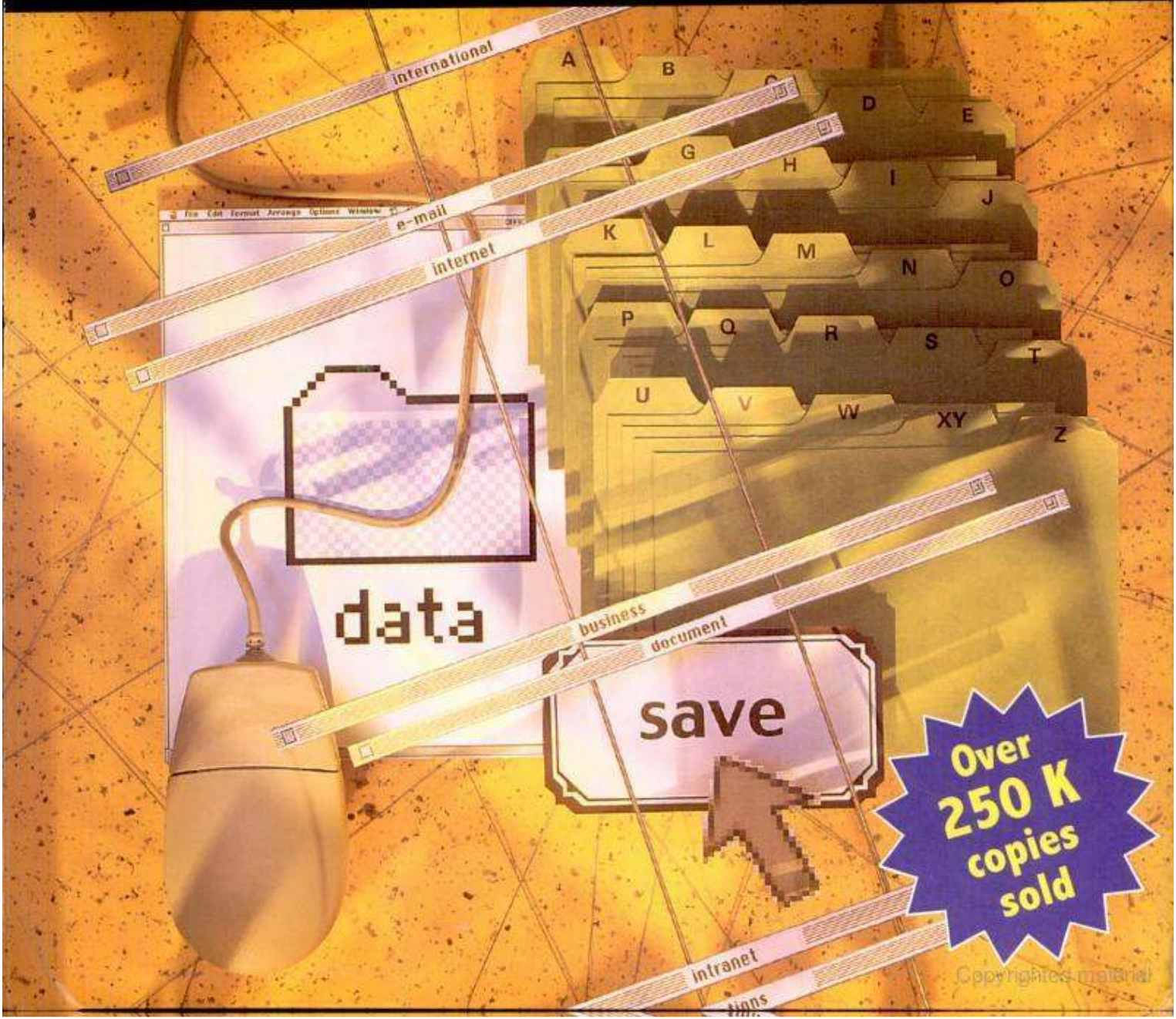
The McGraw-Hill Companies

# UNIX

## Concepts and Applications

FOURTH EDITION

Sumitabha Das



**Tata McGraw-Hill**



**Eighth reprint 2008**
**DCXCRRYXRQRDD**



**ISBN-13: 978-0-07-063546-3**
**ISBN-10: 0-07-063546-3**

Published by Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nagar, New Delhi 110 008, typeset in Times at Script Makers,
19, A1-B, DDA Market, Pashchim Vihar, New Delhi 110 063and printed at
Pashupati Printers Pvt. Ltd., Delhi 110 095

*The McGraw-Hill Companies*

# Conventions Used in This Book

The following conventions have been used in this book:

- Key terms are shown in a large bold font:

  What you see above is an **absolute pathname**, .....

- Commands, program names, user input in examples and system calls are shown in bold constant width font:

  Many commands in **more** including **f** and **b** use a repeat factor.

  The shell features three types of loops—**while**, **until** and **for**.

  Enter your name: **henry**

  The forking mechanism uses the **fork** system call to create a process.

- Apart from command output, filenames, command options, strings, expressions and symbols are shown in constant width font:

  Most commands are located in `/bin` and `/usr/bin`.

  Try doing that with the name `jai sharma`.

  There's adequate scope of using the `-e` and `-f` options.

  The shell looks for the characters >, < and << in the command line.

- Machine and domain names, email addresses and URLs are displayed in italics:

  This copies a file from the remote machine *saturn*.

  User kumar on this host can be addressed as *kumar@calcs.heavens.com*.

  Google offers comprehensive News services at *http://groups.google.com*.

- Placeholders for filenames, terms, menu options and explanatory comments within examples are displayed in italics:

  When compiled with **cc** *filename*, it produces a file named **a.out**.

  .... to develop a set of standard rules (*protocols*)

  Use *Edit>Preferences* to configure Netscape.

  **$ cd ../..** *Moves two levels up*

- The following abbreviations, shortcuts and symbols have been used:

  SVR4—System V Release 4

  `sh`—Bourne shell

  `csh`—C shell

  `ksh`—Korn shell

  `$HOME`/*flname*—The file *flname* in the home directory

  ~/*flname*—The file *flname* in the home directory

  `foo` and `bar`—Generic file and directory names as used on USENET

# Part I

## *For the Beginner ...*

# 1

# Getting Started

This chapter begins the tour of the UNIX universe. To understand UNIX, we'll first have to know what an operating system is, why a computer needs one and how UNIX is vastly different from other operating systems that came before—and after. Through a hands-on session, we'll learn to play with the UNIX system and acquire familiarity with some of its commands that are used everyday for interacting with the system. The experience of the introductory session will help us understand the concepts presented in the next chapter.

UNIX has had a rather turbulent background. Knowing this background will help us understand the objectives that guided its development. Even though UNIX owes its origin to AT&T, contributions made by the academic community and industry have also led to the enrichment and fragmentation of UNIX. The emergence of a windowing (GUI) system, however, has often led users to adopt undesirable shortcuts. For the initiated though, Linux offers a superior and cost-effective solution for the mastery of UNIX.

### *What You Will Learn*

- What an operating system is and how UNIX more than fulfills that role.
- Know the location of the special characters on the keyboard.
- Log in and out of a UNIX system using a username and password.
- Run a few commands like **`date`**, **`who`** and **`cal`**.
- View *processes* with **`ps`**, and *files* with **`ls`**.
- Use a special character like the * to match multiple filenames.
- The role of the *shell* in interpreting these special characters (*metacharacters*).
- A brief background of UNIX and Linux.
- How the Internet contributed to the acceptance of UNIX.

### *Topics of Special Interest*

- Examination of the sequence of steps followed by the shell in executing the command **`ls > list`** to save command output in a file.
- A similar examination of the sequence **`ls | wc`** to connect two commands to form a *pipeline*.

## 1.1 THE OPERATING SYSTEM

We use computers freely, but most of us never bother to know what's inside the box. Why should we? After all, we also use TV and never care to find out how this idiot box manages to convert invisible radio waves to real-life colorful pictures. Yes, you can certainly use spreadsheets and word processors without knowing how these programs access the machine's resources. As long as you continue to get all those reports and charts, do you really need to know anything else?

Then the inevitable happens. The great crash occurs, the machine refuses to boot. The expert tells you that the *operating system* has to be reloaded. You are taken in by surprise. You've heard of software, and you have used lots of them. But what is this thing called the operating system? Is it just another piece of software?

Relax, it is. But it's not just any ordinary software that helps you write letters, but a special one—one that gives life to a machine. Every computer needs some basic intelligence to start with. Unlike mortals, a computer is not born with any. This intelligence is used to provide the essential services for programs that run under its auspices—like using the CPU, allocating memory and accessing devices like the hard disk for reading and writing files.

The computer provides yet another type of service, this time for you—the user. You'll always need to copy or delete a file, or create a directory to house files. You'll need to know the people who are working in the network, or send a mail message to a friend. As a system administrator, you'll also have to back up files. No word processor will do all this for you, neither will your Web browser. All this belongs rightly to the domain of what is known as the *operating system*.

So, what is an operating system? An **operating system** is the software that manages the computer's hardware and provides a convenient and safe environment for running programs. It acts as an interface between programs and the hardware resources that these programs access (like memory, hard disk and printer). It is loaded into memory when a computer is booted and remains active as long as the machine is up.

To grasp the key features of an operating system, let's consider the management tasks it has to perform when we run a program. These operations also depend on the operating system we are using, but the following actions are common to most systems:

- The operating system allocates memory for the program and loads the program to the allocated memory.
- It also loads the CPU registers with control information related to the program. The registers maintain the memory locations where each segment of a program is stored.
- The instructions provided in the program are executed by the CPU. The operating system keeps track of the instruction that was last executed. This enables it to resume a program if it had to be taken out of the CPU before it completed execution.
- If the program needs to access the hardware, it makes a call to the operating system rather than attempt to do the job itself. For instance, if the program needs to read a file on disk, the operating system directs the disk controller to open the file and make the data available to the program.
- After the program has completed execution, the operating system cleans up the memory and registers and makes them available for the next program.

Modern operating systems are *multiprogramming*, i.e. they allow multiple programs to be in memory. However, on computers with a single CPU, only one program can run at any instant. Rather than allow a single program to run to completion without interruption, an operating system generally allows a program to run for a small instant of time, save its current state and then load the next program in the queue. The operating system creates a *process* for each program and then control the switching of these processes.

There have been lots of operating systems in the past, one at least from each hardware vendor. They all contributed in their own way to the chaotic situation that made programs written on one machine totally incapable of running on another. Vendors required consumers to purchase expensive proprietary hardware and software if two dissimilar machines needed to talk to each other. We also had DOS and Windows (in all its manifestations) on our desktop computers providing us with a cheaper and user-friendly way of computing.

## 1.2 THE UNIX OPERATING SYSTEM

Like DOS and Windows, there's another operating system called UNIX. It arrived earlier than the other two, and stayed back late enough to give us the Internet. UNIX is a giant operating system, and is way ahead of them in sheer power. It has practically everything an operating system should have, and several features which other operating systems never had. Its richness and elegance go beyond the commands and tools that constitute it, while simplicity permeates the entire system. It runs on practically every hardware and provided inspiration to the Open Source movement.

However, UNIX also makes many demands of the user. It requires a different type of commitment to understand the subject, even when the user is an experienced computer professional. It introduces certain concepts not known to the computing community before, and uses numerous symbols whose meanings are anything but obvious. It achieves unusual tasks with a few keystrokes, but it takes time to devise a sequence of them for a specific task. Often, it doesn't tell you whether you are right or wrong, and doesn't warn you of the consequences of your actions. That is probably the reason why many people still prefer to stay away from UNIX.

You interact with a UNIX system through a *command interpreter* called the *shell*. Key in a word, and the shell interprets it as a *command* to be executed. A command may already exist on the system as one of several hundred native tools or it could be one written by you. However, the power of UNIX lies in combining these commands in the same way the English language lets you combine words to generate a meaningful idea. As you walk through the chapters of the text, you'll soon discover that this is a major strength of the system.

Kernighan and Pike (*The UNIX Programming Environment*, Prentice-Hall) lamented long ago that "as the UNIX system has spread, the fraction of its users who are skilled in its application has decreased." Many people still use the system as they would use any other operating system, and continue to write comprehensive programs that have already been written before. Beginners with some experience in DOS and Windows think of UNIX in terms of them, quite oblivious of the fact that UNIX has much more to offer. Though references to DOS/Windows have often been made whenever a similar feature was encountered, the similarities end there too. You should not let them get in the way of the UNIX experience.

## 1.3 KNOWING YOUR MACHINE

Unlike DOS and Windows, UNIX can be used by several users concurrently. In other words, a *single* copy of the operating system installed on disk can serve the needs of hundreds of users. If you have access to such a *multiuser* system, then in all probability you'll be sitting with just a terminal or monitor, and a keyboard. Like you, there will be others working on similar terminals. The rest of the equipment will probably be located in a separate room with restricted access. In this arrangement, you are expected to hook on to your account, do your work, disconnect and leave quietly.

Things are quite different, however, when you are the sole user of the system. This could happen if you work on a desktop machine that has its own *CPU* (the Central Processor Unit), *RAM* (Random Access Memory—the memory), hard disk, floppy and CD-ROM drives, printer and the controllers of these devices. If you own the machine, you are directly responsible for its startup, shutdown and maintenance. If you lose a file, it's your job to get it from a backup. If things don't work properly, you have to try all possible means to set them right before you decide to call the maintenance person.

### 1.3.1 The Keyboard

Before you start working, you need to know right now the functions of a number of keys on the keyboard. Many of these keys are either not used by DOS/Windows, or have different functions there. The portion of the keyboard at the left having the *QWERTY* layout resembles your typewriter. You need to be familiar with this section of the keyboard initially, in addition to some other keys in its immediate vicinity. If you know typing, you are on familiar terrain, and keyboard phobia should not get in your way.

Apart from the alphanumeric keys, you'll observe a number of symbols as shown below:

```
` ~ ! @ # $ % ^ & * ( ) - _ = + \ | [ ] { } ; : ' " , . < > / ?
```

Each alphabet, number or symbol is known as a **character**, which represents the smallest piece of information that you can deal with. All of these characters have unique values assigned to them, called the **ASCII value** (ASCII—American Standard Code for Information Interchange). For instance, the letter `A` has the ASCII value 65, while the bang or exclamation mark (`!`) has the value 21.

There are some keys that have no counterparts in the typewriter. Note the key *[Enter]* at the right which is used to terminate a line. On some machines this key may be labeled *[Return]*. The significance of this key is taken up in Section 1.4.4.

When you look at a blank screen, you'll see a blinking object called a **cursor**. When you key in a character, it shows up at the location of the cursor, while moving the cursor itself right. Directly above the *[Enter]* key is the key shown with a ← or labeled *[Backspace]*. You have to press this key to erase one or more characters that you have just entered, using a feature known as **backspacing**. When this key is pressed, the cursor moves over the character placed on its left and removes it from sight.

The prompt here is preceded by the version of the operating system, SunOS 5.8, which is the operating system of Solaris 8. This is a flavor (brand) of UNIX offered by Sun Microsystems, but your system could show a different string here (if at all). The prompt itself could have a prefix showing the machine name. (Yes, every machine has a name in UNIX.)

The login prompt indicates that the terminal is available for someone to **log in** (i.e., connect to the machine). This message also indicates that the previous user has **logged out** (i.e., finished her work and disconnected). Since you now have an account named 'kumar', enter this string at the prompt. Then press the *[Enter]* key after the string:

```
login: kumar[Enter]
Password:
```

The system now requests you to enter the secret code that was handed to you by your administrator. This code should be known to none except yourself. (The administrator doesn't need to know!) Type the secret code and press *[Enter]*:

```
login: kumar
Password:*********[Enter]                    Entry not displayed
```

You may be surprised to observe that the string you entered at the `Password:` prompt isn't displayed on the screen. This is another security feature built into the system that doesn't let someone near you see what you have entered (unless, of course, she has been meticulously monitoring your finger movements!).

If you make mistakes while typing, simply press *[Enter]* one or two times until the login prompt reappears on the screen. Be sure to terminate your responses with *[Enter]* to make the system "see" the input that you have entered.

The string that you entered at the first prompt (`login:`) is known variously as your **login name**, **user-id**, or **username**, and these names will be used interchangeably throughout this book. The secret code that you entered at the next prompt (`Password:`) is known as the **password**. If you enter either of them incorrectly, the system flashes the following message:

```
Login incorrect
login:
```

Another level of security! You simply don't know what went wrong—your login name or your password. The message `Login incorrect` is in fact quite deceptive. In most cases, it's the password that's the culprit. Go back to your secret diary where the password should have been noted and restart the session. When you get both these parameters correct, this is what you could see on a Solaris system:

```
Last login: Thu May  9 06:48:39 from saturn.heavens.com
$ _                                    The cursor shown by the _ character
```

The system now shows the `$` as the **prompt**, with the cursor constantly blinking beside it. This is a typical UNIX prompt, and many UNIX systems use the `$` as the default prompt string. For some users, you might see the `%` instead of the `$`, and the system administrator will in all probability be using the `#`. UNIX allows you to customize the prompt, and it's not unusual to see prompts like these:

**date** is a valid command, and it displays both the date and time. Notice another security feature of UNIX; the command doesn't prompt you to change either the date or time. This facility is available only to the administrator, and the strange thing is that he uses the same command *(15.2.1)* to do it!

So what is **date**? It's one of several hundred programs available in the UNIX system. Whatever you input through the keyboard is interpreted by the system as a **command**, and when you use one, you are in fact commanding the machine to do something. The **date** command instructs the machine to display the current date and time. Incidentally, most UNIX commands are represented as *files* in the system.

---

**Caution:** Tampering with the system date can have adverse effects on a UNIX system. There are many processes that go on in the background without your knowledge, and they are scheduled to start at specific times. If a nonprivileged (ordinary) user (The system administrator is known as the *superuser* simply because he's blessed with enormous powers!) is allowed to change the date and time at will, chaos will ensue.

---

**Note:** Henceforth, we'll use the terms *privileged user*, *superuser* and *system administrator* to refer to the root user account that is used by the administrator for logging in, and *nonprivileged user* to mean all other users. It's often important for us to make this distinction because the root user enjoys certain privileges that are denied others.

---

### 1.4.4 Two Important Observations

You have had your first interaction with the system by running two commands after logging in. Even though one command worked and the other didn't, you had to terminate each by hitting the *[Enter]* key. The text that you type at the terminal remains hidden from the system until this key is pressed. First-time users often fail to appreciate this point because there's no "*[Enter]* key" in the human information system. Humans register speech or text, as it is being spoken or read, in a continuous manner. In this respect, this key resembles the shutter of a camera; nothing gets into the film until the shutter is pressed.

Also note that the completion of a command, successful or otherwise, is indicated by the return of the prompt (here, a $). The presence of the prompt indicates that all work related to the previous command have been completed, and the system is ready to accept the next command. Henceforth, we'll not indicate the *[Enter]* key or the return of the prompt except in not-so-obvious circumstances.

### 1.4.5 tput clear: Clearing the Screen

All UNIX systems offer the **tput** command to clear the screen. This is something you'd often like to do to avoid getting distracted by output or error messages of previous commands. However, when you use **tput** as it is, (i.e., without any additional words), this is what UNIX has to say:

```
$ tput
usage:  tput [-T [term]] capname [parm argument...]
OR:     tput -S <<
```

This message makes little sense to a beginner, so we won't attempt to interpret it right now. However, one thing is obvious; **tput** requires additional input to work properly. To make **tput** work, follow **tput** with the word `clear`:

```
tput clear                                    clear is an argument to tput
```

The screen clears and the prompt and cursor are positioned at the top-left corner. Some systems also offer the **clear** command, but the standard UNIX specifications (like POSIX) don't require UNIX systems to offer this command. You must remember to use **tput clear** to clear the screen because we won't be discussing this command again in this text.

---

**Note:** The additional word used with **tput** isn't a command, but is referred to as an *argument*. Here, `clear` is an argument to **tput**, and the fact that **tput** refused to work alone indicates that it always requires an argument (sometimes more). And, if `clear` is one argument, there could be others. We'll often refer to the *default* behavior of a command to mean the effect of a using a command without any arguments.

---

### 1.4.6 cal: The Calendar

**cal** is a handy tool that you can invoke any time to see the calendar of any specific month, or a complete year. To see the calendar for the month of July, 2006, provide the month number and year as the two arguments to **cal**:

```
$ cal 7 2006                                  Command run with two arguments
      July 2006
Su Mo Tu We Th Fr Sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

With **cal**, you can produce the calendar for any month or year between the years 1 and 9999. This should serve our requirements for some time, right? We'll see more of this command later.

### 1.4.7 who: Who Are the Users?

UNIX is a system that can be concurrently used by multiple users, and you might be interested in knowing the people who are also using the system like you. Use the **who** command:

```
$ who
kumar      console      May  9 09:31    (:0)
vipul      pts/4        May  9 09:31    (:0.0)
raghav     pts/5        May  9 09:32    (saturn.heavens.com)
```

There are currently three users—kumar, vipul and raghav. These are actually the user-ids or usernames they used to log in. The output also includes the username, kumar, which you entered

at the `login:` prompt to gain entry to the system. The second column shows the name of the terminal the user is working on. Just as all users have names, all terminals, disks and printers also have names. You'll see later that these names are represented as *files* in the system. The date and time of login are also shown in the output. Ignore the last column for the time being.

Observe also that the output of **who** doesn't include any headers to indicate what the various columns mean. This is an important feature of the system, and is in some measure responsible for the unfriendly image that UNIX has acquired. After you have completed Chapter 8, you'll discover that it is actually a blessing in disguise.

You logged in with the name kumar, so the system addresses you by this name and associates kumar with whatever work you do. Create a file and the system will make kumar the owner of the file. Execute a program and kumar will be the owner of the *process* (next topic) associated with your program. Send mail to another user and the system will inform the recipient that mail has arrived from kumar.

---

**Note:** UNIX isn't just a repository of commands producing informative output. You can extract useful information from command output for using with other commands. For instance, you can extract the day of the week (here, Fri) from the **date** output and then devise a program that does different things depending on the day the program is invoked. You can also "cut" the user-ids from the **who** output and use the list with the **mailx** command to send mail to all users currently logged in. The facility to perform these useful tasks with one or two lines of code makes UNIX truly different from other operating systems.

---

### 1.4.8 ps: Viewing Processes

We observed that the shell program is always running at your terminal. Every command that you run gives rise to a *process*, and the shell is a process as well. To view all processes that you are responsible for creating, run the **ps** command:

```
$ ps
   PID TTY      TIME CMD
   364 console  0:00 ksh                    Shell running all the time!
```

Unlike **who**, **ps** generates a header followed by a line containing the details of the **ksh** process. When you run several programs, there will be multiple lines in the **ps** output. **ksh** represents the Korn shell (an advanced shell from AT&T) and is constantly running at this terminal. This process has a unique number 364 (called the *process-id* or PID), and when you log out, this process is killed.

---

**Note:** Even though we are using the Korn shell here, you could be using another shell. Instead of **ksh**, you could see **sh** (the primitive Bourne shell), **csh** (C shell—still popular today) or **bash** (Bash shell—a very powerful shell and recommended for use). Throughout this book, we'll be comparing the features of these shells and discover features that are available in one shell but not in another. If a command doesn't produce output as explained in this text, it can often be attributed to the shell.

---

### 1.4.9 ls: Listing Files

Your UNIX system has a large number of files that control its functioning, and users also create files on their own. These files are organized in separate folders called *directories*. You can list the names of the files available in this directory with the **ls** command:

```
$ ls
README                                          Uppercase first
chap01
chap02
chap03
helpdir
progs
```

**ls** displays a list of six files, three of which actually contain the chapters of this textbook. Note that the files are arranged alphabetically with uppercase having precedence over lower (which we call the **ASCII collating sequence**).

Since the files containing the first three chapters have similar filenames, UNIX lets you use a special short-hand notation (*) to access them:

```
$ ls chap*
chap01
chap02
chap03
```

Sometimes, just displaying a list of filenames isn't enough; you need to know more about these files. For that to happen, **ls** has to be used with an *option*, -l, between the command and filenames:

```
$ ls -l chap*                                                    -l is an option
-rw-r--r--   1 kumar    users       5609 Apr 23 09:30 chap01
-rw-r--r--   1 kumar    users      26129 May 14 18:55 chap02
-rw-r--r--   1 kumar    users      37385 May 15 10:30 chap03
```

The argument beginning with a hyphen is known as an *option*. The characteristic feature of most command options is that they begin with a - (hyphen). An option changes the default behavior (i.e. when used without options) of a command, so if **ls** prints a columnar list of files, the -l option makes it display some of the attributes as well.

### 1.4.10 Directing Output to a File

UNIX has simple symbols (called *metacharacters*) for creating and storing information in files. Instead of viewing the output of the **ls** command on the terminal, you can save the information in a file, list, by using a special symbol, > (the right chevron character on your keyboard):

```
$ ls > list
$ _                                    Prompt returns—no display on terminal
```

You see nothing on the terminal except the return of the prompt. The shell is at work here. It has a mechanism of *redirecting* any output, normally coming to the terminal, to a disk file. To check whether the shell has actually done the job, use the **cat** command with the filename as argument:

### 1.4.13 Programming with the Shell

The system also features a programming facility. You can assign a value to a variable at the prompt:

```
$ x=5                                          No spaces on either side of =
$ _
```

and then evaluate the value of this variable with the **echo** command and a $-prefixed variable name:

```
$ echo $x                                      A $ required during evaluation
5
```

Apart from playing with variables, UNIX also provides control structures like conditionals and loops, and you'll see a great deal of that in later chapters.

### 1.4.14 exit: Signing Off

So far, what you have seen is only a small fragment of the UNIX giant, though you have already been exposed to some of its key features. Most of these commands will be considered in some detail in subsequent chapters, and it's a good idea to suspend the session for the time being. You should use the **exit** command to do that:

```
$ exit
login:
```

Alternatively, you may be able to use *[Ctrl-d]* (generated by pressing the *[Ctrl]* key and the character d on the keyboard) to quit the session. The `login:` message confirms that the session has been terminated, thus making it available for the next user.

---

**Note:** Depending on how your environment has been set up, you may not be able to use *[Ctrl-d]* to exit the session. If that happens, try the **logout** command, and if that fails too, use the **exit** command. This command will *always* work.

---

---

**Caution:** Make sure that you log out after your work is complete. If you don't do that, anybody can get hold of your terminal and continue working using your user-id. She may even remove your files! The login prompt signifies a terminated session, so don't leave your place of work until you see this prompt.

---

## 1.5 HOW IT ALL CLICKED

Until UNIX came on the scene, operating systems were designed with a particular machine in mind. They were invariably written in a low-level language (like assembler, which uses humanly unreadable code). The systems were fast but were restricted to the hardware they were designed for. Programs designed for one system simply wouldn't run on another. That was the status of the computer industry when Ken Thompson and Dennis Ritchie, of AT&T fame, authored the UNIX system.

**Note:** The UNIX trademark is owned by The Open Group.

### 1.5.3 The Internet

Even before the advent of SVR4, big things were happening in the U.S. Defense Department. DARPA, a wing of this department, engaged several vendors to develop a reliable communication system using computer technology. Through some brilliant work done by Vinton Cerf and Robert Kahn, DARPA's ARPANET network was made to work using packet-switching technology. In this scenario, data is split into packets, which can take different routes and yet be reassembled in the right order. That was the birth of *TCP/IP*—a set of protocols (rules) used by the Internet for communication.

DARPA commissioned UCB to implement TCP/IP on BSD UNIX. ARPANET converted to TCP/IP in 1983, and in the same year, Berkeley released the first version of UNIX which had TCP/IP built-in. The computer science research community were all using BSD UNIX, and the network expanded like wild fire. The incorporation of TCP/IP into UNIX and its use as the basis of development were two key factors in the rapid growth of the Internet (and UNIX).

### 1.5.4 The Windows Threat

In the meantime, however, Microsoft was making it big with Windows—a *graphical user interface* (GUI) that uses the mouse rather than arcane and complex command options to execute a job. Options could be selected from drop-down menu boxes and radio buttons, which made handling some of the basic operating system functions easier. Windows first swept the desktop market (with Windows 3.1/95/98) and then made significant inroads into the server market (with Windows NT/2000) which had for long been dominated by UNIX.

When UNIX badly needed a Windows-type interface for its survival, the Massachusetts Institute of Technology (MIT) introduced *X Window*—the first windowing system for UNIX. X Window has many of the important features of Microsoft Windows plus a lot more. Every flavor of UNIX now has X along with a host of other tools that can not only handle files and directories but also update the system's configuration files.

**Note:** All said and done, the power of UNIX is derived from its commands and their multiple options. No GUI tool can ever replace the **`find`** command that uses elaborate file attribute-matching schemes to locate files.

## 1.6 LINUX AND GNU

Although UNIX finally turned commercial, Richard Stallman and Linus Torvalds had different ideas. Torvalds is the father of Linux, the free UNIX that has swept the computer world by storm. Stallman runs the Free Software Foundation (formerly known as GNU—a recursive acronym that stands for "GNU's Not Unix"!). Many of the important Linux tools were written and supplied free by GNU.

Linux is distributed under the GNU General Public License which makes it mandatory for developers and sellers to make the source code public. Linux is particularly strong in networking and Internet features, and is an extremely cost-effective solution in setting up an Internet server or a local internet. Today, development on Linux is carried out at several locations across the globe at the behest of the Free Software Foundation.

The most popular GNU/Linux flavors include Red Hat, Caldera, SuSE, Debian and Mandrake. These distributions, which are shipped on multiple CD-ROMs, include a plethora of software—from C and C++ compilers to Java, interpreters like **perl**, **python** and **tcl**, browsers like Netscape, Internet servers, and multimedia software. Much of the software can also be downloaded free from the Internet. All the major computer vendors (barring Microsoft) have committed to support Linux, and many of them have ported their software to this platform. This book also discusses Linux.

## 1.7 CONCLUSION

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."

UNIX is a command-based system, and you have used a number of them already in the hands-on session. These commands have varied usage and often have a large number of options and arguments. Before we take up each UNIX subsystem along with its associated commands, you need to know more about the general characteristics of commands and the documentation associated with them. The next chapter addresses this issue.

### WRAP UP

A computer needs an *operating system* to provide all programs with essential services that involve use of the machine's resources. UNIX is also an operating system but it has more features than is expected out of an operating system.
You enter a UNIX system by entering a username, assigned by the system administrator, and a password. You terminate a session by using the **exit** command or pressing *[Ctrl-d]*.

You can enter any legitimate command at the prompt. The prompt is produced by the *shell*, a program that constantly runs at the terminal as long as the user is logged in.

The **ls** command displays all filenames in a directory. The **ps** command shows the *processes* running in the system.

You can use the * to capture a number of filenames with a simple pattern. The > symbol is used to save the output coming on the screen in a file, while | feeds the output of one command as input to another.

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.

All UNIX flavors today offer a graphical user interface (GUI) in the X Window system. But the strength of UNIX lies in its commands and options.

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

## Test Your Understanding

1.1 Every character has a value associated with it. What is it called?

1.2 Why is the password string not displayed on the terminal?

1.3 The _____ interacts with the hardware and the _____ interacts with the user.

1.4 Name the commands you used in this chapter to display (i) filenames, (ii) processes, (iii) users.

1.5 Enter this sequence: `> user.1st`. Note what happens. Repeat the process by removing the space after the >. Does it make any difference?

1.6 Enter this command sequence: **`ls | wc -l`**. What do you think the output represents?

1.7 Enter the two commands: **`echo *`** and **`ls`**. What do you think **`echo`** did?

1.8 Enter these commands: **`echo "$SHELL"`** and **`echo '$SHELL'`**. What difference do you notice?

1.9 A *program* is synonymous with a *process*. True or False?

1.10 Who are the principal architects of the UNIX operating system?

1.11 Why is UNIX more portable than other operating systems?

1.12 What does X/OPEN represent? Who owns the UNIX trademark today?

1.13 Name some of the contributions made by Berkeley in the development of UNIX.

1.14 What is the windowing system of UNIX known as?

1.15 What is the role of the Free Software Foundation in the development of Linux? Who developed the Linux kernel?

## Flex Your Brain

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 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.

1.4 What exactly happens when you enter this sequence: **`ls > list`**?

1.5 Attempt the variable assignment x=10 by providing a space on either end of the =. Why doesn't it work?

1.6 Run the following commands and then invoke **`ls`**. What do you conclude?

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

1.7 Enter the following commands and note your observations: (i) **`who`** and **`tty`** (ii) **`ps`** and **`echo $$`**

1.8 Name the three commands that you would try in sequence to log yourself out of the system. Which one among them will always work?

1.9 What is the significance of your user-id? Where in the system is the name used?

1.10 What is the one thing that is common to directories, devices, terminals and printers?

1.11 Name some of the duties of the system administrator that you have encountered so far.

1.12 What are the two schools of UNIX that initially guided its development?

- The importance of the POSIX and Single UNIX Specification standards.
- How to use the man documentation in an effective way and interpret the symbols used.
- The use of keyboard sequences to restore normal operation when commands don't work properly.

## 2.1 THE UNIX ARCHITECTURE

The entire UNIX system is supported by a handful of essentially simple, though somewhat abstract concepts. The success of UNIX, according to Thompson and Ritchie, "lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small and yet powerful operating system." UNIX is no longer a small system, but it certainly is a powerful one. Before we examine the features of UNIX, we need to understand its software architecture—its foundation.

### 2.1.1 Division of Labor: Kernel and Shell

Foremost among these "fertile ideas" is the division of labor between two agencies—the *kernel* and *shell*. The kernel interacts with the machine's hardware, and the shell with the user. You have seen both of them in action in the hands-on session though the kernel wasn't mentioned by name. Their relationship is depicted in Fig. 2.1.

The **kernel** is the core of the operating system—a collection of routines mostly written in C. It is loaded into memory when the system is booted and communicates directly with the hardware. User programs (the applications) that need to access the hardware (like the hard disk or terminal) use the services of the kernel, which performs the job on the user's behalf. These programs access the kernel through a set of functions called *system calls*, which are taken up shortly.

Apart from providing support to user programs, the kernel has a great deal of housekeeping to do. It manages the system's memory, schedules processes, decides their priorities, and performs other tasks which you wouldn't like to bother about. The kernel has to do a lot of this work even if no user program is running. It is often called *the* operating system—a program's gateway to the computer's resources.

Computers don't have any inherent capability of translating commands into action. That requires a **command interpreter**, a job that is handled by the "outer part" of the operating system—the **shell**. It is actually the interface between the user and kernel. Even though there's only one kernel running on the system, there could be several shells in action—one for each user who is logged in.

When you enter a command through the keyboard, the shell thoroughly examines the keyboard input for special characters. If it finds any, it rebuilds a simplified command line, and finally communicates with the kernel to see that the command is executed. You have already seen the shell in action when you used the > *(1.4.10)* and | *(1.4.12)* symbols. As a simpler example of how the shell examines and tampers with our input, consider this **echo** command which has lots of spaces between the arguments:

```
$ echo Sun                    Solaris
Sun Solaris
```

### 2.1.2 The File and Process

Two simple entities support the UNIX system—the *file* and *process*—and Kaare Christian (*The UNIX Operating System*, John Wiley) detects two powerful illusions in them: "Files have places and processes have life." Let's briefly examine these two abstractions.

A **file** is just an array of bytes and can contain virtually anything. It is also related to another file by being part of a single hierarchical structure. So you should be able to locate a file with reference to a predetermined place. Further, you as user can also be "placed" at a specific location in this hierarchy (the file system), and you can also "move" from one place to another. This real-life model makes the UNIX file system easily comprehensible.

UNIX doesn't really care to know the type of file you are using. It considers even directories and devices as members of the file system. The dominant file type is text, and the behavior of the system is mainly controlled by text files. UNIX provides a vast array of text manipulation tools that can edit these files without using an editor. The file system, its attributes and text manipulation tools are discussed in several chapters.

The second entity is the **process**, which is the name given to a file when it is executed as a program. You can say that a process is simply the "time image" of an executable file. Like files, processes also belong to a separate hierarchical tree structure. We also treat processes as living organisms which have parents, children and grandchildren, and are born and die. UNIX provides the tools that allow us to control processes, move them between foreground and background, and even kill them. The basics of the process management system are discussed in Chapter 9.

### 2.1.3 The System Calls

The UNIX system—comprising the kernel, shell and applications—is written in C. Though there are over a thousand commands in the system, they all use a handful of functions, called **system calls**, to communicate with the kernel. All UNIX flavors have one thing in common: *They use the same system calls.* These system calls are described in the POSIX specification *(2.3)*. If an operating system uses different system calls, then it won't be UNIX. The reason why Linux can't replace UNIX is that Linux uses the same system calls; Linux is UNIX.

A typical UNIX command writes a file with the **`write`** system call without going into the innards that actually achieve the write operation. Often the same system call can access both a file and a device; the **`open`** system call opens both. These system calls are built into the kernel, and interaction through them represents an efficient means of communication with the system. This also means that once software has been developed on one UNIX system, it can easily be ported to another UNIX machine.

C programmers on a Windows system use the *standard library functions* for everything. You can't use the **`write`** system call on a Windows system; you'll need to use a library function like **`fprintf`** for the purpose. In contrast, the C programmer in the UNIX environment has complete access to the entire system call library as well as the standard library functions. Chapters 23 and 24 deal with the basic system calls that you need to know to program in the UNIX environment.

## 2.2 FEATURES OF UNIX

UNIX is an operating system, so it has all the features an operating system is expected to have. However, UNIX also looks at a few things differently and possesses features unique to itself. The following sections present the major features of this operating system.

### 2.2.1 UNIX: A Multiuser System

From a fundamental point of view, UNIX is a **multiprogramming** system; it permits multiple programs to run and compete for the attention of the CPU. This can happen in two ways:

- Multiple users can run separate jobs.
- A single user can also run multiple jobs.

In fact, you'll see several processes constantly running on a UNIX system. The feature of multiple users working on a single system often baffles Windows users. Windows is essentially a single-user system where the CPU, memory and hard disk are all dedicated to a single user. In UNIX, the resources are actually shared between all users; UNIX is also a **multiuser system**. Multiuser technology is the great socializer that has time for everyone.

For creating the illusory effect, the computer breaks up a unit of time into several segments, and each user is allotted a segment. So at any point in time, the machine will be doing the job of a single user. The moment the allocated time expires, the previous job is kept in abeyance and the next user's job is taken up. This process goes on until the clock has turned full-circle and the first user's job is taken up once again. This the kernel does several times in one second and keeps all ignorant and happy.

### 2.2.2 UNIX: A Multitasking System Too

A single user can also run multiple tasks concurrently; UNIX is a **multitasking** system. It is usual for a user to edit a file, print another one on the printer, send email to a friend and browse the World Wide Web—all without leaving any of the applications. The kernel is designed to handle a user's multiple needs.

In a multitasking environment, a user sees one job running in the *foreground*; the rest run in the *background*. You can switch jobs between background and foreground, suspend, or even terminate them. Programmers can use this feature in a very productive way. You can edit a C program and then suspend it to run the compiler; you don't have to quit the editor to do that. This feature is provided by most shells (except the original Bourne shell).

---

**Note:** Today, we have machines with multiple CPUs that make it possible to actually earmark an entire processor for a single program (in a single-user and single-tasking situation).

---

### 2.2.3 The Building-Block Approach

The designers never attempted to pack too many features into a few tools. Instead, they felt "small is beautiful," and developed a few hundred commands each of which performed one simple job

only. You have already seen how two commands (`ls` and `wc`) were used with the | (pipe) to count the number of files in your directory *(1.4.12)*. No separate command was designed to perform the job. The commands that can be connected in this way are called *filters* because they filter or manipulate data in different ways.

It's through pipes and filters that UNIX implements the small-is-beautiful philosophy. Today, many UNIX tools are designed with the requirement that the output of one tool be used as input to another. That's why the architects of UNIX had to make sure that commands didn't throw out excessive verbiage and clutter the output—one reason why UNIX programs are not interactive. If the output of the `ls` command contained column headers, or if it prompted the user for specific information, this output couldn't have been used as useful input to the `wc` command.

By interconnecting a number of tools, you can have an large number of combinations of their usage. That's why it's better for a command to handle a specialized function rather than solve multiple problems. *Though UNIX started with this concept, it was somewhat forgotten when tools were added to the system later.*

## 2.2.4 The UNIX Toolkit

By one definition, UNIX represents the kernel, but the kernel by itself doesn't do much that can benefit the user. To properly exploit the power of UNIX, you need to use the host of applications that are shipped with every UNIX system. These applications are quite diverse in scope. There are general-purpose tools, text manipulation utilities (called filters), compilers and interpreters, networked applications and system administration tools. You'll also have a choice of shells.

This is one area that's constantly changing with every UNIX release. New tools are being added and the older ones are being removed or modified. The shell and utilities form part of the POSIX specification. There are open-source versions for most of these utilities, and after reading Chapter 22, you should be able to download these tools and configure them to run on your machine.

## 2.2.5 Pattern Matching

UNIX features very sophisticated pattern matching features. You listed the chapters of the text *(1.4.9)* by using the `ls` command with an unusual argument (`chap*`) instead of explicitly specifying all filenames. The * is a special character used by the system to indicate that it can match a number of filenames. If you choose your filenames carefully, you can use a simple expression to access a whole lot of them.

The * (known as a *metacharacter*) isn't the only special character used by the UNIX system; there are several others. UNIX features elaborate pattern matching schemes that use several characters from this metacharacter set. The matching isn't confined to filenames only. Some of the most advanced and useful tools also use a special expression called a *regular expression* that is framed with characters from this set. This book heavily emphasizes the importance of regular expressions, and shows how you can perform complex pattern matching tasks using them.

### 2.2.6 Programming Facility

The UNIX shell is also a programming language; it was designed for a programmer, not a casual end user. It has all the necessary ingredients, like control structures, loops and variables, that establish it as a powerful programming language in its own right. These features are used to design *shell scripts*—programs that can also invoke the UNIX commands discussed in this text.

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, you'll have to know the shell's programming features very well. Proficient UNIX programmers seldom take recourse to any other language (except **perl**) for text manipulation problems. Shell programming is taken up in Chapters 14 and 21.

### 2.2.7 Documentation

UNIX documentation is no longer the sore point it once was. Even though it's sometimes uneven, at most times 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. Thanks to O'Reilly & Associates, one can safely say that there's no feature of UNIX on which a separate textbook is not available. UNIX documentation and the man facility are discussed later in the chapter.

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 fire your queries in case you are stranded with a problem—be it a problem related to shell programming or a network configuration issue. 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. UNIX is easily tamed today.

## 2.3 POSIX AND THE SINGLE UNIX SPECIFICATION

Dennis Ritchie's decision to rewrite UNIX in C didn't quite make UNIX very portable. UNIX fragmentation and the absence of a single conforming standard adversely affected the development of *portable* applications. First, AT&T created the *System V Interface Definition* (SVID). Later, X/Open (now The Open Group), a consortium of vendors and users, created the *X/Open Portability Guide* (XPG). Products conforming to this specification were branded UNIX95, UNIX98 or UNIX03 depending on the version of the specification.

Yet another group of standards, the *Portable Operating System Interface for Computer Environments* (POSIX), were developed at the behest of the Institution of Electrical and Electronics Engineers (IEEE). POSIX refers to operating systems in general, but was based on UNIX. Two of the most-cited standards from the POSIX family are known as *POSIX.1* and *POSIX.2*. POSIX.1 specifies the C application program interface—the system calls. POSIX.2 deals with the shell and utilities.

In 2001, a joint initiative of X/Open and IEEE resulted in the unification of the two standards. This is the *Single UNIX Specification, Version 3* (SUSV3). The "write once, adopt everywhere" approach to this development means that once software has been developed on any POSIX-

When you issue a command, the shell searches this list *in the sequence specified* to locate and execute it. Note that this list also includes the current directory indicated by a singular dot at the end. The following message shows that the **netscape** command is not available in any of these directories:

```
$ netscape
ksh: netscape:  not found
```

The Korn shell is running here and prints the message after failing to locate the file. This doesn't in any way confirm that **netscape** doesn't exist on this system; it could reside in a different directory. In that case we can still run it

- by changing the value of PATH to include that directory.
- by using a pathname (like **/usr/local/bin/netscape** if the command is located in /usr/local/bin).

Windows users also use the same PATH variable to specify the search path, except that Windows uses the ; as the delimiter instead of the colon. We have more to say about pathnames in Chapter 4 and we'll learn to change PATH in Chapter 10.

---

**Note:** The essential UNIX commands for general use are located in the directories /bin and /usr/bin. In Solaris, they are all in /usr/bin.

---

## 2.5 INTERNAL AND EXTERNAL COMMANDS

Since **ls** is a program or file having an independent existence in the /bin directory (or /usr/bin), it is branded as an **external command**. Most commands are external in nature, but there are some which are not really found anywhere, and some which are normally not executed even if they are in one of the directories specified by PATH. Take for instance the **echo** command:

```
$ type echo
echo is a shell builtin
```

**echo** isn't an external command in the sense that, when you type echo, the shell won't look in its PATH to locate it (even if it is there in /bin). Rather, it will execute it from its own set of built-in commands that are not stored as separate files. These built-in commands, of which **echo** is a member, are known as **internal commands**.

You must have noted that it's the shell that actually does all this work. This program starts running for you when you log in, and dies when you log out. The shell is an external command with a difference; it possesses its own set of internal commands. So if a command exists both as an internal command of the shell as well as an external one (in /bin or /usr/bin), the shell will accord top priority to its own internal command of the same name.

This is exactly the case with **echo**, which is also found in /bin, but rarely ever executed because the shell makes sure that the internal **echo** command takes precedence over the external. We'll take up the shell in detail later.

If you use a command with a wrong option, the shell locates the command all right, but the *command* this time finds the option to be wrong:

```
$ ls -z note
ls: illegal option -- z                                    Message from ls, not shell
usage: ls -1RaAdCxmnlogrtucpFbqisfL [files]
```

The above message has been generated by the command, and not by the shell. **ls** does have a large number of options (over 20), but it seems that -z is not one of them. Many commands provide the right syntax and options to use when you use them wrongly.

What you need to keep in mind is something that beginners often forget—the necessity of providing spaces between the command and argument. If you have used **DIR/P** instead of **DIR /P** in DOS, don't expect UNIX to be equally accommodating:

```
$ ls-l
bash: ls-l: command not found
```

Options can normally be combined with only one - sign, i.e., instead of using

```
ls -l -a -t
```

you might as well use

```
ls -lat                                                    Same as ls -l -a -t
```

to obtain the same output. This facility reduces your typing load, which becomes significant when you use commands with several options. The command *parses* (breaks up) the option combination into separate options.

Because UNIX was developed by people who had their own ideas as to what options should look like, there will invariably be exceptions to whatever rules we try to formulate. Some commands won't let you combine options in the way you did just now. There are some that use + as an option prefix instead of -. Some even use the =! Let this not deter you; you would have already built up a lot of muscle before you take on these commands.

## 2.6.2 Filename Arguments

Many UNIX commands use a filename as argument so the command can take input from the file. If a command uses a filename as argument at all, it will generally be its last argument—and after all options. It's also quite common to see many commands working with multiple filenames as arguments:

```
ls -lat chap01 chap02 chap03
cp chap01 chap02 progs                                     cp copies files
rm chap01 chap02                                           rm removes files
```

The command with its arguments and options is known as the **command line**. This line can be considered complete only after the user has hit *[Enter]*. The complete line is then fed to the shell as its input for interpretation and execution.

### 2.6.3 Exceptions

There are, of course, exceptions to the general syntax of commands mentioned above. There are commands (**pwd**) that don't accept any arguments, and some (**who**) that may or may not be specified with arguments. The **ls** command can run without arguments (**ls**), with only options (**ls -l**), with only filenames (**ls chap01 chap02**), or using a combination of both (**ls -la chap01 chap02**). The word *option* turns out to be a misnomer in some instances; some commands compulsorily have to use one (**cut**).

Later on, you'll find that the arguments can take the form of an expression (in **grep**), a set of instructions (in **sed**), or a program (in **awk** and **perl**). You can't really have a catch-all syntax that works for all commands, the syntax pertaining to a specific command is best taken from the UNIX manual. The syntax for some commands have been explicitly specified in this book.

---

**Note:** C programmers and shell scripters need to count the number of arguments in their programs. It helps to be aware at this stage that there are some characters in the command line that are not really arguments—the |, > and <, for instance. In Chapter 8, we'll make an amazing discovery that in the command line **who > user.txt**, `user.txt` is not an argument to **who**!

---

## 2.7 FLEXIBILITY OF COMMAND USAGE

The UNIX system provides a certain degree of flexibility in the usage of commands. A command can often be entered in more than one way, and if you use it judiciously, you can restrict the number of keystrokes to a minimum. In this section, we'll see how permissive the shell is to command usage.

### 2.7.1 Combining Commands

So far, you have been executing commands separately; each command was first processed and executed before the next could be entered. Also, UNIX allows you to specify more than one command in the command line. Each command has to be separated from the other by a ; (semicolon):

```
wc note ; ls -l note
```

When you learn to redirect the output of these commands *(8.5.2)*, you may even like to group them together within parentheses:

```
( wc note ; ls -l note ) > newlist
```

The combined output of the two commands is now sent to the file `newlist`. Whitespace is provided here only for better readability. You might reduce a few keystrokes like this:

```
(wc note;ls -l note)>newlist
```

When a command line contains a semicolon, the shell understands that the command on each side of it needs to be processed separately. The ; here is known as a *metacharacter*, and you'll come across several metacharacters that have special meaning to the shell.

```
User Commands                                   wc(1)
NAME
     wc - display a count of lines, words and characters in a file
SYNOPSIS
     wc [ -c | -m  | -C ]  [ -lw ]  [ file ... ]
DESCRIPTION
     The wc utility  reads  one  or  more  input  files and,  by
     default,  writes the number of newline characters, words and
     bytes contained in  each input file to the standard output.
     The utility also writes a total count for all named  files,
     if more than one input file is specified.
     wc considers a word to be a non-zero-length string  of  char-
     acters  delimited by white space (for example, SPACE, TAB ).
     See iswspace(3C) or isspace(3C).
OPTIONS
     The following options are supported:
     -c        Count bytes.
     -m        Count characters.
     -C        Same as -m.
     -l        Count lines.
     -w        Count words delimited by white space characters or new
               line  characters.  Delimiting  characters are Extended
               Unix Code (EUC) characters from any code  set  defined
               by iswspace().
     If  no option is specified the default is -lwc (count  lines,
     words, and  bytes.)
OPERANDS
     The following operand is supported:
     file  A path name of an input file. If no file operands  are
           specified, the standard input will be used.
USAGE
     See largefile(5) for the description of the behavior of wc when
     encountering files greater than or equal to 2 Gbyte (2 **31 bytes).
EXIT STATUS
     The following exit values are returned:
     0     Successful completion.
     >0    An error occurred.
SEE ALSO
     cksum(1),      isspace(3C),      iswalpha(3C),        iswspace(3C),
     setlocale(3C), attributes(5), environ(5), largefile(5)
```

Fig. 2.2 man page for **wc** (Solaris)

**man** locates its argument from the NAME line of all man pages. **nawk** is a "newer" version of **awk** (has been new for a long time) that is generally found on all modern UNIX systems. Once you know that **awk** is a processing language, you can use **man awk** to view its detailed documentation. Note that both **awk** and **nawk** are found in Section 1. There's a separate chapter in this textbook that discusses **awk**.

Wanting to know what a command does is one thing, but to find out the commands and files associated with a keyword is quite another. What is FTP? Let's use the **apropos** command this time:

```
$ apropos FTP
ftp             ftp (1)         - file transfer program
ftpd            in.ftpd (1m)    - file transfer protocol server
ftpusers        ftpusers (4)    - file listing users to be disallowed ftp login
privileges
in.ftpd         in.ftpd (1m)    - file transfer protocol server
netrc           netrc (4)       - file for ftp remote login data
```

**apropos** lists the commands and files associated with FTP—the protocol used to transfer files between two machines connected in a network. There are two commands here, **ftp** and **in.ftpd**, who cooperate with each other for effecting file transfer. There are also two text files, `ftpusers` and `netrc` (actually, `.netrc`), that are looked up by these commands for authenticating users. Don't worry if you don't understand all this now; we still have some way to go before we take on FTP.

The **whatis** command is also available on many UNIX systems. **man** uses the `-f` option to emulate **whatis** behavior. The command also lists one-liners for a command:

```
$ whatis cp
cp              cp (1)          - copy files
```

This is the command you have to use to copy a file (or directory).

---

**Note:** If you don't have the **apropos** command on your system, you can use **man -k**. You can also use **man -f** in place of **whatis**. The commands search a database that is built separately from man pages. It may or may not be installed on your system. **apropos** and **whatis** are not included in the POSIX specification, but **man -k** is (but not `-f`).

---

### LINUX: The `--help` Option

Some commands have just too many options, and sometimes a quick lookup facility is what you need. Most Linux commands offer the `--help` option that displays a compact listing of all options. You can spot the **find** option you are looking for by using this:

```
$ find --help
Usage: find [path...] [expression]
default path is the current directory; default expression is -print
expression may consist of:
operators (decreasing precedence; -and is implicit where no others are given):
      ( EXPR ) ! EXPR -not EXPR EXPR1 -a EXPR2 EXPR1 -and EXPR2
      EXPR1 -o EXPR2 EXPR1 -or EXPR2 EXPR1 , EXPR2
options (always true): -daystart -depth -follow --help
```

```
      -maxdepth LEVELS -mindepth LEVELS -mount -noleaf --version -xdev
 tests (N can be +N or -N or N): -amin N -anewer FILE -atime N -cmin N
      -cnewer FILE -ctime N -empty -false -fstype TYPE -gid N -group NAME
      -ilname PATTERN -iname PATTERN -inum N -ipath PATTERN -iregex PATTERN
      -links N -lname PATTERN -mmin N -mtime N -name PATTERN -newer FILE
      -nouser -nogroup -path PATTERN -perm [+-]MODE -regex PATTERN
      -size N[bckw] -true -type [bcdpfls] -uid N -used N -user NAME
      -xtype [bcdpfls]
 actions: -exec COMMAND ; -fprint FILE -fprint0 FILE -fprintf FILE FORMAT
      -ok COMMAND ; -print -print0 -printf FORMAT -prune -ls
```

A Linux command invariably offers far more options than its UNIX counterpart. You'll find this lookup facility quite useful when you know the usage of the options but can't recollect the one you require.

## 2.11 WHEN THINGS GO WRONG

Terminals and keyboards have no uniform behavioral pattern. Terminal settings directly impact keyboard operation, and you may sometimes need to check the value of the `TERM` variable. We'll discuss `TERM` later, but as of now, you should at least be able to wriggle out of some common traps. You must know which keys to press when things don't quite work as expected.

*Backspacing Doesn't Work* Consider that you misspelled `passwd` (a legitimate command) as `password`, and when you pressed the backspace key to erase the last three characters, you saw this:

```
$ password^H^H^H
```

Backspacing is not working here; that's why you see the symbol `^H` every time you press the key. This often happens when you log on to a remote machine whose terminal settings are different from your local one. In this case you should try these two key sequences; one of them should see you through:

| | |
|---|---|
| *[Ctrl-h] or [Delete]* | *The erase character* |

*Killing a Line* If the command line contains many mistakes, you could prefer to kill the line altogether without executing it. In that case, use

| | |
|---|---|
| *[Ctrl-u]* | *The line-kill character* |

The line-kill character erases everything in the line and returns the cursor to the beginning of the line.

*Interrupting a Command* Sometimes, a program goes on running for an hour and doesn't seem to complete. You can interrupt the program and bring back the prompt by using either of the two sequences:

| | |
|---|---|
| *[Ctrl-c]* or *[Delete]* | *The interrupt character* |

This is an important key sequence, and in this book, you'll often be advised to use the *interrupt key*. Note, however, that if *[Delete]* works as the erase character on your machine, it can't also be the interrupt character at the same time.

*Terminating a Command's Input* You know that the **cat** command is used with an argument representing the filename *(1.4.10)*. What happens if you omit the filename and simply press *[Enter]*?

```
$ cat[Enter]
```

Nothing happens; the command simply waits for you to enter something. Even if you do some text entry, you must know how to terminate your input. For commands that expect user input, enter a *[Ctrl-d]* to bring back the prompt:

```
$ cat
[Ctrl-d]                                    The end-of-file or eof character
$ _
```

This is another important key sequence; we'll often refer to *[Ctrl-d]* as the *eof* or end-of-file character. Sometimes pressing the interrupt key also works in this situation.

*The Keyboard is Locked* When this happens, you won't be able to key in anything. It could probably be due to accidental pressing of the key sequence *[Ctrl-s]*. Try using *[Ctrl-q]* to release the lock and restore normal keyboard operation. These two sequences are actually used by the system to control the flow of command output.

At times, you may consciously like to use *[Ctrl-s]* and *[Ctrl-q]*. If the display from a command is scrolling too fast for you to see on the terminal, you can halt the output temporarily by pressing *[Ctrl-s]*. To resume scrolling, press *[Ctrl-q]*. With modern hardware where the output scrolls off very fast, this facility is now practically ineffective, but it pays to know what they do because inadvertent pressing of *[Ctrl-s]* can lock your terminal.

*The [Enter] Key Doesn't Work* This key is used to complete the command line. If it doesn't work, you can use either *[Ctrl-j]* or *[Ctrl-m]*. These key sequences generate the linefeed and carriage return characters, respectively.

*The Terminal Behaves in an Erratic Manner* Your terminal settings could be disturbed; it may display everything in uppercase or simply garbage when you press the printable keys. Try using the command **stty sane** to restore sanity. Since the *[Enter]* key may not work either in these situations, use *[Ctrl-j]* or *[Ctrl-m]* to simulate *[Enter]*.

These key functions are summarized in Table 2.2. We have provided names to some of these key sequences (like eof and interrupt), but don't be surprised if you find some of them behaving differently on your system. Much of UNIX is configurable by the user, and you'll learn later to use the **stty** command to change these settings. If you have problems, seek assistance of the system administrator.

---

**Tip:** At this early stage, it may not be possible for you to remember all of these key sequences. But do keep these two keys in mind: *[Ctrl-c]*, the interrupt character, used to interrupt a running program and *[Ctrl-d]*, the eof character, used to terminate a program that's expecting input from the terminal. On machines running Solaris or Linux, *[Ctrl-c]* can interrupt a command even when it is expecting input.

---

Also keep in mind that some UNIX programs (like **mailx**) are interactive and have their own set of internal commands (those understood only by the program). These commands have specific key sequences for termination. You may not remember them, so try using **q**, **quit**, **exit** or *[Ctrl-d]*; one of them might just work.

**Table 2.2** Keyboard Commands to Try When Things Go Wrong

| *Keystroke or Command* | *Function* |
|---|---|
| *[Ctrl-h]* | Erases text (The *erase* character) |
| *[Ctrl-c]* or *[Delete]* | Interrupts a command (The *interrupt* character) |
| *[Ctrl-d]* | Terminates login session or a program that expects its input from the keyboard (The *eof* character) |
| *[Ctrl-s]* | Stops scrolling of screen output and locks keyboard |
| *[Ctrl-q]* | Resumes scrolling of screen output and unlocks keyboard |
| *[Ctrl-u]* | Kills command line without executing it (The *line-kill* character) |
| *[Ctrl-\]* | Kills running command but creates a core file containing the memory image of the program (The *quit* character) |
| *[Ctrl-z]* | Suspends process and returns shell prompt; use **fg** to resume job (The *suspend* character) |
| *[Ctrl-j]* | Alternative to *[Enter]* |
| *[Ctrl-m]* | As above |
| `stty sane` | Restores terminal to normal status (a UNIX command) |

## 2.12 CONCLUSION

This chapter should prepare you well for the forthcoming tour of UNIX. You can now expect to encounter UNIX commands used with a wide variety of options and arguments. The man documentation will be your most valuable help tool and you must develop the habit of looking it up whenever you are stranded with a problem related to command usage. Also, things will go wrong and keyboard sequences won't sometimes work as expected. So don't forget to look up Section 2.11 for remedial action when that happens.

### WRAP UP

The *kernel* addresses the hardware directly. The *shell* interacts with the user. It processes a command, scans it for special characters and rebuilds it in a form that the kernel can understand.

The shell and applications communicate with the kernel using *system calls*, which are special routines built into the kernel.

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

Several users can use the system together (*multiuser*), and a single user can also run multiplejobs concurrently(*multitask*).

- The advantage character-based mailers have over graphic programs.
- The significance of the *mailbox* and *mbox* in the mailing system.

## 3.1 cal: THE CALENDAR

You can invoke the **cal** command to see the calendar of any specific month or a complete year. The facility is totally accurate and takes into account the leap year adjustments that took place in the year 1752. Let's have a look at its syntax drawn from the Solaris man page:

```
cal [  [ month ]  year  ]
```

Everything within rectangular brackets is optional, so we are told *(2.9.1)*. So, **cal** can be used without arguments, in which case it displays the calendar of the current month:

```
$ cal
     August 2005
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

The syntax also tells us that when **cal** is used with arguments, the month is optional but the year is not. To see the calendar for the month of March 2006, you need two arguments:

```
$ cal 03 2006
     March 2006
Su Mo Tu We Th Fr Sa
           1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

You can't hold the calendar of a year in a single screen page; it scrolls off too rapidly before you can use *[Ctrl-s]* to make it pause. To make **cal** pause in the same way **man** pauses, use **cal** with a pager (**more** or **less**) using the | symbol to connect them. A single argument to **cal** is interpreted as the year:

```
cal 2003 | more                                    Or use less instead of more
```

The | symbol connects two commands (in a pipeline) where **more** takes input from the **cal** command. We have used the same symbol in Chapter 1 to connect the **ls** and **wc** commands *(1.4.12)*. You can now scroll forward by pressing the spacebar, or move back using **b**.

## 3.2 date: DISPLAYING THE SYSTEM DATE

The UNIX system maintains an internal clock meant to run perpetually. When the system is shut down, a battery backup keeps the clock ticking. This clock actually stores the number of seconds

Note that the formats also optionally use a number to specify the width that should be used when printing a string or number. You can also use multiple formats in a single **printf** command. But then you'll have to specify as many arguments as there are format strings—and in the right order.

While **printf** can do everything that **echo** does, some of its format strings can convert data from one form to another. Here's how the number 255 is interpreted in octal (base 8) and hexadecimal (base 16):

```
$ printf "The value of 255 is %o in octal and %x in hexadecimal\n" 255 255
The value of 255 is 377 in octal and ff in hexadecimal
```

The %o and %x format strings are also used by **awk** and **perl** (and by C) to convert a decimal integer to octal and hex, respectively; it's good to know them. Note that we specified 255 twice to represent the two arguments because it's the same number that we want to convert to octal and hex.

---

**Note:** C language users should note some syntactical differences in **printf** usage. **printf** is a function in C and hence uses the parentheses to enclose its arguments. Moreover, arguments are separated from one another as well as from the format string by commas. Here's how the previous command line is implemented as a C statement:

```
printf("The value of 255 is %o in octal and %x in hexadecimal\n", 255, 255);
```

The discussion on **printf** should prepare you well for eventually using the **printf** function in C, but remember that this C function uses many format specifiers not used by the UNIX **printf** command.

---

## 3.5 bc: THE CALCULATOR

UNIX provides two types of calculators—a graphical object (the **xcalc** command) that looks like one, and the text-based **bc** command. The former is available in the X Window system and is quite easy to use. The other one is less friendly, extremely powerful and remains one of the system's neglected tools.

When you invoke **bc** without arguments, the cursor keeps on blinking and nothing seems to happen. **bc** belongs to a family of commands (called *filters*) that expect input from the keyboard when used without an argument. Key in the following arithmetic expression and then use *[Ctrl-d]* to quit **bc**:

```
$ bc
12 + 5
17                              Value displayed after computation
[Ctrl-d]                        The eof character
$ _
```

**bc** shows the output of the computation in the next line. Start **bc** again and then make multiple calculations in the same line, using the **;** as delimiter. The output of each computation is, however, shown in a separate line:

```
12*12 ; 2^32                    ^ indicates "to the power of"
144
4294967296                      Maximum memory possible on a 32-bit machine
```

*Sending Mail Noninteractively* Since we often need to send mail from a shell script, we can use a shell feature called *redirection (8.5)* to take the message body from a file and the `-s` option to specify the subject:

```
mailx -s "New System" charlie < message.txt
```

Though POSIX doesn't require **`mailx`** to support options that copy messages to other people, on most systems, **`mailx`** can be used to send copies using the `-c` option. Multiple recipients should be enclosed in quotes:

```
mailx -s "New System" -c "jpm,sumit" charlie < message.txt
```

This command sends a message to charlie with copies to jpm and sumit. Now, pay attention to this: If this command line is placed in a shell script, mail will be sent *without user intervention (no mouse clicks)*. You can now well understand why UNIX is considered to be a versatile system.

---

**Note:** What makes this method of invocation remarkable is that the subject and recipients need not be known in advance, but can be obtained from shell variables. The message body could come from the output of another program. You can use this feature to design automated mailing lists.

---

---

**Tip:** You can send mail to yourself also using your own user-id as argument. Mail sent to oneself in this way serves as a reminder service provided the user is disciplined enough to read all incoming mail immediately on logging in.

---

## 3.8.2 Receiving Mail

All incoming mail is appended to the **mailbox**. This is a text file named after the user-id of the recipient. UNIX systems maintain the mailbox in a directory which is usually `/var/mail` (`/var/spool/mail` in Linux). charlie's mail is appended to `/var/mail/charlie`. By default, **`mailx`** reads this file for viewing received mail.

Referring to the message sent by henry, the shell on charlie's machine regularly checks his mailbox to determine the receipt of new mail. If charlie is currently running a program, the shell waits for program execution to finish before flashing the following message:

```
You have new mail in /var/mail/charlie
```

When charlie logs in, he may also see this message. He now has to invoke the **`mailx`** command in the receiving mode (without using an argument) to see the mail henry has sent him. The system first displays the headers and some credentials of all incoming mail *that's still held in the mailbox*:

```
$ mailx
mailx version 5.0 Wed Jan  5 16:00:40 PST 2000  Type ? for help.
"/var/mail/charlie": 5 messages 2 new 5 unread
U  1 henry@jack.hill.com   Fri Apr  3 16:38  19/567   "sweet dreams"
U  2 MAILER-DAEMON@jack.h  Sat Apr  4 16:33  69/2350  "Warning: could not se"
```

```
U  3 MAILER-DAEMON@jack.h  Thu Apr  9 08:31  63/2066  "Returned mail: Cannot"
N  4 henry@jack.hill.com   Thu Apr 30 10:02  17/515   "Away from work"
>N  5 henry@jack.hill.com   Thu Apr 30 10:39  69/1872  "New System"
? _
```

*The ? prompt*

The pointer (>) is positioned on the fifth message. This is the *current message*. To view the message body, charlie has to input either the message number shown in the second column, or press *[Enter]*. The following message is typically seen on charlie's screen:

```
Message 5:
>From henry@saturn.heavens.com  Tue Jan 13 10:06:14 2003
Date: Tue, 13 Jan 2003 10:06:13 +0530
From: "henry blofeld" <henry@saturn.heavens.com>
To: charlie@saturn.heavens.com
Subject: New System

The new system will start functioning from next month.
Convert your files by next week - henry

? q                                                    Quitting mailx with q
Saved 1 message in /users1/home/staff/charlie/mbox
$ _
```

As we mentioned before, after a message has been seen by the recipient, it moves from the mailbox to the **mbox**, the secondary storage. The name of this file is generally `mbox`, to be found in the user's home directory (where a user is placed on logging in).

---

**Note:** All mail handling commands are presumed to work in a network. That's why the sender's address shows `@saturn.heavens.com` appended. This is the sender's domain name whose significance is taken up in Chapter 17.

---

### 3.8.3 `mailx` Internal Commands

Like the pager used by **man**, the **mailx** command supports a number of internal commands (Table 3.2) that you can enter at the ? prompt. Enter **help** or a **?** at this prompt to see the list of these commands.

You can see the next message by using a **+** (or *[Enter]*), and a **-** to display the previous message. Since messages are numbered, a message can also be accessed by simply entering the number itself:

```
3                                                  Shows message number 3
```

*Replying to mail* The **r** (reply) command enables the recipient to reply when the sent message is on display at her terminal. **mailx** has a way of deducing the sender's details, and consequently, the **r** command is usually not used with an address:

```
? r
To: henry@saturn.heavens.com                  Sender's address automatically inserted
Subject: Re: File Conversion
```

```
I am already through.
[Ctrl-d]
EOT
```

When charlie invokes the **r** command, **mailx** switches to the sending mode. The rules for replying to a message are the same as for sending.

*Saving Messages* Generally, all mail commands act on the current message by default. With the **w** command, you can save one or more messages in separate files rather than the default mbox (the file mbox) used by the system:

```
w note3                    Appends current message to note3
w 1 2 3 note3              Appends first three messages to note3
```

In either case, the message is saved without header information. To save messages with headers, use the **s** command.

*Deleting Mail* To delete a message from the mailbox, use the **d** (delete) command. It simply *marks* mail for deletion; the mail actually gets deleted only when quitting **mailx**.

There's still a lot to know about email. We need to examine mail headers, and domain names that are used in email addresses. We must also understand how multimedia files are sent as part of a mail message. We visit email again in Chapter 17.

**Table 3.2** Internal Commands used by **mailx**

| *Command* | *Action* |
|---|---|
| + | Prints next message |
| - | Prints previous message |
| *N* | Prints message numbered *N* |
| h | Prints headers of all messages |
| d *N* | Deletes message *N* (The current message if *N* is not specified) |
| u *N* | Undeletes message *N* (The current message if *N* is not specified) |
| s *flname* | Saves current message with headers in *flname* ($HOME/mbox if *flname* is not specified) |
| w *flname* | Saves current message without headers in *flname* ($HOME/mbox if *flname* is not specified) |
| m *user* | Forwards mail to *user* |
| r *N* | Replies to sender of message *N* (The current message if *N* is not specified) |
| q | Quits **mailx** program |
| ! *cmd* | Runs UNIX command *cmd* |

## 3.9 passwd: CHANGING YOUR PASSWORD

The remaining commands in this chapter relate to our UNIX system, and we'll first take up the command that changes the user's password. In Chapter 1, you have seen how keying in a wrong password prevents you from accessing the system. If your account doesn't have a password or has

One of the users shown in the first column is obviously the user who invoked the **who** command. To know that specifically, use the arguments `am` and `i` with **who**:

```
$ who am i
kumar      pts/10       Aug  1 07:56    (pc123.heavens.com)
```

---

**Note:** UNIX provides a number of tools (called *filters*) to extract data from command output for further processing. For instance, you can use the **cut** command to extract the first column from the **who** output, and then use this list with **mailx** to send a message to these users. The ability to combine commands to perform tasks that are not possible to achieve using a single command is what makes UNIX so different from other operating systems. We'll be combining commands several times in this text.

---

## 3.11 uname: KNOWING YOUR MACHINE'S CHARACTERISTICS

The **uname** command displays certain features of the operating system running on your machine. By default, it simply displays the name of the operating system:

```
$ uname
SunOS                                          Linux shows Linux
```

This is the operating system used by Sun Solaris. Linux systems simply show the name `Linux`. Using suitable options, you can display certain key features of the operating system, and also the name of the machine. The output will depend on the system you are using.

*The Current Release (-r)* Since UNIX comes in many flavors, vendors have customized a number of commands to behave in the way they want, and not as AT&T decreed. A UNIX command often varies across versions so much so that you'll need to use the `-r` option to find out the version of your operating system:

```
$ uname -r
5.8                                            This is SunOS 5.8
```

This is a machine running SunOS 5.8, the name given to the operating system used by the Solaris 8 environment. If a command doesn't work properly, it could either belong to a different "implementation" (could be BSD) or a different "release" (may be 4.0, i.e., System V Release 4 of AT&T).

*The Machine Name (-n)* If your machine is connected to a network, it must have a name (called *hostname*). If your network is connected to the Internet, then this hostname is a component of your machine's *domain name* (a series of words separated by dots, like *mercury.heavens.com*). The `-n` option tells you the hostname:

```
$ uname -n
mercury                                        The first word of the domain name
```

The same output would be obtained with the **hostname** command. Many UNIX networking utilities use the hostname as argument. To copy files from a remote machine named *mercury* running the FTP service, you have to run **ftp mercury**.

LINUX: `uname -n` may show either the host name (like `mercury`) or the complete domain name (like `mercury.heavens.com`), depending on the flavor of Linux you are using. **`uname`** and **`uname -r`** display the operating system name and version number of the kernel, respectively:

```
$ uname
Linux
$ uname -r
2.4.18-14                                        Kernel version is 2.4
```

The first two numbers of the kernel version (here, 2.4) are something every Linux user must remember. Before installing software, the documentation may require you to use a kernel that is "at least" 2.2. The same software should run on this machine whose kernel version is 2.4.

## 3.12 tty: KNOWING YOUR TERMINAL

Since UNIX treats even terminals as files, it's reasonable to expect a command that tells you the filename of the terminal you are using. It's the **tty** (teletype) command, an obvious reference to the device that has now become obsolete. The command is simple and needs no arguments:

```
$ tty
/dev/pts/10
```

The terminal filename is 10 (a file named 10) resident in the `pts` directory. This directory in turn is under the `/dev` directory. These terminal names were seen on a Solaris machine; your terminal names could be different (say, `/dev/tty01`).

You can use **tty** in a shell script to control the behavior of the script depending on the terminal it is invoked from. If a program must run from only one specified terminal, the script logic must use **tty** to make this decision.

## 3.13 stty: DISPLAYING AND SETTING TERMINAL CHARACTERISTICS

Different terminals have different characteristics, and your terminal may not behave in the way you expect it to. For instance, command interruption may not be possible with *[Ctrl-c]* on your system. Sometimes you may like to change the settings to match the ones used at your previous place of work. The **stty** command helps straighten these things out; it both displays and changes settings.

**stty** uses a very large number of *keywords* (options that look different), but we'll consider only a handful of them. The `-a` (all) option displays the current settings. A trimmed output is presented below:

```
$ stty -a
speed 38400 baud; rows = 25; columns = 80; ypixels = 0; xpixels = 0;
intr = ^c; quit = ^\; erase = ^?; kill = ^u;
eof = ^d; eol = <undef>; eol2 = <undef>; swtch = <undef>;
start = ^q; stop = ^s; susp = ^z; dsusp = ^y;
isig icanon -xcase echo echoe echok -echonl -noflsh
-tostop echoctl -echoprt echoke -defecho -flusho -pendin iexten
```

The output shows, among other things, the baud rate (the speed) of the terminal—in this case 38,400. It also shows many of the parameters that were discussed in a previous chapter *(2.11)*. The other keywords take two forms:

- *keyword* = *value*
- *keyword* or *-keyword*. The - prefix implies that the option is turned off.

The setting `intr = ^c` signifies that *[Ctrl-c]* interrupts a program. The erase character is *[Ctrl-h]* and the kill character is *[Ctrl-u]*. The eof (end-of-file) character is set to *[Ctrl-d]*, the same key sequence that was used with the **bc** *(3.5)* and **mailx** *(3.8)* commands. For commands that accept input from the keyboard, this key signifies the end of input.

Let's understand the significance of some of the other keywords and then use **stty** to change the settings.

## 3.13.1 Changing the Settings

*Whether Backspacing Should Erase Character (*`echoe`*)* If you have worked on a number of terminals, you would have noticed that backspacing over a character sometimes removes it from sight and sometimes doesn't. This is decided by the keyword `echoe`. Since it is set here (no - prefix to it), backspacing removes the character from display.

You can use the same keyword to reverse this setting. Here you need to prefix a - to the `echoe` keyword:

```
stty -echoe
```

Backspacing now doesn't remove a character from sight. This setting is inoperative on some systems.

*Entering a Password through a Shell Script (*`echo`*)* The `echo` setting has to be manipulated to let shell programs accept a password-like string that must not be displayed on the screen. By default, the option is turned on, but you can turn it off in this way:

```
stty -echo                                    Turns off keyboard input
```

With this setting, keyboard entry is not echoed. You should turn it off after the entry is complete by using **stty echo**, which again is not displayed, but makes sure that all subsequent input is.

*Changing the Interrupt Key (*`intr`*)* **stty** also sets the functions for some of the keys. For instance, if you like to use *[Ctrl-c]* as the interrupt key instead of *[Delete]*, you'll have to use

```
stty intr \^c                                 ^ and c
```

Here, the keyword `intr` is followed by a space, the \ (backslash) character, a ^ (caret), and finally the character `c`. This is the way **stty** indicates to the system that the interrupt character is *[Ctrl-c]*.

When you insert control characters into a file, you'll see a ^ symbol apparently prefixed to the character. For example, *[Ctrl-l]* is seen as ^l (or ^L). However, it's actually a single character, occupying two slots on the terminal; no caret is actually present. However, for using a control character in an **stty** setting, you'll have to use a literal caret preceded by a backslash.

3.10 Both your local and remote machines use identical versions of UNIX. How do you confirm whether you are logged on to a remote machine or not?

3.11 How do you determine the erase, kill and eof characters on your system?

3.12 What will you do to ensure that *[Ctrl-c]* interrupts any program? Will it work the next time you log in?

# The File System

In this chapter, we begin our study of one of the two pillars that support UNIX—the file system. UNIX looks at everything as a file and any UNIX system has thousands of files. If you write a program, you add one more file to the system. When you compile it, you add some more. Files grow rapidly, and if they are not organized properly, you'll find it difficult to locate them. Just as an office has separate file cabinets to group files of a similar nature, UNIX also organizes its own files in *directories* and expects you to do that as well.

The file system in UNIX is one of its simple and conceptually clean features. It lets users access other files not belonging to them, but it also offers an adequate security mechanism so outsiders are not able to tamper with a file's contents. In this chapter, you'll learn to create directories, move around within the system, and list filenames in these directories. We'll deal with file attributes, including the ones related to security, in a later chapter.

### *What You Will Learn*

- The initial categorization of files into three types—*ordinary*, *directory* and *device*.
- The features of a UNIX filename.
- The hierarchical structure containing files and directories, and the parent—child relationship that exists between them.
- The significance of the *home directory* and `HOME` variable.
- Navigate the file system with the **`cd`** and **`pwd`** commands.
- Create and remove directories with **`mkdir`** and **`rmdir`**.
- The significance of *absolute* and *relative* pathnames.
- Use **`ls`** to list filenames in a directory in different formats.

### *Topics of Special Interest*

- The significance of the important directories of the UNIX file system from a functional point of view.
- How It Works: A graphic that shows how **`mkdir`** and **`rmdir`** affect a directory.

## 4.1 THE FILE

The **file** is a container for storing information. As a first approximation, we can treat it simply as a sequence of characters. If you name a file `foo` and write three characters `a`, `b` and `c` into it, then `foo` will contain only the string `abc` and nothing else. Unlike the old DOS files, a UNIX file doesn't contain the eof (end-of-file) mark. A file's size is not stored in the file, nor even its name. All file attributes are kept in a separate area of the hard disk, not directly accessible to humans, but only to the kernel.

UNIX treats directories and devices as files as well. A directory is simply a folder where you store filenames and other directories. All physical devices like the hard disk, memory, CD-ROM, printer and modem are treated as files. The shell is also a file, and so is the kernel. And if you are wondering how UNIX treats the main memory in your system, it's a file too!

So we have already divided files into three categories:

- *Ordinary file*—Also known as *regular file*. It contains only data as a stream of characters.
- *Directory file*—It's commonly said that a directory contains files and other directories, but strictly speaking, it contains their *names* and a number associated with each name.
- *Device file*—All devices and peripherals are represented by files. To read or write a device, you have to perform these operations on its associated file.

There are other types of files, but we'll stick to these three for the time being. The reason why we make this distinction between file types is that the significance of a file's attributes often depends on its type. Read permission for an ordinary file means something quite different from that for a directory. Moreover, you can't directly put something into a directory file, and a device file isn't really a stream of characters. While many commands work with all types of files, some don't. For a proper understanding of the file system you must understand the significance of these files.

### 4.1.1 Ordinary (Regular) File

An **ordinary file** or **regular file** is the most common file type. All programs you write belong to this type. An ordinary file itself can be divided into two types:

- Text file
- Binary file

A **text file** contains only printable characters, and you can often view the contents and make sense out of them. All C and Java program sources, shell and **`perl`** scripts are text files. A text file contains lines of characters where every line is terminated with the *newline* character, also known as *linefeed* (LF). When you press *[Enter]* while inserting text, the LF character is appended to every line. You won't see this character normally, but there is a command (**`od`**) which can make it visible.

A **binary file**, on the other hand, contains both printable and unprintable characters that cover the entire ASCII range (0 to 255). Most UNIX commands are binary files, and the object code and executables that you produce by compiling C programs are also binary files. Picture, sound and video files are binary files as well. Displaying such files with a simple **`cat`** command produces unreadable output and may even disturb your terminal's settings.

## 4.7 mkdir: MAKING DIRECTORIES

Directories are created with the **mkdir** (make directory) command. The command is followed by names of the directories to be created. A directory patch is created under the current directory like this:

```
mkdir patch
```

You can create a number of subdirectories with one **mkdir** command:

```
mkdir patch dbs doc                    Three directories created
```

So far, simple enough, but the UNIX system goes further and lets you create directory trees with just one invocation of the command. For instance, the following command creates a directory tree:

```
mkdir pis pis/progs pis/data           Creates the directory tree
```

This creates three subdirectories—pis and two subdirectories under pis. The order of specifying the arguments is important; you obviously can't create a subdirectory before creation of its parent directory. For instance, you can't enter

```
$ mkdir pis/data pis/progs pis
mkdir: Failed to make directory "pis/data"; No such file or directory
mkdir: Failed to make directory "pis/progs"; No such file or directory
```

Note that even though the system failed to create the two subdirectories, progs and data, it has still created the pis directory.

Sometimes, the system refuses to create a directory:

```
$ mkdir test
mkdir: Failed to make directory "test"; Permission denied
```

This can happen due to these reasons:

- The directory test may already exist.
- There may be an ordinary file by that name in the current directory.
- The permissions set for the current directory don't permit the creation of files and directories by the user. You'll most certainly get this message if you try to create a directory in /bin, /etc or any other directory that houses the UNIX system's files.

We'll take up file and directory permissions in Chapter 6 featuring file attributes.

## 4.8 rmdir: REMOVING DIRECTORIES

The **rmdir** (remove directory) command removes directories. You simply have to do this to remove the directory pis:

```
rmdir pis                              Directory must be empty
```

Like **mkdir**, **rmdir** can also delete more than one directory in one shot. For instance, the three directories and subdirectories that were just created with **mkdir** can be removed by using **rmdir** with a reversed set of arguments:

```
rmdir pis/data pis/progs pis
```

Note that when you delete a directory and its subdirectories, a reverse logic has to be applied. The following directory sequence used by **mkdir** is invalid in **rmdir**:

```
$ rmdir pis pis/progs pis/data
rmdir: directory "pis": Directory not empty
```

Have you observed one thing from the error message? **rmdir** has silently deleted the lowest level subdirectories `progs` and `data`. This error message leads to two important rules that you should remember when deleting directories:

- You can't delete a directory *with* **rmdir** unless it is empty. In this case, the `pis` directory couldn't be removed because of the existence of the subdirectories, `progs` and `data`, under it.
- You can't remove a subdirectory unless you are placed in a directory which is hierarchically *above* the one you have chosen to remove.

The first rule follows logically from the example above, but the highlight on **rmdir** has significance that will be explained later. (A directory can also be removed without using **rmdir**.) To illustrate the second cardinal rule, try removing the `progs` directory by executing the command from the same directory itself:

```
$ cd progs
$ pwd
/home/kumar/pis/progs
$ rmdir /home/kumar/pis/progs                    Trying to remove the current directory
rmdir: directory "/home/kumar/pis/progs": Directory does not exist
```

To remove this directory, you must position yourself in the directory above `progs`, i.e., `pis`, and then remove it from there:

```
$ cd /home/kumar/pis
$ pwd
/home/kumar/pis
$ rmdir progs
```

The **mkdir** and **rmdir** commands work only in directories *owned* by the user. Generally, a user is the owner of her home directory, and she can create and remove subdirectories (as well as regular files) in this directory or in any subdirectories created by her. However, she normally won't be able to create or remove files and directories in other users' directories. The concept of ownership will be discussed in Section 6.3.

---

**Note:** A subdirectory can't be removed with **rmdir** unless it's empty, and one is positioned in its parent directory or above it. But we can remove a directory without using **rmdir** also (discussed later).

---

## HOW IT WORKS: How Files and Directories are Created and Removed

As mentioned in Section 4.1.2, a file (ordinary or directory) is associated with a name and a number, called the *inode number*. When a directory is created, an entry comprising these two parameters is made in the file's parent directory. The entry is removed when the directory is removed. The same holds good for ordinary files also. Figure 4.2 highlights the effect of **mkdir** and **rmdir** when creating and removing the subdirectory `progs` in `/home/kumar`.

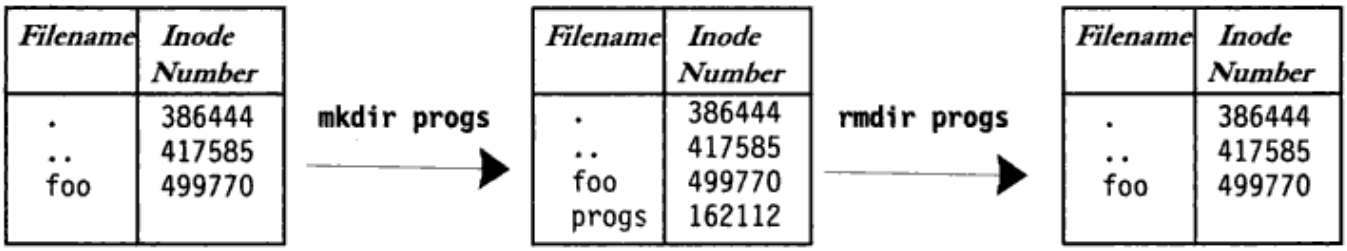



**Fig. 4.2** Directory Entry after **mkdir** and **rmdir**

Later in this chapter, we'll discuss the significance of the entries, `.` and `..`, that you'll find in every directory. In this chapter and Chapters 5 and 11, we'll be progressively monitoring this directory for changes that are caused by some of the file-handling commands.

## 4.9 ABSOLUTE PATHNAMES

Many UNIX commands use file and directory names as arguments, which are presumed to exist in the current directory. For instance, the command

```
cat login.sql
```

will work only if the file `login.sql` exists in your current directory. However, if you are placed in `/usr` and want to access `login.sql` in `/home/kumar`, you can't obviously use the above command, but rather the pathname of the file:

```
cat /home/kumar/login.sql
```

As stated before, if the first character of a pathname is /, the file's location must be determined with respect to root (the first /). Such a pathname, as the one above, is called an **absolute pathname.** When you have more than one / in a pathname, for each such /, you have to descend one level in the file system. Thus, `kumar` is one level below `home`, and two levels below root.

When you specify a file by using frontslashes to demarcate the various levels, you have a mechanism of identifying a file uniquely. No two files in a UNIX system can have identical absolute pathnames. You can have two files with the same name, but in different directories; their pathnames will also be different. Thus, the file `/home/kumar/progs/c2f.pl` can coexist with the file `/home/kumar/safe/c2f.pl`.

### 4.9.1 Using the Absolute Pathname for a Command

More often than not, a UNIX command runs by executing its disk file. When you specify the **date** command, the system has to locate the file `date` from a list of directories specified in the `PATH` variable, and then execute it. However, if you know the location of a particular command, you can precede its name with the complete path. Since **date** resides in `/bin` (or `/usr/bin`), you can also use the absolute pathname:

```
$ /bin/date
Thu Sep  1 09:30:49 IST 2005
```

Nobody runs the **date** command like that. For any command that resides in the directories specified in the `PATH` variable, you don't need to use the absolute pathname. This `PATH`, you'll recall *(2.4.1)*, invariably has the directories `/bin` and `/usr/bin` in its list.

If you execute programs residing in some other directory that isn't in `PATH`, the absolute pathname then needs to be specified. For example, to execute the program **less** residing in `/usr/local/bin`, you need to enter the absolute pathname:

```
/usr/local/bin/less
```

If you are frequently accessing programs in a certain directory, it's better to include the directory itself in `PATH`. The technique of doing that is shown in Section 10.3.

## 4.10 RELATIVE PATHNAMES

You would have noted that in a previous example *(4.8)*, we didn't use an absolute pathname to move to the directory `progs`. Nor did we use one as an argument to **cat** *(4.9)*:

```
cd progs
cat login.sql
```

Here, both `progs` and `login.sql` are presumed to exist in the current directory. Now, if `progs` also contains a directory `scripts` under it, you still won't need an absolute pathname to change to that directory:

```
cd progs/scripts                                    progs is in current directory
```

Here we have a pathname that has a /, but it is not an absolute pathname because it doesn't begin with a /. In these three examples, we used a rudimentary form of relative pathnames though they are generally not labeled as such. Relative pathnames, in the sense they are known, are discussed next.

### 4.10.1 Using . and .. in Relative Pathnames

In a preceding example *(4.8)*, you changed your directory from `/home/kumar/pis/progs` to its parent directory (`/home/kumar/pis`) by using **cd** with an absolute pathname:

```
cd /home/kumar/pis
```

Navigation often becomes easier by using a common ancestor (here, /home) as reference. UNIX offers a shortcut—the **relative pathname**—that uses either the current or parent directory as reference, and specifies the path relative to it. A relative pathname uses one of these cryptic symbols:

- . (a single dot)—This represents the current directory.
- .. (two dots)—This represents the parent directory.

We'll now use the .. to frame relative pathnames. Assuming that you are placed in /home/kumar/progs/data/text, you can use .. as an argument to **cd** to move to the parent directory, /home/kumar/progs/data:

```
$ pwd
/home/kumar/progs/data/text
$ cd ..                                          Moves one level up
$ pwd
/home/kumar/progs/data
```

This method is compact and more useful when ascending the hierarchy. The command **cd ..** translates to this: "Change your directory to the parent of the current directory." You can combine any number of such sets of .. separated by /s. However, when a / is used with .. it acquires a different meaning; instead of moving down a level, it moves one level *up*. For instance, to move to /home, you can always use **cd /home**. Alternatively, you can also use a relative pathname:

```
$ pwd
/home/kumar/pis
$ cd ../..                                       Moves two levels up
$ pwd
/home
```

Now let's turn to the solitary dot that refers to the current directory. Any command which uses the current directory as argument can also work with a single dot. This means that the **cp** command *(5.2)* which also uses a directory as the last argument can be used with a dot:

```
cp ../sharma/.profile .                          A filename can begin with a dot
```

This copies the file .profile to the current directory (.). Note that you didn't have to specify the filename of the copy; it's the same as the original one. This dot is also implicitly included whenever we use a filename as argument, rather than a pathname. For instance, **cd progs** is the same as **cd ./progs**.

---

**Note:** Absolute pathnames can get very long if you are located a number of "generations" away from root. However, whether you should use one depends solely on the number of keystrokes required when compared to a relative pathname. In every case here, the relative pathname required fewer key depressions. Depending on where you are currently placed, an absolute pathname can be faster to type.

---

## 4.11 ls: LISTING DIRECTORY CONTENTS

You have already used the **ls** command *(1.4.9)* to obtain a list of all filenames in the current directory. Let's execute it again:

```
$ ls -x helpdir progs
helpdir:
forms.obd    graphics.obd    reports.obd

progs:
array.pl    cent2fah.pl    n2words.pl    name.pl
```

This time the *contents* of the directories are listed, consisting of the Oracle documentation in the `helpdir` directory and a number of **`perl`** program files in `progs`. Note that **`ls`**, when used with directory names as arguments, doesn't simply show their names as it does with ordinary files.

*Recursive Listing (`-R`)* The `-R` (recursive) option lists all files and subdirectories in a directory tree. Similar to the **`DIR /S`** command of DOS, this traversal of the directory tree is done recursively until there are no subdirectories left:

```
$ ls -xR
08_packets.html  TOC.sh           calendar        cptodos.sh
dept.lst         emp.lst          helpdir         progs
usdsk06x         usdsk07x         usdsk08x        ux2nd06

./helpdir:
forms.hlp        graphics.hlp     reports.hlp

./progs:
array.pl         cent2fah.pl      n2words.pl      name.pl
```

The list shows the filenames in three sections—the ones under the home directory and those under the subdirectories `helpdir` and `progs`. Note the subdirectory naming conventions followed; `./helpdir` indicates that `helpdir` is a subdirectory under `.` (the current directory). Since `/home/kumar` happens to be the current directory, the absolute pathname of this file expands to `/home/kumar/helpdir`.

**Table 4.1** Options to **`ls`**

| *Option* | *Description* |
|---|---|
| `-x` | Multicolumnar output |
| `-F` | Marks executables with *, directories with / and symbolic links with @ |
| `-a` | Shows all filenames beginning with a dot including . and .. |
| `-R` | Recursive list |
| `-r` | Sorts filenames in reverse order (ASCII collating sequence by default) |
| `-l` | Long listing in ASCII collating sequence showing seven attributes of a file *(6.1)* |
| `-d` *dirname* | Lists only *dirname* if *dirname* is a directory *(6.2)* |
| `-t` | Sorts filenames by last modification time *(11.6)* |
| `-lt` | Sorts listing by last modification time *(11.6)* |
| `-u` | Sorts filenames by last access time *(11.6)* |
| `-lu` | Sorts by ASCII collating sequence but listing shows last access time *(11.6)* |
| `-lut` | As above but sorted by last access time *(11.6)* |
| `-i` | Displays inode number *(11.1)* |

## 4.12 THE UNIX FILE SYSTEM

We have learned to use the basic command set for handling files and directories. Let's conclude this chapter by taking a cursory look at the structure of the UNIX file system. This structure has been changing constantly over the years until AT&T proposed one in its SVR4 release. Though vendor implementations vary in detail, broadly the SVR4 structure has been adopted by most vendors.

Refer to Fig. 4.1 which shows a heavily trimmed structure of a standard UNIX file system. In real life, the root directory has many more subdirectories under it than shown, but for our initial comprehension, we'll stick to the ones presented below. It helps, from the administrative point of view at least, to view the entire file system as comprising two groups of files. The first group contains the files that are made available during system installation:

- `/bin` and `/usr/bin`—These are the directories where all the commonly used UNIX commands (binaries, hence the name bin) are found. Note that the `PATH` variable always shows these directories in its list.
- `/sbin` and `/usr/sbin`—If there's a command that you can't execute but the system administrator can, then it would probably be in one of these directories. You won't be able to execute most (some, you can) commands in these directories. Only the system administrator's `PATH` shows these directories.
- `/etc`—This directory contains the configuration files of the system. You can change a very important aspect of system functioning by editing a text file in this directory. Your login name and password are stored in files `/etc/passwd` and `/etc/shadow`.
- `/dev`—This directory contains all device files. These files don't occupy space on disk. There could be more subdirectories like `pts`, `dsk` and `rdsk` in this directory.
- `/lib` and `/usr/lib`—Contain all library files in binary form. You'll need to link your C programs with files in these directories.
- `/usr/include`—Contains the standard header files used by C programs. The statement `#include <stdio.h>` used in most C programs refers to the file `stdio.h` in this directory.
- `/usr/share/man`—This is where the man pages are stored. There are separate subdirectories here (like `man1`, `man2`, etc.) that contain the pages for each section. For instance, the man page of **`ls`** can be found in `/usr/share/man/man1`, where the `1` in `man1` represents Section 1 of the UNIX manual. These subdirectories may have different names on your system (like `sman1`, `sman2`, etc. in Solaris).

Over time, the contents of these directories would change as more software and utilities are added to the system. Users also work with their own files; they write programs, send and receive mail and also create temporary files. These files are available in the second group shown below:

- `/tmp`—The directories where users are allowed to create temporary files. These files are wiped away regularly by the system.
- `/var`—The variable part of the file system. Contains all your print jobs and your outgoing and incoming mail.

- /home—On many systems users are housed here. kumar would have his home directory in /home/kumar. However, your system may use a different location for home directories.

On a busy system, it's in directories belonging to the second group that you could experience rapid depletion of available disk space. You'll learn later to house some of these directory structures on separate *file systems* so that depletion of space in one file system doesn't affect other file systems. File system internals and administration are taken up toward the end of this text.

## 4.13 CONCLUSION

Though UNIX is known to make little distinction between the various types of files, that wasn't really established in this chapter. You used exclusive commands to handle directories (like **pwd**, **cd**, **mkdir** and **rmdir**). These commands have no relevance when applied to ordinary or device files. It appears that UNIX does care to some extent about the type of file it handles. In the next chapter, we look at ordinary files using yet another set of commands meant for them.

### WRAP UP

Files have been assumed to be of three types. An *ordinary* file contains what you put into it. A *directory* maintains the filename and its associated *inode number*. A *device* file contains no data but the kernel uses the attributes of the file to operate the device.

Executable files don't need any specific extensions. A file doesn't contain its attributes nor does it contain the end-of-file mark.

The file system is a hierarchical structure, and the top-most directory is called *root*. Files and directories have a parent—child relationship.

**pwd** tells you the current directory, and **cd** is used to change it. When used by itself, it switches to the *home* directory. The home directory 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.

A pathname is a sequence of directory and filenames separated by slashes. 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.

By default, **ls** displays a list of filenames in *ASCII collating sequence*, which accords priority in this sequence—numbers, uppercase, lowercase. It can also display hidden filenames beginning with a dot (-a) and a recursive list (-R). However, when used with a directory name as argument, **ls** displays the *filenames* in the directory.

## Test Your Understanding

4.1 How long can a UNIX filename be? Which characters can't be used in a filename?

4.2 State two reasons for not having a filename beginning with a hyphen.

4.3 Can the files `note` and `Note` coexist in the same directory?

4.4 In how many ways can you find out what your home directory is?

4.5 Switch to the root directory with **cd** and then run **cd ..** followed by **pwd**. What do you notice?

4.6 What is the easiest way of changing from `/var/spool/lp/admins` to `/var/spool/mail`?

4.7 Explain the significance of these two commands: **ls .. ; ls -d ..**

4.8 Look up the man page of **mkdir** to find out the easiest way of creating this directory structure: `share/man/cat1`

4.9 If **rmdir c_progs** fails, what could be the possible reasons?

4.10 If the file `/bin/echo` exists on your system, are the commands **echo** and **/bin/echo** equivalent?

4.11 How do you run **ls** to (i) mark directories and executables separately, (ii) display also hidden files?

4.12 How will you obtain a complete listing of all files and directories in the whole system?

## Flex Your Brain

4.1 Name the two types of ordinary files and explain the difference between them. Provide three examples of each type of file.

4.2 How does the device file help in accessing the device?

4.3 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**

4.4 If **mkdir test** fails, what could be the possible reasons?

4.5 Which of these files or directories can you create? Explain with reasons: `.`, `..`, `...` and `....`

4.6 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?

4.7 Suppose you have to develop a script that refers to a file in charlie's home directory. How will you specify the location of this file in your script to make sure that it works even when charlie's home directory changes?

4.8 Explain the difference between the commands **cd ~charlie** and **cd ~/charlie**. Is it possible for both commands to work?

4.9 Why do we sometimes run a command like this — **./update.sh** instead of **update.sh**?

4.10 What is the sort order prescribed by the ASCII collating sequence?

4.11 Assuming that you are positioned in the directory `/home/kumar`, what are these commands presumed to do and explain whether they will work at all: (i) **cd ../..** (ii) **mkdir ../bin** (iii) **rmdir ..** (iv) **ls ..**

The AT&T and BSD versions of **more** differ widely in their capabilities and command usage. The POSIX specification on **more** is based on the BSD version. You have to try out the commands shown in Table 5.1, as well as look up the man pages, to know whether they apply to your system. **more** has a fairly useful help screen too; hitting an **h** invokes this screen.

### 5.5.1 Navigation

Irrespective of version, **more** uses the spacebar to scroll forward a page at a time. You can also scroll by small and large increments of lines or screens. To move forward one page, use

```
f or the spacebar
```

and to move back one page, use

```
b
```

### 5.5.2 The Repeat Features

*The Repeat Factor* Many navigation commands in **more**, including **f** and **b**, use a *repeat factor*. This is the term used in **vi** *(7.1.1)* to prefix a number to a **vi** internal command. Use of the repeat factor as a command prefix simply repeats the command that many times. This means you can use **10f** for scrolling forward by 10 pages and **30b** for scrolling back 30 pages. Just remember that the commands themselves are not displayed on the screen—even for a moment.

*Repeating The Last Command (.)* **more** has a repeat command, the dot (same command used by **vi**), that repeats the last command you used. If you scroll forward with **10f**, you can scroll another 10 pages by simply pressing a dot. This is a great convenience available in **more**!

### 5.5.3 Searching for a Pattern

You can perform a search for a pattern with the **/** command followed by the string. For instance, to look for the first **while** loop in your program, you'll have to enter this:

```
/while                                          Press [Enter] also
```

You can repeat this search for viewing the next `while` loop section by pressing **n**, and you can do that repeatedly until you have scanned the entire file. Move back with **b** (using a repeat factor, if necessary) to arrive at the first page.

---

**Note:** The search capability in **more** is not restricted to simple strings. Like many UNIX commands (**grep**, **sed** and **vi**), **more** lets you use a *regular expression* to match multiple similar strings. Regular expressions are discussed in several chapters of this text beginning with Chapter 13.

---

### 5.5.4 Using more in a Pipeline

The **man** syntax doesn't indicate this (except mention that **more** is a filter), but we often use **more** to page the output of another command. The **ls** output won't fit on the screen if there are too many files, so the command has to be used like this:

- The tab character, *[Ctrl-i]*, is shown as \t and the octal value 011.
- The bell character, *[Ctrl-g]*, is shown as 007. Some systems show it as \a.
- The formfeed character, *[Ctrl-l]*, is shown as \f and 014.
- The LF (linefeed or newline) character, *[Ctrl-j]*, is shown as \n and 012. Note that **od** makes the newline character visible too.

Like **wc**, **od** also takes a command's output as its own input, and in Section 5.13, we'll use it to display nonprintable characters in filenames.

## 5.10 cmp: COMPARING TWO FILES

You may often need to know whether two files are identical so one of them can be deleted. There are three commands in the UNIX system that can tell you that. In this section, we'll have a look at the **cmp** (compare) command. Obviously, it needs two filenames as arguments:

```
$ cmp chap01 chap02
chap01 chap02 differ: char 9, line 1
```

The two files are compared byte by byte, and the location of the first mismatch (in the ninth character of the first line) is echoed to the screen. By default, **cmp** doesn't bother about possible subsequent mismatches but displays a detailed list when used with the -l (list) option.

If two files are identical, **cmp** displays no message, but simply returns the prompt. You can try it out with two copies of the same file:

```
$ cmp chap01 chap01
$ _
```

This follows the UNIX tradition of quiet behavior. This behavior is also very important because the comparison has returned a *true* value, which can be subsequently used in a shell script to control the flow of a program.

## 5.11 comm: WHAT IS COMMON?

Suppose you have two lists of people and you are asked to find out the names available in one and not in the other, or even those common to both. **comm** is the command you need for this work. It requires two *sorted* files, and lists the differing entries in different columns. Let's try it on these two files:

```
$ cat file1
c.k. shukla
chanchal singhvi
s.n. dasgupta
sumit chakrobarty
```

```
$ cat file2
anil aggarwal
barun sengupta
c.k. shukla
lalit chowdury
s.n. dasgupta
```

Both files are sorted and have some differences. When you run **comm**, it displays a three-columnar output:

```
$ comm file[12]                          Comparing file1 and file2
        anil aggarwal
        barun sengupta
                c.k. shukla
chanchal singhvi
        lalit chowdury
                s.n. dasgupta
sumit chakrobarty
```

The first column contains two lines unique to the first file, and the second column shows three lines unique to the second file. The third column displays two lines common (hence its name) to both files.

This output provides a good summary to look at, but is not of much use to other commands that take **comm**'s output as their input. These commands require single-column output from **comm**, and **comm** can produce it using the options -1, -2 or -3. To drop a particular column, simply use its column number as an option prefix. You can also combine options and display only those lines that are common:

```
comm -3 foo1 foo2                        Selects lines not common to both files
comm -13 foo1 foo2                       Selects lines present only in second file
```

The last example and one more with the other matching option (-23) has more practical value than you may think, but we'll not discuss their application in this text.

## 5.12 diff: CONVERTING ONE FILE TO OTHER

**diff** is the third command that can be used to display file differences. Unlike its fellow members, **cmp** and **comm**, it also tells you which lines in one file have to be *changed* to make the two files identical. When used with the same files, it produces a detailed output:

```
$ diff file1 file2                       Or diff file[12]
0a1,2                                    Append after line 0 of first file
> anil aggarwal                          this line
> barun sengupta                         and this line
2c4                                      Change line 2 of first file
< chanchal singhvi                       Replacing this line
---                                      with
> lalit chowdury                         this line
4d5                                      Delete line 4 of first file
< sumit chakrobarty                      containing this line
```

**diff** uses certain special symbols and **instructions** to indicate the changes that are required to make two files identical. You should understand these instructions as they are used by the **sed** command, one of the most powerful commands on the system.

Each instruction uses an **address** combined with an **action** that is applied to the first file. The instruction **0a1,2** means appending two lines after line 0, which become lines 1 and 2 in the second file. **2c4** changes line 2 which is line 4 in the second file. **4d5** deletes line 4.

**Tip:** If you are simply interested in knowing whether two files are identical or not, use **cmp** without any options.

*Maintaining Several Versions of a File (-e)* **diff -e** produces a set of instructions only (similar to the above), but these instructions can be used with the **ed** editor (not discussed in this text) to convert one file to the other. This facility saves disk space by letting us store the oldest file in its entirety, and only the changes between consecutive versions. We have a better option of doing that in the *Source Code Control System* (SCCS), but **diff** remains quite useful if the differences are few. SCCS is discussed in Chapter 22.

## 5.13 dos2unix AND unix2dos: CONVERTING BETWEEN DOS AND UNIX

Life being the way it is, you'll encounter DOS/Windows files in the course of your work. Sometimes, you'll need to move files between Windows and UNIX systems. Windows files use the same format as DOS, where the end of line is signified by two characters—CR (\r) and LF (\n). UNIX files, on the other hand, use only LF. Here are two lines from a DOS file, foo, viewed on a UNIX system with the **vi** editor:

```
Line 1^M                                    The [Ctrl-m] character at end
Line 2^M
```

There's a ^M (*[Ctrl-m]*) representing the CR sequence at the end of each line. An octal dump confirms this:

```
$ od -bc foo
0000000 114 151 156 145 040 061 015 012 114 151 156 145 040 062 015 012
          L   i   n   e       1  \r  \n   L   i   n   e       2  \r  \n
```

The CR-LF combination is represented by the octal values 015-012 and the escape sequence \r\n. Conversion of this file to UNIX is just a simple matter of removing the \r. This is often done automatically when downloading a UNIX file from a Windows machine using **ftp**, but sometimes you have to do that job yourself.

For this purpose, some UNIX systems feature two utilities—**dos2unix** and **unix2dos**—for converting files between DOS and UNIX. Sometimes, systems differ in their implementation. This is how we use **dos2unix** to convert this file foo to UNIX format on a Solaris system:

```
dos2unix foo foo.dos
```

The output is written to foo.dos. When you use **od** again, you'll find that the CR character is gone:

```
$ od -bc foo.dos
0000000 114 151 156 145 040 061 012 114 151 156 145 040 062 012
          L   i   n   e       1  \n   L   i   n   e       2  \n
```

One some systems (like Solaris), the first and second filenames could be the same. Others (like Linux) require only one filename, in which case the command rewrites the input file. Some may even require redirection. Browse through the man page to identify the form that works on your machine:

This option can be used for decompression also. To decompress all files in this directory you need to use **gunzip -r progs** or **gzip -dr progs**.

---

**Tip:** To view compressed text files, you really don't need to "gunzip" (decompress) them. Use the **gzcat** and **gzmore** (or **zcat** and **zmore**) commands if they are available on your system. In most cases, the commands run **gunzip -c**.

---

---

**Note:** For some years, **gzip** reigned as the most favored compression agent. Today, we have a better agent in **bzip2** (and **bunzip2**). **bzip2** is slower than **gzip** and creates .bz2 files. We are beginning to see .bz2 files on the Internet. **bzip2** options are modeled on **gzip**, so if you know **gzip** you also know **bzip2**.

---

## 5.16 tar: THE ARCHIVAL PROGRAM

For creating a disk archive that contains a group of files or an entire directory structure, we need to use **tar**. The command is taken up in some detail in Chapter 15 to back up files to tape (or floppy), but in this section we need to know how the command is used to create a disk archive. For this minimal use of **tar** we need to know these *key* options:

| | |
|---|---|
| -c | Create an archive |
| -x | Extract files from archive |
| -t | Display files in archive |
| -f *arch* | Specify the archive *arch* |

Only one these key options can be used at a time. We'll also learn to use **gzip** and **gunzip** to compress and decompress the archive created with **tar**.

### 5.16.1 Creating an Archive (-c)

To create an archive, we need to specify the name of the archive (with -f), the copy or write operation (-c) and the filenames as arguments. Additionally, we'll use the -v (verbose) option to display the progress while **tar** works. This is how we create a file archive, archive.tar, from the two uncompressed files used previously:

```
$ tar -cvf archive.tar libc.html User_Guide.ps
a libc.html 3785K                          -v (verbose) displays list
a User_Guide.ps 364K                       a indicates append
```

By convention, we use the .tar extension, so you'll remember to use the same **tar** command for extraction. We created an archive containing two ordinary files, but **tar** also behaves recursively to back up one or more directories. In the following example, **tar** fills the archive progs.tar with three directory structures:

```
tar -cvf progs.tar c_progs java_progs shell_scripts
```

We'll soon use the same **tar** command to extract files from this archive. But before we do that, let's see how we can compress this archive.

**lp** prints a file and can *directly* print Postscript documents. You can cancel any submitted job with **cancel**. Linux uses the **lpr** command instead of **lp**.

**file** identifies the file type beyond the normal three categories. **wc** counts the number of lines, words and characters. **od** displays the octal value of each character and is used to display invisible characters.

We discussed three file comparison utilities. **cmp** tells you where the first difference was encountered. **comm** shows the lines that are common and optionally shows you lines unique to either or both *sorted* files. **diff** lists file differences as a sequence of **sed**-like instructions.

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.

We also discussed several compression and archival tools. **gzip** and **gunzip** compresses and decompresses individual files (extension—.gz). **tar** always works recursively to archive a group of files into an *archive*. **tar** and **gzip** are often used together to create compressed archives (extension—.tar.gz).

**zip** and **unzip** can perform all functions that are found in **gzip**, **gunzip** and **tar**. **zip** alone can create a compressed archive from directory structures (-r).

## Test Your Understanding

5.1 What will **cat foo foo foo** display?

5.2 How will you copy a directory structure bar1 to bar2? Does it make any difference if bar2 exists?

5.3 Run the command **tty** 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?

5.4 How will you remove a directory tree even when it's not empty and without using **rmdir**?

5.5 How does the command **mv bar1 bar2** behave, where both bar1 and bar2 are directories, when (i) bar2 exists, (ii) bar2 doesn't exist?

5.6 Use the **file** command on all files in the /dev directory. Can you group these files into two categories?

5.7 How do you print three copies of /etc/passwd on the printer named arjun?

5.8 Run the **script** command and then issue a few commands before you run **exit**. What do you see when you run **cat -v typescript**?

5.9 How will you find out the ASCII octal values of the numerals and alphabets?

5.10 How will you display *only* the lines common to two files?

5.11 The command **cmp foo1 foo2** displays nothing. What does it indicate?

## Flex Your Brain

5.1 Describe the contents of a directory, explaining the mechanism by which its entries are updated. Why is the size of a directory usually small?

*File Size* The fifth column shows the size of the file in bytes, i.e., the amount of data it contains. The important thing to remember is that it is only a character count of the file and not a measure of disk space that it occupies. The space occupied by a file on disk is usually larger than this figure since files are written to disk in blocks of 1024 bytes or more. In other words, even though the file `dept.lst` contains 84 bytes, it would occupy 1024 bytes on disk on systems that use a block size of 1024 bytes. We'll discuss the significance of block size much later in the text.

The two directories show smaller file sizes (512 bytes each). This is to be expected as a directory maintains a list of filenames along with an identification number (the inode number) for each file. The size of the directory file depends on the size of this list—whatever be the size of the files themselves.

*Last Modification Time* The sixth, seventh and eighth columns indicate the last modification time of the file, which is stored to the nearest second. A file is said to be modified only if its contents have changed in any way. If you change only the permissions or ownership of the file, the modification time remains unchanged. If the file is less than a year old since its last modification date, the year won't be displayed. Note that the file `genie.sh` has been modified more than a year ago.

You'll often need to run automated tools that make decisions based on a file's modification time. This column shows two other time stamps when **`ls`** is used with other options. The time stamps are discussed in Chapter 11.

*Filename* The last column displays the filenames arranged in ASCII collating sequence. You already know *(4.2)* that UNIX filenames can be very long (up to 255 characters). If you would like to see an important file at the top of the listing, then choose its name in uppercase—at least, its first letter.

The order of the list can be changed by combining the `-l` option with other options. In the rest of the chapter, we'll discuss permissions and ownership, and also learn how to change them.

## 6.2 THE -d OPTION: LISTING DIRECTORY ATTRIBUTES

You'll recall *(4.11.1)* that **`ls`**, when used with directory names, lists files in the directory rather than the directory itself. To force **`ls`** to list the attributes of a directory, rather than its contents, you need to use the `-d` (directory) option:

```
$ ls -ld helpdir progs
drwxr-xr-x   2 kumar    metal         512 May  9 10:31 helpdir
drwxr-xr-x   2 kumar    metal         512 May  9 09:57 progs
```

Directories are easily identified in the listing by the first character of the first column, which here shows a `d`. For ordinary files, this slot always shows a `-` (hyphen), and for device files, either a `b` or `c`. The significance of the attributes of a directory differ a good deal from an ordinary file. Directories will be considered in some detail in Chapter 11.

```
$ cat /usr/bin/startx > xstart                    Actually copies the file startx
$ ls -l xstart
-rw-r--r--   1 kumar    metal        1906 Sep  5 23:38 xstart
```

It seems that, by default, a file doesn't also have execute permission. So how does one execute such a file? To do that, the permissions of the file need to be changed. This is done with **chmod**.

The **chmod** (change mode) command is used to set the permissions of one or more files for all three categories of users (user, group and others). It can be run only by the user (the owner) and the superuser. The command can be used in two ways:

- In a relative manner by specifying the changes to the current permissions.
- In an absolute manner by specifying the final permissions.

We'll consider both ways of using **chmod** in the following sections.

### 6.5.1 Relative Permissions

When changing permissions in a relative manner, **chmod** only changes the permissions specified in the command line *and leaves the other permissions unchanged*. In this mode it uses the following syntax:

chmod *category operation permission filename(s)*

**chmod** takes as its argument an expression comprising some letters and symbols that completely describe the user category and the type of permission being assigned or removed. The expression contains three components:

- User *category* (user, group, others)
- The *operation* to be performed (assign or remove a permission)
- The type of *permission* (read, write, execute)

By using suitable abbreviations for each of these components, you can frame a compact expression and then use it as an argument to **chmod**. The abbreviations used for these three components are shown in Table 6.1.

Now let's consider an example. To assign execute permission to the user (We won't remind again that user here is the owner.) of the file xstart, we need to frame a suitable expression by using appropriate characters from each of the three columns of Table 6.1. Since the file needs to be executable only by the user, the expression required is u+x:

```
$ chmod u+x xstart
$ ls -l xstart
-rwxr--r--   1 kumar    metal        1906 May 10 20:30 xstart
```

The command assigns (+) execute (x) permission to the user (u), but other permissions remain unchanged. You can now execute the file if you are the owner of the file but the other categories (i.e., group and others) still can't. To enable all of them to execute this file, you have to use multiple characters to represent the user category (ugo):

```
$ chmod ugo+x xstart ; ls -l xstart
-rwxr-xr-x   1 kumar    metal        1906 May 10 20:30 xstart
```

The string ugo combines all the three categories—user, group and others. UNIX also offers a shorthand symbol a (all) to act as a synonym for the string. And, as if that wasn't enough, there's an even shorter form that combines these three categories. When it is not specified, the permission applies to all categories. So the previous sequence can be replaced by either of the following:

```
chmod a+x xstart          a implies ugo
chmod +x xstart           By default, a is implied
```

**chmod** accepts multiple filenames in the command line. When you need to assign the same set of permissions to a group of files, this is what you should do:

```
chmod u+x note note1 note3
```

Permissions are removed with the - operator. To remove the read permission from both group and others, use the expression go-r:

```
$ ls -l xstart
-rwxr-xr-x   1 kumar    metal        1906 May 10 20:30 xstart
$ chmod go-r xstart ; ls -l xstart
-rwx--x--x   1 kumar    metal        1906 May 10 20:30 xstart
```

**chmod** also accepts multiple expressions delimited by commas. For instance, to restore the original permissions to the file xstart, you have to remove the execute permission from all (a-x) and assign read permission to group and others (go+r):

```
$ chmod a-x,go+r xstart ; ls -l xstart
-rw-r--r--   1 kumar    metal        1906 May 10 20:30 xstart
```

More than one permission can also be set; **u+rwx** is a valid **chmod** expression. So setting write and execute permissions for others is no problem:

```
$ chmod o+wx xstart ; ls -l xstart
-rw-r--rwx   1 kumar    metal        1906 May 10 20:30 xstart
```

We described relative permissions here, but **chmod** also uses an absolute assignment system, which is taken up in the next topic.

**Table 6.1** Abbreviations Used by **chmod**

| *Category* | *Operation* | *Permission* |
|---|---|---|
| u—User | +—Assigns permission | r—Read permission |
| g—Group | -—Removes permission | w—Write permission |
| o—Others | =—Assigns absolute permission | x—Execute permission |
| a—All (ugo) | | |

To assign all permissions to the owner, read and write permissions to the group, and only execute permission to the others, use this:

```
chmod 761 xstart
```

Now it should be quite easy for you to understand that the expression 777 signifies all permissions for all categories, while 000 indicates absence of all permissions for all categories. But can we delete a file with permissions 000? Yes, we can. Can we prevent a file with permissions 777 from being deleted? We can do that too. We'll soon learn that it's the directory that determines whether a file can be deleted, not the file itself.

---

**Note:** Remember that a file's permissions can only be changed by the owner (understood by **chmod** as user) of the file. One user can't change the protection modes of files belonging to another user. However, the system administrator can tamper with all file attributes including permissions, irrespective of their ownership.

---

### 6.5.3 The Security Implications

To understand the security implications behind these permissions and the role played by **chmod**, consider the default permissions of the file `xstart` that was shown at the beginning of Section 6.5:

```
-rw-r--r--   1 kumar    metal        1906 May 10 20:30 xstart
```

These permissions are fairly safe; only the user can edit the file. What are the implications if we remove all permissions in either of these ways?

```
chmod u-rw,go-r xstart
chmod 000 xstart
```

The listing in either case will look like this:

```
----------   1 kumar    metal        1906 May 10 20:30 xstart
```

This setting renders the file virtually useless; you simply can't do anything useful with it. *But the user can still delete this file!* To understand why that can happen, you need to understand directory permissions and how they are related to file permissions.

On the other hand, you must not be too generous (and careless, too) to have all permissions enabled for all categories of users, using either of these commands:

```
chmod a+rwx xstart
chmod 777 xstart
```

The resulting permissions setting is simply dangerous:

```
-rwxrwxrwx   1 kumar    metal        1906 May 10 20:30 xstart
```

It's the universal write permission here that concerns us most. This file can be written by all. You shouldn't be able to read, write or execute every file. If that were possible, you can never have a

secure system. The UNIX system, by default, never allows that, and no sensible user will compromise security so easily.

---

**Note:** We ignored the directory permissions in our discussions, but they also have a major role to play in setting a file's access rights. No matter how careful you are with your file permissions, a faulty directory permission will affect the security of all files housed in that directory. It doesn't matter who owns the file or whether the file itself has write permission for that user. Directory permissions are taken up later.

---

### 6.5.4 Using chmod Recursively (-R)

It's possible to make **chmod** descend a directory hierarchy and apply the expression to every file and subdirectory it finds. This is done with the `-R` (recursive) option:

```
chmod -R a+x shell_scripts
```

This makes all files and subdirectories found in the tree-walk (that commences from the `shell_scripts` directory) executable by all users. You can provide multiple directory and filenames, and if you want to use **chmod** on your home directory tree, then "cd" to it and use it in one of these ways:

```
chmod -R 755 .          Works on hidden files also
chmod -R a+x *          Leaves out hidden files
```

When you know the shell metacharacters well, you'll appreciate that the * doesn't match filenames beginning with a dot. The dot is generally a safer bet but note that both commands change the permissions of directories also. What do permissions mean when they are applied to a directory? The directory is taken up first in Section 6.6 and then again in Chapter 11.

## 6.6 DIRECTORY PERMISSIONS

Directories also have their own permissions and the significance of these permissions differ a great deal from those of ordinary files. You may not have expected this, but be aware that read and write access to an ordinary file are also influenced by the permissions of the directory housing them. It's possible that a file can't be accessed even though it has read permission, and can be removed even when it's write-protected. In fact, it's very easy to make it behave that way.

If the default directory permissions are not altered, the **chmod** theory still applies. However, if they are changed, unusual things can happen. Though directory permissions are taken up later *(11.4)*, it's worthwhile to know what the default permissions are on your system:

```
$ mkdir c_progs; ls -ld c_progs
drwxr-xr-x   2 kumar    metal         512 May  9 09:57 c_progs
```

The default permissions of a directory on this system are `rwxr-xr-x` (or 755); that's what it should be. *A directory must never be writable by group and others.* If you find that your files are being tampered with even though they appear to be protected, check up the directory permissions. If the permissions differ from what you see here, look up Chapter 11 for remedial action.

**Caution:** If a directory has write permission for group and others also, be assured that every user can remove every file in the directory! As a rule, you must not make directories universally writable unless you have definite reasons to do so.

**Note:** The default file and directory permissions on your machine could be different from what has been assumed here. The defaults are determined by the *umask* setting of your shell. This topic is discussed in Section 11.5.

## 6.7 CHANGING FILE OWNERSHIP

File ownership is a feature often ignored by many users. By now you know well enough that when a user kumar of the metal group creates a file `foo`, he becomes the owner of `foo`, and metal becomes the group owner. It's only kumar who can change the major file attributes like its permissions and group ownership. No member of the metal group (except kumar) can change these attributes. However, when sharma copies `foo`, the ownership of the copy is transferred to sharma, and he can then manipulate the attributes of the copy at will.

There are two commands meant to change the ownership of a file or directory—**chown** and **chgrp**. UNIX systems differ in the way they restrict the usage of these two commands. On BSD-based systems, only the system administrator can change a file's owner with **chown**. On the same systems, the restrictions are less severe when it comes to changing groups with **chgrp**. On other systems, only the owner can change both.

### 6.7.1 chown: Changing File Owner

We'll first consider the behavior of BSD-based **chown** (change owner) that has been adopted by many systems including Solaris and Linux. The command is used in this way:

```
chown options owner [:group] file(s)
```

**chown** transfers ownership of a file to a user, and it seems that it can optionally change the group as well. The command requires the user-id (UID) of the recipient, followed by one or more filenames. Changing ownership requires superuser permission, so let's first change our status to that of superuser with the **su** command:

```
$ su
Password: ********                    This is the root password!
# _                                   This is another shell
```

After the password is successfully entered, **su** returns a # prompt, the same prompt used by root. **su** lets us acquire superuser status if we know the root password. To now renounce the ownership of the file `note` to sharma, use **chown** in the following way:

```
# ls -l note
-rwxr----x   1 kumar    metal        347 May 10 20:30 note
# chown sharma note ; ls -l note
-rwxr----x   1 sharma   metal        347 May 10 20:30 note
```

## Flex Your Brain

6.1 How will you obtain a complete listing of all files and directories in the whole system and save the output in a file?

6.2 Explain briefly the significance of the first four fields of the **ls -l** output. Who can change these attributes? Is there any attribute that can be changed *only* by the superuser?

6.3 Explain the significance of the following commands: (i) **ls -ld .** (ii) **ls -l ..**

6.4 The commands **ls bar** and **ls -d bar** display the same output—the string bar. This can happen in two ways. Explain.

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

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

6.7 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?

6.8 Explain the consequences, from the security viewpoint, of a file having the permissions (i) 000 (ii) 777. Assume that the directory has write permission.

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

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

6.10 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.

6.11 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**?

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

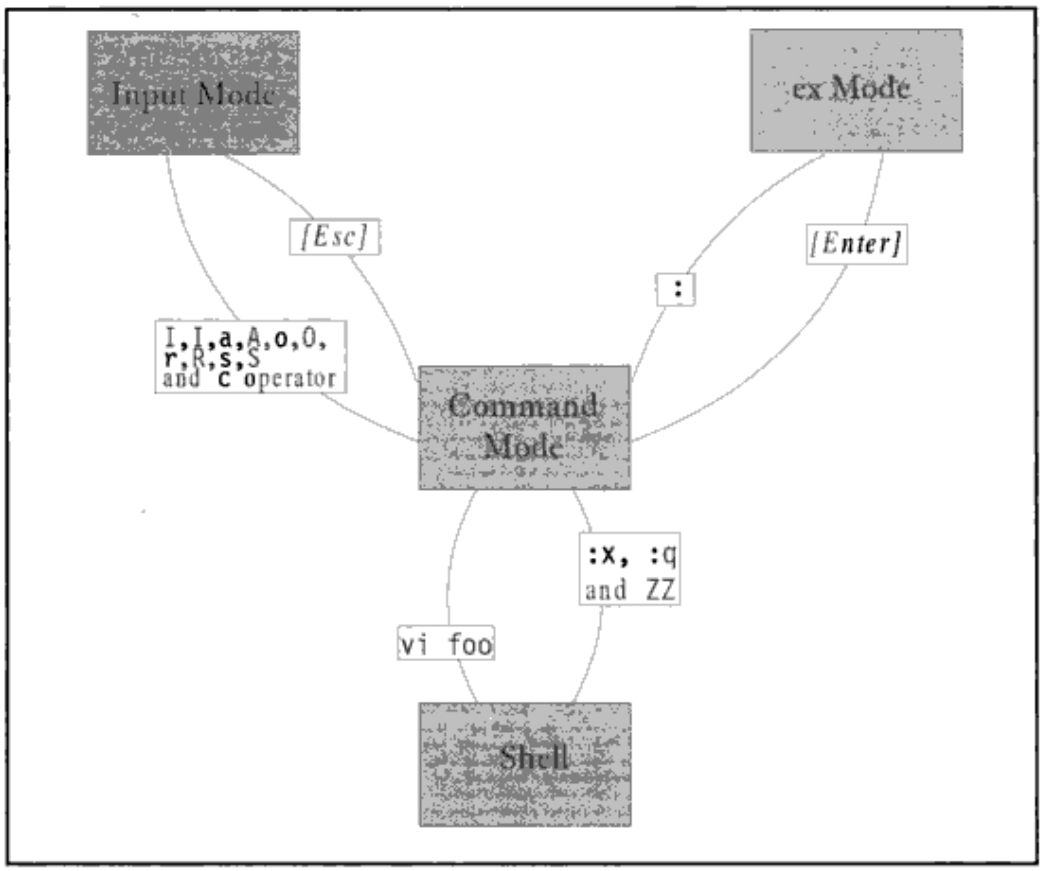



**Fig. 7.2** The Three Modes

### 7.1.3 A Few Tips First

We are about to take off, but before we do that, a few tips at this stage will stand you in good stead. You must keep them in mind at all times when you are doing work with **vi**:

- *Undo whenever you make a mistake.* If you have made a mistake in editing, either by wrongly deleting text or inserting it at a wrong location, then as a first measure, just press *[Esc]* and then **u** to undo the last action. If that makes matters worse, use **u** again. Linux users should instead use *[Ctrl-r]*.
- *Clearing the screen* If the screen gets garbled for some reason, use *[Ctrl-l]* (el) in the Command Mode to redraw the screen. If you hit *[Ctrl-l]* in the Input Mode, you'll see the symbol ^L on the screen. Use the backspace key to wipe it out, press *[Esc]* and then hit *[Ctrl-l]*.
- *Don't use [CapsLock]* **vi** commands are case-sensitive; **a** and **A** are different commands. Even if you activate this key to enter a large block of text in uppercase, make sure you deactivate it after text entry is complete.
- *Avoid using the PC navigation keys* As far as possible, avoid using all the standard navigation keys like Up, Down, Left and Right, *[PageUp]* and *[PageDown]*. Many of them could fail when you use **vi** over a network connection. **vi** provides an elaborate set of keys for navigation purposes.

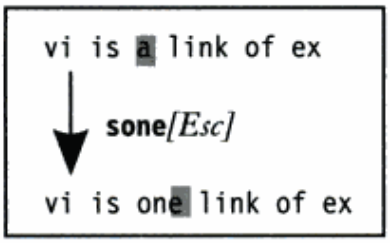



Fig. 7.8 Replacing Text with **s**

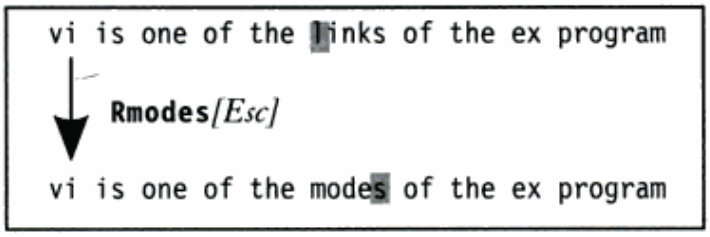



Fig. 7.9 Replacing Text with **R**

followed by the character that replaces the one under the cursor (Fig. 7.7). You can replace a single character only in this way. **vi** momentarily switches from Command Mode to Input Mode when **r** is pressed. It returns to Command Mode as soon as the new character is entered. There's no need to press *[Esc]* when using **r** and the replacement character, since **vi** expects a single character anyway.

When you want to replace the letter `d` with `10f` in a **printf** statement in C, you need to replace one character with three. In that case, press

`s` *Replaces one character with many*

**vi** deletes the character under the cursor and switches to Input Mode. It may also show a `$` at that location to indicate that replacement will not affect text on its right. Now enter **10f** and press *[Esc]*. To replace multiple characters, use a repeat factor. **3s** replaces three characters with new text. Use of **s** is shown in Fig. 7.8.

**R** and **S** act in a similar manner compared to their lowercase ones except that they act on a larger group of characters:

`R` Replaces all text on the right of the cursor position.

`S` Replaces the entire line irrespective of cursor position. (Existing line disappears)

Try using the repeat factor with **R** and **S** and see whether you can take advantage of this feature. Use of **R** is shown in Fig. 7.9.

You have now been able to enter the Input Mode in ten ways. The functions of these ten keys are summarized in Table 7.1.

---

**Caution:** You must press *[Esc]* to switch to Command Mode after you have keyed in text. Repeated pressing of *[Esc]* won't make any difference to **vi** except that it has a built-in capability to indicate with a beep if a key has been pressed unnecessarily. Try this by pressing *[Esc]* several times. You are now in the Command Mode.

---

**LINUX:** A superb text completion feature is available in **vim**. If the string `printf` is available in the file, you don't need to enter the entire string eve. Just key in as much as is necessary to make the string unique (say, up to `pr`), and then press

*[Ctrl-p]* `vim` *attempts to complete string*

`vim` expands `pr` to `printf` if this is the *only* word beginning with `pr`. In case there are other words, repeated pressing of the key shows all matching words in turn. In case you have to view the list backwards, use *[Ctrl-n]*.

**Table 7.1** Input Mode Commands

| *Command* | *Function* |
|---|---|
| `i` | Inserts text to left of cursor (Existing text shifted right) |
| `a` | Appends text to right of cursor (Existing text shifted right) |
| `I` | Inserts text at beginning of line (Existing text shifted right) |
| `A` | Appends text at end of line |
| `o` | Opens line below |
| `O` | Opens line above |
| `rch` | Replaces single character under cursor with *ch* (No *[Esc]* required) |
| `R` | Replaces text from cursor to right (Existing text overwritten) |
| `s` | Replaces single character under cursor with any number of characters |
| `S` | Replaces entire line |

## 7.3 SAVING TEXT AND QUITTING—THE ex MODE

When you edit a file using **`vi`**—or for that matter, any editor—the original file isn't disturbed as such, but only a copy of it that is placed in a **buffer** (a temporary form of storage). From time to time, you should save your work by writing the buffer contents to disk to keep the disk file current (or, as we say, in *sync*). When we talk of saving a file, we actually mean saving this buffer. You may also need to quit **`vi`** after or without saving the buffer. These features are adequately handled by the ex Mode. The basic file handling features are shown in Table 7.2.

### 7.3.1 Saving Your Work (`:w`)

We have already used the ex Mode command, **`:x`**, to save the buffer and exit the editor *(7.1)*. For extended sessions with **`vi`**, you must able to save the buffer and *remain* in the editor. From time to time, you must use the **`:w`** command to write the buffer to disk. Enter a `:`, which appears on the last line of the screen, then `w` and finally *[Enter]*:

```
:w[Enter]
"sometext", 8 lines, 275 characters
```

You can now continue your editing work normally, but make sure that you execute this command regularly. With the **`:w`** command you can optionally specify a filename as well. In that case, the contents are separately written to another file.

---

**Tip:** It's common practice to ignore the `readonly` label on opening a file that doesn't have the write permission bit set. When you attempt to save the file with **`:w`**, **`vi`** retorts with the message `File is read only`. You should have been careful in the first place, but there's hope: Just save the file with a different name (say, **`:w foo`**) after making sure that `foo` doesn't exist. Look up Table 7.2 for the command to use when `foo` exists.

---

### 7.3.2 Saving and Quitting (:x and :wq)

The previous command returns you to the Command Mode so you can continue editing. However, to save and quit the editor (i.e., return to the shell), use the **:x** (exit) command instead:

```
:x[Enter]
"sometext", 8 lines, 303 characters
$ _
```

You can also use **:wq** to save and quit the editor. But that requires an additional keystroke and is not recommended for use.

---

**Tip:** The best way to save and quit the editor is to use **ZZ**, a Command Mode command, instead of **:x** or **:wq**.

---

### 7.3.3 Aborting Editing (:q)

It's also possible to abort the editing process and quit the editing mode without saving the buffer. The **:q** (quit) command does that job:

```
:q[Enter]                                   Won't work if buffer is unsaved
$ _
```

**vi** also has a safety mechanism that prevents you from aborting accidentally if you have modified the file (buffer) in any way. The following message is typical when you try to do so:

```
No write since last change (:quit! overrides)
```

If the buffer has been changed and you still want to abandon the changes, then use

```
:q!                                         Ignores all changes made and quits
```

to return you to the prompt irrespective of the status of the buffer—no questions asked. **vi** suggests appending a ! to an ex Mode command every time it feels that you could be doing something that is potentially unsafe.

---

**Note:** In general, any ex Mode command used with a ! signifies an abort of some type. It can be used to switch to another file without saving the current one, or reload the last saved version of a file. You can even use it to overwrite a separate file.

---

### 7.3.4 Writing Selected Lines

The **:w** command is an abbreviated way of executing the ex Mode instruction **:1,$w**. w can be prefixed by one or two *addresses* separated by a comma. The command

```
:10,50w n2words.pl                          Writes 41 lines to another file
```

saves lines 10 through 50 to the file `n2words.pl`. You can save a single line as well:

```
:5w n2words.pl                              Writes 5th line to another file
```

**Note:** You can't be assured of complete recovery every time. Sometimes, **vi** may show you absolute junk when using the `-r` option (or `:recover`). In that case, don't save the file and simply quit (with **:q!**). Start **vi** again normally; recovery is not possible here. Linux users should note that in these situations, they need to delete the file having a `.swp` extension manually; otherwise the file will not be editable.

**Table 7.2** Save and Exit Commands of the ex Mode

| *Command* | *Action* |
|---|---|
| `:w` | Saves file and remains in editing mode |
| `:x` | Saves file and quits editing mode |
| `:wq` | As above |
| `:w n2w.pl` | Like *Save As* ..... in Microsoft Windows |
| `:w! n2w.pl` | As above, but overwrites existing file |
| `:q` | Quits editing mode when no changes are made to file |
| `:q!` | Quits editing mode but after abandoning changes |
| `:`*n1*`,`*n2*`w build.sql` | Writes lines *n1* to *n2* to file `build.sql` |
| `:.w build.sql` | Writes current line to file `build.sql` |
| `:$w build.sql` | Writes last line to file `build.sql` |
| `:!`*cmd* | Runs *cmd* command and returns to Command Mode |
| `:sh` | Escapes to UNIX shell |
| `:recover` | Recovers file from a crash |

## 7.4 NAVIGATION

We'll now consider the functions of the Command Mode. This is the mode you come to when you have finished entering or changing your text. A Command Mode command doesn't show up on screen but simply performs a function. We begin with navigation. Don't forget to avoid the cursor control keys for navigation as advised in Section 7.1.3.

### 7.4.1 Movement in the Four Directions (h, j, k and l)

**vi** provides the keys **h**, **j**, **k** and **l** to move the cursor in the four directions. These keys are placed adjacent to one another in the middle row of the keyboard. Without a repeat factor, they move the cursor by one position. Use these keys for moving the cursor vertically:

- `k` Moves cursor up
- `j` Moves cursor down

To move the cursor along a line, use these commands:

- `h` Moves cursor left
- `l` Moves cursor right

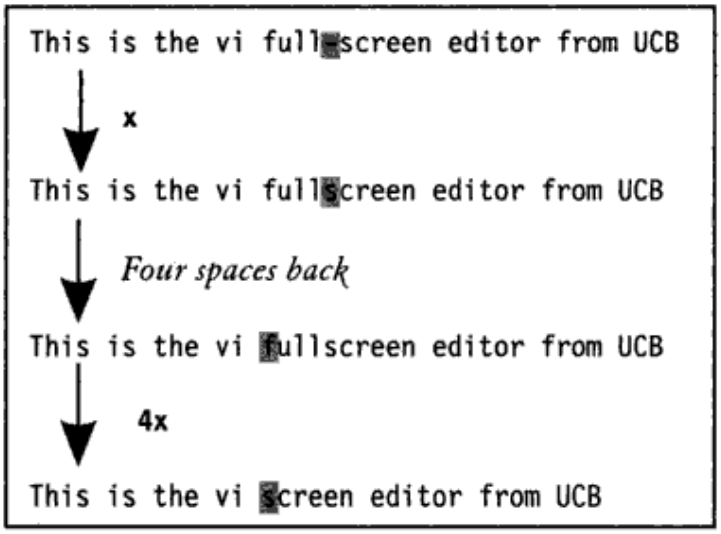



**Fig. 7.12** Deleting Text with **x**

### 7.5.1 Deleting Text (x and dd)

The simplest text deletion is achieved with the **x** command. This command deletes the character under the cursor. Move the cursor to the character that needs to be deleted and then press

```
x                                    Deletes a single character
```

The character under the cursor gets deleted, and the text on the right shifts left to fill up the space. A repeat factor also applies here, so **4x** deletes the current character as well as three characters from the *right* (Fig. 7.12).

A Windows Notepad user would be surprised to note that when the cursor is at the end of a line, **x** doesn't pull up the following line but works instead on text on the *left* of the cursor. Deletion of text from the left is otherwise handled by the **X** command. Keep it pressed, and you'll see that you have erased all text to the beginning of the line.

Entire lines are removed with the **dd** "command" (rather a doubled operator). Move the cursor to any line and then press

```
dd                                   Cursor can be anywhere in line
```

**6dd** deletes the current line and five lines below. Fig. 7.13 illustrates the use of **dd** both with and without a repeat factor. There are other forms of deletion available in **vi** and you'll know them all after you have understood the **d** operator well *(20.1.1)*.

### 7.5.2 Moving Text (p)

Text movement requires you to perform an additional task: Put the text at the new location with **p** or **P**. **vi** uses these two commands for all "put" operations that follow delete or copy operations. The significance of **p** and **P** depends on whether they are used on parts of lines or complete lines. We need some examples to illustrate their behavior.

searches backward for the most previous instance of the pattern. The wraparound feature also applies here but in the reverse manner.

### 7.8.1 Repeating the Last Pattern Search (n and N)

The **n** and **N** commands repeat a search where **n** and **N** don't exactly play the roles you'd expect them to. For repeating a search in the direction the previous search was made with **/** or **?**, use

`n` *Repeats search in same direction of original search*

The cursor will be positioned at the beginning of the pattern. In this way, you can press **n** repeatedly to scan all instances of the string. **N** reverses the direction pursued by **n**, which means that you can use it to retrace your search path. The search and repeat actions are illustrated in Fig. 7.14 and the commands are summarized in Table 7.3.

---

**Note:** **n** doesn't necessarily repeat a search in the forward direction; the direction depends on the search command used. If you used **?printf** to search in the reverse direction in the first place, then **n** also follows the same direction. In that case, **N** will repeat the search in the forward direction, and not **n**.

---

**Tip:** The three commands, **/** (search), **n** (repeat search) and **.** (repeat last editing command), form a wonderful trio of search—search-repeat—edit-repeat commands. You'll often be tempted to use this trio in many situations where you want the same change to be carried out at a number of places.

For instance, if you want to replace some occurrences of `int` with `double`, then first search for `int` with **/int**, change `int` to `double` with **3s**, repeat the search with **n**, and press the **.** wherever you want the replacement to take place. Yes, you wouldn't like `printf` to also show up (`int` is embedded there), which means you need to use a *regular expression* to throw `printf` out. Like **more**, **vi** also recognizes regular expressions as search patterns; these expressions are first discussed in Section 13.2.

---

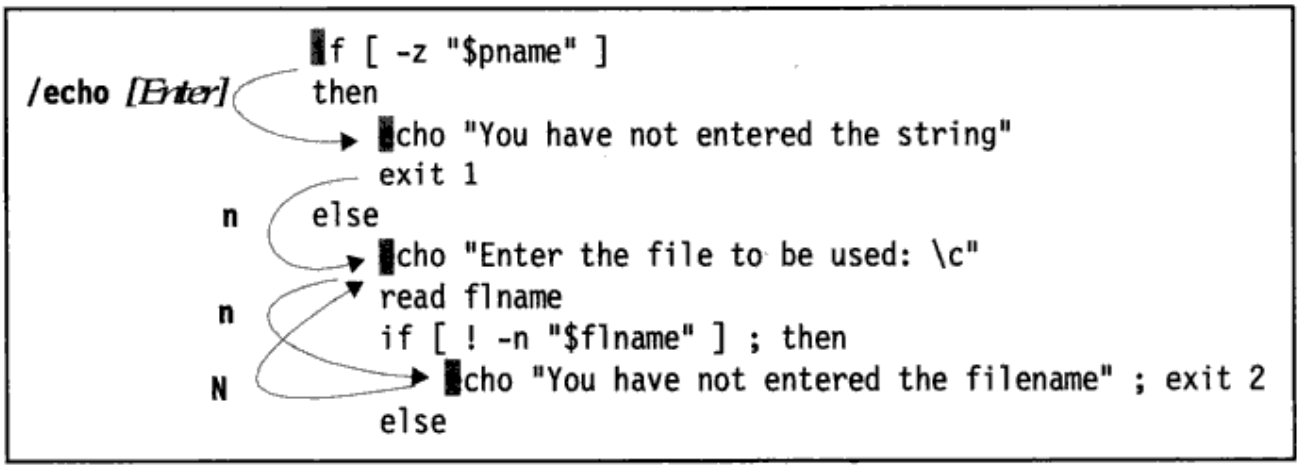



**Fig. 7.14** Search and Repeat with **/** and **n**

Most of the Input and Command Mode commands also work with a *repeat factor* which performs the command multiple times.

The Input Mode is used to insert (**i** and **I**), append (**a** and **A**), replace (**r** and **R**), change (**s** or **S**) text and open a line (**o** and **O**). The mode is terminated by pressing *[Esc]*.

Using the ex Mode you can save your work (**:w**), exit the editor after saving (**:x**) and quit without saving (**:q** and **:q!**). The current line in this mode is represented by a dot and the last line by a **$**.

Navigation is performed in the Command Mode. You can move in the four directions (**h**, **j**, **k** and **l**) or move along a line, using a *word* as a navigation unit. You can move back (**b**) and forward (**w**) to the beginning of a word.

Editing functions are also performed in the Command Mode. You can delete characters (**x** and **X**) and lines (**dd**), and yank or copy lines (**yy**). Deleted and yanked text can be put at another location (**p** and **P**).

**vi** can repeat (**.**) and undo (**u**) the last editing instruction. **vim** in Linux can perform multiple levels of undo and redo with **u** and *[Ctrl-r]*, respectively.

You can search for a pattern (**/** and **?**) and also repeat (**n** and **N**) the search in both directions. The **/**, **n** and **.** commands form a very useful trio for interactive replacement work.

The ex Mode is also used for substitution (**:s**). Both search and replace operations also use *regular expressions* for matching multiple patterns.

## Test Your Understanding

7.1 You pressed **50k** to move the cursor 50 lines up but you see **50k** input as text. What mistake did you make and how do you remove the three characters?

7.2 How will you replace has with have in the current line?

7.3 How will you insert a line (i) above the current line, (ii) below the current line?

7.4 How do you abort an editing session?

7.5 Name three ways exiting a **vi** session after saving your work.

7.6 How will you quickly move to the fifth word of a line and replace its four characters with the string `counter`?

7.7 Find out the number of words in this string as interpreted by (i) `vi`, (ii) wc—29.02.2000 is_last_day_of_February.

7.8 In the current line, how do you take your cursor to (i) the 40th character, (ii) the beginning, (iii) the end?

7.9 Explain which of the following commands can be repeated or undone: (i) **40k** (ii) *[Ctrl-f]* (iii) **5x** iv) **J**

7.10 You have wrongly entered the word `Comptuer`. How will you correct it to `Computer`?

7.11 From a conceptual point of view, how are **d** and **y** different from Command Mode commands like **j** and **$**?

7.12 How do you combine five lines into a single line?

7.13 How will you search for a pattern `printf` and then repeat the search in the opposite direction the original search was made?

7.14 Every time you press a `.` (dot), you see a blank line inserted below your current line. Why does that happen?

## Flex Your Brain

7.1 Name the three modes of **vi** and explain how you can switch from one mode to another.

7.2 How will you add /* at the beginning of a line and */ at the end?

7.3 How do you remove the characters that you just inserted above without using the undo feature?

7.4 How do you move to line number 100 and then write the remaining lines (including that line) to a separate file?

7.5 **vi** refuses to quit with **:q**. What does that indicate and how do you exit anyway?

7.6 Explain what the following commands do: (i) **:.,10w foo** (ii) **:$w! foo**. In which mode are the commands executed and what difference does it make if `foo` exists?

7.7 Assuming that the cursor is at the beginning of the line, name the commands required to replace (i) `echo 'Filename: \c'` with `echo -n "Filename: "` (ii) `printf("File not found\n");` with `fprintf(stderr, "File not found\n");`

7.8 In the midst of your work, how can you see the list of users logged in? If you have a number of UNIX commands to execute, which course of action will you take?

7.9 Name the sequence of commands to execute to move to the line containing the string `#include`, deleting four lines there, and then placing the deleted lines at the beginning of the file.

7.10 Mention the sequence of commands to execute that will replace `printf(` with `fprintf(stderr,`? How will you repeat the action globally?

7.11 How do **u** and **U** differ? When will **U** fail to work?

7.12 Name the commands required to noninteractively replace all occurrences of `cnt` with `count` in (i) the first 10 lines, (ii) the current line, (iii) all lines. How do you repeat the exercise in an interactive manner?

7.13 If the power to the machine is cut off while a **vi** session is active, how does it affect your work? What salvage operation will you try?

7.14 You made some changes to a read-only file and then find that you can't save the buffer. What course of action will you take without quitting the editor?

7.15 Copy `/etc/passwd` to `passwd`. Name the **vi** commands required to save the first 10 lines in `passwd1`, the next 10 in `passwd2` and the rest in `passwd3`.

7.16 List the editing and navigation commands required to convert the following text:

```
# include<errno.h>
void quit (char *message)
{
        printf("Error encountered\n");
        printf("error number %d, ", errno);
```

```
        printf("quitting program\n");
        exit(1);
}
```

to this:

```
#include <stdio.h>
#include <errno.h>
void quit (char *message, int exit_status) {
        /* printf("Error encountered\n"); */
        fprintf(stderr, "Error number %d, quitting program\n", errno);
        exit(exit_status);
}
```

# 8

# The Shell

This chapter introduces the agency that sits between the user and the UNIX system. It is called the *shell*. All the wonderful things that you can do with UNIX are possible because the shell does a lot of work on our behalf that could be tedious for us to do on our own. The shell looks for some special symbols in the command line, performs the tasks associated with them and finally executes the command. For example, it opens a file to save command output whenever it sees the > symbol.

The shell is a unique and multi-faceted program. It is a command interpreter and a programming language rolled into one. From another viewpoint, it is also a process that creates an environment for you to work in. All of these features deserve separate chapters for discussion, and you'll find the shell discussed at a number of places in this book. In this chapter, we focus on the shell's basic interpretive activities. We have seen some of these activities in previous chapters (like `rm *` or `ls | wc`), but it is here that we need to examine them closely.

### *What You Will Learn*

- An overview of the shell's interpretive cycle.
- The significance of *metacharacters* and their use in *wild-cards* for matching multiple filenames.
- The use of *escaping* and *quoting* to remove the meaning of a metacharacter.
- The significance of the three *standard files (streams)* that are available to every command.
- How the shell manipulates the default source and destination of these streams to implement *redirection* and *pipelines*.
- Understand what *filters* are and why they are so important in UNIX.
- The significance of the files `/dev/null` and `/dev/tty`.
- The use of *command substitution* to obtain the arguments of a command from the standard output of another.
- Shell variables and why they are so useful.

### *Topics of Special Interest*

- The difference between use of double and single quotes.
- The importance of making the command ignorant of the source of its input and destination of its output.

**Note:** DOS/Windows users may be surprised to know that the * may occur anywhere in a filename and not merely at the end. Thus, *chap* matches all the following filenames—chap newchap chap03 chap03.txt.

**Caution:** Be careful when you use the * with **rm** to remove files. You could land yourself in a real mess if, instead of typing **rm *.o** which removes all the C object files, you inadvertently introduce a space between * and .o:

```
$ rm * .o                                          Very dangerous!
rm: .o: No such file or directory
```

The error message here masks a disaster that has just occurred; **rm** has removed all files in this directory! In such situations, you should pause and check the command line before you finally press *[Enter]*.

The next wild-card is the ?, which matches a single character. When used with same string chap (chap?), the shell matches all five-character filenames beginning with chap. Appending another ? creates the pattern chap??, which matches six-character filenames. Use both expressions separately, and the meaning becomes obvious:

```
$ ls chap?
chapx  chapy  chapz
$ ls chap??
chap01  chap02  chap03  chap04  chap15  chap16  chap17
```

Both the * and ? operate with some restrictions that are taken up in the next topic.

## 8.3.2 Matching the Dot

The behavior of the * and ? in relation to the dot isn't as straightforward as it may seem. The * doesn't match all files *beginning* with a . (dot) or the / of a pathname. If you want to list all hidden filenames in your directory having at least three characters after the dot, the dot must be matched explicitly:

```
$ ls .???*
.bash_profile   .exrc   .netscape  .profile
```

However, if the filename contains a dot anywhere but at the beginning, it need not be matched explicitly. For example, the expression emp*lst matches a dot embedded in the filename:

```
$ ls emp*lst
emp.lst    emp1.lst    emp22lst  emp2.lst    empn.lst
```

**Note:** There are two things that the * and ? can't match. First, they don't match a filename beginning with a dot, but they can match any number of embedded dots. For instance, apache*gz matches apache_1.3.20.tar.gz. Second, these characters don't match the / in a pathname. You can't use **cd /usr?local** to switch to /usr/local.

```
$ rm chap0\[1-3\]
$ ls chap0\[1-3\]
chap0[1-3] not found                                    File removed
```

*Escaping the Space* Apart from metacharacters, there are other characters that are special—like the space character. The shell uses it to delimit command line arguments. So to remove the file `My Document.doc` which has a space embedded, a similar reasoning should be followed:

```
rm My\ Document.doc                                     Without the \ rm would see two files
```

*Escaping the \ Itself* Sometimes you may need to interpret the \ itself literally. You need another \ before it, that's all:

```
$ echo \\
\
$ echo The newline character is \\n
The newline character is \n
```

*Escaping the Newline Character* The newline character is also special; it marks the end of the command line. Some command lines that use several arguments can be long enough to overflow to the next line. To ensure better readability, you need to split the wrapped line into two lines, but make sure that you input a \ before you press *[Enter]*:

```
$ find /usr/local/bin /usr/bin -name "*.pl" -mtime +7 -size +1024 \[Enter]
> -size -2048 -atime +25 -print                                  Note the >
```

This is the **find** command at work, a command often used with several arguments. The \ here escapes the meaning of the newline character generated by *[Enter]*. It also produces the second prompt (which could be a > or a ?), which indicates that the command line is incomplete. Command lines involving multiple commands (as in pipelines) can also be very long. For better readability, you should split them into multiple lines wherever possible.

---

**Note:** The space, \ and LF (the newline character generated by *[Enter]*) are also special and need to be escaped if the shell is to be prevented from interpreting them in the way it normally does.

---

## 8.4.2 Quoting

There's another way to turn off the meaning of a metacharacter. When a command argument is enclosed in quotes, the meanings of all enclosed special characters are turned off. Here's how we can run some of the previous commands, using a mix of single- and double-quoting this time:

```
echo '\'                                    Displays a \
rm 'chap*'                                  Removes file chap*
rm "My Document.doc"                        Removes file My Document.doc
```

Escaping also turns out to be a tedious affair when there are just too many characters to protect. Quoting is often a better solution. The following example shows the protection of four special characters using single quotes:

```
$ echo 'The characters |, <, > and $ are also special'
The characters |, <, > and $ are also special
```

The shell can effect redirection of this stream when it sees the > or >> symbols in the command line. You can replace the default destination (the terminal) with any file by using the > (right chevron) operator, followed by the filename:

```
$ wc sample.txt > newfile
$ cat newfile
       3      14      71  sample.txt
```

The first command sends the word count of `sample.txt` to `newfile`; nothing appears on the terminal screen. If the output file doesn't exist, the shell creates it before executing the command. If it exists, the shell overwrites it, so use this operator with caution. The shell also provides the >> symbol (the right chevron used twice) to append to a file:

```
wc sample.txt >>newfile                    Doesn't disturb existing contents
```

## HOW IT WORKS: How Output Redirection Works

Command: `wc sample.txt > newfile`

1. On seeing the >, the shell opens the disk file, `newfile`, for writing.
2. It unplugs the standard output file from its default destination and assigns it to `newfile`.
3. **wc** (and not the shell) opens the file `sample.txt` for reading.
4. **wc** writes to standard output which has earlier been reassigned by the shell to `newfile`.

And all this happens without **wc** knowing that it is in fact writing to `newfile`! Any command that uses standard output is ignorant about the destination of its output also.

Redirection also becomes a useful feature when concatenating the standard output of a number of files. The following example that uses a wild-card saves all C programs in a single file:

```
cat *.c > c_progs_all.txt
```

This concatenated output stream provides no visual indication of the name of the files, so we can do somewhat better by preceding this output with a list of filenames. A single > does the job, but this requires the use of the ( and ) symbols for grouping commands:

```
( ls -x *.c ; echo ; cat *.c ) > c_progs_all.txt
```

The **echo** command in the middle serves to insert a blank line between the multicolumn file list and the code listings. The program names now feature in the front page, so you have a table of contents preceding your code listings. But better still would be to precede each program listing with its filename. You can do this using a loop construct after you have learned shell programming.

The standard output of one command can also be used by another command as its standard input. This is the third destination of standard output and is taken up in the discussion on pipes *(8.7)*.

**Note:** When the output of a command is redirected to a file, the output file is created by the shell before the command is executed. Any idea what **`cat foo > foo`** does?

Why did we run **cmp** to compare two files if we are not interested in its output? Later, you'll learn how to examine a special shell variable (`$?`) to know whether two files are identical or not. It's this value that we are often interested in and not the actual output that lists the differences.

`/dev/tty` The second special file in the UNIX system is the one indicating one's terminal—`/dev/tty`. But make no mistake: *This is not the file that represents standard output or standard error.* Commands generally don't write to this file, but you'll need to redirect some statements in shell scripts to this file.

Consider, for instance, that romeo is working on terminal `/dev/pts/1` and juliet on `/dev/pts/2`. However, both romeo and juliet can refer to their own terminals with the same filename—`/dev/tty`. Thus, if romeo issues the command

```
who >/dev/tty
```

the list of current users is sent to the terminal he is currently using—`/dev/pts/1`. Similarly, juliet can use an identical command to see the output on her terminal, `/dev/pts/2`. Like `/dev/null`, `/dev/tty` can be accessed independently by several users without conflict.

You may ask why one should need to specifically redirect output to one's own terminal since the default output goes to the terminal anyway. The answer is that sometimes you need to specify that explicitly as the following real-world example suggests.

Consider redirecting a shell script to a file, say, by using **`foo.sh > redirect.txt`**. Redirecting a script implies redirecting the standard output of all statements in the script. That's not always desirable. Your script may contain some **echo** commands that provide helpful messages for the user, and you would obviously like to see them on the terminal. If these statements are explicitly redirected to `/dev/tty` inside the script, redirecting the script won't affect these statements. We'll use this feature later in our shell scripts.

Apart from its use in redirection, `/dev/tty` can also be used as an argument to some UNIX commands. Section 8.8 makes use of this feature, while some situations are presented in Chapter 14 (featuring shell programming).

---

**Note:** The size of `/dev/null` is always zero and all terminals can be represented by `/dev/tty`.

---

## 8.7 PIPES

Standard input and standard output constitute two separate streams that can be individually manipulated by the shell. If that be so, can't the shell connect these streams so that one command takes input from the other? That is not only possible but forms the cornerstone of the building-block approach that UNIX advocates to solve all text manipulation problems.

You know the **who** command produces a list of users, one user per line. Let's use redirection to save this output in a file:

## 8.10.2 Where to Use Shell Variables

*Setting Pathnames* If a pathname is used several times in a script, you should assign it to a variable. You can then use it as an argument to any command. Let's use it with **cd** in this manner:

```
$ progs='/home/kumar/c_progs'
$ cd $progs ; pwd
/home/kumar/c_progs
```

A shell script would generally contain this definition at the beginning, and then it could be used everywhere—both in the script and in other scripts run from that script. It means lesser typing, but there's another advantage. In a later reorganization, if the location of `c_progs` changes to, say, `/export/home/kumar/c_progs`, then you simply need to change the variable definition, and everything will work in the same way as before.

*Using Command Substitution* You can also use the feature of command substitution to set variables:

```
$ mydir=`pwd` ; echo $mydir
/home/kumar/c_progs
```

You can store the size of a file in a variable too:

```
size=`wc -c < foo.txt`
```

We used the < symbol to leave out the filename in the value assigned to size. If we had used `foo.txt` as an argument instead, `size` would have contained a two-word string.

*Concatenating Variables and Strings* In your shell scripts, you'll often need to concatenate a variable with another variable or string. In a later chapter, we'll use this feature to change a file's extension. To concatenate two variables you can either place them side by side:

```
base=foo ; ext=.c                    Two assignments in one line
file=$base$ext                       This is foo.c
```

or use curly braces to delimit them:

```
file=${base}$ext
```

In either case, you can finally run a compilation command in this way:

```
cc -o $base $file                    Creates executable foo from foo.c
```

A similar technique can be used to concatenate a variable and a string. Note that some situations require you to use quotes:

```
file=$base'.c'                       This is foo.c; quotes not required
file=${base}.c                       Same but more readable
file=$base'01'                       This is foo01; quotes required
```

## Test Your Understanding

12.1 How will you (i) doublespace a file, (ii) produce a list of all files in the current directory without headers but in three columns?

12.2 What happens when you use **head** with multiple filenames?

12.3 How do you display the **ps** output without the header line?

12.4 You need to run a program, `a.out`, that continuously writes to a file `foo`. How will you run the program and then monitor the growth of this file from the *same* terminal?

12.5 Will this command work? **`cut -d: -c1 -f2 foo`**

12.6 Write a **sort** sequence to order `emp.lst` on the month of birth.

12.7 Produce from `emp.lst`, a list of the birth years along with the number of people born in that year.

12.8 Generate a numbered code list for the departments in `shortlist` in the form *code_number code_description* (like `1 admin`).

12.9 Devise a sequence to reset the `PATH` variable so that the first directory is removed from its list.

12.10 How do you remove repeated lines from an unsorted file where the repeated lines are (i) contiguous, (ii) not contiguous?

12.11 How do you convert the contents of the file `emp.lst` to uppercase?

## Flex Your Brain

12.1 How will you use **pr**, **sort** and **cut** to read a file in reverse?

12.2 Write a Korn or Bash shell alias that always brings up the last modified file for editing with **vi**.

12.3 How do you select from a file (i) lines 5 to 10, (ii) last but one line?

12.4 How do you display the **date** output with each field on a separate line? How do you now join the fields to get back the original output?

12.5 How do you display a list of all processes without the **ps** header line where processes with the same name are grouped together?

12.6 How will you find out the number of times the character ? occurs in a file?

12.7 Extract the names of the users from `/etc/passwd` after ignoring the first 10 entries.

12.8 How do you display a listing of all directories in the `PATH` list?

12.9 Devise a **sort** command to order the file `/etc/passwd` on GID (primary) and UID (secondary) so that users with the same GID are placed together. Users with a lower UID should be placed higher in the list.

12.10 Devise a pipeline which lists the five largest files in the current directory.

12.11 Use a pipeline and command substitution to set the length of a line in `emp.lst` to a variable.

12.12 You have two files, `foo1` and `foo2`, copied from two `/etc/passwd` files on two machines. How do you print a list of users who are (i) present in `foo1` but not in `foo2`, (ii) present in `foo2` and not in `foo1`, (iii) present in both files?

12.13 Assuming that a user may be logged in more than once, how do you (i) list only those users, (ii) mail root a sorted list of all users currently logged in, where a user is mailed only once?

12.14 How are these two commands similar and different? **`sort -u foo ; uniq foo`**

12.15 A feature provided in a user's startup file appends the output of the **`date`** command to a file `foo` whenever a user logs in. How can the user print a report showing the day along with the number of times she logged in on that day?

## 14.6 THE if CONDITIONAL

The **if** statement makes two-way decisions depending on the fulfillment of a certain condition. In the shell, the statement uses the following forms, much like the one used in other languages:

| Form 1 | Form 2 | Form 3 |
|---|---|---|
| if *command is successful*<br>then<br>  *execute commands*<br>else<br>  *execute commands*<br>fi | if *command is successful*<br>then<br>  *execute commands*<br>fi | if *command is successful*<br>then<br>  *execute commands*<br>elif *command is successful*<br>then...<br>else...<br>fi |

As in BASIC, **if** also requires a **then**. It evaluates the success or failure of the *command* that is specified in its "command line." If *command* succeeds, the sequence of commands following it is executed. If *command* fails, then the **else** statement (if present) is executed. This statement is not always required, as shown in Form 2. Every **if** is closed with a corresponding **fi**, and you'll encounter an error if one is not present.

What makes shell programming so powerful is that a command's exit status solely determines the course of action pursued by many of the shell's important constructs like **if** and **while**. All commands return an exit status as we saw with **cat** and **grep**, so you can imagine where shell programming can lead us.

In the next script, **emp3.sh** (Fig. 14.4), **grep** is first executed and a simple **if-else** construct tests the exit status of **grep**. This time we'll search /etc/passwd for the existence of two users; one exists in the file and the other doesn't:

```
$ emp3.sh ftp
ftp:*:325:15:FTP User:/users1/home/ftp:/bin/true
Pattern found - Job Over
$ emp3.sh mail
Pattern not found
```

```
#!/bin/sh
# emp3.sh: Using if and else
#
if grep "^$1" /etc/passwd 2>/dev/null   # Search username at beginning of line
then
        echo "Pattern found - Job Over"
else
        echo "Pattern not found"
fi
```

Fig. 14.4 **emp3.sh**

We'll discuss the third form of the **if** statement when we discuss **test**. The condition placed in the command line of the **if** statement will henceforth be referred to as the **control command**. You can use **if** in this way with any executable program. Amazing power indeed!

## 14.7 USING test AND [] TO EVALUATE EXPRESSIONS

When you use **if** to evaluate expressions, you need the **test** statement because the true or false values returned by expressions can't be *directly* handled by **if**. **test** uses certain operators to evaluate the condition on its right and returns either a true or false exit status, which is then used by **if** for making decisions. **test** works in three ways:

- Compares two numbers.
- Compares two strings or a single one for a null value.
- Checks a file's attributes.

These tests can be made by **test** in association with the shell's other statements also, but for the present we'll stick with **if**. **test** doesn't display any output but simply sets the parameter $?. In the following sections, we'll check this value.

### 14.7.1 Numeric Comparison

The numerical comparison operators (Table 14.2) used by **test** have a form different from what you would have seen anywhere. They always begin with a - (hyphen), followed by a two-letter string, and enclosed on either side by whitespace. Here's a typical operator:

`-ne` Not equal

The operators are quite mnemonic; `-eq` implies equal to, `-gt` implies greater than, and so on. Numeric comparison in the shell is confined to integer values only; decimal values are simply truncated. To illustrate how numeric tests are performed, we'll assign some values to three variables and numerically compare them:

```
$ x=5; y=7; z=7.2
$ test $x -eq $y ; echo $?
1                                   Not equal
$ test $x -lt $y ; echo $?
0                                   True
$ test $z -gt $y ; echo $?
1                                   7.2 is not greater than 7!
$ test $z -eq $y ; echo $?
0                                   7.2 is equal to 7!
```

The last two tests prove conclusively that numeric comparison is restricted to integers only. Having used **test** as a standalone feature, you can now use it as **if**'s control command. The next script, **emp3a.sh** (Fig. 14.5) uses **test** in an **if-elif-else-fi** construct (Form 3) to evaluate the shell parameter, $#. It displays the usage when no arguments are input, runs **grep** if two arguments are entered and displays an error message otherwise.

17

# Networking Tools

The realization that standalone computers made no sense made the network possible. When there were too many of them, people realized that standalone networks made little sense either, and that they also needed to talk to one another. UNIX has played a predominant role in the development of TCP/IP as a communications technology. Network communication became so pervasive that the technology was ultimately made available to all operating systems and eventually led to the formation of the Internet. The Net is running on TCP/IP since 1983.

This chapter discusses the tools used in a TCP/IP network. Some of these tools, like **telnet** and **ftp**, belong to the original DARPA set which we cover only briefly. Rather, we'll examine the basics of cryptography, and how its principles are incorporated into the secure shell (SSH). We also need to study the mechanism behind email and the Web service, and how both applications have benefited from MIME technology.

### *What You Will Learn*

- The basics of TCP/IP and the way it splits data into packets before reassembling them.
- Understand how a host is addressed by the *hostname* and *IP address*.
- The function of `/etc/hosts` in resolving hostnames to IP addresses.
- Use **telnet** and **ftp** for remote login and file transfer.
- The concept of *domains* and the Internet domain hierarchy.
- The role of *name servers*, *resolvers* and a distributed database in performing hostname-IP address mappings.
- The role of *hypertext*, *URL* and *HTML* in the HTTP protocol.

### *Topics of Special Interest*

- The basics of *cryptography*.
- The use of *symmetric* and *asymmetric* keys for encryption and decryption.
- The mechanism behind SSH (the secure shell) and the tools **ssh**, **slogin**, **sftp** and **scp**.
- The significance of *daemons* and *ports* in the client-server scheme.

### 16.10.2 The -xrm Option: Overriding the Resources

There are situations when you need not only to ignore the systemwide default settings but also the ones you have put in your `.Xdefaults`. A specific instance of an application may also require a different setting. You can sometimes override these settings by invoking the command with suitable options. However, not all resources have corresponding option equivalents, but a special option `-xrm` can let you specify any resource value. For example, you can change the background color of **xclock** with the `-bg` option. But you can also use the resource specification for the class:

```
xclock -xrm 'xclock*background: lightblue' &
```

Besides these simple resources, there are other settings related to the translation of events. A typical event could be the interpretation of mouse clicks or movements. They are not at all intuitive and a discussion on them is beyond the scope of this text.

## 16.11 CONCLUSION

Even if you are a habitual command line user, you'll often need to use X to view PDF and Postscript documents. This chapter covered X briefly simply because it is not central to the UNIX philosophy. We discussed just as much of X as is needed to use the file manager, view the documentation, handle multimedia mail attachments and browse the World Wide Web. We would be discussing the last two functions in Chapter 17.

#### *WRAP UP*

The X Window system splits an application into two components—*client* and *server*. The server program controls the monitor, keyboard and mouse, while the application itself runs as the client.

A special client, the window manager, is invoked at the start of the session to make window management functions available. The standard window manager for UNIX has been Motif (**mwm**), but it is now being replaced by **dtwm** of CDE.

The *Common Desktop Environment* (CDE) provides a standard look for both the desktop and window manager. It features a Front Panel from where you can launch many applications.

X is started with the **startx** or **xinit** commands. X can also be invoked by the root user with the **xdm** command.

The most important X client is **xterm**, which displays a simple terminal window that runs a shell. Further commands can be invoked at the shell prompt of this window. CDE and other desktop environments use better terminal emulation programs.

You can copy text from any window by merely selecting it with the mouse button. The copied text is pasted back by clicking the middle button. Multiple sections of copied text can be stored in the **xclipboard** client from where it can be pasted anywhere.

X normally runs on a network, and it is possible for a client to run on one machine and have its display on another. **xhosts** controls access to the server, while the environment variable, `DISPLAY`, determines where the client should display its output. The `-display` option overrides and resets the `DISPLAY` setting.

X programs run with a number of common options. You can position and specify the size and position of a window (`-geometry`) and its foreground and background colors (`-fg` and `-bg`). You can start a program as an icon (`-icon`) and provide a name (`-name`) or title (`-title`).

There are several clients available in X. The most commonly used client is the file manager that functions like Microsoft Windows Explorer. X offers a clock (**xclock**) and a calculator (**xcalc**). **xload** displays the system load and is often used with remote machines. **xkill** kills a window.

X can be easily customized. You can start X clients from `~/.xinitrc`, the startup file used by `xinit`. X resources enable you to change practically any X feature. These features can be stored in `~/.Xdefaults`, and **xrdb** can be used any time to read this file. These settings can also be overridden with the `-xrm` option available in every X client.

## Test Your Understanding

16.1 What does an X display comprise?

16.2 Who places the frames, borders and buttons on the windows? Is it a server or a client?

16.3 Can an X client like **xterm** running on a Solaris machine display its output on a HP-UX machine?

16.4 How is text copied in an **xterm** window?

16.5 What is the function of the **xhost** command? What does **xhost +** signify?

16.6 What circumstances led to the development of the CDE?

16.7 What is the function of the four rectangles that you see on the Front Panel at the center of Fig. 16.2?

16.8 What happens if all the commands in `.xinitrc` are placed in the background?

16.9 How will you override a resource setting when invoking a client?

## Flex Your Brain

16.1 How does X solve the problem of running the same program on displays having different characteristics?

16.2 How is the client-server mechanism in X different from others?

16.3 Explain the role of the window manager in X. Can you work without it?

16.4 Describe how text is copied using (i) the general features of X, (ii) **xclipboard**.

16.5 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?

16.6 How is the `DISPLAY` variable more convenient to use than the `-display` option?

`mime.types` on the sender's side and `mailcap` on the receiver's side. When you attach a PDF document to an outgoing message, your MUA looks up the file's extension in `mime.types` to determine the Content-Type header. Here's a sample entry for a PDF file (extension: `.pdf`):

```
application/pdf          pdf
```

The MUA sets the content type for the PDF portion of the message to `application/pdf`. At the receiving end, the MUA may not have the capability to handle this content type. It then looks up the file `mailcap` for the **helper application** (an external program) that is specified for this content type. Here's an entry from this file:

```
application/pdf; acroread %s                    Note the delimiter is ;
```

This entry directs the MUA to call up the Acrobat Reader to view the PDF document. Many UNIX systems maintain a systemwide `mailcap` database in `/etc`, but many MUAs (like Netscape Messenger) maintain their own. If `mailcap` doesn't specify a helper application for a content type, then the MUA would seek your approval for saving the file to disk.

Even though MIME was designed to make it possible to deliver multimedia attachments with mail messages, the standard applies equally well to newsgroup messages and Web resources. We'll revisit MIME when we discuss HTTP.

## 17.13 THE WORLD WIDE WEB

The World Wide Web is the Internet's latest and finest service yet. It was originally conceived by Tim Berners-Lee at CERN, Switzerland as a simple mechanism for interconnecting documents spread across the globe. It quickly went beyond the original vision of its creator, and today functions as a "one-stop shop" for practically everything that's discussed in this chapter. The Web kept the traditional Internet services (email, anonymous FTP and Net News) alive, but completely obsoleted its immediate ancestors, Archie and Gopher.

Even though the Web appears to be a conglomeration of multiple services, it works within the framework of the simple client-server model. Web service uses the **Hyper Text Transfer Protocol (HTTP)**, and Web servers, also known as *HTTP servers* listen for requests at port 80. The Web's access (client) tool is called the **browser**. A Web browser fetches a document (or any *resource*) residing on Web servers and formats it using the formatting instructions provided in the document itself. It also displays pictures if they are in GIF, JPEG or PNG formats. If there's a format it can't understand, it will call up a *plugin* or a *helper application* to handle it.

The World Wide Web is indeed a "web"—a vast collection of linked **hypertext** documents. This linkage is based on the understanding that if a resource is available on one server, it makes no sense to have it on another. These links are specified by *Uniform Resource Locators* (URLs). In this way, the user "wanders and roams" without needing to know where she is, and initiates a new connection with a simple keystroke or mouse click.

Web documents are written in the **Hyper Text Markup Language (HTML)**, a text-based portable language. HTML can highlight any portion of text to be displayed with some attributes (like bold,

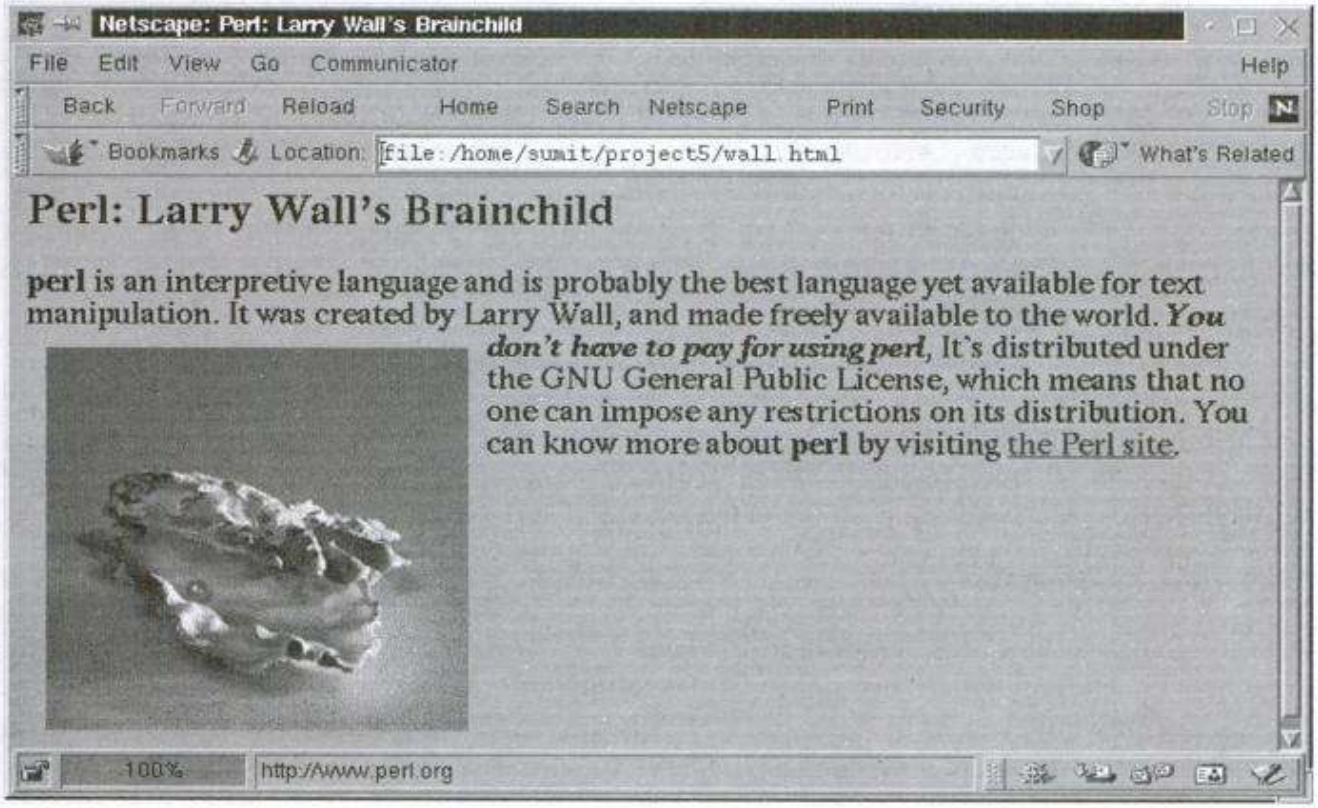



**Fig. 17.2** An HTML Document Viewed with Netscape

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. Netscape Navigator/Mozilla is the standard graphic browser for UNIX and Linux systems today.

## 17.14 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 *(17.12)* 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 by looking up the file `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 17.12 that specified `acroread` (the executable for Acrobat Reader) for `application/pdf`. Unlike in Windows, UNIX Netscape/Mozilla doesn't have this file configured well, so you'll have to fill it up yourself.

## 17.15 CONCLUSION

Even though networking wasn't part of the original UNIX scheme of things, UNIX provided the platform for its development. Many of the features found in the networking tools have their roots in UNIX. Thus, it would be fair to state that UNIX is the language of the Internet. In a sense, the Internet continued the good work done by the UNIX architects, one reason why this giant bubble didn't finally burst.

### WRAP UP

TCP/IP splits data into packets and ensures reliable transmission with full error control. Packets are *routed* between hosts of two networks using a *router* which is part of both networks.

Every host in a network is represented by a unique *hostname* and a unique *IP address*. An IP address consists of four dot-separated octets. The name-address mappings are kept in `/etc/hosts` on *all* machines of a small network.

TCP/IP works in the *client-server* model. Server programs are known as *daemons*, which run in the background and listen for request at certain *ports*. Servers use fixed port numbers for a service, but clients use random port numbers at their end.

The DARPA set of TCP/IP applications comprise **`telnet`** and **`ftp`**. **`telnet`** is used to run commands on a remote machine and have the display on the local machine. **`ftp`** is meant for file transfer. You can upload one or more files (**`put`** and **`mput`**) or download them (**`get`** and **`mget`**).

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 login in a secure manner (**`ssh`** and **`slogin`**), transfer files (**`scp`** and **`sftp`**) and run a command remotely (**`ssh`**).

Large networks and the Internet use the *Domain Name System* (DNS) where each host is represented by its *fully qualified domain name* (FQDN). The FQDN-IP address mappings are maintained across a number of *name servers* which are queried by a *resolver* to obtain the IP address of a host.

## 18.2 SPLITTING A LINE INTO FIELDS

**awk** uses the special parameter, $0, to indicate the entire line. It also identifies fields by $1, $2, $3. Since these parameters also have a special meaning to the shell, single-quoting an **awk** program protects them from interpretation by the shell.

Unlike the other UNIX filters which operate on fields, **awk** uses a contiguous sequence of spaces and tabs as a *single* delimiter. But the sample database *(12.1)* uses the |, so we must use the -F option to specify it in our programs. You can use **awk** to print the name, designation, department and salary of all the sales people:

```
$ awk -F"|" '/sales/ { print $2,$3,$4,$6 }' emp.lst
a.k. shukla       g.m.      sales      6000
chanchal singhvi  director  sales      6700
s.n. dasgupta     manager   sales      5600
anil aggarwal     manager   sales      5000
```

Notice that a , (comma) has been used to delimit the field specifications. This ensures that each field is separated from the other by a space. If you don't put the comma, the fields will be glued together.

So far, the programs have produced readable output, but that is because the file emp.lst contains fixed-length lines. Henceforth, the input for most **awk** programs used in this chapter will come from the file empn.lst which we created with **sed** in Section 13.10. This file is similar to emp.lst except that the lines are of variable length. A few lines of the file show the total absence of spaces around the |:

```
$ head -n 2 empn.lst
3212|shyam saksena|d.g.m.|accounts|12/12/55|6000|6213
6213|karuna ganguly|g.m.|accounts|05/06/62|6300|6213
```

With this file as input, we'll use **awk** with a line address (single or double) to select lines. If you want to select lines 3 to 6, all you have to do is use the built-in variable NR to specify the line numbers:

```
$ awk -F"|" 'NR == 3, NR == 6 { print NR, $2,$3,$6 }' empn.lst
3 n.k. gupta chairman 5400
4 v.k. agrawal g.m. 9000
5 j.b. saxena g.m. 8000
6 sumit chakrobarty d.g.m. 6000
```

This is **awk**'s way of implementing the **sed** instruction **3,6p**. The statement NR == 3 is really a condition that is being tested, rather than an assignment; this should appear obvious to C programmers. NR is one of those built-in variables used in **awk** programs, and == is one of the many operators employed in comparison tests.

---

**Note:** **awk** is the only filter that uses whitespace as the default delimiter instead of a single space or tab.

---

## 18.3 printf: FORMATTING OUTPUT

The above output is unformatted, but with the C-like **printf** statement, you can use **awk** as a stream formatter. **awk** accepts most of the formats used by the **printf** function used in C, but in this chapter, the %s format will be used for string data, and %d for numeric. You can now produce a list of all the agarwals:

```
$ awk -F"|" '/[aA]gg?[ar]+wal/ {
> printf "%3d %-20s %-12s %d\n",NR,$2,$3,$6 }' empn.lst
  4 v.k. agrawal         g.m.         9000
  9 sudhir Agarwal       executive    7500
 15 anil aggarwal        manager      5000
```

Like in **sed**, an **awk** command is considered complete only when the quote is closed. The name and designation have been printed in spaces 20 and 12 characters wide, respectively; the - symbol left-justifies the output. The line number is three characters wide, right-justified. Note that **printf** requires \n to print a newline after each line. Using the various formats in an **awk** program, you can have complete control over the way the output is presented.

### 18.3.1 Redirecting Standard Output

Every **print** and **printf** statement can be separately redirected with the > and | symbols. However, make sure the filename or command that follows these symbols is enclosed within double quotes. For example, the following statement sorts the output of the **printf** statement:

```
printf "%s %-10s %-12s %-8s\n", $1, $3, $4, $6 | "sort"
```

If you use redirection instead, the filename should be enclosed in quotes in a similar manner:

```
printf "%s %-10s %-12s %-8s\n", $1, $3, $4, $6 > "mslist"
```

**awk** thus provides the flexibility of separately manipulating the different output streams. But don't forget the quotes!

## 18.4 VARIABLES AND EXPRESSIONS

Throughout this chapter, we'll be using variables and expressions with **awk**. Expressions comprise strings, numbers, variables and entities that are built by combining them with operators. (x + 5)*12 is an expression. Unlike in programming languages, awk doesn't have char, int, long, double and so forth as primitive data types. Every expression can be interpreted either as a string or a number, and **awk** makes the necessary conversion according to context.

**awk** also allows the use of user-defined variables but without declaring them. Variables are case-sensitive; x is different from X. A variable is deemed to be declared the first time it is used. Unlike shell variables, **awk** variables don't use the $ either in assignment or in evaluation:

```
x = "5"
print x
```

A user-defined variable needs no initialization. It is implicitly initialized to zero or a null string. As mentioned before, **awk** has a mechanism of identifying the type and initial value of a variable from its context.

Strings in **awk** are always double-quoted and can contain any character. Like **echo**, **awk** strings can include escape sequences and octal values, but strings can also include hex values. There's one difference however; octal and hex values are preceded by only \ and \x, respectively:

```
x ="\t\tBELL\7"
print x                                    Prints two tabs, the string BELL and sounds a beep
```

**awk** provides no operator for concatenating strings. Strings are concatenated by simply placing them side-by-side:

```
x = "sun" ; y = "com"
print x y                                  Prints suncom
print x "." y                              Prints sun.com
```

Concatenation is not affected by the type of variable. A numeric and string value can be concatenated just as easily. The following examples demonstrate how **awk** makes automatic conversions when concatenating and adding variables:

```
x = "5" ; y = 6 ; z = "A"
print x y                                  y converted to string; prints 56
print x + y                                x converted to number; prints 11
print y + z                                z converted to numeric 0; prints 6
```

Even though we assigned "5" (a string) to x, we could use it for numeric computation. Also observe that when a number is added to a string, **awk** converts the string to zero since it doesn't have numerals.

Expressions also have true and false values associated with them. Any nonempty string is true; so is any positive number. The statement

```
if (x)
```

is true if x is a nonnull string or positive number.

---

**Note:** Variables are neither declared nor their type specified. **awk** identifies their type and initializes them to zero or null strings. String variables are always double-quoted, but can contain escape sequences. Nonprintable characters can be represented by their octal or hex values.

---

## 18.5 THE COMPARISON OPERATORS

How do you print the three fields for the directors and the chairman? Since the designation field is $3, you have to use it in the selection criteria:

```
$ awk -F"|" '$3 == "director" || $3 == "chairman" {
> printf "%-20s %-12s %d\n", $2,$3,$6 }' empn.lst
n.k. gupta           chairman     5400
lalit chowdury       director     8200
```

- Devise useful shell functions required for everyday use.
- Use **exec** to access files with file descriptors in the same way system calls access them.

## 21.1 SHELLS AND SUB-SHELLS

When the shell executes a shell script, it first spawns a sub-shell, which in turn executes the commands in the script. When script execution is complete, the child shell withers away and returns control to the parent shell. You can also *explicitly* invoke a sub-shell to execute a shell script. The command representing the shell itself (**sh**, **ksh** or **bash**) can be used to read the statements in **join.sh**:

```
sh join.sh                    Shell also accepts script name as argument
sh < join.sh                  Its standard input can also be redirected
```

Thus a shell script run with **sh**, **ksh** or **bash** need not have execute permission. This technique, however, is applicable for executing only shell scripts and not executables. You certainly can't use **sh < a.out**.

Even though the shell accepts a script name as argument, we generally don't run shell scripts in the way shown above. We simply key in the script name from the shell prompt and run it as an executable. In this case, the current shell uses a sub-shell of the same type to execute it. However, if the script contains the interpreter line in this form:

```
#!/usr/bin/ksh
```

then, even though the login shell may be Bourne, it will use the Korn shell to execute the script. Specification of the interpreter line also helps us identify the shell the script is meant to use. We have specified the interpreter line in every script used in Chapter 14; we'll continue this practice in this chapter also.

## 21.2 () AND {}: SUB-SHELL OR CURRENT SHELL?

The shell uses two types of operators to group commands. You must understand clearly the consequences of using one group in preference to the other:

- The () Statements enclosed within parentheses are executed in a sub-shell.
- The {} Statements enclosed within curly braces are executed in the current shell only.

You have used the first type *(8.5.2)* to collectively redirect the standard output of two commands with a single redirection symbol in a manner similar to this:

```
( a.sh ; b.sh ; c.sh ) > d.sh
```

Sub-shell considerations are not important here, so we can use either form, but some applications require a set of commands to be run *without* spawning a child shell. To consider an example, let's use both grouping operators with the **cd** and **pwd** commands. Check your current directory and then change it with **cd**:

```
$ pwd
/home/kumar
```

```
$ ( cd progs ; pwd )
/home/kumar/progs
$ pwd
/home/kumar                                    Back to original directory
```

Working from a sub-shell, **cd** changed the working directory (one of the environmental parameters) to /home/kumar/progs. The parent (login shell) can't adopt this change, so the original directory is back in place. The same command group—this time using the {} operators—tells a different story:

```
$ pwd
/home/kumar
$ { cd progs ; pwd ; }
/home/kumar/progs
$ pwd
/home/kumar/progs                              Directory change is now permanent
```

The two commands have now been executed without spawning a shell; no separate environment was created, and the change of directory became permanent even after the execution of the command group. Note that we need to precede the closing brace with a ; if both { and } appear in the same line.

An often-used sequence used in many shell scripts checks the number of command line arguments and terminates the script with **exit** if the test fails. For instance, a sequence like this:

```
if [ $# -ne 3 ] ; then
    echo "You have not keyed in 3 arguments"
    exit 3
fi
```

can be easily replaced with this sequence using curly braces:

```
[ $# -ne 3 ] && { echo "You have not keyed in 3 arguments" ; exit 3 ; }
```

Why can't we use () instead of {} here? The **exit** statement can terminate a script only if it runs in the same shell that's running the script. This is the case when **exit** runs inside the {}, but not when it runs inside (). An **exit** inside () will stop executing the remaining statements in the group, but that won't automatically terminate a script.

## 21.3 export: EXPORTING SHELL VARIABLES

By default, the values stored in shell variables are local to the shell and are not passed on to a child shell. But the shell can also **export** these variables (with the **export** statement) recursively to all child processes so that, once defined, they are available globally. You have used this statement before, but now you should understand why you have done so.

Consider a simple script which displays the value of a variable x, assigns a new value to it and then displays the new value again:

```
$ cat var.sh
echo The value of x is $x
x=20                                    # Now change the value of x
echo The new value of x is $x
```

Most users have the impression that you must log out and log in if you have made a change to the `.profile`. You actually don't need to do that; simply execute the edited file with the **.** command. You'll need this facility later to execute files containing shell functions.

---

**Note:** The dot command executes a script without using a sub-shell. It also doesn't require the script to have execute permission.

---

## 21.5 let: COMPUTATION—A SECOND LOOK (KORN AND BASH)

Korn and Bash come with a built-in integer handling facility that totally dispenses with the need to use **expr**. You can compute with the **let** statement which is used here both with and without quotes:

```
$ let sum=256+128                    No whitespace after variable
$ let sum="256 + 128"                No whitespace after variable
```

If you use whitespace for imparting better readability, then quote the expression. In either case, `sum` is assigned the result of the expression:

```
$ echo $sum
384
```

Let's see how **let** handles variables. First define three variables; a single **let** does it:

```
$ let x=12 y=18 z=5
$ let z=x+y+$z                       $ not required by let
$ echo $z
35
```

**let** permits you to get rid of the **$** altogether when making an assignment. Since this computational feature is built in, scripts run much faster than when used with **expr**. Later, we'll be using **let** in place of **expr** in one of our scripts.

*A Second Form of Computing with (( and ))* The Korn shell and Bash use the (( )) operators that replace the **let** statement itself:

```
$ x=22 y=28 z=5
$ z=$((x+y + z))                     Whitespace is unimportant
$ echo $z
55
$ z=$((z+1))
$ echo $z
56
```

POSIX recommends the use of (( and )) rather than **let**, and this form is likely to become a standard feature. It's easier to use too because a single dollar can replace multiple ones.

## 21.6 ARRAYS (KORN AND BASH)

Korn and Bash support one-dimensional arrays where the first element has the index 0. Here's how you set and evaluate the value of the third element of the array `prompt`:

```
$ prompt[2]="Enter your name: "
$ echo ${prompt[2]}
Enter your name:
```

Note that evaluation is done with curly braces, and `prompt[2]` is treated just like a variable. It, however, doesn't conflict with a variable `prompt` that you may also define in the same shell. For assigning a group of elements, use a space-delimited list having either of these two forms:

```
set -A month_arr 0 31 29 31 30 31 30 31 31 30 31 30 31          Korn only
month_arr=(0 31 29 31 30 31 30 31 31 30 31 30 31)               Bash only
```

In either case, the array stores the number of days available in each of the 12 months. The first element had to be deliberately assigned to zero for obvious reasons. Finding out the number of days in June is simple:

```
$ echo ${month_arr[6]}
30
```

Using the `@` or `*` as subscript, you can display all elements of the array as well as the number of elements. The forms are similar except for the presence of the # in one:

```
$ echo ${month_arr[@]}
0 31 29 31 30 31 30 31 31 30 31 30 31
$ echo ${#month_arr[@]}                                  Length of the array
13
```

Can we use arrays to validate an entered date? The next script, **`dateval.sh`** (Fig. 21.1), does just that. It takes into account the leap year changes (except the one that takes place at the turn of every fourth century).

The first option of the outer **`case`** construct checks for a null response. The second option uses the expression `$n/$n/$n` to check for an eight-character string in the form *dd/mm/yy*. Using a changed value of `IFS`, the components of the date are set to three positional parameters and checked for valid months. The second **`case`** construct makes the leap year check and then uses an array to validate the day. The **`continue`** statements take you to the beginning of the loop whenever the test fails the validity check. Let's test the script:

```
$ dateval.sh
Enter a date: [Enter]
No date entered
Enter a date: 28/13/00
Illegal month
Enter a date: 31/04/00
Illegal day
Enter a date: 29/02/01
2001 is not a leap year
Enter a date: 29/02/00
29/02/00 is a valid date
[Ctrl-c]
```

Since the script has no exit path, we had to use the interrupt key to terminate execution.

This form should appear familiar to you as **perl** uses a similar form to evaluate the length of an array *(19.8)*.

## 21.7.2 Extracting a String by Pattern Matching

You can extract a substring using a special pattern matching feature. These functions make use of two characters—# and %. Their selection seems to have been based on mnemonic considerations. # is used to match at the beginning and % at the end, and both are used inside curly braces when evaluating a variable.

To remove the extension from a filename, previously you had to use an external command—**basename** *(14.12.2)*. This time, you can use a variable's ${*variable%pattern*} format to do that:

```
$ filename=quotation.txt
$ echo ${filename%txt}
quotation.                                          txt stripped off
```

The % symbol after the variable name deletes the *shortest* string that matches the variable's contents at the *end*. Had there been two %s instead of one, the expression would have matched the longest one. Let's now use %% with wild-cards to extract the hostname from an FQDN:

```
$ fqdn=java.sun.com
$ echo ${fqdn%%.*}
java
```

You'll recall that **basename** can also extract the base filename from a pathname. This means deleting the longest string that matches the pattern */, but at the beginning:

```
$ filename="/var/mail/henry"
$ echo ${filename##*/}
henry
```

This deletes the segment, /var/mail—the longest pattern that matches the pattern */ at the beginning. The pattern matching forms of Korn and Bash are listed in Table 21.1.

**Table 21.1** Pattern Matching Operators Used by Korn and Bash

| *Form* | *Evaluates to segment remaining after deleting* |
|---|---|
| ${*var#pat*} | Shortest segment that matches *pat* at beginning of $*var* |
| ${*var##pat*} | Longest segment that matches *pat* at beginning of $*var* |
| ${*var%pat*} | Shortest segment that matches *pat* at end of $*var* |
| ${*var%%pat*} | Longest segment that matches *pat* at end of $*var* |

## 21.8 CONDITIONAL PARAMETER SUBSTITUTION

To continue on the subject of variable evaluation, you can evaluate a variable depending on whether it has a null or defined value. This feature is known as **parameter substitution**, and is available in the Bourne shell also. It takes this general form:

${<*var*>:<*opt*><*stg*>}

```
/* Program: reverse_read2.c -- Reads a file in reverse - uses error handling  */

#include <fcntl.h>                    /* For O_RDONLY */
#include <unistd.h>                   /* For STDOUT_FILENO */
#include <errno.h>                    /* For ENOENT, errno, etc. */
#include <stdio.h>                    /* For ENOENT, errno, etc. */

int main(int argc, char **argv) {
    int size, fd;
    char buf;                                    /* Single-character buffer */
    char *mesg = "Not enough arguments\n";

    if (argc != 2) {                  /* Our own user-defined error message */
       write(STDERR_FILENO, mesg, strlen(mesg));  /* Crude form of error */
       exit(1);                                   /* handling using write */
    }                                             /* Use fprintf instead */

    if ((fd = open(argv[1], O_RDONLY)) == -1) {
       if (errno == ENOENT) {                    / *Checking for specific error*/
          fprintf(stderr, "%s\n", strerror(errno)); /* perror is better */
          exit(2);
       } else {
          perror(argv[1]);                  /* Using two library functions */
          exit(3);                          /* perror and exit. Often the */
       }                                    /* preferred way */
    }

    lseek(fd, 1, SEEK_END);                 /* Pointer taken to EOF + 1 first */
    while (lseek(fd, -2, SEEK_CUR) >= 0) { /* and then back by two bytes */
       if (read(fd, &buf, 1) != 1) {        /* A signal can create error here */
          perror("read");
          exit(4);
       }
       if (write(STDOUT_FILENO, &buf, 1) != 1) { /* Disk may run out of space */
          perror("write");
          exit(5);
       }
    }
    close(fd);                           /* Can have error here too */
    exit(0);                             /* exit doesn't return - hence no error */
}
```

Fig. 23.6 **reverse_read2.c**

It's often the case that when you use **perror**, you also quit the program immediately thereafter. Henceforth, we'll be using the **quit** function to handle most error conditions. Enter this definition for **quit** in a file, **quit.c**:

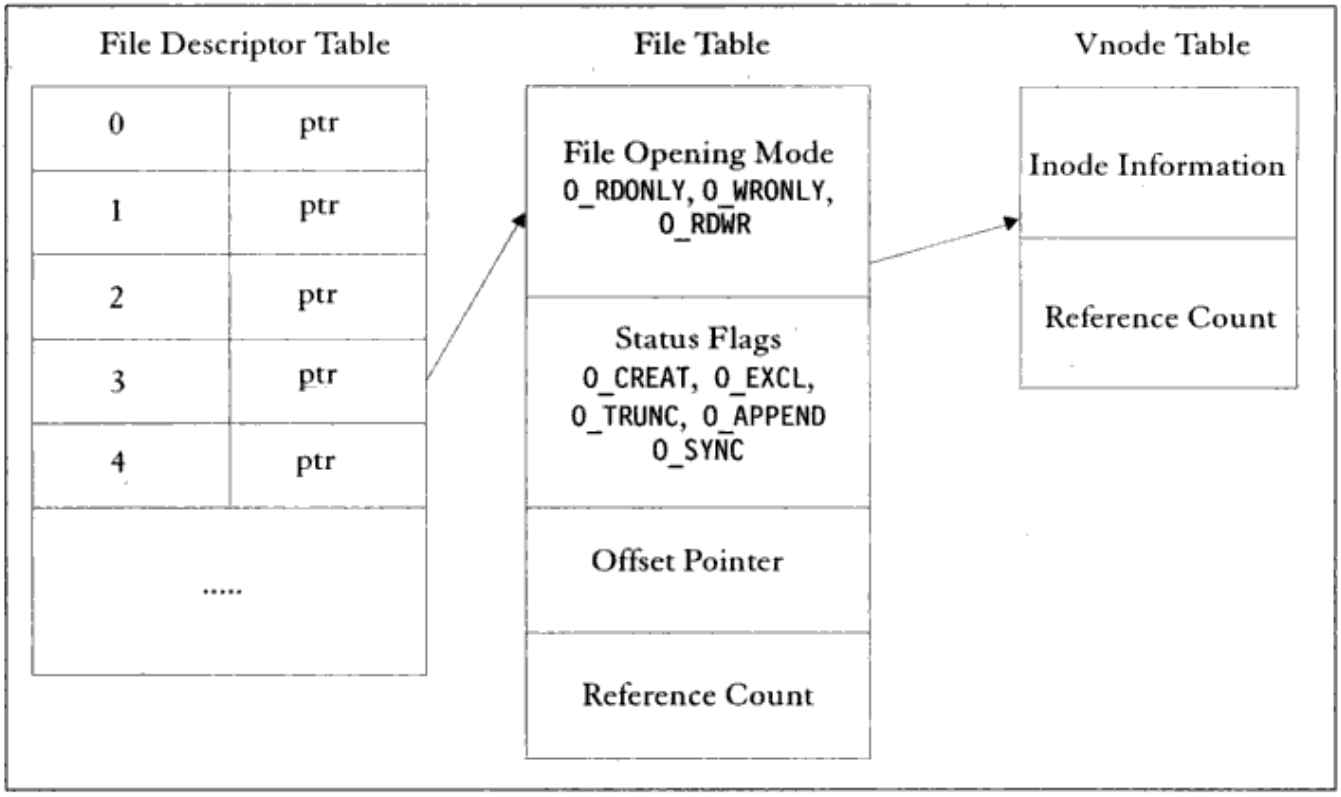



**Fig. 23.7** File Sharing—The Three Tables

the descriptor will be closed when the process does an exec *(9.4)* to run a separate program. By default, a descriptor is not closed when the process does an exec.

## 23.7.2 The File Table

Every entry in the file descriptor table points to a **file table**. This table contains all data that are relevant to an opened file. More specifically, it contains

- The mode of opening (like `O_RDONLY`).
- The status flags (like `O_CREAT`, `O_TRUNC`, etc.).
- The offset pointer location that determines the byte position to be used by the next read or write operation.
- A reference count that indicates the number of processes or calls that point to this table.

Every symbolic constant used as the second argument to **open** is made available in the file table. Since the file descriptor table and file table appear to share a one-to-one relationship, you could ask this: Why can't the mode, flags and offset be kept in the file descriptor table rather than in the file table? The answer is that UNIX also allows two or more file descriptors to point to the same file table entry (many-to-one). This happens when you duplicate the file descriptor (with the **dup** and **dup2** system calls) and when you create a process (with **fork**). We'll consider the implications of this replication in the next chapter.

## 23.8 DIRECTORY NAVIGATION

There are two system calls that perform the action of the **cd** command. They are **chdir** and **fchdir**, which use a pathname and a file descriptor, respectively as argument:

```
int chdir(const char *path);
int fchdir(int fildes);
```

We'll be using **chdir** in our next example. The current directory is obtained by the **getcwd** library function. Some UNIX systems feature other functions (like **getwd**), but POSIX recommends the use of **getcwd** which must use this syntax only:

```
extern char *getcwd(char *buf, size_t size);
```

Here, *buf* is defined as a character array of *size* bytes. After invocation of **getcwd**, the pathname of the current directory is made available in *buf*.

The next program, **dir.c** (Fig. 23.9), uses **chdir** to switch to a directory. It also invokes **getcwd** to obtain the pathname of the current directory, both before and after the switch. The buffers that store the directory pathnames are provided with one extra slot for storing the null character.

The program makes use of the **quit** function that we just developed. We'll now have to link the object code for **quit.o** with this program using

```
/* Program: dir.c -- Directory navigation with chdir and getcwd */

#include <stdio.h>
#define PATH_LENGTH 200

void quit(char *, int);                          /* Prototype definition */

int main(int argc, char **argv) {
    char olddir[PATH_LENGTH + 1];                /* Extra character for null */
    char newdir[PATH_LENGTH + 1];

    if (getcwd(olddir, PATH_LENGTH) == -1)       /* Getting current directory */
        quit("getcwd", 1);
    printf("pwd: %s\n", olddir);

    if ((chdir(argv[1]) == -1))                  /* Changing to another directory */
        quit("chdir", 2);
    printf("cd: %s\n", argv[1]);

    getcwd(newdir, PATH_LENGTH);                 /* Getting new directory */
    printf("pwd: %s\n", newdir);
    exit(0);
}
```

**Fig. 23.9 dir.c**

The exec operation can be performed by six members of a family of one system call and five library functions, which we'll refer to simply as "exec" or the "exec family". There's no system call named exec; rather, there's only one—**execve**, on top of which five library functions are built.

The entire set can be grouped into two parts, which we'll call the "execl" set and the "execv" set, because the function names begin with the string **exec** followed by either an `l` or a `v`. Two of the members also have the names **execl** and **execv**; the other four are simple derivatives of them and have more similarities than differences.

---

**Tip:** First, commit to memory this simple statement: The `l` in **execl** (and its variants) represents a fixed *list* of arguments, while the `v` in **execv** (and its variants) signifies a *variable* number of arguments.

---

### 24.8.1 execl: The Key Member of the "l" Series

As noted in the Tip above, **execl** is used with a list comprising the command name and its arguments:

```
int execl(const char *path, const char *arg0, ... /*, (char *) 0 */);
```

The syntax may appear daunting, but it is not; in fact, it is quite simple. We use **execl** when we know the number of arguments in advance. The first argument is the pathname (*path*) which could be an absolute or a relative pathname. The arguments to the command to run are represented as separate arguments beginning with the name of the command (**arg0*). The ellipsis representation in the syntax (`... /*`) points to the varying number of arguments.

To consider an example, here's how we use **execl** to run the **wc -l** command with the filename `foo` as argument:

```
execl("/bin/wc", "wc", "-l", "foo", (char *) 0);
```

**execl** doesn't use `PATH` to locate **wc**, so we must specify the pathname as its first argument. *The remaining arguments are specified exactly in the way they will appear as* **main***'s arguments in* **wc**. So, `argv[0]` in **wc**'s **main** is **wc**, the name of the command itself.

#### HOW IT WORKS: Why a Null Pointer is Required

To understand why we follow the argument list with a null pointer (`(char *) 0`), let's first understand how arguments are passed to a C program. By convention, we use this syntax for **main** when a program is run with arguments:

```
int main(int argc, char *argv[]) {
```

The startup routine that eventually runs **main** populates `*argv[]` (an array of pointers to `char`) with the string arguments specified in the command line. A null pointer is also placed at the end of the array. The number of arguments (excluding the null pointer) is then evaluated and assigned to `argc`. When **main** runs, it knows the number of arguments passed to it.

When we use exec to run a program, there's no provision to specify the number of arguments (no `argc`); exec has to fill up `argc` "by hand." The only way for **execl** to know the size of the argument list is to keep counting until it encounters the null pointer. That's why this pointer must be specified in every member of the **execl** series.

---

Let's use **execl** in our next program, **execl.c** (Fig. 24.7), to run the **wc -l -c** command with /etc/passwd as argument. Because a successful **execl** never returns, the **printf** statement is not executed:

```
$ a.out
     166    9953 /etc/passwd
```

We can also use NULL in place of (char *) 0. Note that to be able to use **execl**, we must know the number of arguments in advance because each argument is specified as a separate argument to **execl**. Often, the size of the argument list is known only at runtime. The solution is **execv**, which we'll take up next.

## 24.8.2 execv: The Key Member of the "v" Series

To run a command with any number of arguments, you must use one of the functions of the "execv" set. In this section, we discuss the **execv** function before we discover the advantages that are found in its variants. **execv** needs an array to work with:

```
int execv(const char *path, char *const argv[]);
```

Like in **execl**, *path* represents the pathname of the command to run. The second argument represents an array of pointers to char (of the same type as **main**'s argv[]). The array is populated by addresses that point to strings representing the command name and its arguments, in the form they are passed to the **main** function of the program to be exec'd. In this case also, the last element of *argv*[] must be a null pointer.

We'll use **execv** in the next program, **execv.c** (Fig. 24.8), to run the **grep** command with two options to look up the author's name in /etc/passwd. In this program, we populate the array *cmdargs[] with the strings comprising the command line to be executed by **execv**. Note that the first argument to **execv** still remains the pathname of the command. Here's the run:

```
$ a.out
15:sumit:x:102:10::/users1/home/staff/sumit:/usr/bin/bash
```

You might say that this sequence could have been run by **execl** also, but observe the way we used **execv** in the program. Since we pass the address of an array element as its second argument (cmdargs being the same as &cmdargs[0]), it implies that we can input *any* command line during runtime. Our next program will be used in that way.

```
/* Program: execl.c -- Uses execl to run wc */
#include <stdio.h>
int main(void) {
    execl("/bin/wc", "wc", "-l", "-c", "/etc/passwd", (char *) 0);
    printf("execl error\n");
}
```

**Fig. 24.7 execl.c**

This command line is accessed by **execv** as the address of argv[2]. The program doesn't terminate immediately after exec in the way the previous two programs did:

```
$ a.out /bin/grep grep -i -n SUMIT /etc/passwd
15:sumit:x:102:10::/users1/home/staff/sumit:/usr/bin/bash
Exit status: 0
```

The shell does a similar thing with our input except that we don't provide input to the shell in the way we did above. Even if we ignore this program's inability to handle redirection and wild-card interpretation, it still has the following drawbacks:

- We need to know the location of the command file because neither **execl** nor **execv** will use PATH to locate it.
- The command name is specified twice—as the first two arguments.
- These calls can't be used to run a shell script but only binary executables.
- The program has to be invoked every time we need to run a command.

To be able to run programs in a simpler and easier way, we need to discuss the derivatives of **execl** and **execv**. There are four of them, and after we examine them in the next section, we 'll use one of them to design a rudimentary shell program that has none of the limitations presented above.

### 24.8.4 The Other Members of the "l" and "v" Series

**execlp** *and* **execvp** The requirement to provide the pathname of the command makes the previous exec calls somewhat inconvenient to use. Fortunately, help is at hand in the form of the **execlp** and **execvp** functions that use PATH to locate the command. They behave exactly like their other counterparts but overcome two of the four limitations discussed in the previous section. First, the first argument need not be a pathname; it can be a command name. Second, these functions can also run a shell (or **awk** or **perl**) script. Here's their syntax:

```
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
int execvp(const char *file, char *const argv[]);
```

Note that *pathname* has now become *file*; the other arguments remain the same. To show how **execlp** works, just replace the line containing the **execl** call in the program **execl.c** (Fig. 24.7), with this one:

```
execlp("wc", "wc", "-l", "foo", (char *) 0);
```

Now the first and second arguments are the same. To run the program, **execv.c** (Fig. 24.8), that uses **execv**, just change **execv** to **execvp** without disturbing the arguments. When running it, the first two command line arguments can again be the same. You'll find these calls easier to work with.

---

**Note:** To run shell, **awk** or **perl** scripts with exec, use **execlp** or **execvp**. By default, exec spawns a Bourne shell to read the commands in the script, but you can override that by providing an interpreter line at the top of the script. For instance, if you have #!/bin/ksh as the interpreter line, exec will call up the Korn shell.

---

```
/* Program: shell.c -- Accepts user input as a command to be executed. Uses
                       the strtok library function for parsing command line */
#include <stdio.h>
#include <unistd.h>
#include <string.h>                     /* For strtok */
#include <wait.h>

#define BUFSIZE 200                     /* Maximum size of command line */
#define ARGVSIZE 40                     /* Maximum number of arguments */
#define DELIM "\n\t\r "                 /* White-space delimiters for strtok */

int main(int argc, char **argv) {
    int i, n ;
    char buf[BUFSIZE+1];                /* Stores the entered command line */
    char *clargs[ARGVSIZE];             /* Stores the argument strings */
    int returnval ;                     /* Used by wait */
    for (;;) {                          /* Loop forever */
        n = 1;
        write(STDOUT_FILENO, "Shell> ", 7);  /* Display a prompt */
        read(STDIN_FILENO, buf, BUFSIZE);    /* Read user input into buf */
        if (!strcmp(buf, "exit\n"))
            exit(0);                        /* Terminate if user enters exit */
                                            /* Now parse buf to extract the */
        clargs[0] = strtok(buf, DELIM);     /* first word */
                                            /* Continue parsing until ... */
        while ((clargs[n] = strtok(NULL, DELIM)) != NULL)
            n++;                            /* ... all words are extracted */

        clargs[n] = NULL;               /* Set last argument pointer to NULL */
        switch(fork()) {
           case 0:                      /* Run command in child */
                if ((execvp(clargs[0], &clargs[0])) < 0)
                    exit(200);          /* We'll check this value later */
           default:                      /* In the parent */
                wait(&returnval);        /* After the command has completed ... */
                printf("Exit status of command: %d\n", WEXITSTATUS(returnval));
                for (i = 0; i <=n; i++) /* ... initialize both ... */
                    clargs[i] = "\0";   /* the argument array  ... */
                for (i = 0; i <BUFSIZE+1; i++)
                    buf[i] = '\0';       /* .... and buffer that stores command */
        }                                /* line, so next command can work with */
    }                                    /* an initialized buffer and argument */
}                                        /* array. */
```

**Fig. 24.10 shell.c**

```
/* Program: dup.c -- Uses dup to achieve both input and output redirection
                     Closes standard streams first before using dup */
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#define MODE600 (S_IRUSR | S_IWUSR)

int main(int argc, char **argv) {
    int fd1, fd2;
    fd1 = open(argv[1], O_RDONLY);
    fd2 = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, MODE600);

    close(STDIN_FILENO);
    dup(fd1);                       /* This should return descriptor 0 */
    close(STDOUT_FILENO);
    dup(fd2);                       /* This should return descriptor 1 */

    execvp(argv[3], &argv[3]);      /* Execute any filter */
    printf("Failed to exec filter");
}
```

**Fig. 24.11 dup.c**

create a file in this time interval. In that case, that file will be allocated the descriptor that we wanted from **dup**. The program will then fail.

To overcome this problem, we use the **dup2** system call that uses two arguments:

```
int dup2(int fildes, int fildes2);
```

**dup2** replicates *filedes* to *filedes2* and returns it. If *filedes2* is already open, **dup2** closes it first. **dup2** thus combines the actions of **close** and **dup** that were used in succession in the previous program, except that **dup2** performs both functions as a single atomic operation *(23.1.3)*; a signal can't interrupt an atomic operation.

There are two problems with the previous program. First, doing an exec in a single process leaves us with nothing more to do since exec doesn't return. Further, by closing standard output, we ensured that we can't write to the terminal again. Ideally, files to be used for redirection should be opened in a separate child process. The child should manipulate the descriptors before exec'ing the program which uses these descriptors.

To fall in line with the shell's approach to redirection, let's repeat the previous exercise, using **dup2** this time. In this program, **dup2.c** (Fig. 24.12), file opening, descriptor manipulation and also the exec are done in a child process. The parent simply forks and waits for the child to die. The program uses the **quit** function developed in the previous chapter to handle error messages.

To demonstrate how the parent correctly obtains the exit status of the command run by the child let's use the program to run **grep** twice:

```
/* Program: dup2.c -- Opens files in the parent and uses
                      dup2 in the child to reassign the descriptors */
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <wait.h>

#define OPENFLAGS (O_WRONLY | O_CREAT | O_TRUNC)
#define MODE600 (S_IRUSR | S_IWUSR)

void quit(char *message, int exit_status) ;

int main(int argc, char **argv) {
   int fd1, fd2, rv, exit_status;

   if (fork() == 0) {              /* Child */
      if ((fd1 = open(argv[1], O_RDONLY)) == -1)
         quit("Error in opening file for reading\n", 1);
      if ((fd2 = open(argv[2], OPENFLAGS, MODE600)) == -1)
         quit("Error in opening file for writing\n", 1);
      dup2(fd1,0);                 /* Closes standard input simultaneously */
      dup2(fd2,1);                 /* Closes standard output simultaneously */
      execvp(argv[3], &argv[3]);   /* Execute command */
      quit("exec error", 2);
   } else {                        /* Parent */
      wait(&rv);                   /* Or use waitpid(-1, &rv, 0) */
      printf("Exit status: %d\n", WEXITSTATUS(rv));
   }
}
```

**Fig. 24.12 dup2.c**

```
$ a.out /etc/passwd passwd.cnt grep joker
Exit status: 1                                   joker not found in /etc/passwd
$ a.out /etc/passwd passwd.cnt grep sumit
Exit status: 0                                   sumit found in /etc/passwd
$ cat passwd.cnt
sumit:x:500:500:sumitabha das:/home/sumit:/bin/bash
```

**grep** here is used only with the pattern as argument; it obtained its input from standard input, which was redirected using **dup2** to originate from /etc/passwd. Further, the standard output of **grep** was also redirected with **dup2** to passwd.cnt.

### 24.10.3 fcntl: Recommended over dup and dup2

POSIX calls **dup** and **dup2** "redundant" functions, and advocates the use of the **fcntl** system call. Space constraints don't permit a thorough examination of this versatile system call, but you should

Although a pipe is most commonly shared by two processes, a trivial example shows its use in a single process:

```
int n, fd[2];
char buf[100];
pipe(fd);                                  Fills up fd[2] with 2 descriptors
write(fd[1], "abcdefgh", 8 );              Writing to one file descriptor
n = read(fd[0], buf, 100);                 and reading it back from another
write(STDOUT_FILENO, buf, n);              Printing what was read from pipe
```

The **pipe** call here generates two file descriptors, `fd[0]` and `fd[1]`. We simply write the string `abcdefgh` to `fd[1]`, read it back from `fd[0]`, and then write the same string to standard output. There's not much to learn from here except that numerous possibilities open up when a pipe is used by two processes. This is taken up next.

## 24.11.1 Using pipe with fork

To make **pipe** work in tandem with **fork**, the usual procedure is to create the pipe before forking a process. Because **fork** duplicates all open descriptors, a call to **pipe** before a **fork** connects two descriptors to each end of the pipe. To use the pipe, we don't need all four of them, but only one at each end. Data here can flow in either direction, but assuming that the parent writes to the pipe and the child reads it, we must close the pipe's read end in the parent and write end in the child. This is what the program, **pipe.c** (Fig. 24.13), does.

The **pipe** call returns two descriptors, `fd[0]` and `fd[1]`, which should have the values 3 and 4 in this program. This example assumes data flowing from parent to child, so the parent doesn't need the read end (`fd[0]`), while the child doesn't need the write end (`fd[1]`), the reason why these descriptors have been closed. The program outputs the string that's written to the pipe:

```
$ a.out
Writing to pipe
```

This is the string **write** wrote to `fd[1]` and **read** gathered from `fd[0]`. In real-life though, you use separate programs on either side of the shell's pipe symbol, |. So you'll naturally expect to be able to do a similar thing with **pipe**. This means that you have to connect the standard output of one program to `fd[1]` and standard input of the other to `fd[0]`. How does one do that?

## 24.11.2 pipe2.c: Running UNIX Commands in a Pipe

The next program, **pipe2.c** (Fig. 24.14), addresses this issue. Apart from closing the unneeded file descriptors associated with the pipe, the program also uses **dup2** to replicate the other descriptors—both in the parent and the child. We reverse the data flow here—from the child to the parent—just to prove that the direction of flow is irrelevant.

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 descriptor for the read end of the pipe. We don't need it 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; so we close it.

- `respawn`—Makes sure a process restarts on termination. This is always required for the **`getty`** process.
- `boot`—Executes only when `inittab` is read the first time. **`init`** ignores any run-level fields placed here.
- `off`—Kills the process if it is running.
- `ctrlaltdel`—Executes the **`shutdown`** command (Linux only).

*Using* `telinit q` As administrator, you can also insert or modify statements in `/etc/inittab`. You can change the default run level, or add and modify entries when adding a new terminal or modem to the system. But then you have to use the **`telinit`** command to let **`init`** reread its configuration file:

```
telinit q                                    A directive to init
```

**`init`** and **`telinit`** are symbolically linked, and in most cases, they can be used interchangeably. However, **`telinit`** uses the `-t` option to indicate to **`init`** the number of seconds that **`init`** has to wait before it starts killing processes—something that happens during system shutdown.

### 25.8.3 `init` and `getty`

`/etc/inittab` will always have at least one line that specifies running a program to produce a login prompt on the console and other terminals (if supported). We deliberately didn't include this line in the sample lines shown previously because Solaris uses a nonstandard program for handling this function. Rather, these lines from a Linux machine illustrate the relationship between **`init`** and the **`mingetty`** (the "getty" of Linux) program:

```
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
```

Either line provides this directive: "For run levels 2, 3, 4 or 5, run the **`mingetty`** program (with `tty1` or `tty2` as argument) and recreate (`respawn`) the process when it dies, i.e., when the user logs out." It's because of lines like these that you see the `login:` prompt on your terminal. Recall from Section 9.4.1 that the **`getty`** process execs **`login`**, which in turn execs a shell.

LINUX: Because of `/etc/inittab`, you can have multiple *virtual* consoles on your Linux machine. Use *[Ctrl][Alt]* and a function key to bring up a new screen. In this way, you can have multiple login sessions on a single machine.

### 25.8.4 The *rc* Scripts

**`init`** and `/etc/inittab` completely control the way the system is booted and powered down. Moreover, when the system changes a run level, **`init`** looks up `inittab` to identify the processes that should and shouldn't be running for the new run level. It first kills the processes that shouldn't be running and then spawns those that should be.

Every `inittab` specifies the execution of some **rc (run command) scripts** placed in `/etc` or `/sbin`. These scripts have the names **`rc0`**, **`rc1`**, **`rc2`**—one for each run level. This is evident from the following lines in `/etc/inittab` that were also shown at the beginning of Section 25.8.2:

```
s0:0:wait:/sbin/rc0          >/dev/msglog 2<>/dev/msglog </dev/console
s2:23:wait:/sbin/rc2             >/dev/msglog 2<>/dev/msglog </dev/console
s3:3:wait:/sbin/rc3          >/dev/msglog 2<>/dev/msglog </dev/console
```

**init** executes the script specific to that run level. However, **rc2** runs in both states 2 and 3. Each rc script further specifies the execution of a series of scripts in the directory /etc/rc*n*.d. This means that for run level 2, **init** executes **/etc/rc2**, which in turn executes the scripts in /etc/rc2.d.

Now, let's turn our attention to the scripts in the /etc/rc*n*.d directories. These directories hold two types of files as shown by this listing of /etc/rc2.d:

```
K07dmi              S69inet             S74xntpd            S90wbem
K07snmpdx           S70uucp             S75cron             S92volmgt
K16apache           S71ldap.client      S75savecore         S93cacheos.finish
K28nfs.server       S71rpc              S76nscd             S94ncalogd
S01MOUNTFSYS        S72autoinstall      S80lp               S95ncad
S20sysetup          S72slpd             S85power            S99dtlogin
S21perf             S73cachefs.daemon   S88sendmail
```

The scripts here fully initialize the system by mounting file systems, setting up the network and activating the other daemons. They are executed in two batches. When the system enters run level 2, **rc2** executes (in ASCII sequence) all scripts beginning with K (the "kill" scripts) with the stop argument. This kills all processes that shouldn't be running at this level. It then executes the scripts beginning with S (the "start" scripts) with the start argument. If you look up the **rc2** script, you'll find two **for** loops doing that.

As system administrator, you must understand how these scripts work. You should also be able to identify the script that starts a specific service. The filenames generally provides the clue, but sometimes you may need to use **grep** to look for names of specific daemons (like **httpd**, **sendmail**, **lpsched**, **inetd**, etc.) in these files.

---

**Note:** A script in rc*n*.d beginning with S is meant to start a service, while one beginning with K is meant to kill one. The kill scripts are executed before the start ones. They are all run from the corresponding **/etc/rc***n* (or **/etc/rc.d/rc** with the argument *n* in Linux) script where *n* signifies the run level.

---

### LINUX: The rc Files

The initialization files in Linux were originally based on BSD, but now have a strong System V flavor. However, the rc files and directories here are all under one roof—/etc/rc.d. Moreover, instead of using **rc***n*, Linux uses a single file, **rc**, with different arguments as shown below in these lines from /etc/inittab:

```
l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
```

All scripts in the rc*n*.d directories are executed from **/etc/rc.d/rc**. The sequence for Linux goes like this: For switching to run level *n*, **init** executes **/etc/rc.d/rc** *n*, which executes the scripts in /etc/rc.d/rc*n*.d.

A UNIX *file system* is held in a *partition* and consists of the *boot block*, *superblock*, inode and data blocks. The boot block contains a small program that eventually loads the kernel. The superblock contains details of free inodes and data blocks. Every inode contains 13 to 15 disk block addresses that keep track of all blocks used by a file.

Every UNIX system has the swap and root file systems. Most systems 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).

On Linux systems, **fdisk** is used to create, delete and activate partitions. Linux allows the creation of one *extended* partition to hold multiple *logical* partitions.

**mkfs** creates file systems. Each file system is unknown to the root file system until it is *mounted* (**mount**). **mount** takes mounting instructions from /etc/fstab (/etc/vfstab in Solaris). **umount** unmounts file systems.

**fsck** checks the integrity of file systems. It corrects inconsistencies by either deleting files or moving the unconnected ones to /lost+found. The root file system must be checked in single-user mode.

The **init** daemon maintains the system at a *run level*. The single-user mode (S or 1) is used for checking file systems. The multiuser mode (2 or 3) starts the system daemons and activates the network. **init** reads /etc/inittab which contains instructions to run **getty** at all terminals and the system's *rc* scripts. Processes are killed with the "K" scripts and started with the "S" ones.

**shutdown** uses **sync** to write the memory contents of the superblock to the file system. Sometimes, when the disk status is newer, "syncing" should not be done.

The administrator exploits **find**'s -newer option for incremental backups, and -mount for backing up the root file system without traversing other file systems. He should have a table of contents for each backup.

## Test Your Understanding

25.1 How will you ensure that the password is changed after every four weeks?

25.2 A user after logging in is unable to change directories or create files in her home directory. How can that happen?

25.3 Discuss the significance of the boot, root and swap file systems.

25.4 Which file system can't be unmounted and why?

25.5 _____ creates, modifies and deletes partitions. _____ creates a file system. _____ makes a consistency check of one or more file systems.

25.6 When can **mount** work with only one argument? When is unmounting of a file system not possible?

25.7 Where is the file system mounting information kept in (i) SVR4, (ii) Linux?

25.8 What is *run level*? How do you determine and set the default run level of your system?

25.9 What should you do immediately after you have made changes to /etc/inittab?

## Computation

Unlike the Bourne shell, integer computing is built into the C shell. The arithmetic operators are standard (+, -, *, / and %). While variable assignment can be made with **set** or **setenv**, this shell offers a special operator, @, for performing computation:

```
% set x=5
% @ y = 10                          A space after @
% @ sum=$x + $y
% @ product = $x * $y
% @ quotient = $y/$x                Space around / required
@: Badly formed number
% @ quotient = $y / $x
```

Incrementing numbers is done in these ways:

```
@ x = $x + 1
@ x++
```

The @ must be followed by whitespace even if the = need not have any around it. The arithmetic operators must also be surrounded by whitespace.

## Arrays and Lists

As discussed in Chapter 10, the C shell uses the **set** statement to define a local variable and **setenv** to define an environment variable. By default, **set** displays all local variables, but note that one variable (path) is set and evaluated differently:

```
% set path = (/bin /usr/bin /usr/local/bin /usr/dt/bin .)
% echo $path
/bin /usr/bin /usr/local/bin /usr/dt/bin .
```

Like argv, path is an array or list of five elements. The first element is accessed by $path[1], the second by $path[2], and so on. The number of elements in the list is indicated by $#path:

```
% echo $path
/bin /usr/bin /usr/local/bin /usr/dt/bin .
% echo $path[3]
/usr/local/bin
% echo $#path
5
```

Values can be put into an array with the **set** statement and **shift** also works with arrays:

```
% set numb = ( 9876 2345 6213 )    Like set 9876 2345 6213
% echo $numb[1]
9876
% echo $#numb                      Entire list stored in $numb[*]
3
% shift numb                       Uses array name too
% echo $numb[1]
2345
```

```
END
set choice = $<
switch ($choice)
   case 1:                          # Note the :
       ls -l ; breaksw
   case 2:
       ps -f ; breaksw
   case 3:
       exit
   default:                         # Use when previous matches fail
       echo "Invalid option"
endsw
```

The **breaksw** keyword moves control out of the construct after a successful match is found. If this word is not provided, then control "falls through" the end of the construct and performs all actions associated with *all* **case** options. The **default** keyword is used as the last option.

## The while and foreach Loops

There are two loops—**while** and **foreach** (instead of **for**). Both loops have three major differences with their counterparts in the other shells:

- The loop condition (or the list) is to be enclosed within parentheses.
- The **do** keyword is not used.
- The loop is terminated with **end** instead of **done**.

Let's consider the **while** loop first. This simple sequence entered at the prompt runs the **ps** command four times:

```
% set x = 1
% while ( $x < 5 )                    Can also use while { true }
?   ps -f                             PS2 for C shell is ?
?   sleep 5
?   @ x++
? end
```

The **foreach** loop also has differences with its Bourne rival, but has been emulated by **perl**. The keyword **foreach** replaces **for**, and the **in** keyword is not required. The example in Section 14.12 can be reframed like this:

```
% foreach file (chap20 chap21 chap22 chap23)
?    cp $file ${file}.bak
?    echo $file copied to $file.bak
? end
```

Here, each component of the four-item list is assigned to the variable `file` until the list is exhausted. There are other ways of using a list:

```
foreach item ( `cat clist` )
foreach fname ( *.c )                All C programs in current directory
foreach fname ( $* )                 Script arguments
```

Appendix **A**

# The C Shell—Programming Constructs

The C shell was developed at UCB by Bill Joy, the architect of **vi**. This shell is used today mainly for its interpretive features. These features have also influenced the design of the Korn and Bash shells. The latter shells have completely superseded the C shell in power and versatility because they have more powerful programming constructs. If you are looking for an improved C shell, then use the Tcsh shell that is available in Linux.

## Specifying the Interpreter

There are two ways of running a C shell script. Either use the **csh** command with the script name:

```
csh script_name
```

or invoke it by name but only after after providing the interpreter line at the top of the script:

```
#!/bin/csh
```

If the latter technique is used but the script doesn't have the interpreter line, the Bourne shell is used. (Linux uses Bash in this situation.) Every C shell script also executes the rc file, `~/.cshrc`.

## Interactive and Noninteractive Scripts

To make a script pause to read the standard input, you need to assign the special parameter, `$<`, to a variable. Input, which can comprise multiple words, is treated as a quoted string:

```
#!/bin/csh
echo "Enter filename: \c"
set flname = $<
echo "File: $flname"
```

A script can also be run noninteractively by passing command line arguments that are saved in a list or array named `$argv`. The arguments are individually accessed with `$argv[1]`, `$argv[2]` etc. Further, `$#argv` is set to the number of arguments. To maintain compatibility with the other shells, the C shell also allows the use of `$1` and `$2`, and so forth. Tcsh also allows the use of `$#`.

**metacharacter** Term used to describe any character that means something special to the shell. The meaning is reversed by preceding the character with a \. Concept also extends to special characters used by certain commands as part of their syntax.

**minor number** One of the parameters of the listing of a device file which indicates the special characteristics of the device. Can be interpreted to mean the parameters passed on to the device driver. See also **major number**.

**modification time** One of the time stamps of a file stored in the inode which represents the date and time the contents of a file were last modified. One of the attributes displayed by the listing.

**mounting** The process of attaching a standalone **file system** to another file system. Also features a command by the name **`mount`**. See also **unmounting**.

**MULTICS** An operating system whose development work was aborted to give way to the UNIX operating system. Many of the features of UNIX owe their origin to MULTICS.

**Multipurpose Internet Mail Extensions (MIME)** A standard used on the Internet to encode and decode binary files. Also useful in encoding multiple data formats in a *single* mail message. Features two headers—Content-Type and Content-Transfer-Encoding.

**name server** A dedicated service used on the Internet to convert the **fully qualified domain name** of a host to its IP address and vice versa. A name server is queried by a **resolver** and may either provide the answer or the address of another name server.

**newline** The character generated by hitting *[Enter]*. Used as the delimiter between two lines and forms one of the characters of **whitespace**.

**newsgroup** An offline discussion group on the Internet which originated from the UNIX-based USENET. Netscape Messenger also handles newsgroups.

**nonprivileged user** An ordinary user having no superuser privileges.

**option** A string normally beginning with a -, which changes the default behavior of a command. Multiple options can generally be combined with a single - symbol.

**ordinary file** The most common file of the UNIX system represented by programs, data and text. It contains data but not the end-of-file mark. Also known as **regular file**.

**orphan** A process whose parent has died. Orphans are adopted by **`init`**.

**others** A category of user understood by the **`chmod`** command. A user who is neither the owner nor a group owner of a file belongs to this class. One set of file permissions is associated with this category. See also **owner** and **group**.

**overlay** Same as **exec**.

**owner** The creator of a file having complete authority of determining its contents and permissions. Understood as **user** by the **`chmod`** command. The string and numeric representations are stored in `/etc/passwd`. See also **group** and **others**.

**packet** Term applied to describe a fragmented unit of data in a TCP/IP network.

**pager** A tool that displays output one screen at a time. **more** and **less** are the standard pagers on UNIX and Linux systems.

**Parent Process-ID (PPID)** The **process-id** of the parent process which is stored in the **process table** for every process.

**passphrase** A secret string used to protect a user's **private key**. Unlike a **password**, a passphrase can contain spaces.

**password** A secret string used by a user for authentication during login. The code is not flashed on the terminal, but is stored in an encrypted manner in `/etc/shadow`. Also features a command with a similar name (**passwd**) to change the password.

**PATH** A shell variable that contains a colon-delimited list of directories that the shell will look through to locate a command invoked by a user. The `PATH` generally includes `/bin` and `/usr/bin` for nonprivileged users and `/sbin` and `/usr/sbin` for the superuser.

**pathname** A sequence of one or more filenames using a / as a delimiter. All except the last filename have to be directories. See also **relative pathname** and **absolute pathname**.

**pending signals mask** A field maintained in the **process table** that shows the signals that have been received for a process. The kernel looks up this field and the **signal disposition** table to determine the action to be taken.

**ping** The sending of packets to a remote host to check the connectivity of the network. Also features a command by that name.

**pipe** A buffered object that allows one-way data transmission through its two ends using flow control. Whatever is written to one end is read from the other. Features a system call by that name. Used to create a **pipeline**.

**pipeline** A sequence of two or more commands used with the | symbol so that the input of one command comes from the output of the other. See also **pipe**.

**plugin** A small program that is installed in a browser to handle special file formats which can't be handled by the browser. Unlike a **helper application**, a plugin can't be invoked separately.

**port number** A number used to identify a TCP/IP application and defined in `/etc/services`. Servers use fixed port numbers but clients use random port numbers. A packet has two port numbers, one for each end of the channel.

**positional parameters** The external arguments to a shell script which are read into a series of special variables designated as `$1`, `$2`, `$3`, etc. These parameters can be renumbered with the **shift** command.

**POSIX** A set of standard interfaces that are based on the UNIX operating system. POSIX compliance ensures that a set of programs developed on one machine can be moved to another without recoding. POSIX.1 represents the standard for the application programming interface for the C language. POSIX.2 provides the interface for the shell and utilities.

**router** A special device that routes packets from one network to another.

**run level** Term used to refer to the various states that a UNIX system can be in. The run level is determined by the argument of the **init** command, and the action to be taken is specified in /etc/inittab. Different **rc scripts** are executed depending on the value of this run level.

**secure shell (SSH)** A suite of networking tools that enable remote login, file transfer and command execution. Unlike with the older tools like **telnet** and **ftp**, communication with the secure shell is totally encrypted.

**server** See **client-server architecture**.

**Set-User-ID (SUID)** A special mode of a file indicated by the letter **s** in the permissions field. The effective user-id of a process having this bit set is the owner of the file and not the user running the program. This property lets users modify important system files by using a specific command, rather than directly.

**shared library** A group of object files that are loaded into memory only during runtime. Several programs can share the same library code. See also **static library**.

**shell** The command interpreter of the UNIX system which runs perpetually at every occupied terminal. The shell processes a user request and interacts with the kernel to execute the command. It also possesses a programming capability.

**shell function** A group of statements executed as a bunch in the current shell. A shell function accepts parameters and can return only a numeric value.

**shell script** An ordinary file containing a set of commands, which is executed in an interpretive manner in a sub-shell. All the shell's internal commands and external UNIX commands can be specified in a script.

**signal** The notification made by the kernel that an event has occurred. A signal has a default disposition (action to take) but it can be overridden by a user-defined **signal handler**. Signals SIGKILL and SIGSTOP can't be ignored or handled otherwise. See also **signal disposition** and **signal handler**.

**signal disposition** The action to be taken when a signal occurs. Every signal has a default disposition, maintained in the signal disposition table, which could be to terminate, stop the process or to ignore the signal. The disposition can be changed by using a **signal handler** except for the SIGKILL and SIGSTOP signals.

**signal handler** A user-defined function in a C program that catches a signal and makes it behave in a manner that is different from the default. Signals SIGKILL and SIGSTOP can't be caught.

**signature file** A file named .signature in a user's home directory. It is used to enter a person's details that must accompany every mail message. Most mail user agents are configured to automatically attach the file with every outgoing message.

**umask** A number maintained in the shell that determines a file's default permissions. This number is subtracted from the system's default to obtain a file's actual permissions. The value can be displayed and set by using a command of the same name.

**Uniform Resource Locator (URL)** A string of characters that specifies a resource on the Web. Comprises the protocol, the FQDN of the site and the path name of the file.

**unmounting** The process of disengaging a file system from the main file system. The **umount** command performs this unmounting. See also **mounting**.

**user** The owner of a file as understood by the **chmod** command. See also **group** and **others**.

**user mode** A mode of the CPU when running a program. In this mode, the program has no access to the memory locations used by the kernel. See also **kernel mode**.

**User-ID (UID)** The name used by a user to gain access to the system. A list of authorized names is maintained in `/etc/passwd` along with their numeric representations. Also known as **login name** and **username**. See also **group-id**.

**username** Same as **user-id**.

**virtual console** A system of using multiple screens and logins from a single UNIX machine. A new screen is opened by using *[Alt]* and a function key.

**vnode table** The image of the inode in memory. Contains apart from the inode information, a reference count that shows the number of processes that point to the table. A file can't be properly deleted as long as this table is open.

**wait** Term used to refer to the inaction of a parent process while a child is running. Normally, the parent waits for the death of the child to pick up its **exit status**. Also features a shell built-in command and a system call by that name.

**wake** Term used to indicate the termination of a dormant activity when an event occurs. The kernel wakes up a sleeping process when a specific event has occurred (like the completion of I/O).

**Web page** An HTML document containing text and graphics that is presented in the form of a page at every Web site. A Web page has links with other pages—often on different machines.

**Web server** A TCP/IP application that runs the HTTP protocol. The World Wide Web serves all resources through Web servers.

**whitespace** A contiguous sequence of spaces, tabs or newlines. The default value of the `IFS` variable. Used as delimiter by the shell to parse command line arguments and by the **set** statement to assign its arguments to positional parameters.

**wild-card** A special character used by the *shell* to match a group of filenames with a single expression. The * and ? are commonly used wild-card characters. See also **regular expression**.

**word** A contiguous string of characters not containing whitespace. **wc** can count words, and **vi** enables cursor movement using a word as a navigational unit.

**World Wide Web** A service on the Internet featuring a collection of linked documents and images. The browser (client) fetches these resources from a Web server using the HTTP protocol.

**wraparound** A feature provided by the **vi** editor for resuming the search for a pattern from the other end of a file. The entire file is thus searched irrespective of the cursor position at the time of commencement of search.

**X client** An X program which performs a specific function and uses the X server for display. **xterm** is a common X client found in every X Window system.

**X server** The program in X Window which controls the display including the monitor, mouse and keyboard. X clients write their output to this program. If the display changes, only the server needs to change and not the clients.

**X Window System** The graphical component of the UNIX system. X clients write their output to the server which is responsible for their display on separate windows.

**zipped file** Any file that is compressed with the **gzip**, **zip** or **bzip2** commands. They are decompressed with **gunzip**, **unzip** and **bunzip2**.

**zombie** A dead process whose exit status has not been picked up by its parent using **wait**. Zombies clog the process table and can't be killed.

# Index



## A

## B

## D



## E

## M



## N



## O

## V

## W



## X



## Z

## The ls Command (Include -l option for listing)

| *Command* | *Displays* |
|---|---|
| `ls -a` | All filenames including those beginning with a dot |
| `ls -d .*` | Only filenames beginning with a dot |
| `ls -R` | Recursive file list |
| `ls -l` | Long listing in ASCII collating sequence |
| `ls /etc` | Filenames in /etc |
| `ls -d /etc` | Only /etc |
| `ls -ld /etc` | Listing of /etc |
| `ls -l /etc` | Listing of all filenames in /etc |
| `ls -lRa /` | Recursive listing of all filenames in file system |
| `ls -t` | Filenames sorted by last modification time |
| `ls -u` | Filenames sorted by last access time |
| `ls -i` | Inode number |

## Switching Directories (Setting CDPATH makes navigation easier)

| *Command* | *Action* |
|---|---|
| `cd ~sharma` | Switches to home directory of sharma |
| `cd ~/sharma` | Switches to directory $HOME/sharma |
| `cd -` | Toggles between current and previous directory |
| `cd` | Switches to $HOME |

## Wild-cards vs Regular Expressions (EREs available in grep -E, egrep and awk)

| *Wild-card* | *Regular Expression* | *Matches* |
|---|---|---|
| * | .* | Any number of characters including none |
| ? | . | A single character |
| - | * | Zero or more occurrences of the previous character |
| - | g* | Nothing or g, gg, ggg, etc. |
| - | g? | Nothing or g (ERE) |
| - | g+ | g, gg, ggg, etc. (ERE) |
| [ijk] | [ijk] | i, j or k |
| [b-m] | [b-m] | Any character between b and m |
| [!b-m] | [^b-m] | Any character not between b and m |
| [!a-zA-Z0-9] | [^a-zA-Z0-9] | Any non-alphanumeric character |
| - | ^#include | #include at beginning of line |
| - | */$ | */ at end of line |
| - | ^MARK$ | MARK as only word in line |
| - | ^$ | Lines containing nothing |
| - | GIF\|JPEG | GIF or JPEG (ERE) |
| - | (lock\|ver)wood | lockwood or verwood (ERE) |