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.

Another important key is the one labeled *[Ctrl]* (called Control) which you'll find in duplicate on the lower side of the keyboard. This key is never used singly but always in combination with other keys. We'll be using this key several times in this book. For instance, whenever you are advised to use *[Ctrl-s]* to stop a scrolling display, you should first press the *[Ctrl]* key and then the key labeled s, while *[Ctrl]* is still kept pressed.

At the top-left corner, you can see the *[Esc]* key (called Escape), which you'll require to use when performing file editing with a text editor such as **vi**. This key often takes you to the previous menu in a menu-based program. Then there is the *[Delete]* key, which some systems (like SCO UNIX) use to interrupt a program in the same way *[Ctrl-c]* is used on other UNIX systems (and DOS). On a system running Solaris or Linux, you'll probably be using *[Ctrl-c]*, rather than *[Delete]*, for interrupting a program.

In the same line as the *[Esc]* key are the twelve function keys labeled *[F1]*, *[F2]*, etc., up to *[F12]*. You won't require these keys initially, but much later (for mapping keys in **vi**, for example). The cursor control keys (the ones with four arrows) are required for recalling previous commands in the Bash shell.

1.4 A BRIEF SESSION

Seeing is believing as they say, so without further ado, let's get down to business and see what a UNIX session is really like. If you are migrating from the DOS/Windows environment, you have a bit of mental preparation to do before you start. UNIX isn't very friendly, and the messages that it throws up on the screen can sometimes be confusing. Often, there are no messages at all, and you'll then have to figure out yourself whether the command worked properly. Absorb as much as you can from this session, think for a while, and then move on to the other chapters.

1.4.1 Logging in with Username and Password

UNIX is security-conscious, and can be used only by those persons who maintain an *account* with the computer system. The list of accounts is maintained separately in the computer. You can't simply sit down at any terminal and start banging away if your name doesn't feature in the list. That's both inadvisable and an impossible thing to do.

Because the system can be used by many users, someone has to be given charge of administration of the system. This person is known as the **system administrator**, and he is the person who will grant you the authority to use the system. He opens an account with a name for your use, and gives you a secret code that you have to enter when the system prompts you for it. If you are running UNIX on your desktop, then remember that you are the administrator of the machine.

If you have the name "kumar" for this account, you'll be expected to enter this name when you see a *prompt* similar to this on the terminal:

```
SunOS 5.8                                    A Sun machine running Solaris 8

login:
```

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:

```
[/home/kumar]
kumar@saturn:/home/kumar >
```

Both prompts show your "location" in the file system. This should be familiar to DOS users who have used the statement `PROMPT [$p$g]` in the `AUTOEXEC.BAT` file to customize the prompt string. A customized prompt, as you'll see later, helps in a number of ways. For the time being, be content with whatever prompt you have to work with; it won't cause you any harm.

Note: As soon as you log in, a program called the *shell* starts to run at your terminal, and keeps running until you terminate the session. As mentioned before, the shell is your system's *command interpreter*. It displays the prompt and accepts all your input from the keyboard. The UNIX system offers a variety of shells (like Bourne shell, C shell, Korn shell and Bash) for you to choose from. Whether your prompt can display your "location" entirely depends on the shell you use. When the administrator opens a user account, he also sets a specific shell for the user.

Caution: On your first login, the system may force you to change your password. On the other hand, the system may not prompt for the password at all, in which case you'll be taken to the prompt straightaway. This presents a security breach, and you should immediately assign a password to your own account. If others have acquired knowledge of your password, you should change it. Both assignment and change of password can be done with a command named **passwd** *(3.9)*. You'll also learn later the terrible consequences that you may have to face if people with mischievous intent somehow come to know what your password is.

1.4.2 The Command

We have now been able to successfully log on to a UNIX system. What do we do now? Let's enter something from the keyboard, press *[Enter]*, and then see what happens:

```
$ unix[Enter]                          A message from the shell
ksh: unix: command not found           Shell returns prompt
$ _
```

What the system expected was a *command* that it knows, and `unix` doesn't appear to be a legitimate command. Nevertheless, the prompt returns to accept the next command. UNIX has a large family of commands, some of them quite powerful, and a UNIX expert is expected to know many of them. We'll now acquaint ourselves with a few of these commands.

1.4.3 date: Displaying Both Date and Time

Now that the previous command was rejected by the system, let's use one that it recognizes. UNIX has a **date** command that shows the date and time in the form used on the Internet. Type the four characters in `date` and press *[Enter]*:

```
$ date[Enter]
Fri Aug 19 13:28:34 IST 2005
$ _
```

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
      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
```

Command run with two arguments

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
chap01
chap02
chap03
helpdir
progs
```

Uppercase first

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*
-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
```

-l is an option

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:

cat displays a file's contents

```
$ cat list
README
chap01
chap02
chap03
helpdir
progs
```

This single line of code lets us catch a glimpse of the UNIX magic. The shell sees the > *before* **ls** runs, so it gets to act first. It then opens the file following the > (here, `list`). **ls** runs next and it looks up the "table of contents" of the directory to find out the filenames in it. But the shell has already manipulated things in such a way that the **ls** output doesn't come to the terminal but to the file opened by the shell on its behalf.

Note: You list files in a directory with **ls** and display file contents with **cat**. You can get more details of a file using the `-l` option with **ls**.

1.4.11 wc: Counting Number of Lines in a File

How many lines are there in the file? The **wc** command answers this question:

```
$ wc list
       6       6      42 list
```

Observe once again the brevity that typically characterizes UNIX; it merely echoes three numbers along with the filename. You have to have the manual or this text in front of you to know that the file `list` contains 6 lines, 6 words and 42 characters.

1.4.12 Feeding Output of One Command to Another

Now you should see something that is often hailed as the finest feature of the system. Previously, you used **ls** to list files, and then the > symbol to save the output in the file `list`. You then counted the number of lines, words and characters in this file with **wc**. In this way, you could *indirectly* count the number of files in the directory. The shell can do better; its manipulative capability enables a *direct* count without creating an intermediate file. Using the | symbol, it connects two commands to create a *pipeline*:

```
$ ls | wc
       6       6      42
```

No filename this time!

See how you can arrive at the same result (except for the filename), this time by simply connecting the output of **ls** to the input of **wc**. No intermediate file is now needed. On seeing the symbol, | (known as a *pipe*), the shell performs the job of connecting the two commands. If you can connect a number of UNIX commands in this way, you can perform difficult tasks quite easily.

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.

In 1969, AT&T withdrew its team from the MULTICS project, which was engaged in the development of a flexible operating system that would run continuously and be used remotely. Thompson and Ritchie then designed and built a small system having an elegant file system, a command interpreter (the shell) and a set of utilities. However, what they wanted was a general operating system running on more than one type of hardware. In 1973, they rewrote the entire system in C—a high-level (more readable than assembler) language that was invented by Ritchie himself. Portability became one of the strong features of UNIX.

1.5.1 Berkeley: The Second School

A U.S. government decree (subsequently revoked) prevented AT&T from selling computer software. The company had no option but to distribute the product to academic and research institutions at a nominal fee, but without any support. The University of California, Berkeley (UCB), created a UNIX of its own. They called it *BSD UNIX* (Berkeley Software Distribution). These versions became quite popular worldwide, especially in universities and engineering circles. Later, UCB gave up all development work on UNIX.

Berkeley filled the gaps left behind by AT&T, and then later decided to rewrite the whole operating system in the way they wanted. They created the standard editor of the UNIX system (`vi`) and a popular shell (C shell). Berkeley also created a better file system, a more versatile mail feature and a better method of linking files (symbolic links). Later, they also offered with their standard distribution a networking protocol software (TCP/IP) that made the Internet possible. Like AT&T, they also offered it practically free to many companies.

1.5.2 The Others

When computer science graduates left academics for the commercial world, they carried their UNIX aspirations with them. It was just a matter of time before business circles developed interest in the product, and they too joined in its development in a spree of unparalleled innovation. UNIX had turned commercial.

Sun used the BSD System as a foundation for developing their own brand of UNIX (then *SunOS*). Today, their version of UNIX is known as *Solaris*. Others had their own brands; IBM had *AIX*, HP offered *HP-UX*, while DEC produced *Digital UNIX*—and now *Tru64 UNIX*. Then the Linux wave arrived, and most of these vendors are offering Linux too. Today, most supercomputers run UNIX, and handheld devices are increasingly using Linux. By making itself available on most hardware, UNIX now gives the customer the "freedom of choice", as opposed to the "freedom from choice" (the way Scott McNealy, the CEO of Sun Microsystems, describes it).

As each vendor modified and enhanced UNIX to create its own version, the original UNIX lost its identity as a separate product. The BSD releases were much different from the AT&T System V releases, and the incompatibilities steadily mounted. While standards were being developed as to what a product had to satisfy to be called UNIX, AT&T took it upon themselves to rework mainly the BSD product, and ultimately unify their own System V 3.2, BSD, SunOS and XENIX flavors into its last release—*System V Release 4* (SVR4). Shortly thereafter, AT&T sold its UNIX business to Novell, who later turned over the UNIX trademark to a standards body called *X/OPEN*, now merged with *The Open Group*.

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.

2

The UNIX Architecture and Command Usage

UNIX is now more than 25 years old, but fortunately the essentials have remained the same. To appreciate the material presented in the remaining chapters, you must understand the UNIX architecture and its key features. This chapter also introduces the two agencies (the kernel and shell) who between themselves handle all the work of the system.

A major part of the job of learning UNIX is to master the essential command set. UNIX has a vast repertoire of commands that can solve many tasks either by working singly or in combination. We'll examine the generalized UNIX command syntax and understand the significance of its options and arguments. We'll also learn to look up the command documentation from the man pages. UNIX is very well documented, so there's hardly a way that a problem can remain unsolved for long.

WHAT YOU WILL LEARN

- The *multiprogramming*, *multiuser* and *multitasking* nature of UNIX.
- What *system calls* are and how they enrich the UNIX programming environment.
- How UNIX encourages the use of modular tools that can be connected to form complex tasks.
- The other features of UNIX—its vast collection of tools, pattern matching and wide variety of its documentation sources.
- How the shell uses the `PATH` variable to locate commands.
- The difference between *external* and *internal* commands.
- The breakup of a command into *arguments* and *options*.
- The use of the **`man`** command to look up the documentation of a command.
- Obtain contextual information on any topic using the **`apropos`** and **`whatis`** commands.

TOPICS OF SPECIAL INTEREST

- How two programs—the *kernel* and *shell*—work in a cooperative manner to handle everything.
- The significance of two abstractions—the *file* and *process*—in the UNIX system.

- 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
```

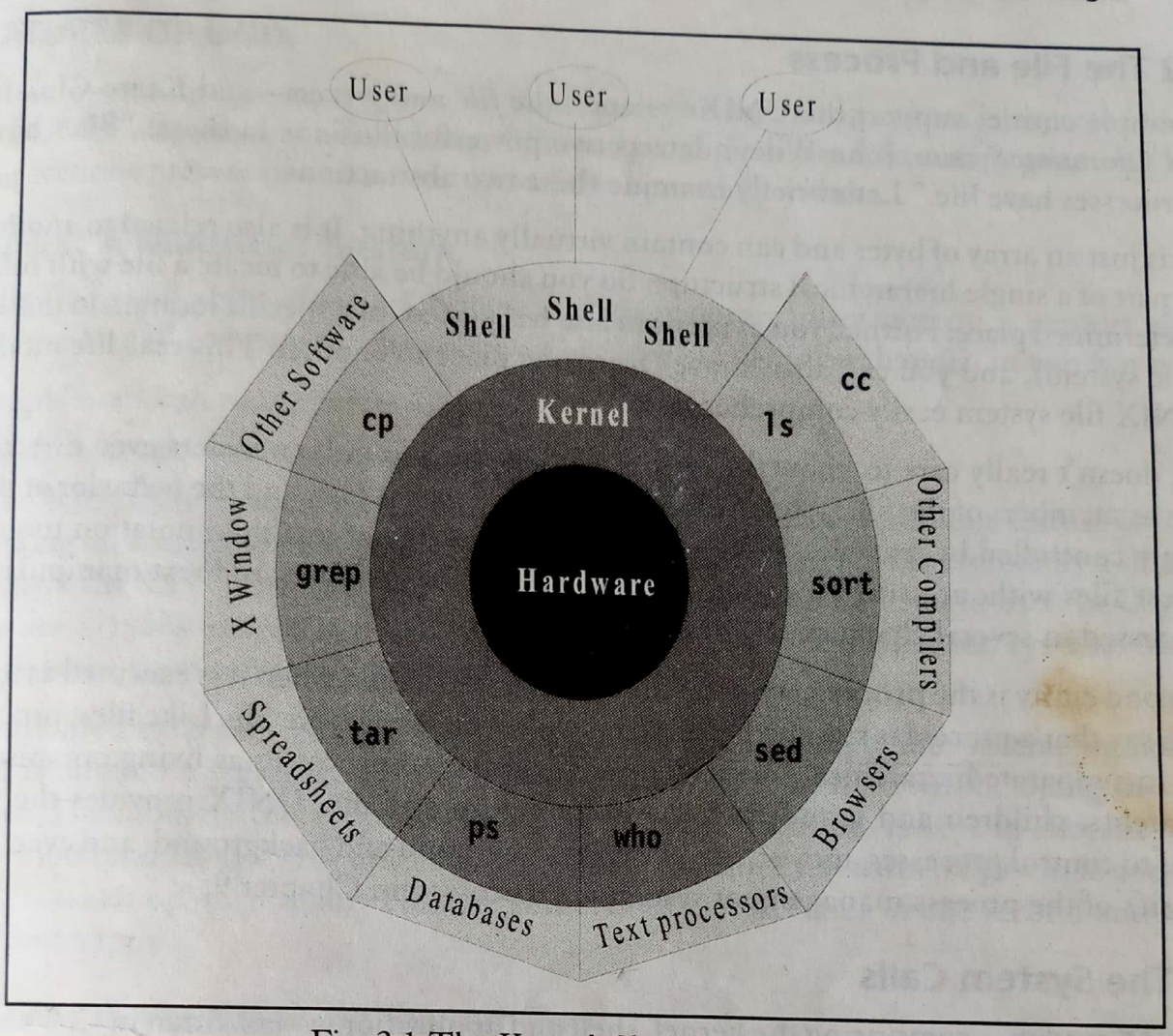

Fig. 2.1 The Kernel-Shell Relationship

In the course of processing, the shell compresses all multiple contiguous spaces in the above command line to a single one. The **echo** command then runs with spaces compressed. But since you may not like this tampering to happen all the time, the shell itself provides features to prevent its own interference. This will be taken up later.

Note: The kernel is represented by the file **/stand/unix**, **/unix** or **/kernel/genunix** (Solaris), depending on the system you are using. The system's bootstrap program loads the kernel into memory at startup. The shell is represented by **sh** (Bourne shell), **csh** (C shell), **ksh** (Korn shell) or **bash** (Bash shell). One of these shells will run to serve you when you log in. To know the one that is running for you, use the command **echo $SHELL**.

LINUX: The kernel is represented by the file **/boot/vmlinuz**. Linux uses the the Bash shell by default, though it offers the C shell and Korn shell as well.

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-

compliant UNIX system, it can be easily ported to another POSIX-compliant UNIX machine with minimum modifications. We make references to POSIX throughout this text, but these references should be interpreted to mean the SUSV3 as well.

Tip: The Single UNIX Specification, Version 3 is available at *http://www.unix.org/version3/pr.html*. You must frequently consult this document when you use a command, an option or a system call to confirm whether the usage is mandated by the specification.

2.4 LOCATING COMMANDS

The UNIX system is command-based, i.e., things happen because of the commands that you key in. UNIX commands are seldom more than four characters long. All UNIX commands are single words like **ls**, **cat**, **who**, etc. These names are all in lowercase, and you must start shedding your old DOS habits of being indifferent to case. For instance, if you enter LS instead of ls, this is how the system will respond:

```
$ LS
bash: LS: command not found
```

This message is from the shell (here, the Bash shell). There's obviously no command named **LS** on the UNIX system. This seems to suggest that there's a "predetermined" list of such commands that the shell first searches before it flashes the message above. These commands are essentially *files* containing programs, mainly written in C. Files are stored in *directories*. For instance, the **ls** command is a file (or program) found in the directory /bin.

The easiest way of knowing the location of an executable program is to use the **type** command:

Won't work in the C shell

```
$ type ls
ls is /bin/ls
```

When you execute the ls command, the shell locates this file in the /bin directory and makes arrangements to execute it. **type** looks up only the directories specified in the PATH variable (discussed next).

2.4.1 The PATH

The sequence of directories that the shell searches to look for a command is specified in its own PATH variable. Use **echo** to evaluate this variable and you'll see a directory list separated by colons:

```
$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/ccs/bin:/usr/local/java/bin:.
```

There are six directories in this colon-separated list. To consider the second one, /usr/bin represents a hierarchy of three directory names. The first / indicates the top-most directory called root, so usr is below the root directory and bin is below usr. *Don't confuse the root directory with the root user account.*

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.

2.6 COMMAND STRUCTURE

To exploit the power for which UNIX is well-known, you must know the syntax of the important commands. You have already encountered commands that were used with multiple words (like **tput clear** and **cat list**). The first word (viz., tput and cat) is actually the command; the additional words are called **arguments**. That is to say, clear is an argument to the **tput** command. Because a command often accepts several types of arguments, it can be made to behave in numerous ways.

Commands and arguments have to be separated by spaces or tabs to enable the system to interpret them as *words*. You can use any number of them to separate the words that form a command. A contiguous string of spaces and tabs together form what is known as **whitespace**. Where the system permits the use of one whitespace character to separate words, it generally permits several, as in this command:

```
cat                README
```

The shell possesses a special mechanism of compressing these multiple consecutive spaces or tabs to a single space. You have already seen this happen in Section 2.1.1.

UNIX arguments range from the simple to the complex. They consist of options, expressions, instructions, filenames, etc. The vast majority of commands have simple arguments, but some like **dd** and **find** can make you feel uneasy for a while. You'll have a taste of all of these types as you proceed.

Note: Whitespace refers to a contiguous set of spaces, tabs and newline characters. The tab character is generated by hitting the *[Tab]* key on your keyboard, or using *[Ctrl-i]* in case *[Tab]* doesn't work. The newline character is generated by hitting the *[Enter]* key.

2.6.1 Options

There's a special type of argument that's mostly used with a - sign. For instance, when you use

```
ls -l note
```

-l is an argument to **ls** by definition, but more importantly, it's a special argument known as an **option**. An option is normally preceded by a minus sign (-) to distinguish it from filenames. There must not be any whitespace between - and l. If you inadvertently provide that, many commands could land you in a great deal of trouble!

Options are also arguments, but given a special name because their list is predetermined. When we use **echo hello dolly**, hello and dolly are also arguments, but they are not predetermined. In fact, you can use millions of words legitimately with **echo** and they'll still remain arguments and not options.

LINUX: While most UNIX options use a single - as the option prefix (e.g. -l), Linux also offers options that use two hyphens and a multi-character word. For instance, it offers the synonym **ls --all** in addition to **ls -a**. Though it means more typing load for the user, the words are quite meaningful and easy to remember; it's easier to remember --all than -a.

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.

2.7.2 A Command Line Can Overflow or Be Split into Multiple Lines

A command is often keyed in. Though the terminal width is restricted to 80 characters, that doesn't prevent you from entering a command, or a sequence of them, in one line even though the total width may exceed 80 characters. The command simply overflows to the next line though it is still in a single *logical* line.

Sometimes, you'll find it necessary or desirable to split a long command line into multiple lines. In that case, the shell issues a **secondary prompt**, usually >, to indicate to you that the command line isn't complete. This is easily shown with the **echo** command:

```
$ echo "This is
> a three-line
> text message"
This is
a three-line
text message
```

A second prompt (>) appears

Tip: Whenever you find the > appear after you have pressed *[Enter]*, it will generally be due to the absence of a matching quote or parenthesis. In case you find that the problem persists even after providing it, just kill the command line with either *[Ctrl-c]* or *[Ctrl-u]*.

2.7.3 Entering a Command Before Previous Command Has Finished

UNIX provides a full-duplex terminal which lets you type a command at any time, and rest assured that the system will interpret it. When you run a long program, the prompt won't appear until program execution is complete. Subsequent commands can be entered at the keyboard (as many commands as you wish) without waiting for the prompt; they may not be even displayed on your screen!

The reason why that happens is this: The command that you key in isn't seen by the shell as its input when it is busy running another program. The input remains stored in a **buffer** (a temporary storage in memory) that's maintained by the kernel for all keyboard input. The command is passed on to the shell for interpretation after the previous program has completed its run. If you type correctly, simply don't bother even if the output from the previous program garbles the display!

2.8 man: BROWSING THE MANUAL PAGES ON-LINE

All said and done, the syntax of some UNIX commands can still be confusing—even to the expert. You may not remember either the command or the required option that will perform a specific job. This is quite understandable considering that there are several hundred commands in /bin and /usr/bin on any UNIX system (around 600 in Solaris).

UNIX offers an online help facility in the **man** command. **man** displays the documentation—often called the **man documentation**— of practically every command on the system. For example, to seek help on the **wc** command, simply run **man** with wc as argument:

```
man wc
```

Help on the `wc` *command*

The entire man page for **wc** is dumped on the screen (Fig. 2.2). **man** presents the first page and pauses. It does this by sending its output to a **pager** program, which displays this output one page (screen) at a time. The pager is actually a UNIX command, and **man** is always preconfigured to be used with a specific pager. UNIX systems currently use these pager programs:

- **more**, Berkeley's pager, that's now available universally as a superior alternative to the original AT&T **pg** command (now obsolete). We'll be considering **more** in this text.
- **less**, the standard pager used on Linux systems, but also available for all UNIX platforms. **less** is modeled on the **vi** editor and is more powerful than **more** because it replicates many of **vi**'s navigational and search functions. The features of **less** are described briefly in Section 5.5.

On a man page that uses **more** as the pager, you'll see a prompt at the bottom-left of the screen which looks something like this:

```
--More--(26%)
```

`less` *shows a* `:` *as the prompt*

At this prompt you can press a key to perform navigation, or search for a string. The key you press is interpreted as one of **man**'s (rather, the pager's) **internal commands**, and the character often doesn't show up on the screen. Many UNIX utilities like **vi** and **mail** also have their own internal commands. A set of internal commands used by **more** is listed in Table 5.1. We'll discuss only a few of them related to navigation and string search.

To quit the pager, and ultimately **man**, press **q**. You'll be returned the shell's prompt.

2.8.1 Navigation and Search

The navigation commands are numerous and often vary across UNIX implementations. For the time being, you should know these two commands which should work on all systems:

- **f** or spacebar, which advances the display by one screen of text at a time.
- **b**, which moves back one screen.

The man documentation is sometimes quite extensive, and the search facility lets you locate a page containing a keyword quite easily. For example, you can call up the page containing the word `clobber` by using the string with the / (frontslash):

```
/clobber[Enter]
```

The / and search string show up on the screen this time, and when you press *[Enter]*, you are taken to the page containing `clobber`. If that's not the page you are looking for, you can repeat the search by pressing **n**. Some pager versions even highlight the search term in reverse video.

2.9 UNDERSTANDING THE man DOCUMENTATION

Vendors organize the man documentation differently, but in general you could see eight sections of the UNIX manual (Table 2.1). Later enhancements have added subsections (like 1C, 1M, 3N

etc.), but we'll ignore them in this text. Occasional references to other sections can also be reflected in the SEE ALSO section of a man page. You can see from the table that the documentation is not restricted to commands; important system files used by these commands also have separate man pages.

Most of the commands discussed in this text are available in Section 1, and **man** searches the manuals starting from Section 1. If it locates a keyword in one section it won't continue the search even if the keyword occurs in another section. When a keyword is found in multiple sections, you should use the section number additionally as an argument. Depending on the UNIX flavor you are using, you may also need to prefix the `-s` option to the section number:

```
man 4 passwd                    passwd also occurs in Section 4
man -s4 passwd                  Solaris uses the -s option
```

This displays the documentation for a configuration file named `/etc/passwd`, from Section 4. There's also an entry for **passwd** in Section 1, but if we had used **man passwd** (without the section number), **man** would have looked up Section 1 only and wouldn't have looked at Section 4 at all.

Note: There are two chapters in this text featuring the important system calls and some standard library functions. Sections 2 and 3 provide detailed documentation on their usage. To look up the **read** system call, you'll have to use **man 2 read** or **man -s2 read**.

2.9.1 Understanding a man Page

A man page is divided into a number of compulsory and optional sections. Every command doesn't have all sections, but the first three (NAME, SYNOPSIS and DESCRIPTION) are generally seen in all man pages. NAME presents a one-line introduction to the command. SYNOPSIS shows the syntax used by the command, and DESCRIPTION (often the largest section) provides a detailed description.

The SYNOPSIS Section is the one that we need to examine closely, and we'll do that with reference to the man page of the **wc** command shown in Fig. 2.2. Here you'll find the syntax—the options and arguments used with the command. The SYNOPSIS follows certain conventions and rules which every user must understand:

- If a command argument is enclosed in rectangular brackets, then it is optional; otherwise, the argument is required. The **wc** man page shows all of its arguments enclosed in three such groups. This means that **wc** can be used without arguments.
- The ellipsis (a set of three dots) implies that there can be more instances of the preceding word. The expression `[ file ... ]` signifies that **wc** can be used with more than one filename as argument.
- If you find the | character in any of these areas, it means that only one of the options shown on either side of the pipe can be used. Here, only one of the options, `-c`, `-m` and `-C`, can be used.

All options used by the command are listed in the OPTIONS section. Often, difficult options are supported by suitable examples. There's a separate section named EXIT STATUS which lists possible error conditions and their numeric representation. You need to understand the significance

```
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)

of these numbers when writing shell scripts in order to determine the actual cause of termination of a command.

2.9.2 Using man to Understand man

Since **man** is also a UNIX command like **ls** or **cat**, you'll probably first like to know how **man** itself is used. Use the same command to view its own documentation:

```
man man                                    Viewing man pages with man
```

From this man page you'll know that you can choose your pager too. The variable, PAGER, controls the pager **man** uses, and if you set it to less, then **man** will use **less** as its pager. This is how you set PAGER at the command prompt before you invoke **man**:

```
PAGER=less ; export PAGER                  Set this shell variable and export it
man wc                                     and then run man
```

To evaluate the value of PAGER, use the command **echo $PAGER**. This setting is valid only for the current session. In later chapters, you'll understand the significance of the **export** statement and also learn to make this setting permanent so that its assigned value remains valid for all sessions.

Note: On some systems, **echo $PAGER** may not show you any value at all, in which case **man** is using a default pager. Some systems set this variable in the file /etc/default/man instead.

Table 2.1 Organization of the man Documentation

Section	*Subject (SVR4)*	*Subject (Linux)*
1	User programs	User programs
2	Kernel's system calls	Kernel's system calls
3	Library functions	Library functions
4	Administrative file formats	Special files (in /dev)
5	Miscellaneous	Administrative file formats
6	Games	Games
7	Special files (in /dev)	Macro packages and conventions
8	Administration commands	Administration commands

2.10 FURTHER HELP WITH man -k, apropos AND whatis

The POSIX specification requires **man** to support only one option (-k). Most UNIX systems also offer the **apropos** command that emulates **man -k**. When used with this option, **man** searches a summary database and prints a one-line description of the command. To know what **awk** does, use man like this:

```
$ man -k awk Same as apropos awk
awk             awk (1)         - pattern scanning and processing language
nawk            nawk (1)        - pattern scanning and processing language
```

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 multiple jobs concurrently (*multitask*).

3

General-Purpose Utilities

The best way to start acquiring knowledge of the UNIX command set is to try your hand at some of the general-purpose utilities of the system. These commands have diverse functions, but can be broadly divided into two categories. Some of them act as handy accessories that can perform calculations or handle your mail, for instance. Others tell you the state the system is in, and even manipulate it.

Every command featured in this chapter is useful, and has not been included here for cosmetic effect. Many of them have been reused in later chapters, especially in shell programming. A few are true dark horses. You'll need these commands in all situations in your daily life at the machine. The commands are simple to use, have very few options (except for **stty**), and don't require you to know much about the files they may access.

What You Will Learn

- Display the calendar of a month or year with **cal**.
- Display the current system date and time in a variety of formats using **date**.
- Use **echo** with escape sequences to display a message on the terminal.
- Use **bc** as a calculator with a decimal, octal or hexadecimal number as base.
- The basics of electronic mail and its addressing scheme.
- Handle your mail with the character-based **mailx** program.
- Change your password with **passwd**.
- Display the list of users currently working on the system with **who**.
- Display the characteristics of your operating system with **uname**.
- Display the filename of your terminal with **tty**.
- Use **stty** to display and change your terminal's settings.

Topics of Special Interest

- Features of the POSIX-recommended **printf** command as a portable replacement of **echo**.
- Record your login session including all keystrokes with **script**.

- 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

elapsed since the **Epoch**; this is January 1, 1970. A 32-bit counter stores these seconds (except on 64-bit machines), and the counter will overflow sometime in 2038.

You can display the current date with the **date** command, which shows the date and time to the nearest second:

```
$ date
Wed Aug 31 16:22:40 IST 2005
```

Time zone here is IST

The command can also be used with suitable format specifiers as arguments. Each format is preceded by the + symbol, followed by the % operator, and a single character describing the format. For instance, you can print only the month using the format +%m:

```
$ date +%m
08
```

or the month name:

```
$ date +%h
Aug
```

or you can combine them in one command:

```
$ date +"%h %m"
Aug 08
```

There are many other format specifiers, and the useful ones are listed below:

- `d`—The day of the month (1 to 31).
- `y`—The last two digits of the year.
- `H`, `M` and `S`—The hour, minute and second, respectively.
- `D`—The date in the format *mm/dd/yy*.
- `T`—The time in the format *hh:mm:ss*.

When you use multiple format specifiers (as shown in the previous example), you must enclos them within quotes (single or double), and use a single + symbol before it.

Note: You can't change the date as an ordinary user, but the system administrator uses the sam command with a different syntax to set the system date! This is discussed in Chapter 15.

3.3 echo: DISPLAYING A MESSAGE

We have used the **echo** command a number of times already in this text. This command is ofte used in shell scripts to display diagnostic messages on the terminal, or to issue prompts for takir user input. So far, we have used it in two ways:

- To display a message (like **echo Sun Solaris**).
- To evaluate shell variables (like **echo $SHELL**).

Originally, **echo** was an external command *(2.5)*, but today all shells have **echo** built-in. There are some differences in **echo**'s behavior across the shells, and most of these differences relate to the way **echo** interprets certain strings known as escape sequences.

An **escape sequence** is generally a two character-string beginning with a \ (backslash). For instance, \c is an escape sequence. When this escape sequence is placed at the end of a string used as an argument to **echo**, the command interprets the sequence as a directive to place the cursor and prompt in the same line that displays the output:

```
$ echo "Enter filename: \c"
Enter filename: $_                    Prompt and cursor in same line
```

This is how **echo** is used in a shell script to accept input from the terminal. Like \c, there are other escape sequences (Table 3.1). Here are two commonly used ones:

\t—A tab which pushes text to the right by eight character positions.

\n—A newline which creates the effect of pressing *[Enter]*.

All escape sequences are not two-character strings. ASCII characters can also be represented by their octal values (numbers using the base 8 contrasted with the standard decimal system which uses the base 10). **echo** interprets a number as octal when it is preceded by \0. For instance, *[Ctrl-g]*, which results in the sounding of a beep, has the octal value 7 (i.e., \07). You can use this value as an argument to **echo**, but only after preceding it with \0:

```
$ echo '\07'                          Double quotes will also do
..... beep heard .....
```

This is the first time we see ASCII octal values used by a UNIX command. Later, you'll see the **tr**, **awk** and **perl** commands also using octal values.

Caution: echo escape sequences are a feature of System V. BSD doesn't recognize them but it supports the -n option as an alternative to the \c sequence:

```
echo "Enter filename: \c"             System V
echo -n "Enter filename: "           BSD
```

Even though we don't use the disk version nowadays, the bad news is that the shells too respond in different ways to these escape sequences. Rather than go into these details, a word of caution from POSIX would be appropriate: Use **printf**.

LINUX: Bash, the standard shell used in Linux, interprets the escape sequences only when **echo** is used with the -e option:

```
echo -e "Enter your name:\c"
```

We'll be using these escape sequences extensively in this text, so if you are a Bash user (which most Linux users are), you must either commit this option to memory or make a special setting in the shell that makes **echo** behave in the normal way *(14.1—Tip)*.

Table 3.1 Escape Sequences Used by **echo** and **printf**

Escape Sequence	*Significance*
\a	Bell
\b	Backspace
\c	No newline (cursor in same line)
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Tab
\v	Vertical tab
\\	Backslash
\0*n*	ASCII character represented by the octal value *n*, where *n* can't exceed 0377 (decimal value 255)

3.4 printf: AN ALTERNATIVE TO echo

The **printf** command is available on most modern UNIX systems, and is the one you should use instead of **echo** (unless you have to maintain a lot of legacy code that use **echo**). Like **echo**, it exists as an external command; it's only the Bash shell that has **printf** built-in. The command in its simplest form can be used in the same way as **echo**:

```
$ printf "No filename entered\n"
No filename entered
$ _
```

\n is explicitly specified

printf also accepts all escape sequences used by **echo**, but unlike **echo**, it doesn't automatically insert a newline unless the \n is used explicitly. Though we don't need to use quotes in this example, it's good discipline to use them. **printf** also uses formatted strings in the same way the C language function of the same name uses them:

```
$ printf "My current shell is %s\n" $SHELL
My current shell is /usr/bin/bash
```

No comma before $

The %s format string acts as a placeholder for the value of $SHELL (the argument), and **printf** replaces %s with the value of $SHELL. %s is the standard format used for printing strings. **printf** uses many of the formats used by C's **printf** function. Here are some of the commonly used ones:

%s — String
%30s — As above but printed in a space 30 characters wide
%d — Decimal integer
%6d — As above but printed in a space 6 characters wide
%o — Octal integer
%x — Hexadecimal integer
%f — Floating point number

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

bc performs only integer computation and truncates the decimal portion that it sees. This shows up clearly when you divide two numbers:

```
9/5
1                                    Decimal portion truncated
```

To enable floating-point computation, you have to set `scale` to the number of digits of precision before you key in the expression:

```
scale=2                              Truncates to 2 decimal places
17/7
2.42                                 Not rounded off, result is actually 2.42857.....
```

bc is quite useful in converting numbers from one base to another. For instance, when setting IP addresses *(17.1.3)* in a network, you may need to convert binary numbers to decimal. Set `ibase` (input base) to 2 before you provide the number:

```
ibase=2
11001010
202                                  Output in decimal—base 10
```

The reverse is also possible, this time with `obase`:

```
obase=2
14
1110                                 Binary of 14
```

In this way, you can convert from one base to the other (not exceeding 16). **bc** also comes with a library for performing scientific calculations. It can handle very, very large numbers. If a computation results in a 900-digit number, **bc** will show each and every digit!

3.6 script: RECORDING YOUR SESSION

This command, virtually unknown to many UNIX users, lets you "record" your login session in a file. This command is not included in POSIX, but you'll find it useful to store in a file all keystrokes as well as output and error messages. You can later view the file. If you are doing some important work and wish to keep a log of all your activities, you should invoke this command immediately after you log in:

```
$ script
Script started, file is typescript
$ _                                  Another shell—child of login shell
```

The prompt returns and all your keystrokes (including the one used to backspace) that you now enter here get recorded in the file `typescript`. After your recording is over, you can terminate the session by entering **exit**:

```
$ exit
Script done, file is typescript      Or use [Ctrl-d]
$ _                                  Back to login shell
```

You can now view this file with the **cat** command. **script** overwrites any previous typescript that may exist. If you want to append to it, or want to use a different log file, you can consider using these arguments:

```
script -a                    Appends to existing file typescript
script logfile               Logs activities to file logfile
```

There are some activities that won't be recorded properly, for instance, the commands used in the full-screen mode (like **vi** and **emacs**).

3.7 EMAIL BASICS

A UNIX system is used by multiple users, so communication through the system seems natural and necessary. It's no wonder that electronic mail (email) is one of the first applications that UNIX users are familiar with. Email is fast and cheap, and can be used today to exchange graphics, sound and video files. Because a message is usually received in a few seconds, post office mail is often referred to as "snail mail".

An email message never appears on your terminal the moment it is received. It is deposited in your *mailbox* even when you are not logged in. The shell regularly checks this mailbox, and when it detects the arrival of new mail, it issues a message similar to this:

```
You have new mail in /var/mail/romeo          romeo is the user
```

You can see this message either when you log in or after a program run by you completes execution. It's good practice to see incoming mail immediately upon arrival rather than defer it for future viewing. After viewing it, you can do these things with it:

- Reply to the sender and all recipients.
- Forward it to others.
- Save it in a mailbox folder or separate file.
- Delete it.
- Print it.
- Add the address of the sender and all recipients to the address book.
- Call up a helper application to view it if it is not in plain text format.

The characteristic feature of most character-based mail programs (like **mailx**, **mail**, **pine**, etc.) is that mail doesn't remain in the mailbox after it is viewed. The message moves from the mailbox to the *mbox* (a secondary file), where it remains until it is explicitly deleted.

3.7.1 Mail Addressing Scheme

A recipient is identified by her email address. The addressing scheme in the early days simply used the recipient's username as the email address. However, in a single network, the email address also uses the machine name as a component. Here are the two forms:

```
mailx henry                    henry on same host
mailx henry@saturn             henry on networked host saturn
```

The second example uses a combination of the username and machine name (called *hostname*); this email address is unique in a network. A similar form is also used in Internet addressing except that the Internet uses *domain names* (called FQDNs) instead of simple hostnames:

```
mailx henry@heavens.com        Email address unique on the Internet
```

Because of the way domains (like *heavens*) are allotted, no two individuals or organizations can have the same email address on the Internet. We'll now discuss **mailx**, a standard mail handling tool.

3.8 mailx: THE UNIVERSAL MAILER

One of the strengths of UNIX lies in its command line tools. Even though we may use a sophisticated mailer (like Netscape or Mozilla) for handling our mail, we also need a simple mail program that can run noninteractively from shell scripts. We should be able to execute this program with its various command line options, using redirection and piping, to generate the mail headers and message body on the fly. In this chapter, we discuss **mailx**, a character-based mail agent that can do that.

mailx finds place in the POSIX specification, which no longer requires UNIX systems to support the earliest mail agent of all—the **mail** command. Linux doesn't support **mailx**, so you must use **mail** or BSD-based **Mail**, which have a lot in common with **mailx**.

There are two ways of invoking **mailx**—in the sending and receiving modes. In the sending mode, **mailx** is used with the email address of the recipient as argument. In the receiving mode, you *generally* use it without arguments to handle your received mail. In this mode, you can perform all mail functions listed at the beginning of Section 3.7. In the following paragraphs, we'll discuss both modes while communicating with another user on the same host.

3.8.1 Sending Mail

In the sending mode, **mailx** turns interactive, prompting for the subject first. You have to key it in before entering the message body. The same conventions apply: Use the standard input to key in your input and then use *[Ctrl-d]* (or a solitary .). This is how user henry sends mail to charlie:

```
$ mailx charlie                charlie is on same host
Subject: New System
The new system will start functioning from next month.
Convert your files by next week - henry
[Ctrl-d]
EOT                            Some systems use a dot here
                               System indicates end of text
```

Sending a message is as simple as this. The sent message doesn't directly appear on charlie's terminal but lands in his mailbox, which is usually /var/mail/charlie. How charlie handles this mail is considered shortly.

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
Saved 1 message in /users1/home/staff/charlie/mbox
$ _
```

Quitting `mailx` *with* `q`

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
Subject: Re: File Conversion
```

Sender's address automatically inserted

```
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 that is already known to others, you should change it immediately. This is done with the **passwd** command:

```
$ passwd
passwd:  Changing password for kumar
Enter login password: *******                    Asks for old password
New password: ********
Re-enter new password: ********
passwd (SYSTEM): passwd successfully changed for kumar
```

passwd (note the spelling) expects you to respond three times. First, it prompts for the old password. Next, it checks whether you have entered a valid password, and if you have, it then prompts for the new password. Enter the new password using the password naming rules applicable to your system. Finally, **passwd** asks you to reenter the new password. If everything goes smoothly, the new password is registered by the system.

When you enter a password, the string is *encrypted* by the system. Encryption generates a string of seemingly random characters that UNIX subsequently uses to determine the authenticity of a password. This encryption is stored in a file named shadow in the /etc directory. Even if a user is able to see the encryption in the file, she can't work backwards and derive the original password string from the encryption.

Note: This is the first time you have changed the *state* of the system; you have indirectly modified a file (shadow) *that is otherwise not available to you for direct editing*. There's a special feature of UNIX that allows you to do that, and we'll be examining it at the end of Part I of this text.

3.9.1 Password Framing Rules and Discipline

Depending on the way they are configured, many systems conduct certain checks on the string that you enter as password. There is often a minimum length of the string. Many systems insist on using a mix of letters with numerals. They may either disallow you from framing easy-to-remember passwords or advise you against choosing a bad password. The following messages are quite common:

```
passwd(SYSTEM): Password too short - must be at least 6 characters.
passwd(SYSTEM): Passwords must differ by at least 3 positions.
passwd(SYSTEM): The first 6 characters of the password must contain at least
two alphabetic characters and at least one numeric or special character.
BAD PASSWORD: it is based on a dictionary word.
BAD PASSWORD: is too similar to the old one.
passwd(SYSTEM): Too many failures - try later.
```

These messages suggest that you are not able to choose any password you like. These are some of the rules that you are expected to follow when handling your own password:

- Don't choose a password similar to the old one.
- Don't use commonly used names like names of friends, relatives, pets and so forth. A system may check with its own dictionary and throw out those passwords that are easily guessed.

- Use a mix of alphabetic or numeric characters. Enterprise UNIX systems won't allow passwords that are wholly alphabetic or numeric.
- Don't write down the password in an easily accessible document.
- Change the password regularly.

You must remember your password, but if you still forget it, rush to your system administrator. You'll learn later of the terrible consequences that you may have to face if people with mischievous intent somehow come to know what your password is. The command also behaves differently when used by the system administrator; it doesn't ask for the old password. The **passwd** command is revisited in Chapter 15.

3.10 who: WHO ARE THE USERS?

UNIX maintains an account of all users who are logged on to the system. It's often a good idea to know their user-ids so you can mail them messages. The **who** command displays an informative listing of these users:

```
$ who
root     console    Aug  1 07:51    (:0)
kumar    pts/10     Aug  1 07:56    (pc123.heavens.com)
sharma   pts/6      Aug  1 02:10    (pc125.heavens.com)
project  pts/8      Aug  1 02:16    (pc125.heavens.com)
sachin   pts/14     Aug  1 08:36    (mercury.heavens.com)
```

The first column shows the usernames (or user-ids) of five users currently working on the system. The second column shows the device names of their respective terminals. These are actually the filenames associated with the terminals. kumar's terminal has the name pts/10 (a file named 10 in the pts directory). The third, fourth and fifth columns show the date and time of logging in. The last column shows the machine name from where the user logged in. Users can log in remotely to a UNIX system, and all users here except root have logged in remotely from four different machines.

While it's a general feature of most UNIX commands to avoid cluttering the display with header information, this command does have a header option (-H). This option prints the column headers, and when combined with the -u option, provides a more detailed list:

```
$ who -Hu
NAME     LINE       TIME          IDLE    PID  COMMENTS
root     console    Aug  1 07:51  0:48  11040      (:0)
kumar    pts/10     Aug  1 07:56  0:33  11200      (pc123.heavens.com)
sachin   pts/14     Aug  1 08:36   .    13678      (mercury.heavens.com)
```

Two users have logged out, so it seems. The first five columns are the same as before, but it's the sixth one (IDLE) that is interesting. A . against sachin shows that activity has occurred in the last one minute before the command was invoked. kumar seems to be idling for the last 33 minutes. The PID is the process-id, a number that uniquely identifies a process. You have seen it when you used the **ps** command in Section 1.4.8 to list the processes running at your terminal.

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.

Changing the End-of-File Key (eof) When using **mailx**, you used *[Ctrl-d]* to terminate input. This eof character is also selectable. Instead of *[Ctrl-d]*, you can use *[Ctrl-a]* as the eof character:

```
stty eof \^a
```

[Ctrl-a] will now terminate input for those commands that expect input from the keyboard when invoked in a particular way. The **cat**, **bc** and **mailx** commands can be made to work in this way.

When Everything Else Fails (sane) **stty** also provides another argument to set the terminal characteristics to values that will work on most terminals. Use the word sane as a single argument to the command:

```
stty sane                                    Restores sanity to the terminal
```

There are other options, but you are advised against tampering with too many settings.

3.14 CONCLUSION

The commands featured in this chapter were all used in standalone mode. But UNIX provides a number of commands (called *filters*) that can be combined with one another to process command output. In fact, UNIX is most effective when used in these ways. In later chapters, we'll often be combining commands to perform useful tasks. The next four chapters examine one of the two illusions *(2.1.2)* that support the UNIX system—files.

WRAP UP

cal produces the calendar of a month or year.

date can display any component of the system date and time in a number of formats.

echo displays a message on the screen. It can work with escape sequences like \c and \t as well as octal numbers like \007. The command has portability problems, the reason why **printf** should be used. **printf** works like **echo** but can also use format specifiers like %d and %s.

bc is the calculator. You can use any numbering system or degree of precision you want. You can also store intermediate results in variables.

script is the UNIX system's recorder which logs all activities of a user in a separate file.

Electronic mail can be used to send messages to a user on the same host, any host in a network or on the Internet. A mail message is appended to a *mailbox* from where it moves to *mbox*.

mailx is invoked with the email address of the recipient in the sending mode, and without arguments in the receiving mode. The command can also be used noninteractively from shell scripts.

passwd is used to change a user's password but is not displayed on the screen. The command prompts for the old password before the new one.

who shows the users working on the system and the time of logging in.

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