X method of doing things will soon become apparent.

SUMMARY

UNIX commands are case-sensitive but are generally in lowercase. They need not have any specific extensions. Commands for general use are located in the directories `/bin` and `/usr/bin`. The shell variable, `PATH`, specifies the search list of directories for locating commands.

The shell treats a command either as *external* when it exists on disk or *internal* when it is built into the shell. Commands like **man** and **mailx** also have their own internal commands.

The *command line* comprises the command, and its *options* and *arguments*. Commands and arguments are separated by *whitespace*. Multiple commands can be delimited with a **;**, and a command sequence can be split into multiple lines.

Use the **man** command to look up the documentation for a command, a configuration file, or a system call. Most commands are found in Section 1. You'll find system calls and library functions in Sections 2 and 3.

echo displays a message on the screen. It supports *escape sequences* (like `\c` and `\007`). The command has portability problems, the reason why **printf** should be used. **printf** also supports format specifiers (like `%d`).

script logs all user activities in a separate file named `typescript`.

A mail message is saved in a text file called *mailbox*. Mail is moved to the *mbox* after it is viewed. **mailx**, a command-line mail program, can be used interactively and also noninteractively from shell scripts.

date displays any component of the system date and time in a number of formats. **passwd** changes a user's password. The system administrator can change the system date and the password of any user.

uname reveals details of your machine's operating system (`-r` and `-s`). It also displays the hostname (`-n`) that is used by networking commands.

who displays the users working on the system.

stty displays and sets various terminal attributes. It defines the key that interrupts a program (`intr`), suspends a job (`susp`), and marks the end-of-file (`eof`). **stty sane** sets the terminal to some standard values.

The X Window System provides a Graphical User Interface (GUI) for users to run programs that involve graphics. X also provides several applications including a *terminal emulator* and a file management program.

SELF-TEST

2.1 Enter a : and press *[Enter]*. Next run **type :**. What do you conclude?

2.2 UNIX commands must be in lowercase and must not have extensions. True or false?

2.3 Name three UNIX commands whose names are more than five characters long.
2.4 Find out whether these commands are internal or external: `echo`, `date`, `pwd`, and `ls`.
2.5 If two commands with the same filename exist in two directories in `PATH`, how can they be executed?
2.6 How is the current directory indicated in `PATH`?
2.7 How many options are there in this command? `ls -lut chap01 note3`
2.8 If you find yourself using options preceded by two hyphens (like `--all`), which flavor of UNIX could you be using?
2.9 What is the name given to the command, and its options and arguments?
2.10 How do you find out the version number of your operating system?
2.11 Why are the directories `/bin` and `/usr/bin` usually found first in `PATH`?
2.12 What is *whitespace*? Explain the treatment the shell metes out to a command that contains a lot of whitespace.
2.13 Do you need to wait for a command to finish before entering the next one?
2.14 Why doesn't this command run in the way it is meant to?
`printf "Filename: %s\n", fname`
2.15 What is a *pager*? Name the two standard pagers used by **man**.
2.16 You located the string `crontab` in a man page by searching with **/crontab***[Enter]*. How do you find out the other occurrences of this string in the page?
2.17 You don't know the name of the command that could do a job. What do you do?
2.18 How do you find out the users who are idling from the man documentation of **who**?
2.19 What is the difference between the *mailbox* and *mbox*?
2.20 The **passwd** command didn't prompt for the old password. When do you think that can happen? Where is the password stored?
2.21 Can you change the system date with the **date** command?
2.22 Enter the **uname** command without any arguments. What do you think the output represents?
2.23 How will you record your login session in the file `foo`?
2.24 Interpret the following output of **who am i:**

```
romeo    pts/10    Aug  1 07:56    (pc123.heavens.com)
```

2.25 How do you determine the erase, kill, and eof characters on your system?
2.26 You suddenly find your keyboard is displaying uppercase letters even though your *[CapsLock]* key is set properly. What should you try?

EXERCISES

2.1 Enter a # before a command and press *[Enter]*. What do you see, and how do you think you can take advantage of the behavior?
2.2 Name three major differences between UNIX commands and Windows programs.
2.3 A program file named `foo` exists in the current directory, but when we try to execute it by entering `foo`, we see the message `foo: command not found`. Explain how that can happen.

2.4 If a command resides in a directory which is not in `PATH`, there are at least two ways you can still execute it. Explain.
2.5 Where are the commands used by the system administrator located?
2.6 You won't find the **cd** command either in `/bin` or `/usr/bin`. How is it executed then?
2.7 If you find the **echo** command in `/bin`, would you still call it an external command?
2.8 Is an option also an argument? How many arguments are there in this command?
`cat < foo > bar`
2.9 Why shouldn't you have a filename beginning with a `-`?
2.10 Reduce the number of keystrokes to execute this command:
`tar -t -v -f /dev/fd0`.
2.11 Look up the **tar** man page to find out whether the command **tar -cvfb 20 foo.tar *.c** is legitimate or not. Will the command work without the `-` symbol?
2.12 Both commands below try to open the file `foo`, but the error messages are a little different. What could be the reason?

```
$ cat foo
cat: foo: No such file or directory
$ cat < foo
bash: foo: No such file or directory
```

2.13 Invoke the commands **echo hello dolly** and **echo "hello dolly"** (three spaces between `hello` and `dolly`). Explain the difference in command behavior.
2.14 What does the secondary prompt look like, and when does it appear?
2.15 What do the | and the three dots in the SYNOPSIS section of these man pages indicate as shown below?

```
/usr/xpg4/bin/tail [ -f | -r ]
/usr/bin/ls [ -aAbcCdfFgilLmnopqrRstux1 ] [ file ... ]
```

2.16 If a command, filename, and a system call have the same name and are available in Sections 1, 5, and 2 respectively, how will you display the man pages of each one of them?
2.17 Your system doesn't have the **apropos** command. What will you do?
2.18 The command **echo "Filename: \c"** didn't place the cursor at the end of the line. How will you modify the command to behave correctly if your shell is (i) Bash, (ii) any other shell?
2.19 What is an *escape sequence*? Name three escape sequences used by the **echo** command, and explain the significance of each.
2.20 Use **printf** to find out the hex and octal values of 255.
2.21 Run **ps**, then the **script** command, and then run **ps** again. What do you notice?
2.22 In what way is the **mailx** command superior to a GUI program like Netscape or Mozilla?
2.23 Can you have the same user-id more than once in the **who** output?
2.24 Both your local and remote machines use identical versions of UNIX. How do you confirm whether you are logged in to a remote machine or not?

2.25 Which command does the nonprivileged user use to change the system date and time?
2.26 Display the current date in the form *dd/mm/yyyy*.
2.27 You need to accept a secret code through a shell script. What command will you run in the script to make sure that your keyboard input is not displayed? How do you then revert to the normal setting?
2.28 Explain why it is possible to key in the next command before the previous command has completed execution.
2.29 What will you do to ensure that *[Ctrl-c]* interrupts a program? Will it work the next time you log in?

CHAPTER 3

The File System

UNIX looks at everything as a file, and any UNIX system has thousands of files. For convenience, we make a distinction between ordinary files and directories that house groups of files. In this chapter, we'll create directories, navigate the file system, and list files in a directory. We'll also examine the structure of the standard UNIX file system.

In addition, we'll copy, move, and delete files, and understand how these actions affect the directory. Some commands exhibit *recursive* behavior by descending a directory structure to perform some action. Because we frequently encounter Windows systems, we need to be able to move files between UNIX and Windows systems. As Internet users, we also need to handle compressed files that we download. However, we don't tamper with the major file attributes in this chapter.

Objectives

- Understand the initial categorization of files into *ordinary, directory,* and *device*.
- Learn the hierarchical structure of the file system, and how UNIX organizes its own data.
- Understand the significance of the *home directory* and *current directory*.
- Create and remove directories with **mkdir** and **rmdir**.
- Navigate the file system with **cd** and **pwd**.
- Become aware of the significance of *absolute* and *relative* pathnames.
- List files with **ls**.
- Copy, rename, and delete files with **cp**, **mv**, and **rm**.
- View text files with **cat** and **more**.
- Count the number of lines, words, and characters with **wc**.
- Learn how UNIX handles printing and using **lp** and **lpr**.
- Display the nonprintable characters in a file with **od**.
- Convert between UNIX and DOS formats with **unix2dos** and **dos2unix**.
- Compress files with **gzip** and create *archives* comprising multiple files with **tar**.
- Perform both compressing and archiving with **zip**.

3.1 The File

The *file* is a container for storing information. As a first approximation, we can treat it simply as a sequence of characters. UNIX doesn't impose any structure for the data held

Viewing the Archive (-v) You can view the compressed archive with the `-v` option. The list shows both the compressed and uncompressed size of each file in the archive along with the percentage of compression achieved:

```
$ unzip -v archive.zip
Archive:  archive.zip
 Length   Method   Size   Ratio   Date     Time   CRC-32   Name
 ------   ------   ----   -----   ----     ----   ------   ----
3875302   Defl:N  788068   80%   08-24-02  19:49  fae93ded  libc.html
 372267   Defl:N  128309   66%   08-24-02  19:48  7839e6b3  User_Guide.ps
 ------           ------   ---                              -------
4247569           916377   78%                              2 files
```

3.24 Other Ways of Using These Commands

The commands discussed in this chapter don't always take input from files. Some commands (like **more** and **lp**) use, as alternate sources of input, the keyboard or the output of another command. Most of the other commands (like **wc**, **cat**, **od**, **gzip**, and **tar**) can also send output to a file or serve as input to another command. Some examples in this chapter (and previous ones) have shown this to be possible with the > and | symbols. The discussion of these techniques is taken up in Chapter 7.

SUMMARY

We considered three types of files—*ordinary, directory,* and *device*. A directory maintains the inode number and name for each file. The kernel uses the attributes of a device file to operate the device. File attributes are maintained in the inode.

A filename is restricted to 255 characters and can use practically any character. Executable files don't need any specific extensions.

UNIX supports a hierarchical file system where the top-most directory is called *root*. An *absolute* pathname begins with a / and denotes the file's location with respect to root. A *relative* pathname uses the symbols `.` and `..` to represent the file's location relative to the current and parent directory, respectively.

pwd tells you the current directory, and **cd** is used to change it or to switch to the *home* directory. This directory is set in `/etc/passwd` and is available in the shell variable `HOME`. A file `foo` in the home directory is often referred to as `$HOME/foo` or `~/foo`.

mkdir and **rmdir** are used to create or remove directories. To remove a directory `bar` with **rmdir**, `bar` must be empty and you must be positioned above `bar`.

By default, **ls** displays filenames in *ASCII collating sequence* (numbers, uppercase, lowercase). It can also display hidden filenames beginning with a dot (`-a`). When used with a directory name as argument, **ls** displays the *filenames* in the directory.

You can copy files with **cp**, remove them with **rm**, and rename them with **mv**. All of them can be used interactively (`-i`), and the first two can be used to work on a complete directory tree (`-r` or `-R`) i.e., recursively. **rm -r** can remove a directory tree even if is not empty.

cat and **more** are used to display the contents of a file. **more** supports a number of internal commands that enable paging and searching for a pattern. Linux offers **less** as a superior pager.

lp submits a job for printing which is actually carried out by a separate program. Linux and many UNIX systems use the **lpr** command for printing. Both can be *directly* used to print Postscript documents.

wc counts the number of lines, words, and characters. **od** displays the octal value of each character and is used to display invisible characters.

The **dos2unix** and **unix2dos** commands convert files between DOS and UNIX. DOS files use CR-LF as the line terminator, while UNIX uses only LF.

gzip and **gunzip** compresses and decompresses individual files (extension: `.gz`). **tar** can archive a directory tree and is often used with **gzip** to create compressed archives (extension: `.tar.gz`). **zip** and **unzip** use `.zip` files. **zip** alone can create a compressed archive from directory structures (`-r`). **bzip2** is better than them (extension: `.bz2`).

SELF-TEST

3.1 How long can a UNIX filename be? What characters can't be used in a filename?
3.2 State two reasons for not having a filename beginning with a hyphen.
3.3 Name the two types of ordinary files, and explain the difference between them. Provide three examples of each type of file.
3.4 Can the files `note` and `Note` coexist in the same directory?
3.5 Frame **cd** commands to change from (i) `/var/spool/lp/admins` to `/var/spool/mail`, (ii) `/usr/include/sys` to `/usr`.
3.6 Switch to the root directory with **cd,** and then run **cd ..** followed by **pwd.** What do you notice?
3.7 Explain the significance of these two commands: `ls .. ; ls -d ...`
3.8 Can you execute any command in `/sbin` and `/usr/sbin` by using the absolute pathname?
3.9 If the file `/bin/echo` exists on your system, are the commands **echo** and **/bin/echo** equivalent?
3.10 Look up the man pages of **mkdir** to find out the easiest way of creating this directory structure: `share/man/cat1`.
3.11 If **mkdir test** fails, what could be the possible reasons?
3.12 How do you run **ls** to (i) mark directories and executables separately, (ii) also display hidden files?
3.13 What will **cat foo foo foo** display?
3.14 A file contains nonprintable characters. How do you view them?
3.15 How will you copy a directory structure `bar1` to `bar2`? Does it make any difference if `bar2` exists?
3.16 Assuming that `bar` is a directory, explain what the command **rm -rf bar** does. How is the command different from **rmdir bar**?
3.17 How do you print the file `/etc/passwd` on the printer named laser on System V (i) to generate three copies, (ii) and know that the file has been printed?

3.18 How will you find out the ASCII octal values of the numerals and alphabets?
3.19 Run the **wc** command with two or more filenames as arguments. What do you see?

EXERCISES

3.1 Describe the contents of a directory, explaining the mechanism by which its entries are updated by **cp**, **mv**, and **rm**. Why is the size of a directory usually small?
3.2 How does the device file help in accessing the device?
3.3 Which of these commands will work? Explain with reasons: (i) `mkdir a/b/c`, (ii) `mkdir a a/b`, (iii) `rmdir a/b/c`, (iv) `rmdir a a/b`, (v) `mkdir /bin/foo`.
3.4 The command **rmdir c_progs** failed. State three possible reasons.
3.5 Using **echo**, try creating a file containing (i) one, (ii) two, and (iii) three dots. What do you conclude?
3.6 The command **rmdir bar** fails with the message that the directory is not empty. On running **ls bar**, no files are displayed. Why did the **rmdir** command fail?
3.7 How does the command **mv bar1 bar2** behave, where both `bar1` and `bar2` are directories, when (i) `bar2` exists and (ii) `bar2` doesn't exist?
3.8 Explain the difference between the commands **cd ~charlie** and **cd ~/charlie**. Is it possible for both commands to work?
3.9 charlie uses `/usr/charlie` as his home directory and many of his scripts refer to the pathname `/usr/charlie/html`. Later, the home directory is changed to `/home/charlie`, thus breaking all his scripts. How could charlie have avoided this problem?
3.10 Why do we sometimes run a command like this—`./update.sh` instead of `update.sh`?
3.11 What is the sort order prescribed by the ASCII collating sequence?
3.12 The commands **ls bar** and **ls -d bar** display the same output—the string `bar`. This can happen in two ways. Explain.
3.13 Assuming that you are positioned in the directory `/home/romeo`, what are these commands presumed to do, and explain whether they will work at all: (i) `cd ../..`, (ii) `mkdir ../bin`, (iii) `rmdir ..`, (iv) `ls ...`
3.14 Explain what the following commands do: (i) `cd`, (ii) `cd $HOME`, (iii) `cd ~`.
3.15 The command **cp hosts backup/hosts.bak** didn't work even though all files exist. Name three possible reasons.
3.16 You have a directory structure `$HOME/a/a/b/c` where the first `a` is empty. How do you remove it and move the lower directories up?
3.17 Explain what the following commands do: (i) `rm *`, (ii) `rm -i *`, (iii) `rm -rf *`.
3.18 What is the significance of these commands? (i) `mv $HOME/include .`, (ii) `cp -r bar1 bar2`, (iii) `mv * ../bin`.
3.19 Will the command **cp foo bar** work if (i) `foo` is an ordinary file and `bar` is a directory, (ii) both `foo` and `bar` are directories?
3.20 Explain the significance of the repeat factor used in **more**. How do you search for the pattern `include` in a file and repeat the search? What is the difference between this repeat command and the dot command?

3.21 Look up the man page for the **file** command, and then use it on all files in the `/dev` directory. Can you group these files into two categories?
3.22 How do DOS and UNIX text files differ? Name the utilities that convert files between these two formats?
3.23 Run the **script** command, and then issue a few commands before you run **exit**. What do you see when you run **cat -v typescript**?
3.24 Run the **tty** command, and note the device name of your terminal. Now use this device name (say, `/dev/pts/6`) in the command **cp /etc/passwd /dev/pts/6**. What do you observe?
3.25 How do you use **tar** to add two files, `foo.html` and `bar.html`, to an archive, `archive.tar`, and then compress the archive? How will you reverse the entire process and extract the files in their original uncompressed form?
3.26 Name three advantages **zip** has over **gzip**.
3.27 How do you send a complete directory structure to someone by email using (i) **tar**, (ii) **zip**? How does the recipient handle it? Which method is superior and why? Does **gzip** help in any way?
3.28 What is meant by *recursive* behavior of a command? Name four commands, along with a suitable example of each, that can operate recursively.

find uses an AND condition (an implied -a operator between -perm and -type) to select directories that provide all access rights to everyone. It selects files only if both selection criteria (-perm and -type) are fulfilled.

Finding Unused Files (-mtime and -atime) Files tend to build up incessantly on disk. Some of them remain unaccessed or unmodified for months—even years. **find**'s options can easily match a file's modification (-mtime) and access (-atime) times to select them. The -mtime option helps in backup operations by providing a list of those files that have been modified, say, in less than 2 days:

```
find . -mtime -2 -print
```

Here, -2 means *less* than 2 days. To select from the /home directory all files that have not been accessed for more than a year, a positive value has to be used with -atime:

```
find /home -atime +365 -print
```


Note

+365 means greater than 365 days; -365 means less than 365 days. For specifying exactly 365, use 365.

4.11.2 The find Operators (!, -o, and -a)

There are three operators that are commonly used with **find**. The ! operator is used before an option to negate its meaning. So,

```
find . ! -name "*.c" -print
```

selects all but the C program files. To look for both shell and **perl** scripts, use the -o operator which represents an OR condition. We need to use an escaped pair of parentheses here:

```
find /home \( -name "*.sh" -o -name "*.pl" \) -print
```

The (and) are special characters that are interpreted by the shell to run commands in a group *(7.6.2)*. The same characters are used by **find** to group expressions using the -o and -a operators, the reason why they need to be escaped.

The -a operator represents an AND condition, and is implied by default whenever two selection criteria are placed together.

4.11.3 Operators of the Action Component

Displaying the Listing (-ls) The -print option belongs to the *action* component of the **find** syntax. In real life, you'll often want to take some action on the selected files and not just display the filenames. For instance, you may want to view the listing with the -ls option:

```
$ find . -type f -mtime +2 -mtime -5 -ls                    -a option implied
475336 1 -rw-r--r-- 1 romeo users 716 Aug 17 10:31 ./c_progs/fileinout.c
```


find here runs the **ls -lids** command to display a special listing of those regular files that are modified in more than two days and less than five days. In this example, we see two options in the selection criteria (both -mtime) simulating an AND condition. It's the same as using \(-mtime +2 -a -mtime -5 \).

Taking Action on Selected Files (-exec and -ok) The -exec option allows you to run any UNIX command on the selected files. -exec takes the command to execute as its own argument, followed by {} and finally the rather cryptic symbols \; (backslash and semicolon). This is how you can reuse a previous **find** command quite meaningfully:

```
find $HOME -type f -atime +365 -exec rm {} \;                    Note the usage
```

This will use **rm** to remove all ordinary files unaccessed for more than a year. This can be a risky thing to do, so you can consider using **rm**'s -i option. But all commands don't have interactive options, in which case, you should use **find**'s -ok option:

```
$ find $HOME -type f -atime +365 -ok mv {} $HOME/safe \;
< mv ... ./archive.tar.gz > ? y
< mv ... ./yourunix02.txt > ? n
< mv ... ./yourunix04.txt > ? y
  .......
```

mv turns interactive with -i but only if the destination file exists. Here, -ok seeks confirmation for every selected file to be moved to the $HOME/safe directory irrespective of whether the files exist at the destination or not. A y deletes the file.

find is the system administrator's tool, and in Chapter 19, you'll see it used for a number of tasks. It is specially suitable for backing up files and for use in tandem with the **xargs** command (See Going Further of Chapter 7).


Note

The pair of {} is a placeholder for a filename. So, -exec cp {} {}.bak provides a .bak extension to all selected files. Don't forget to use the \; symbols at the end of every -exec or -ok option.

SUMMARY

The **ls -l** command displays the *listing* containing seven file attributes. **ls -ld** used with a directory name lists directory attributes.

A file can have read, write, or execute permission, and there are three sets of such permissions for the *user, group,* and *others*. A file's owner uses **chmod** to alter file permissions. The permissions can be *relative* or *absolute*. The octal digit 7 includes read (4), write (2), and execute permissions (1).

Permissions have different significance for directories. Read permission means that the filenames stored in the directory are readable. Write permission implies that you are permitted to create or remove files in the directory. Execute (or *search*) permission means that you can change to that directory with the **cd** command.

The *umask* setting determines the default permissions that will be used when creating a file or a directory.

Multiple *file systems*, each with its own root directory are *mounted* at boot time to appear as a single file system. A file's attributes are stored in the *inode* which is identified by the *inode number*. The inode number is unique in a single file system.

A file can have more than one name or *link*, and is linked with **ln**. Two linked filenames have the same inode number. A *symbolic link* contains the pathname of another file or directory and is created with **ln -s**. The file pointed to can reside on another file system. **rm** removes both types of links.

Hard links provide protection against accidental deletion but removing the file pointed to by a symlink can be dangerous. Both links enable you to write program code that does different things depending on the name by which the file is invoked.

chown and **chgrp** are used to transfer ownership and group ownership, respectively. They can be used by the owner of the file on AT&T systems. On BSD systems, **chown** can be used only by the superuser, and a user can use **chgrp** to change her group to another to which she also belongs.

A file has three time stamps including the time of last modification and access.

find looks for files by matching one or more file attributes. A file can be specified by type (`-type`), name (`-name`), permissions (`-perm`), or by its time stamps (`-mtime` and `-atime`). The `-print` option is commonly used, but any UNIX command can be run on the selected files with or without user confirmation (`-ls`, `-exec`, and `-ok`).

SELF-TEST

4.1 What do you understand by the *listing* of a file? How will you save the complete listing of all files and directories (including the hidden ones) in the system?

4.2 Show the octal representation of these permissions: (i) `rwxr-xrw-`, (ii) `rw-r-----`, (iii) `--x-w-r--`.

4.3 What will the permissions string look like for these octal values? (i) 567, (ii) 623, (iii) 421

4.4 What does a group member require to be able to remove a file?

4.5 If a file's permissions are 000, can the superuser still read and write it?

4.6 You removed the write permission of a file from group and others, and yet they could delete your file. How could that happen?

4.7 Try creating a directory in the system directories `/bin` and `/tmp` and explain your observations.

4.8 Copy a file with permissions 444. Copy it again and explain your observations.

4.9 How do you ensure that all ordinary files created by you have `rw-rw----` as the default permissions?

4.10 How do you display the inode number of a file?

4.11 What does the inode store? Which important file attribute is not maintained in the inode? Where is it stored then?

4.12 What do you mean by saying that a file has three *links*?

4.13 How do you remove (i) a hard link, (ii) a symbolic link pointing to a directory?

4.14 How do you link all C source files in the current directory and place the links in another directory, `bar`?

4.15 A symbolic link has the same inode number as the file it is linked to. True or false?

4.16 How do you link `fool` to `foo2` using (i) a hard link, (ii) a symbolic link? If you delete `foo2`, does it make any difference?

4.17 Copy the file `/etc/passwd` to your current directory and then observe the listing of the copy. Which attributes have changed?

4.18 Where are the UID and GID of a file stored?

4.19 How is **chown** different from **chgrp** on a BSD-based system when it comes to renouncing ownership?

4.20 Explain with reference to the dot and * what the following commands do: (i) `chown -R project .`, (ii) `chgrp -R project *`.

4.21 When you invoke **ls -l foo** the access time of `foo` changes. True or false?

4.22 View the access time of a file with **ls -lu foo** before appending the **date** command output to it using **date >> foo**. Observe the access time again. What do you see?

4.23 Devise a **find** command to locate in `/docs` and `/usr/docs` all filenames that (i) begin with `z`, (ii) have the extension `.html` or `.java`..

EXERCISES

4.1 A file contains 1026 bytes. How many bytes of disk space does it occupy?

4.2 Does the owner always belong to the same group as the group owner of a file?

4.3 Explain the significance of the following commands: (i) `ls -ld .`, (ii) `ls -l ..`

4.4 Create a file `foo`. How do you assign all permissions to the owner and remove all permissions from others using (i) relative assignment and (ii) absolute assignment? Do you need to make any assumptions about `foo`'s default permissions?

4.5 From the security viewpoint, explain the consequences of creating a file with permissions (i) 000, (ii) 777.

4.6 Examine the output of the two commands below on a BSD-based system. Explain whether romeo can (i) edit, (ii) delete, (iii) change permissions, (iv) change ownership of `foo`:

```
$ who am i ; ls -l foo
romeo
-r--rw----    1 sumit    romeo         78 Jan 27 16:57 foo
```

4.7 Assuming that a file's current permissions are `rw-r-xr--`, specify the **chmod** expression required to change them to (i) `rwxrwxrwx`, (ii) `r--r-----`, (iii) `---r--r--`, (iv) `---------`, using both relative and absolute methods of assigning permissions.

4.8 Use **chmod -w .** and then try to create and remove a file in the current directory. Can you do that? Is the command the same as **chmod a-w foo**?

4.9 You tried to copy a file `foo` from another user's directory, but you got the error message `cannot create file foo`. You have write permission in your own directory. What could be the reason, and how do you copy the file?

4.10 What do you do to ensure that no one is able to see the names of the files you have?

4.11 The command **cd bar** failed where `bar` is a directory. How can that happen?

4.12 If a file has the permissions 000, you may or may not be able to delete the file. Explain how both situations can happen. Does the execute permission have any role to play here?

4.13 If the owner doesn't have write permission on a file but her group has, can she (i) edit it, (ii) delete it?

4.14 If **umask** shows the value (i) 000, (ii) 002, what implications do they have from the security viewpoint?

4.15 The UNIX file system has many root directories even though it actually shows one. True or false?

4.16 What change takes place in the inode and directory when a filename is connected by a hard link?

4.17 If **ls -li** shows two filenames with the same inode number, what does that indicate?

4.18 What happens when you invoke the command **ln foo bar** if (i) bar doesn't exist, (ii) bar exists as an ordinary file, (iii) bar exists as a directory?

4.19 How can you make out whether two files are copies or links?

4.20 Explain two application areas of hard links. What are the two main disadvantages of the hard link?

4.21 You have a number of programs in $HOME/progs which are called by other programs. You have now decided to move these programs to $HOME/internet/progs. How can you ensure that users don't notice this change?

4.22 Explain the significance of *fast symbolic links* and *dangling symbolic links*.

4.23 Explain how **ls** obtains the (i) filename, (ii) name of owner, (iii) name of group owner when displaying the listing.

4.24 How will you determine whether your system uses the BSD or AT&T version of **chown** and **chgrp**?

4.25 The owner can change all attributes of a file on a BSD-based system. Explain whether the statement is true or false. Is there any attribute that can be changed *only* by the superuser?

4.26 What are the three time stamps maintained in the inode, and how do you display two of them for the file foo?

4.27 How can you find out whether a program has been executed today?

4.28 Explain the difference between (i) ls -l and ls -lt, (ii) ls -lu and ls -lut.

4.29 Use **find** to locate from your home directory tree all (i) files with the extension .html or .HTML, (ii) files having the inode number 9076, (iii) directories having permissions 666, (iv) files modified yesterday. Will any of these commands fail?

4.30 Use **find** to (i) move all files modified within the last 24 hours to the posix directory under your parent directory, (ii) locate all files named a.out or core in your home directory tree and remove them interactively, (iii) locate the file login.sql in the /oracle directory tree, and then copy it to your own directory, (iv) change all directory permissions to 755 and all file permissions to 644 in your home directory tree.

CHAPTER 5

The vi/vim Editor

No matter what work you do with the UNIX system, you'll eventually write some C programs or shell (or **perl**) scripts. You may have to edit some of the system files at times. For all of this you must learn to use an editor, and UNIX provides a very old and versatile one—**vi**. Bill Joy created this editor for the BSD system. The program is now standard on all UNIX systems. Bram Moolenaar improved it and called it **vim** (**vi** **im**proved). In this text, we discuss **vi** and also note the features of **vim**, available in Linux.

Like any editor, **vi** supports a number of internal commands for navigation and text editing. It also permits copying and moving text both within a file and from one file to another. The commands are cryptic but often mnemonic. **vi** makes complete use of the keyboard where practically every key has a function. There are numerous features available in this editor, but a working knowledge of it is all that you are required to have initially. The advanced features of **vi** are taken up in Appendix C.

Objectives

- Know the three modes in which **vi** operates for sharing the workload.
- Repeat a command multiple times using a *repeat factor*.
- Insert, append, and replace text in the *Input Mode*.
- Save the buffer and quit the editor using the *ex Mode*.
- Perform navigation in a relative and absolute manner in the *Command Mode*.
- The concept of a *word* as a navigation unit for movement along a line.
- Learn simple editing functions like deleting characters and changing the case of text.
- Understand the use of *operator–command* combinations to delete, yank (copy) and move text.
- Copy and move text from one file to another.
- Undo the last editing action and repeat the last command.
- Search for a pattern, and repeat the search both forward and back.
- Replace one string with another.
- Master the three-function sequence to (i) search for a pattern, (ii) take some action, and (iii) repeat the search and action.
- Customize **vi** using the **:set** command and the file ~/.exrc.
- Become familiar with two powerful features available in **vim**—word completion and multiple undoing.
- Map your keys and define abbreviations *(Going Further)*

Commands usually have limits on the number of arguments they can handle. **xargs** uses the -n option to provide the specified number of arguments for a single invocation of the command:

```
find / -name core -size +1024 -print | xargs -n20 rm -f
```

If **find** locates 100 files, **rm** will be invoked five times—each time with 20 filenames as arguments. A useful tool indeed!

SUMMARY

The shell is a program that runs when a user logs in and terminates when she logs out. It scans the command line for *metacharacters* and rebuilds it before turning it over to the kernel for execution. The shell may or may not wait for the command to terminate.

The shell matches filenames with *wild cards*. It can match any number of characters (*) or a single one (?). It can also match a *character class* ([ ]) and negate a match ([!]). The * doesn't match a filename beginning with a dot.

A wild card is *escaped* with a \ to be treated literally, and if there are a number of them, then they should be *quoted*. Single quotes protect all special characters, while double quotes enable command substitution and variable evaluation.

Files are accessed with small integers called *file descriptors*. The shell makes available three files representing *standard input, standard output,* and *standard error* to every command that it runs. It manipulates the default source and destination of these streams by assigning them to disk files.

The file /dev/null never grows in size, and every user can access her own terminal as /dev/tty.

Pipes connect the standard output of one command to the standard input of another. Commands using standard output and standard input are called *filters*. A combination of filters placed in pipelines can be used to perform complex tasks which the commands can't perform individually.

The external **tee** command duplicates its input. It saves one to a file and writes the other to the standard output.

Command substitution enables a command's standard output to become the arguments of another command.

The shell supports *variables* which are evaluated by prefixing a $ to the variable name. The variables that control the workings of the UNIX system are known as *environment variables*.

The shell is also a scripting language, and a group of commands can be placed in a *shell script* to be run in a batch.

SELF-TEST

7.1 Why does the shell need to expand wild cards? How does it treat the * when used as an argument to a command (like **echo ***)?

7.2 What is the significance of the command **ls *.***? Does it match filenames that begin with a dot?

GOING FURTHER

7.3 How do you remove only the hidden files of your directory? Does **rm *** remove these files as well?

7.4 Match the filenames chapa, chapb, chapc, chapx, chapy, and chapz with a wild-card expression.

7.5 Is the wild-card expression [3-h]* valid?

7.6 Devise a command that copies all files named chap01, chap02, chap03, and so forth through chap26 to the parent directory. Can a single wild-card pattern match them all?

7.7 Frame wild-card patterns (i) where the last character is not numeric, (ii) that have at least four characters.

7.8 When will **cd *** work?

7.9 Which UNIX command uses wild cards as part of its syntax?

7.10 How do you split a long command sequence into multiple lines?

7.11 Name the three sources and destinations of standard input and standard output.

7.12 Is the output of the command **cat foo1 foo2 >/dev/tty** directed to the standard output?

7.13 Is this a legitimate command, and what does it appear to do? >foo <bar bc

7.14 How do you save your entire home directory structure including the hidden files in a separate file?

7.15 What is the file /dev/null used for?

7.16 The commands **cat** and **wc**, when used without arguments, don't seem to do anything. What does that indicate, and how do you return the shell prompt?

7.17 How do you create a filename containing just one space character? How can you "see" the space in the **ls** output?

7.18 How do you find out the number of (i) users logged in, (ii) directories in your home directory tree?

7.19 Enter the commands **echo "$SHELL"** and **echo '$SHELL'**. What difference do you notice?

7.20 Command substitution requires the command to use (i) standard input, (ii) standard output, (iii) both, (iv) none of these.

7.21 Attempt the variable assignment x = 10 (space on both sides of the =). Does it work if you are not using the C shell?

7.22 To append .c to a variable x, you have to use the expression (i) $x.c, (ii) $x".c", (iii) ${x}.c, (iv) any of these, (v) only the first two.

EXERCISES

7.1 What happens when you use (i) **cat > foo** if foo contains data, (ii) **who >> foo** if foo doesn't exist, (iii) **cat foo > foo**, (iv) **echo 1> foo?**

7.2 What does the shell do with the metacharacters it finds in the command line? When is the command finally executed?

7.3 Devise wild-card patterns to match the following filenames: (i) foo1, foo2 and Foo5, (ii) quit.c, quit.o and quit.h, (iii) watch.htm, watch.HTML and Watch.html, (iv) all filenames that begin with a dot and end with .swp.

7.4 Explain what the commands **ls .*** and **ls *.** display. Does it make any difference if the -d option is added?

7.5 How do you remove from the current directory all ordinary files that (i) are hidden, (ii) begin and end with #, (iii) have numerals as the first three characters, (iv) have single-character extensions? Will the commands work in all shells?

7.6 Devise wild-card patterns to match all filenames comprising at least three characters (i) where the first character is numeric and the last character is not alphabetic, (ii) not beginning with a dot, (iii) containing 2004 as an embedded string except at the beginning or end.

7.7 Explain what these wild-card patterns match: (i) `[A-z]????*`, (ii) `*[0-9]*`, (iii) `*[!0-9]`, (iv) `*.[!s][!h]`.

7.8 A directory `bar` contains a number of files including one named `-foo`. How do you remove the file?

7.9 You have a file named * and a directory named `My Documents` in the current directory. How do you remove them with a single command using (i) escaping, (ii) quoting?

7.10 Explain the significance of single- and double-quoting including when one is preferred to the other. What are the two consequences of using double quotes?

7.11 When will **`wc < chap0[1-5]`** work? How can you remove `chap0[1-5]` if you have a file of that name?

7.12 Explain why the error message is seen at the terminal in spite of having used the 2> symbol:

```
$ cat < foo 2>bar
ksh: cannot open foo: No such file or directory
```

7.13 How do the commands **`wc foo`** and **`wc < foo`** differ? Who opens the file in each case?

7.14 You want to concatenate two files, `fool` and `foo2`, but also insert some text after `fool` and before `foo2` from the terminal. How will you do this?

7.15 Execute the command **`ls > newlist`**. What interesting observation can you make from the contents of `newlist`?

7.16 How will you add the tags `<html>` and `</html>` to the beginning and end respectively of `foo.html`?

7.17 What are *file descriptors*? Why is 2> used as the redirection symbol for standard error?

7.18 Create a file `foo` with the statement **`echo "File not found"`** in it. Explain two ways of providing redirection to this statement so that the message comes to the terminal even if you run **`foo > /dev/null`**.

7.19 How do the programs `prog1`, `prog2`, and `prog3` need to handle their standard files so they can work like this? `prog1 | prog2 | prog3`.

7.20 Use command substitution to print the (i) calendar of the current month, (ii) listing of a group of filenames stored in a file.

7.21 Explain the behavior of this command: `echo '`find $HOME -type d -print | wc -l`' > list`. How do you modify it to work correctly?

7.22 When will the command **`cd `find . -type l -name scripts -print``** work? If it does, what do **`pwd`** and **`/bin/pwd`** display?

7.23 What is a *filter*? For the statement **`` `foo` ``** to work, does **`foo`** have to be a filter?

7.24 Look up the **`tar`** and **`gzip`** documentation to find out how a group of files can be archived and compressed without creating an intermediate file.

7.25 How will you store in a variable `count` (i) the total size of all C source files (`.c`), (ii) the total number of lines in a file?

7.26 Interpret these statements and the message displayed (if any): (i) `$x=5`, (ii) `directory='pwd'=`pwd``.

7.27 A file `foo` contains a list of filenames. Devise a single statement, with suitable explanation, that stores in a variable `count` the total character count of the *contents* of these files. (HINT: Both command substitution and **`cat`** have to be used twice.)


Caution

If you use **crontab -** to provide input through the standard input and then decide to abort it, you should terminate it with the interrupt key applicable to your terminal, rather than *[Ctrl-d]*. If you forget to do that, you'll remove all entries from your existing crontab file!

cron's strength lies in its unusual number matching system. You can match one or more numbers if you keep in mind these rules:

- A * used in any of the first five fields matches any valid value.
- A set of numbers is delimited by a comma. `3,6,9` is a valid field specification.
- Ranges are possible and need not be restricted to a single digit. `00-10` includes all integer values between 0 and 10.

Things don't appear so simple when crontab fields conflict with one another. Take, for instance, this entry:

```
00-10 17 * 3,6,9,12 5  find / -newer .last_time -print > backuplist
```

The first two fields indicate that the command is to run every minute from 17:00 hours to 17:10 hours. The third field (being a *) specifies that it should run every day. The fourth field (`3,6,9,12`), however, restricts the operation to four months of the year. The fifth field limits execution to every Friday.

So, who overrides whom? Here, "Friday" overrides "every day." The **find** command will thus be executed every minute in the first 10 minutes after 5 p.m., every Friday of the months March, June, September, and December (of every year).

So, what are the rules that determine which fields have the ultimate say? This question arises when a * occurs in the third, fourth, or fifth fields. The rules are clearly laid down by POSIX and Table 8.4 shows all possible combinations of these fields.


Caution

Unless you are sure, never use a * in the minute field. You'll receive a mail every minute, and this could completely use up your mail quota if the command produces high-volume output.

cron is mainly used by the system administrator to perform housekeeping chores, like removing outdated files or collecting data on system performance. It's also extremely useful to periodically dial up to an Internet mail server to send and retrieve mail.


Linux

The number matching system goes beyond POSIX requirements. It allows the use of step values which enable us to use compact expressions. You can use `3-12/3` instead of `3,6,9,12` that was used in our examples. Moreover, a * comes in handy here; `*/10` in the minutes field specifies execution every 10 minutes. The crontab file also supports a `MAILTO` variable which sends mail to the user whose name is assigned to the variable. The mail is suppressed if we set `MAILTO=""`.

cron looks in a control file in `/var/spool/cron` in Red Hat. It additionally looks up `/etc/crontab` which specifies the user as an additional field (the sixth). This file generally specifies the execution of files in the directories `cron.hourly`, `cron.daily`, `cron.weekly`, and `cron.monthly` (in `/etc`).

TABLE 8.4 *Sample crontab Entries (First five fields only)*

Fields	*Matches*
*When a * occurs in any of the third, fourth, and fifth fields*	
`00-10 17 * * *`	Every day
`00-10 17 * 3,6,9,12 *`	Every day but restricted to four months
`00-10 17 10,20,30 * *`	Three days in a month
`00-10 17 * * 1,3`	Monday and Wednesday
`00-10 17 * 3,6,9,12 1,3`	Either every day of four months or Monday and Wednesday of every month
`00-10 17 10,20,30 * 1,3`	Either three days of every month or Monday and Wednesday of every month
Other Examples	
`0,30 * * * *`	Every 30 minutes on the half-hour.
`0 0 * * *`	Midnight every day.
`55 17 * * 4`	Every Thursday at 17:55 hours.
`30 0 10,20 * *`	00:30 hours on the tenth and twentieth of every month.
`00,30 09-17 * * 1-5`	On weekdays every half hour between 9 and 17 hours.

anacron **cron** assumes that the machine is run continuously, so if the machine is not up when a job is scheduled to run, **cron** makes no amends for the missed opportunity. The job will have to wait for its next scheduled run. The **anacron** command is often more suitable than **cron. anacron** periodically inspects its control file (`/etc/anacrontab`) to see if there's a job which has "missed the bus." If it finds one, it executes the job.

8.13.1 Controlling Access to cron

All users may not be able to use **cron**. As with **at** and **batch**, the authorization to use it is controlled by two files, `cron.allow` and `cron.deny`. If `cron.allow` is present, only users included in this file are allowed to use this facility. If this file is not present, `cron.deny` is checked to determine the users who are prohibited. In case neither of them is present, depending on the system configuration, either the system administrator only is authorized to use **cron** or all users are allowed access.

SUMMARY

A process is an instance of a running program. It is identified by the *process-id* (PID) and its *parent PID* (PPID). Process attributes are maintained in the *process table* in memory.

Because of multitasking, a process can *spawn* multiple processes. The login shell is a process (PID = $$) that keeps running as long as the user is logged in.

You can list your own processes with **ps**, view the process ancestry (`-f`), all users' processes (`-a`), and all system processes (`-e`). BSD uses a different set of options.

System processes, often called *daemons,* are generally not attached to a terminal and not invoked specifically by a user. **init** is the parent of most daemons and all users' shells.

A process is created by *forking,* which creates a copy (a child) of itself. The child then uses *exec* to overwrite itself with the image of the program to be run.

The child turns into a *zombie* on termination. The kernel doesn't remove its process table entry until the parent picks up the *exit status* of the child. Premature death of the parent turns the child into an *orphan,* and **init** takes over the parentage of all orphans.

The child's environment inherits some parameters from the parent, like the real and effective UID and GID, the file descriptors, the current directory, and environment variables. However, changes in the child are not made available in the parent.

Built-in shell commands like **pwd** and **cd** don't fork a separate process. Shell scripts use a sub-shell to run the commands in a script.

The UNIX kernel communicates with a process by sending it a *signal.* Signals can be generated from the keyboard or by the **kill** command. You can kill a process with **kill,** and use **kill -s KILL** if a simple **kill** doesn't do the job.

A job can be run in the background. **nohup** ensures that a background job remains alive even after the user has logged out.

The C shell, Korn and Bash shells enable job control. You can move jobs between foreground and background (**fg** and **bg**) and suspend (*[Ctrl-z]*) them. You can list jobs (**jobs**) and also kill them (**kill**).

You can schedule a job for one-time execution with **at**, or run it when the system load permits with **batch. cron** lets you schedule jobs so that they run repeatedly. It takes input from a user's *crontab* file where the schedule and frequency of execution is specified by five fields using a special number matching system.

SELF-TEST

8.1 What is the significance of the PID and PPID? Without using **ps**, how do you find out the PID of your login shell?
8.2 How do you display all processes running on your system?
8.3 Which programs are executed by spawning a shell? What does the second shell do?
8.4 Name some commands that don't require a separate process.
8.5 Name the two system calls required to run a program.
8.6 How will you find out the complete command lines of all processes run by user timothy?
8.7 Run **ps** with the appropriate option, and note some processes that have no controlling terminal.
8.8 How will you use **kill** to ensure that a process is killed?
8.9 How will you kill the last background job without knowing its PID?
8.10 How do you display the signal list on your system?
8.11 Should you run a command like this? `nohup compute.sh`
8.12 The **jobs** command displayed the message `jobs: not found`. When does that normally happen?
8.13 In the midst of an editing session with **vi** or **emacs**, how do you make a temporary exit to the shell and then revert to the editor?

8.14 How do you find out the name of the job scheduled to be executed with **at** and **batch**?
8.15 Frame an **at** command to run the script `dial.sh` tomorrow at 8 p.m.
8.16 Interpret the following crontab entry:

```
30 21 * * * find /tmp /usr/tmp -atime +30 -exec rm -f {} \;
```

8.17 You invoked the **crontab** command to make a crontab entry and then changed your mind. How do you terminate the standard input that **crontab** is now expecting?
8.18 How does the system administrator become the exclusive user of **at** and **cron**?

EXERCISES

8.1 Mention the significance of the two parameters, $$ and $!. Explain the differing behavior of the command **echo $$** when run from the shell prompt and inside a shell script.
8.2 Mention the similarities that you find between processes and files.
8.3 If two users execute the same program, are the memory requirements doubled?
8.4 What are the two options available to a parent after it has spawned a child? How can the shell be made to behave in both ways?
8.5 Explain the significance of this command: `ps -e | wc -l`.
8.6 Explain the attributes of *daemon* processes using three examples. How do you display and identify them?
8.7 Which process will you look for in the **ps** output if you are not able to (i) print, (ii) send out mail, (iii) log in using the secure shell?
8.8 Unlike the built-in commands, **pwd** and **echo**, which also exist as separate disk files, why is there no file named **cd** on any UNIX system?
8.9 Which process do you think may have the maximum number of children? What is its PID? Can you divide its children into two categories?
8.10 How is a process created? Mention briefly the role of the **fork** and *exec* system calls in process creation.
8.11 Name five important process attributes that are inherited by the child from its parent.
8.12 A shell script `foo` contains the statement **echo "$PATH $x"**. Now define **x=5** at the prompt, and then run the script. Explain your observations and how you can rectify the behavior.
8.13 What is a *zombie,* and how is it killed?
8.14 Explain whether the following are true or false: (i) A script can be made to ignore all signals. (ii) The parent process always picks up the exit status of its children. (iii) One program can give rise to multiple processes.
8.15 What is the difference between a process run with & and one run with **nohup**?
8.16 What are *signals*? Name two ways of generating signals from the keyboard. Why should we use **kill** with signal names rather than their numbers?
8.17 What is the difference between a *job* and a *process*? How do you (i) suspend the foreground job, (ii) move a suspended job to the background, (iii) bring back a suspended job to the foreground?

8.18 Interpret these crontab entries and explain if they will work: (i) `* * * * * dial.sh`, (ii) `00-60 22-24 30 2 * find.sh`, (iii) `30 21 * * * find /tmp /usr/tmp -atime +30 -exec rm -f {} \;`.

8.19 Frame a crontab entry to execute the `connect.sh` script every 30 minutes on every Monday, Wednesday, and Friday between the times of 8 a.m. and 6 p.m.

8.20 Create a directory `foo`, and then run a shell script containing the two commands **`cd foo ; pwd`**. Explain the behavior of the script.

8.21 What does the **`exit`** command do? Why doesn't it log you out when run in your login shell like this? `( exit )`

8.22 The cron facility on your system is not working. How do you check whether the process is running at all and whether you are authorized to use **`cron`**?

8.23 The administrator has decided that most users will be allowed to use **`at`** and **`cron`**. What should she change that requires minimum effort?

CHAPTER 9

The Shell—Customizing the Environment

The shell is different from other programs. Apart from interpreting metacharacters, it presents an environment that you can customize to suit your needs. These needs include devising shortcuts, manipulating shell variables, and setting up startup scripts. A properly setup shell makes working easier, but the degree of customization possible also depends on the shell you use.

This chapter presents the environment-related features of the Bash shell, but also examines the differences with three other shells—Bourne shell, C shell, and Korn shell. After reading this chapter, you may want to select your shell. To aid you in this task, let it be said right here that you'll have a headstart over others if you select either Korn or Bash as your login shell.

Objectives

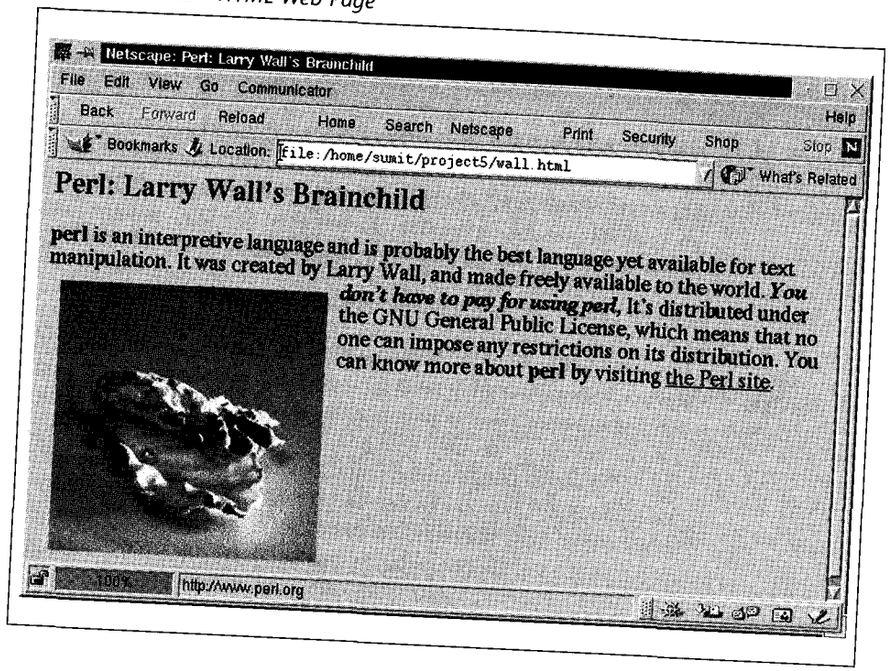
- Learn the evolution of the four shells—Bourne shell, C shell, Korn shell, and Bash.
- Discover the difference between *local* and *environment* variables.
- Examine some environment variables like `PATH`, `SHELL`, `MAIL`, and so forth.
- Use *aliases* to invoke commands with short names.
- Use the *history* mechanism to recall, edit, and run previously executed commands.
- Edit any previous command line using the **`vi`**-like *in-line editing* feature.
- Use the *tilde substitution* feature to shorten pathnames that refer to the home directory.
- Prevent accidental overwriting of files and logging out using **`set -o`**.
- Make environment settings permanent using *profiles* and *rc* scripts.
- Manipulate the directory stack *(Going Further)*.

9.1 The Shells

The UNIX shell is both an interpreter and a scripting language. This is one way of saying that a shell can be interactive or noninteractive. When you log in, an **interactive shell** presents a prompt and waits for your requests. This type of shell supports job control, aliases, and history. An interactive shell runs a **noninteractive shell** when executing a shell script.

Every feature used in a shell script can also be used in an interactive shell, but the reverse is not true. Job control and history have no meaning in a shell script. In this chapter, we are mostly concerned with interactive shells.

FIGURE 14.2 *HTML Web Page*



In the HTML source on page 421, the word `perl` appears twice in boldface as shown in Fig. 14.2. The browser ignores extra spaces and blank lines, and combines multiple adjacent spaces to a single space.

Two tags provide the actual hyptertext capability—`<IMG>` and `<A>`. Both of them take on *attributes* in the form *attribute=value*. The `<IMG>` tag and its `SRC` attribute are used to specify the URL of a graphic. The browser fetches the graphic file from the server (which could be a different one) and displays it inline within the Web page. Here, the tag places the picture of a pearl on the page.

The anchor tag, `<A>`, and the `HREF` attribute behave in a similar manner except that they allow you to click on a section of text or an image to fetch another resource. Here, the words `the Perl site` appear underlined, indicating a hyperlink. Clicking on it replaces the current page with the home page of *www.perl.org*.

Because HTML documents are text files, they are portable, and you can view them on *any* browser on *any* machine running *any* operating system. They are also small in size and thus are ideally suited for use on the Web where network bandwidth is often a constraint.

14.13.5 The Web Browser

The Web browser is the HTTP client. It accepts a URL either from the URL window or from a bookmark, and fetches the resource from the server. If the document contains

`<IMG SRC>` tags, the browser fetches the images the tags link to—using a single Keep-Alive connection, wherever possible. Every browser is also expected to offer these features:

- Step back and forth through documents viewed in a session.
- Save HTML files (and graphics) to the local machine.
- Bookmark important URLs so they can be fetched later without actually entering the URL.
- Support other application protocols like FTP and TELNET.
- Automatically invoke helper applications and special software (plugins) when encountering a file format it can't handle.

Like email clients, the earliest Web browsers were character-based, and the **`lynx`** browser remained popular until the advent of graphics and X Window. Netscape Navigator is the standard graphic browser for UNIX systems today. Linux users have a wider choice in Navigator, Mozilla, Konqueror, (part of KDE) and Firefox.

14.14 Multimedia on the Web: MIME Revisited

Web documents today feature a variety of multimedia objects like Java applets, RealAudio, RealVideo, and Shockwave technology. MIME technology *(14.11)* also applies to multimedia files on the Web. However, these files are sent by Web servers not as multipart messages but as independent files. The server sends the content type to the client before it sends the file. It does this by looking up `mime.types` that associates the content type with the file's extension, as shown below for a PDF document:

```
type=application/acrobat   exts=pdf                    Solaris
application/pdf            pdf                         Linux
```

When a browser encounters an unfamiliar data format, it first sees whether there is a **plugin** in its arsenal. A plugin is a piece of software installed ("plugged") in the browser. It is normally small in size and has the minimal features required for simple viewing (or, in case of audio and video, playing). You can't invoke a plugin separately as you can call up a helper application (explained next) like Acrobat Reader. When a file is viewed with a plugin, it appears inline with the HTML text, and not in a separate window.

If the browser is not able to locate a plugin for a specific content type, it looks up `mailcap` to determine the **helper application**. This is a separate standalone application that can also be invoked separately from the UNIX command line. We saw one entry in this file in Section 14.11 that specified `acroread` for `application/pdf`. Unlike in Windows, UNIX Netscape doesn't have this file configured well, so you'll have to fill it up yourself.

SUMMARY

TCP/IP is a suite of protocols that connects heterogeneous machines in a network. It splits data into packets and ensures reliable transmission with full error control. Packets pass through *routers* to reach their destination.

A host is represented by a unique *hostname* and a unique *IP address* comprising four dot-separated octets. A host can be accessed both by its IP address and hostname, but TCP/IP packets contain only IP addresses.

The hostname-IP address translation is performed by `/etc/hosts` or the *Domain Name System* (DNS). The hosts file is maintained on *all* machines of a network. DNS understands a host by its *fully qualified domain name* (FQDN) and distributes the mappings across a number of *name servers*. The *resolver* queries the hosts file or DNS to perform the translation.

TCP/IP works in the *client-server* model. Server programs are known as *daemons*, which run in the background and listen for requests at certain *ports*.

`telnet` is used to run commands on a remote machine and display the output on the local machine. **`ftp`** transfers files between two hosts. You can upload one or more files (**`put`** and **`mput`**) or download them (**`get`** and **`mget`**). *Anonymous FTP* lets you download files from the Internet.

The *secure shell* is more secure than **`telnet`** and **`ftp`** as it encrypts the entire session including the password. It uses a *symmetric key* for encryption of bulk data, but uses *asymmetric keys* (public and private) for host and user authentication and key distribution. You can log in in a secure manner (**`ssh`** and **`slogin`**), transfer files (**`scp`** and **`sftp`**), and run a command remotely (**`ssh`**).

Internet mail is handled by three agencies. You read and compose mail using a *Mail User Agent* (MUA). The Mail Transport Agent (MTA) transports mail to the MTA at the receiving end using the *Simple Mail Transfer Protocol* (SMTP). The *Mail Delivery Agent* (MDA) delivers the mail to the user's mailbox.

The Web works on the *Hyper Text Transfer Protocol* (HTTP) at port 80. Web documents written in the *Hyper Text Markup Language* use *hypertext* to link one document with another resource. An HTML document is cross-platform and can be viewed in any environment.

The *Uniform Resource Locator* (URL) combines the FQDN of the site with a pathname. It can point to a static resource like a file or a program to be run, using the *Common Gateway Interface* (CGI). **`perl`** is the language of choice for CGI programming.

The *Multipurpose Internet Mail Extensions* (MIME) standard enables transmission of binary data in both email and HTTP. The `Content-Type:` and `Content-Transfer-Encoding:` headers together define the type of data and encoding techniques used. The file `mime.types` associates the content type with a file's extension, and `mailcap` specifies the helper application that will handle a specific content type.

SELF-TEST

14.1 Why is TCP termed a *reliable* protocol?
14.2 What is the significance of the port number? How will you find out the port number **`finger`** uses?
14.3 Why are the TELNET and FTP services increasingly being disabled on most networks? What are they being replaced with?
14.4 How can you be sure whether you are working on the local machine or have used **`telnet`** or **`ssh`** to log on to a remote machine?

14.5 You copied a graphics file with **`ftp`**, and the file appears corrupted. What could be the possible reason?
14.6 With which command do you upload files to an anonymous FTP site?
14.7 What is a *brute force attack?* Why does the security of data mainly depend on the size of the key?
14.8 To send a large volume of data securely over a network connection, what form of encryption would you adopt?
14.9 What is the difference between a *password* and a *passphrase*? Why is it necessary to have a passphrase?
14.10 Using **`scp`**, how will you noninteractively copy all files from juliet's home directory on host *saturn* without knowing the absolute pathname of her home directory?
14.11 What does this command do? `ssh jupiter date \> .date`
14.12 How does X solve the problem of running the same program on different displays with different characteristics?
14.13 Can an X client like **`xterm`** running on a Solaris machine display its output on a HP-UX machine?
14.14 What is the problem with `/etc/hosts`?
14.15 Name three top-level domains that have been added to the Internet namespace in the year 2000. Is the domain name *WWW.suse.COm* valid?
14.16 Explain the significance of the *MUA* and *MTA*. Whom does the MTA hand over mail to?
14.17 How are binary files included in mail messages even though SMTP handles only 7-bit data? Name the two mail headers that play an important role here.
14.18 The browser can display three types of images without needing external help. What are they?
14.19 What is *hypertext?* Is it confined to text only?
14.20 What is *HTTP*? Which port number does it use?
14.21 What are *CGI programs?* How are they invoked?
14.22 How do you access the home page of the Web server running on your own machine?

EXERCISES

14.1 How is a TCP/IP network different from a telephone network?
14.2 What is an *FQDN?* Why are hostnames not used on the Internet, but only FQDNs?
14.3 Describe the role of the resolver when handling (i) simple hostnames, (ii) FQDNs.
14.4 Name three important features of DNS. What advantages does DNS have over the hosts file?
14.5 Explain the role of a name server. What does a name server do if it can't handle an FQDN?
14.6 When you change your local directory from inside **`ftp`**, will the changed directory still be in place after you quit **`ftp`**, and why?
14.7 When A sends data to B over a network connection using public key cryptography, how does A achieve the goals of (i) authentication, (ii) confidentiality?

14.8 Public key cryptography is more suitable for key distribution than bulk data encryption. Explain how you can use this mechanism to distribute a symmetric key.
14.9 For using SSH, why does a host also need to have a public and private key?
14.10 Explain how you can generate a public/private key pair for yourself.
14.11 Explain how the **ssh-agent** and **ssh-add** programs enable noninteractive logins.
14.12 Cite two reasons why **scp** is preferable to **ftp**.
14.13 How is the client-server mechanism in X different from others?
14.14 How can romeo running Netscape on his machine *saturn* write its output to juliet's display on a remote machine *uranus*? Do both users need to run X?
14.15 Why is the `DISPLAY` variable more convenient to use than the `-display` option?
14.16 Explain how the general mail handling scheme changes when a user connects to the mail server over a dialup line.
14.17 Explain the significance of each word in the acronym *URL*. What happens if you leave out the port number in the URL?
14.18 Why is HTTP called a *stateless* protocol? What is meant by the *Keep-Alive* feature?
14.19 Why is the HTML format specially suitable for Web documents?
14.20 Can you use *WWW.PLANETS.COM/CATALOG.HTML* instead of *www.planets.com/catalog.html* as the URL?
14.21 To download a Web page with 10 graphics, how many connections are required in (i) HTTP 1.0, (ii) HTTP 1.1?
14.22 If a browser passes data from an HTML form to the server, how does the server handle the data?
14.23 What is a *helper application,* and how does it differ from a *plugin?* Explain the role of the files, `mime.types` and `mailcap`, when using a helper application.
14.24 What is *MIME?* How are the limitations of SMTP in handling mail attachments overcome by MIME?

CHAPTER 15

perl—The Master Manipulator

perl is UNIX's latest major acquisition, and one of its finest. Developed by Larry Wall, this Practical Extraction and Report Language is often hailed as the "Swiss Army Officer's Knife" of the UNIX system. In **perl**, Wall invented a catchall tool that does several things well. **perl** is standard on Linux and also offered on Solaris. However, it is free, and executables are available for all UNIX flavors (*http://www.perl.com*).

perl is both a scripting language and the mother of all filters. It combines the power of C, the UNIX shell, and its power filters—**grep**, **tr**, **sed**, and **awk**. It has all the control structures and regular expressions that you could find anywhere. It is exceedingly cryptic, even by UNIX standards, and can solve text manipulation problems with very compact code. In spite of offering so much, **perl** is faster than the shell and **awk** (but not C).

Objectives

- Gain an overview of a sample **perl** program.
- Understand how **perl** treats variables and constants and changes their type when required.
- Learn how to use the concatenation and repetition operators (`.` and `x`).
- Read files both in the command line and inside a script from command line arguments.
- Understand the significance of the default variable, `$_`, and how its presence can be felt everywhere.
- Use *lists* and *scalar arrays*, and the functions to manipulate them.
- Use the **foreach** loop for working with a list.
- Split and join a line with **split** and **join**.
- Handle *associative arrays* with a nonnumeric subscript.
- Examine **perl**'s enlarged regular expression set which uses special escape sequences.
- Filter data with the **s** and **tr** commands.
- Use *filehandles* to access a file or stream.
- Test the file attributes.
- Develop *subroutines* for repeated use.
- Gain an overview of CGI and how **perl** is suitable for the task. *(Going Further)*

16.11.4 Making a Reassignment

We'll now use a helpful debugger feature to change the value of a variable without recompiling the program. This requires the **assign** command, so let's learn to use it before we run the program again with **rerun**:

```
(dbx) help assign
assign (command)
assign <var> = <exp>   # Assign the value of the <exp> to <var>
(dbx) rerun
Running: a.out
(process id 1542)
Enter a multiword string: gdb is a better debugger
stopped in main at line 16 in file "parsestring.c"
  16      clargs[0] =  strtok(buf, DELIM);    /* first word */
```

The program stops at the breakpoint defined earlier. Let's now assign 1 to n:

```
(dbx) print n
n = 0
(dbx) assign n = 1
(dbx) print n
n = 1
```

Now let's issue a series of **step** commands till we reach line 22:

```
(dbx) step
stopped in main at line 17 in file "parsestring.c"
   17     while ((clargs[n] = strtok(NULL, DELIM)) != NULL)
  ... more step commands ......
   22        printf("Argument %d is %s\n",i,  clargs[i]);
(dbx) step
Argument 0 is gdb is a better debugger
```

You can also use **step** *n* to execute *n* lines. The **for** loop has been executed once, but that is expected because the **while** loop was not executed at all. We now quit **dbx** with **quit**, and then make these changes to lines 5 and 11 with the **vi** or **emacs** editor:

```
#define DELIM " \n\t\r"                    Introduce a space before \n
n = 1;                                     Instead of n = 0;
```

Alternatively, we could have edited the source file itself with the **edit** command without leaving **dbx**. Don't forget to compile the program with the **-g** option of the compiler. When you trace the variable n (next topic), you'll find that both loops work properly.

16.11.5 Tracing a Variable

The **trace** command displays information about an event. It often produces a lot of output, but we can use **trace** selectively to trace a variable or a function. Enter the debugger once more with **dbx a.out** and find out how **trace** is used:


```
(dbx) help trace
trace at <line#>        # Trace given source line
trace in <func>         # Trace calls to and returns from the given function
trace change <var>      # Trace changes to the variable
When the specified event occurs, a "trace" is printed.
```

Let's now place a trace on the variable n. The next **run** command shows three iterations of the **while** loop and four of the **for** loop:

```
(dbx) trace change n
(2) trace change n -in main
(dbx) run
Running: a.out
(process id 1603)
initially (at line "parsestring.c":12): n = -4260708
after line "parsestring.c":11: n = 1
Enter a multiword string: But X/Open requires dbx
after line "parsestring.c":18: n = 2                    while loop
after line "parsestring.c":18: n = 3                    is executed
after line "parsestring.c":18: n = 4                    three times
Argument 0 is But                                       for loop
Argument 1 is X/Open                                    is executed
Argument 2 is requires                                  four
Argument 3 is dbx                                       times
execution completed, exit code is 1
(dbx) quit
$_
```

Now that both loops are working properly, you can run the program without using **dbx**. Once you are satisified with the behavior of the program, recompile it without the -g option.

Before we close, let's briefly examine the other mode of this debugger. **dbx** can also work in post-mortem mode. When a program crashes, the operating system saves the memory image of the program at the time of the crash in a file named core. **dbx** can be used to analyze this file to determine the cause of the crash, often by identifying the signal that was generated at that time.

SUMMARY

A C program is *compiled* and *assembled* to create an object (.o) file and *linked* to create the executable. Object files should be retained for multisource programs to avoid recompilation of unchanged sources. Functions should be placed in separate files to be reusable.

make monitors the last modification times of the executable, object, source, and header files to determine the sources that need to be recompiled. It looks up a *makefile* for associating a *target* with a *dependency*, and then takes a defined action if the dependency is found to be newer than target.

The **ar** command combines a group of object files into a *static library* or *archive*. **make** can automatically recompile a module and replace it in the archive. A *shared library* or *shared object* is not linked to the executable but is loaded during runtime.

The *Source Code Control System (SCCS)* saves the first version in full and its differences with subsequent versions in an encoded *SCCS file*. A *delta* (version) is checked out (**get**) and checked in (**delta**) after editing. Revision of an intermediate version creates a *branch delta*. The **admin** command is used both for creation of an SCCS file and access control.

The *Revision Control System (RCS)* saves the latest *revision*. A revision is checked in with **ci** and checked out with **co**. The **rcs** command controls version locking, user access and is also used to remove versions.

dbx is used for analyzing core dumps and for debugging programs. You can set *breakpoints* (**stop**) and execute one statement at a time (**step**). A variable can be displayed (**print**) and reassigned in the debugger itself (**assign**). You can also trace a variable and display its value whenever it is encountered (**trace**). GNU **gdb** is also a powerful debugger.

SELF-TEST

16.1 Name the three phases a program has to go through before an executable is created from it.

16.2 Name the two commands invoked by the **cc** command to create an executable.

16.3 Place the function definitions of **arg_check** and **quit** *(16.1.1)* in a single file, **foo.c**. Can the main program, **rec_deposit** (in **rec_deposit.c**), access them?

16.4 Why does a static library have the `lib` prefix? What suffix does it have? Where are the system's library and include files available?

16.5 The command **cc -c foo.c** compiles without error. Explain why **make** could still generate an error with a makefile that contains the following entry:

```
foo.o: foo.c
    cc -c foo.c
```

16.6 How is **libc.a** different from other libraries?

16.7 Mention two advantages a shared library has over a static library.

16.8 Create a file `foo` containing a line of text. Mention the commands needed to (i) create the SCCS file for `foo`, (ii) check out an editable version of `foo`, (iii) check in `foo` after editing.

16.9 Look up the man page of **sccsdiff**, and then create two deltas of a file. How will you use the command to display their differences?

16.10 Why do we enter comments when using the **delta** command? How can we see them?

16.11 Explain when you need to create a branch delta. Is 1.2.1 a branch delta?

EXERCISES

16.1 How does an executable C program differ from its associated object files?

16.2 It makes no sense to save object files if other programs are not going to use them. Right or wrong?

16.3 Explain the significance of the `-c`, `-o`, `-l`, and `-g` options of the C compiler.

16.4 Modify the application presented in Section 16.1.2 to implement the following:

(i) The **compute** function should be in a separate file, **compute.c**, and its **include** statement in **compute.h**.

(ii) All function prototypes should be defined in a single file, **prototype.h**.

Modify `makefile2` discussed in Section 16.2.2 accordingly.

16.5 Look up the man page of **make** to find out how it can be invoked to display the command line that it would execute without actually executing it.

16.6 Explain how **make** may run without error with a makefile that contains only the following entry. What command will it run?

```
foo:
```

16.7 A **make** rule doesn't always have a dependency, and the target need not be a disk file. Explain with an example of a makefile entry.

16.8 Specify the commands that will (i) create an archive named **foobar.a** containing the object files **foo1.o** and **foo2.o**, (ii) delete **foo2.o** from the archive. Can this archive be used with the `-l` option to **cc**?

16.9 What does this entry in a makefile mean? What command does **make** run if **a.h** is modified?

```
foo.a(a.o): a.h
```

16.10 SCCS and RCS basically use the same mechanism to store file versions, but they work in opposite directions. Explain. Which system do you think reconstructs a version faster?

16.11 You have three versions of a program named **foo1.c**, **foo2.c**, and **foo3.c**. Mention the steps needed to check in all three versions to a single SCCS file.

16.12 Mention the set of commands that produce the deltas 1.1, 1.2, 1.3, and 1.2.1.1 of **foo.c**. Now use **get -r9 s.foo.c** and **get -e -r9 s.foo.c**. What do you observe?

16.13 Which file does **sact** obtain activity information from? When is the file created and deleted?

16.14 You checked out a delta with **get -e s.foo.c** and then realized that you shouldn't have done so. What should you do now and why?

FIGURE 17.10 **atimemtime.c**

```
/* Program: atimemtime.c --
                        Sets a file's time stamps to those of another file */
#include <sys/stat.h>
#include <fcntl.h>
#include <utime.h>                              /* For struct utimbuf */

int main(int argc, char **argv) {
   struct stat statbuf;         /* To obtain time stamps for an existing file */
   struct utimbuf timebuf;      /* To set time stamps for another file */

   arg_check(3, argc, "Two filenames required\n", 1) ;

   if (lstat(argv[1], &statbuf) == -1)
      quit("stat", 1);

   timebuf.actime  = statbuf.st_atime;  /* Setting members of timebuf with */
   timebuf.modtime = statbuf.st_mtime;  /* values obtained from statbuf */

   if (open(argv[2], O_RDWR | O_CREAT, 0644) == -1)
      quit("open", 2);
   close(argv[2]);                  /* Previously used open only to create it */

   if (utime(argv[2], &timebuf) == -1)  /* Sets both time stamps for file */
     quit("utime", 3);                  /* that was just created */

   exit(0);
}
```

The last access time for this file (obtained with **ls -lu**) is generally the time we logged in, and it's a good idea to save this time by creating another file with identical time stamps. We'll move the **a.out** executable to the home directory before running it:

```
$ mv a.out $HOME; cd ; a.out .profile .logintime
$ ls -l .logintime ; ls -lu .logintime
-rw-r--r--   1 sumit    staff          0 Dec 12 20:14 .logintime
-rw-r--r--   1 sumit    staff          0 Feb  2 12:33 .logintime
```

Note that the time stamps for the two files are identical. Using a C program, we have done something that we couldn't do using UNIX commands and the shell.

You can now start writing programs that use these system calls. But we still have some way to go. We must be able to create processes, run programs in them, open files in one process, and pass on the descriptors to the child. We also need to manipulate these descriptors to implement redirection and piping. The programmer's view of the process is presented in Chapter 18.

SUMMARY

A *system call* is a routine built in the kernel to perform a function that requires communication with the hardware. It switches the CPU to *kernel mode* from *user mode*. *Library functions* are built on top of the system calls.

When a system calls returns an error (often, -1), it sets a global variable, `errno`, to an integer whose associated text can be printed with **perror**.

open returns a *file descriptor* that is used by the other I/O calls. It sets the opening *mode* (read, write, etc.) and *status flags* that can create (`O_CREAT`) or truncate (`O_TRUNC`) a file or allow data to be appended (`O_APPEND`).

read and **write** use a buffer whose size should be set equal to the size of the kernel buffer for best performance. **lseek** moves the file offset pointer, and can take it beyond EOF to create a *sparse* file. Unlike **read**, **write** returns immediately even though the actual writing can take place later.

File permissions specified with **open** are modified by either the shell's umask setting or a previous **umask** system call. **umask** returns the previous value of the mask.

Directories are usually handled with library functions because of the nonstandard format of the directory structure. **opendir** returns a pointer to a `DIR` structure that is used by **readdir** to read a directory entry. **readdir** returns a pointer to a `dirent` structure that contains the filename and inode number as its members. **chdir** changes the current directory and **getcwd** returns the pathname of the current directory.

Files can be hard linked (**link**) and symbolically linked (**symlink**), but **unlink** removes both. When **unlink** is invoked on an open file, the kernel removes the directory entry, but its data blocks are deallocated only when the file is closed.

The inode information is maintained in the `stat` structure which is populated by **stat**, **lstat**, and **fstat**. The file type and its permissions are maintained in a single field, `st_mode`, which can be split into separate components using the `S_IFMT` mask. The `S_IS`*xxx* macros can be used to test for specific file types.

access tests a file's access rights (which includes the test for its existence) using the *real* UID and *real* GID of the process.

The **chmod** and **chown** calls do the same jobs as their command counterparts. **utime** is used to set a file's access and modification time stamps using a structure of type `utimbuf`.

SELF-TEST

Use system calls wherever possible. However, you may use **printf**, **perror**, *and the directory handling library functions.*

17.1 Explain the difference between *system calls* and *library functions*. What happens in the CPU when a system call is invoked?

17.2 Why must we *immediately* check the value of `errno` after a system call fails, rather than later?
17.3 What is a *file descriptor*, and what is it used for?
17.4 Check the man pages of **open**, **dup**, **dup2**, **pipe**, and **fcntl**, and see if you find anything they have in common.
17.5 Write a program that ascertains the size of the file descriptor table by opening a file repeatedly. Display the cause of the error that led to the aborting of the program.
17.6 Specify how the **open** call can be used to emulate the function performed by the shell's (i) >, (ii) >> symbols.
17.7 Group the following symbolic constants into two categories and explain the significance of the categories: (i) `O_RDONLY`, (ii) `O_CREAT`, (iii) `O_SYNC`, (iv) `O_RDWR`, (v) `O_TRUNC`, (vi) `O_APPEND`, (vii) `O_WRONLY`. What is the role of the `O_SYNC` flag when using **write**?
17.8 Write a program that uses a filename as argument and display its contents in uppercase.
17.9 Write a program that (i) creates a file `foo` with permissions 666 and a directory `bar` with permissions 777, (ii) removes `foo` and `bar`.
17.10 Write a program that accepts a directory name as argument and creates it if it doesn't already exist. If there is an ordinary file by that name, then the program should remove it before creating the directory.
17.11 Write a program that accepts a directory name as argument, changes to that directory, and displays its absolute pathname. Is the change permanent?
17.12 Write a program that lists from the current directory all ordinary filenames whose size exceeds 100,000 bytes. It should also remove all zero-sized files.
17.13 Write a program that sets the user mask to zero before creating a file `foo`. Now, change the permissions of `foo` to (i) 764, (ii) 440. The previous value of the mask should be restored before program termination.

EXERCISES

Use system calls wherever possible. However, you may use **printf**, **perror**, **strerror**, *and the directory handling library functions.*

17.1 Look up the man page of **strerror** before you write a program that prints all possible error numbers and their corresponding text as shown in Table 17.1. The number of error messages on your system is held in the `extern` variable, `sys_nerr`.
17.2 Explain what an *atomic operation* is. Specify the statement that opens a file and (i) truncates it if it exists, (ii) creates it if it doesn't. What is the advantage of using **open** to create a file instead of **creat** which is designed only for that purpose?
17.3 Write a program that copies a file using the source and destination as arguments. The destination can also be a directory.
17.4 Modify the program in 17.8 (Self-Test) so that the output is displayed in lowercase when invoked by the name **lower**, and uppercase when invoked as **upper**. What else do you need to do to run it?
17.5 Explain why the selection of the buffer size used by **read** and **write** is crucial in writing efficient programs.

17.6 Write two programs that read `/etc/passwd` using (i) a single-character buffer, (ii) a buffer of 2048 bytes with **read**. Use the **time** command with each and compare their performance.
17.7 Write a program that displays the current value of the user mask but leaves it unchanged.
17.8 Using **access**, devise an advisory locking mechanism which allows two programs, **lock1.c** and **lock2.c**, to read a file `foo` only if the file `.lockfile` doesn't exist. Both programs will first create the lock file if it doesn't exist, and remove it before termination.
17.9 Write a program to split the contents of a file specified as argument into multiple files so that each file contains at most 10,000 bytes. Name the files `foo.1`, `foo.2`, and so forth if `foo` is the argument.
17.10 Write a program that uses error checking to perform the following on an existing file `foo`: (i) opens `foo` and then deletes it without closing it, (ii) reads `foo` and displays its output, (iii) opens `foo` again. After the program has completed execution, check whether `foo` has actually been deleted. Explain your observations with reference to the behavior of the **unlink** system call.
17.11 Write a program that moves a group of ordinary files to a directory. The filenames are provided as arguments, and the last argument is a directory. Provide adequate checks to ensure that the files exist as ordinary files and the directory is created if it doesn't exist.
17.12 Write a program that does the following: (i) creates a file `foo` with permissions 644, (ii) assigns write permission to the group, (iii) removes the read permission from others. Look up the **system** library function, and use it to display the listing at each stage.
17.13 Write a program that removes the read, write, and execute permissions for others for all files in the current directory that are owned by the user running the program. (HINT: Use **getuid** to obtain your own UID.)
17.14 Write a program to create a file `fool` with the same permissions, modification time, and access time as another file, `foo2`.
17.15 Write a program that uses a filename as argument and checks each of the 12 permission bits. The program should display a message if the bit is set. For instance, if the user has read permission, then it should display `User-readable`. Develop the code in a modular manner using two separate functions, A and B:

(i) A will populate a `stat` structure with the attributes of the file and print its permissions in octal format as a 4-character string.
(ii) B will extract each permission bit from `stat.st_mode` and then print a message like `User-readable` if the respective bit is set.

To understand how the program works, let's first examine the sequence of statements that are executed in the child process. We first close `fd[0]`, the read end, since the child writes (not reads) to the pipe. Next, we replicate `fd[1]` with **dup2** to give us the descriptor used by standard output. At this stage, the file descriptor for standard output points to the write end of the pipe. This means we don't need the original descriptor (`fd[1]`) that was connected to the same end of the pipe.

Having now closed both the original read and write ends of the pipe, we are left with only the descriptor for standard output that is now connected to the pipe's write end. Invoking **execvp** to run the **cat** command ensures that **cat**'s output is connected to the pipe's write end.

If we apply a similar line of reasoning to the statements in the parent, we'll end up in a situation where the standard input of **tr** is connected to the read end of the pipe. We have been able to establish a pipeline between **cat** and **tr**. On running the program, you should see the entries in `/etc/hosts.equiv`, but after conversion to uppercase:

```
$ a.out
SATURN
EARTH
MERCURY
JUPITER
```

Compare this output with that obtained from the program, **reverse_read.c** (Fig. 17.3), which displayed the contents of `/etc/hosts.equiv` in reverse order.

Two processes can use a pipe for communication only if they share a common ancestor. But UNIX also supports *named pipes* and *sockets* for two unrelated processes to communicate with each other. Besides, SVR4 offers *semaphores, shared memory,* and *message queues* as advanced forms of IPC. Space constraints don't permit inclusion of these topics, but you have a lot to explore on your own.

SUMMARY

A process runs in its own *virtual address space* comprising the text, data, and stack. Part of the process address space is also reserved for the kernel.

The *process table* contains all control information related to a process. The table contains both the *pending signals mask* and the *signal disposition* for every signal that the process may receive.

The environment variables are available in the variable, `environ[]`. A variable can be set with **setenv** or **putenv** and retrieved with **getenv**.

The **fork** system call creates a process by replicating the existing address space. **fork** returns twice—zero in the child and its own PID in the parent. The *exec* family replaces the complete address space of the current process with that of a new program. Shell and **perl** scripts can be run by **execlp** and **execvp**. A successful exec doesn't return.

A process exits by invoking **exit** with an argument that represents the *exit status*. This number is retrieved by the parent with **wait** or **waitpid**. Unlike **wait**, **waitpid** can wait for a specific child to die and also need not block till the child dies.

If the parent dies before the child, the child turns into an *orphan,* which is immediately adopted by **init**. If the parent is alive but doesn't invoke **wait**, the child is transformed into a *zombie*. A zombie is a dead process whose address space has been freed but not the entry in the process table.

The kernel maintains three tables in memory when a file is opened. The *file descriptor table* stores all open descriptors. The *file table* stores the opening mode, status flags and file offset. The *vnode table* contains the inode information. A table is not freed until its reference count drops to zero. A forked child process inherits the descriptors but shares the file table.

dup replicates a file descriptor and returns the lowest unallocated value. **dup2** allows us to choose the descriptor we want by closing it if it is already open. In either case, original and copy both share the same file table. POSIX recommends the use of **fcntl** rather than these two calls.

A *signal* makes a process aware of the occurrence of an event. The process may allow the default disposition to occur, ignore the signal, or invoke a *signal handler*. A handler is *installed* with **sigaction**. The signals `SIGKILL` and `SIGSTOP` can't be caught.

pipe creates a buffered object that returns two file descriptors. Data written to one descriptor is read back from the other. To create a pipeline of two commands, you need to create a pipe before invoking **fork**.

SELF-TEST

18.1 Why do we say that the address space of a process is *virtual?* Which segment of the address space do you think is loaded from the program file?

18.2 When and why does a process voluntarily relinquish control of the CPU?

18.3 What is the value returned by **fork**? Why was it designed to behave that way?

18.4 Name the system calls discussed in this chapter that return one or more file descriptors.

18.5 Write a program that forks twice. Display the PIDs and PPIDs of all three processes.

18.6 Write a program that executes the command **wc -l -c /etc/passwd** using (i) **execl**, (ii) **execv**. What changes do you need to make if you use **execlp** and **execvp** instead?

18.7 Is it necessary for the parent to wait for the death of the child? What happens if it doesn't?

18.8 Write a program that accepts two small numbers (< 50) as arguments and then sums the two in a child process. The sum should be returned by the child to the parent as its exit status, and the parent should print the sum.

18.9 Write a shell script containing only one statement: **exit 123456**. Run the script and then invoke **echo $?** from the shell. Explain why the value provided in the script is different from the output.

18.10 A file may have more than one vnode table in memory. True or false?

18.11 Write a program that uses **write** to output the message `hello dolly` to the standard error. Manipulate the file descriptors so that this message can be saved by using **a.out > foo** rather than **a.out 2>foo**.

18.12 Why can't a background process be terminated with the interrupt key?

18.13 Use the **kill** command to find the number of signals available on your system and then write a program that ignores all of them. Is it possible to do so?
18.14 Explain why a pipe can connect two related processes only.

EXERCISES

18.1 What is the significance of the stack and heap segments in the address space of a process?
18.2 View the man page of **size**, and then run the command on any executable like **/bin/cat**. Explain the significance of the output columns.
18.3 What is the role of the Memory Management Unit in process switching. Why can't one process corrupt the address space of another?
18.4 Write a program that displays all environment variables.
18.5 Write a program that sets an `int` variable `x` to 100 before forking a child. Next perform the following in the child and parent:
Child:

(i) Display the value of `x`, reset it to 200, and display it again.
(ii) Display the value of `PATH`, reset it to only `.`, and display it again.
(iii) Change the current directory to `/etc` and display the absolute pathname of the changed directory.

Parent:

(i) Sleep for 2 seconds.
(ii) Display the value of `x`, `PATH`, and the pathname of the current directory.

Explain your observations. Why was the parent made to sleep for two seconds?
18.6 Redirect the output of **fork.c** (Fig. 18.3) to a file and explain the change in behavior.
18.7 Create a shell script that prints the values of `HOME`, `PATH`, `MAIL`, and `TERM`. Next, write a program that uses exec to run this script so that it prints null values for `MAIL` and `TERM`.
18.8 Explain which of these process attributes change with a fork and exec: (i) PID, (ii) PPID, (iii) file descriptors, (iv) standard I/O buffers.
18.9 The completion of process execution doesn't mean that the process is dead. True or false?
18.10 Write a program where the parent dies after creating a child. Display the value of the PPID in the child. Explain your observations.
18.11 Why are the attributes of an open file held in two tables rather than one?
18.12 Does each entry in the file descriptor table have a separate file table associated with it?
18.13 What are the structural changes that take place in memory when (i) a file is opened twice, (ii) its descriptor is replicated. In how many ways can you replicate the descriptor?
18.14 Modify the program in 18.8 (SELF-TEST) so that process A creates B and B creates C. The summation should be performed in C and the result returned to B as the exit status. B should double the summed value and return the product to A as the exit status. Will the program work with large numbers?

18.15 Name two advantages **waitpid** has over **wait**. How do you use **waitpid** to emulate the behavior of **wait**?
18.16 Explain how the kernel treats zombies and orphans.
18.17 Write a program that repeatedly prints the `Shell>` prompt to accept a UNIX command as argument. The command line, which can't contain shell metacharacters, will be executed by an exec function. The program will terminate when the user enters **exit**. Also try running the program with some of the shell's internal commands (like **umask**) and explain your observations.
18.18 Look up the man page for any shell and understand the significance of the `-c` option. Next, write a program that prompts for a command, which is executed with **exec** and **sh -c**. Try using the program with both external and internal (shell) commands. Does this program behave properly, even when using wild cards and pipes?
18.19 In what ways can a process behave when it receives a signal? What is special about the `SIGSTOP` and `SIGKILL` signals?
18.20 Invoke the command **vi foo &**, and explain why you can't input text to the editor.

19.13.3 Displaying the Archive (-t)

The `-t` key option displays the contents of the device in a long format similar to the listing:

```
# tar -tvf /dev/rdsk/f0q18dt
rwxr-xr-x  203/50    472 Jun  4 09:35  1991 ./dentry1.sh
rwxr-xr-x  203/50    554 Jun  4 09:52  1991 ./dentry2.sh
rwxr-xr-x  203/50   2229 Jun  4 13:59  1991 ./func.sh
```

The files here have been backed up with relative pathnames. Each filename here is preceded by `./`. If you don't remember this but want to extract the file `func.sh` from the diskette, you'll probably first try this:

```
# tar -xvf /dev/rdsk/f0q18dt func.sh
tar: func.sh: Not found in archive
```

tar failed to find the file because it existed there as `./func.sh` and not `func.sh`. Put the `./` before the filename, and you are sure to get it this time. Remember this whenever you encounter extraction errors as above.

19.13.4 Other Options

There are a number of other options of **tar** that are worth considering:

- The `-r` key option is used to append a file to an archive. This implies that an archive can contain several versions of the same file!
- The `-u` key option also adds a file to an archive but only if the file is not already there or is being replaced with a newer version.
- The `-w` option permits interactive copying and restoration. It prints the name of the file and prompts for the action to be taken (`y` or `n`).
- Some versions of **tar** use a special option to pick up filenames from a file. You might want to use this facility when you have a list of over a hundred files, which is impractical (and sometimes, impossible) to enter in the command line. Unfortunately, this option is not standard; Solaris uses `-I` and Linux uses `-T`.

The GNU **tar** command is more powerful than its System V counterpart and supports a host of exclusive options. Unfortunately, there is sometimes a mismatch with the options used by System V. The `-M` option is used for a multivolume backup (e.g., **tar -cvf /dev/fd0H1440 -M *.**). There are two options (-z and -Z) related to compression that we have already discussed *(3.22—Linux)*.

SUMMARY

The system administrator or *superuser* uses the root user account, though any user can also invoke **su** to acquire superuser powers. The superuser can change the attributes of any file, kill any process, and change any user's password. The current directory doesn't feature in `PATH`.

A user is identified by the UID and GID, and root has 0 as the UID. A user can be added (**useradd**), modified (**usermod**), and removed from the system (**userdel**). User details are maintained in `/etc/passwd` and `/etc/group`. The password is stored in an encrypted manner in `/etc/shadow`.

For enforcing security, the administrator may assign a restricted shell so the user can execute only a fixed set of commands. The *set-user-id* (SUID) bit of a program makes its process run with the powers of the program's *owner*. The *sticky bit* set on a directory allows users to create and remove files owned by them in that directory, but not remove or edit files belonging to others.

During system startup, the **init** process reads `/etc/inittab` to run **getty** at all terminals and the system's *rc* scripts. These scripts mount file systems and start the system's daemons. **init** also becomes the parent of all login shells. **shutdown** uses **init** to kill all processes, unmount file systems, and write file system information to disk.

Devices can be *block special* (which use the buffer cache) or *character special* (which don't). A device file is also represented by a *major number* which represents the device driver, and *minor number* which signifies the parameters passed to the device driver. The same device can often be accessed with different filenames.

A UNIX *file system* comprises the *boot block, superblock, inode,* and *data blocks*. The superblock contains global information on the file system, including details of free inodes and data blocks. The memory copies of the superblock and inodes are regularly written to disk by the **update** daemon which calls **sync**.

Most systems today use the ufs file system which permit multiple superblocks, symbolic links and disk quotas. Linux uses the ext2 and ext3 file systems. There are different file system types for CD-ROMs (hsfs or iso9660), DOS disks (pcfs, vfat, or msdos), and a pseudo-file system for processes (proc or procfs).

A file system is unknown to the root file system until it is *mounted* (**mount**). **umount** unmounts file systems but only from above the mount point. **fsck** checks the integrity of file systems.

The administrator has to monitor the disk usage and ensure that adequate free space is available. **df** displays the free disk space for each file system. **du** lists the detailed usage of each file or directory. The administrator also uses **find** to locate large files (`-size`).

Floppy diskettes have to be formatted (**format** or **fdformat**) before they can be used. **dd** uses a character device to copy diskettes and tapes. UNIX provides an entire group of commands to handle DOS diskettes. Their names begin with the string `dos` (SVR4) or `m` (Linux).

tar is suitable for backing up a directory tree. It uses key options for copying to the media (`-c`), restoring from it (`-x`), and displaying the archive (`-t`). GNU **tar** adds compression to the archiving activity.

SELF-TEST

19.1 Where are the administrator's commands primarily located? Which directory is not found in the administrator's `PATH` even though nonprivileged users have it in theirs?

19.2 How does the behavior of the **passwd** command change when invoked by the superuser?

19.3 Two shell variables are assigned by **login** after reading /etc/passwd. What are they?
19.4 Specify the command line that changes romeo's shell from /bin/csh to /bin/bash.
19.5 Why was the password encryption moved from /etc/passwd to /etc/shadow?
19.6 A user after logging in is unable to change directories or create files in her home directory. How can this happen?
19.7 The letters s and t were seen in the permissions field of a listing. What do they indicate?
19.8 Explain the mechanism used by **ls** to display the name of the owner and group owner in the listing.
19.9 How will you use **find** to locate all SUID programs in /bin and /usr/bin?
19.10 What is meant by *run level*? How do you display the run level for your system?
19.11 Which file does **init** take its instructions from? How are the changes made to that file activated?
19.12 Name some services that are not available when the machine is in single-user mode.
19.13 How will you use **shutdown** to bring down the system immediately? What shortcut does Linux offer?
19.14 Mention the significance of the boot and swap file systems.
19.15 Which file system can't be unmounted and why?
19.16 What is the **fsck** command used for?
19.17 What is the difference between the **find** options -perm 1000 and -perm -1000?
19.18 How can the system administrator arrange to monitor the free disk space every hour on a working day between 9 a.m. and 10 p.m.?
19.19 How will you find out the total disk usage of the current directory tree?
19.20 How do you copy all HTML files to a DOS floppy in (i) SVR4, (ii) Linux?
19.21 The command **tar xvf /dev/fd0 *.c** displays an error message even though the diskette contains a number of .c files. Explain the two ways that can lead to the message.

EXERCISES

19.1 Why is the **su** command terminated with **exit**? What is the difference between **su** and **su - romeo**?
19.2 Name five administrative functions that can't be performed by a nonprivileged user.
19.3 Look up the man page of **passwd** to find out how the command can be used to change the password every four weeks.
19.4 How can you create another user with the same powers as root?
19.5 Specify the command lines needed to create a user john with UID 212 and GID dialout (a new group). john will use Bash as his shell and be placed in the /home directory. How can you later change john's shell to Korn without editing /etc/passwd?
19.6 A user romeo belongs to the student group and yet /etc/group doesn't show his name beside the group name. What does that indicate?
19.7 Name five features of the restricted shell.

19.8 How is a user able to update /etc/shadow with **passwd** even though the file doesn't have write permission?
19.9 How will you arrange for a group of users to write to the same directory and yet not be able to remove one another's files?
19.10 What are the two important functions of **init**? Explain how the shell process is created.
19.11 How do you determine the default run level? What is the difference between run levels 0 and 6?
19.12 What is the significance of the *start* and *kill* scripts? How are they organized on (i) an SVR4 system, (ii) Linux?
19.13 Write a shell script that shows whether the printer daemon is running irrespective of whether the system is using SVR4 or Linux.
19.14 Explain what these commands do:

(i) `find / -perm -4000 -print`
(ii) `find / -type d -perm -1000 -exec ls -ld {} \;`
(iii) `find / -type f -size +2048 -mtime +365 -print`

19.15 Why do we install the UNIX system on multiple partitions?
19.16 What is meant by *mounting*? When is unmounting of a file system not possible?
19.17 How do UNIX systems counter superblock corruption?
19.18 Discuss the role of the **sync** command in maintaining the system in a consistent state. When must you not use **sync**?
19.19 Name the important features of the *ufs* file system. What is the significance of the *proc* file system?
19.20 Write a shell script to copy a floppy diskette.
19.21 Specify the **tar** command line that (i) prevents files from being overwritten during restoration, (ii) renames each file interactively during restoration, (iii) appends to an existing archive during copying.
19.22 You need to back up all files that you have worked with today. How do you plan the backup activity using **tar**?