Davis & Rajkumar
5th edition

UNIX/Linux Commands and Utilities

WHEN YOU FINISH READING THIS CHAPTER YOU SHOULD BE ABLE TO:

- Relate the UNIX shell to the command processor introduced in Chapter 5.
- Describe the general form of a UNIX command.
- Briefly describe a hierarchical directory structure.
- Distinguish between a path name and a file name.
- Distinguish between the root directory, your home directory, and your working directory.
- Explain the significance of the (.) and (..) file names.
- Distinguish between creating a directory and creating a file.
- Change working directories.
- Explain how wild-card characters are used.
- Briefly explain redirection.
- Explain filters and pipes.
- Briefly explain what a shell script is.
- Explain the advantage of running selected programs in the background.

UNIX

UNIX was developed at Bell Laboratories, a division of AT&T, in the 1970s. Largely the work of two individuals, Ken Thompson and Dennis Ritchie, the system's main thrust was providing a convenient working environment for programming. Today, it is an important standard that has influenced the design of many modern operating systems. For example, MS-DOS incorporates numerous UNIX features. Experienced programmers consider UNIX simple, elegant, and easy to learn. Beginners, on the other hand, sometimes find it terse and not very friendly.

Linux

In 1991, Linus Torvald, then a student at the University at Helsinki, created a version of Unix to run on the Intel 386 chip. He released the source code on the Internet as **Linux**. Since then it has been refined and modified and today incorporates contributions from hundreds of software developers around the world. Linux is open-source software. Openness implies that the complete source code is freely available and the operating system can be extended through common interfaces. Today Linux has been ported to run on most platforms including the personal computer.

The initial stable version of the Linux, **kernel** 1.0, was released in 1994. Kernel refers to the core of the operating system and not the applications such as the shell, compilers, and other programs that run on the kernel. Commercial vendors such as RedHat, Caldera, and SuSe have taken these stable versions and packaged the software for easy installation on a wide variety of personal computers and added other enhancements that make Linux useful. The version numbers of these vendors are different. For example, all of the examples in this chapter are run under RedHat's version 6.1, which uses a Linux kernel 2.2.12.

The UNIX Shell

UNIX commands are processed by a **shell** that lies between the user and the resident operating system (Figure 10.1). The shell is not really part of the operating system, so it can be changed. Professional programmers might choose a technical shell. Beginners might prefer selecting commands from a menu or pointing at pictures (icons). The idea of a command processor that is independent from the operating system was an important UNIX innovation.

Two shells are in common use. The standard shell, sometimes called the **Bourne shell**, was developed at Bell Laboratories. A second, the **C shell**, is related to the C programming language. They are similar; in fact, you can follow the chapter tutorial and complete the end-of-chapter exercises using either one. As you become more experienced, you will want to write shell programs, and at that point, the differences become significant.

Figure 10.2 shows the general form of a UNIX command. The system **prompt** (often, a dollar sign for the Bourne shell or a percent sign for the C shell) is displayed by UNIX. Command names are generally terse (ed for editor, cp for copy a file), but meaningful. One or more spaces separate the command name from the options. If they are included, the options are usually preceded by a minus sign to distinguish them from the arguments. Most options are designated by a single lower case letter, and multiple options can be coded. One or more spaces separate the options from the arguments, which generally consist of one or more file names.

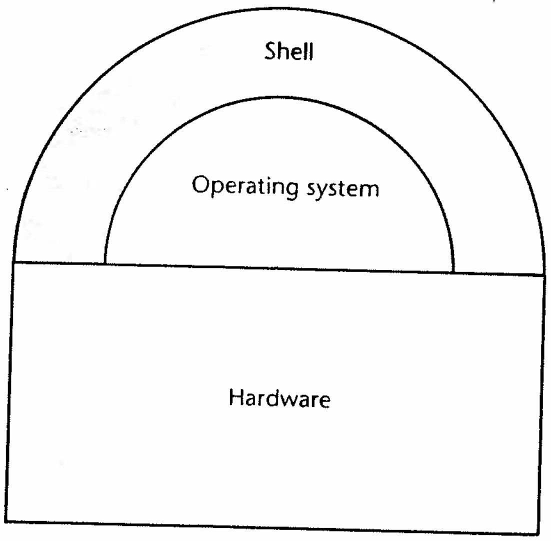


Figure 10.1
UNIX commands are processed by a shell that lies between the user and the resident operating system.

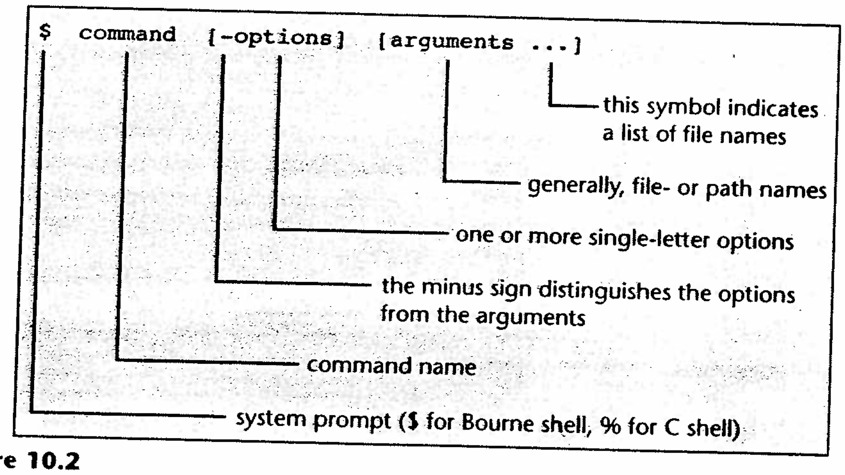

$ command [-options] [arguments ...]
this symbol indicates a list of file names
generally, file- or path names
one or more single-letter options
the minus sign distinguishes the options from the arguments
command name
system prompt ($ for Bourne shell, % for C shell)

Figure 10.2
The general form of a UNIX command.

The Chapter Tutorial

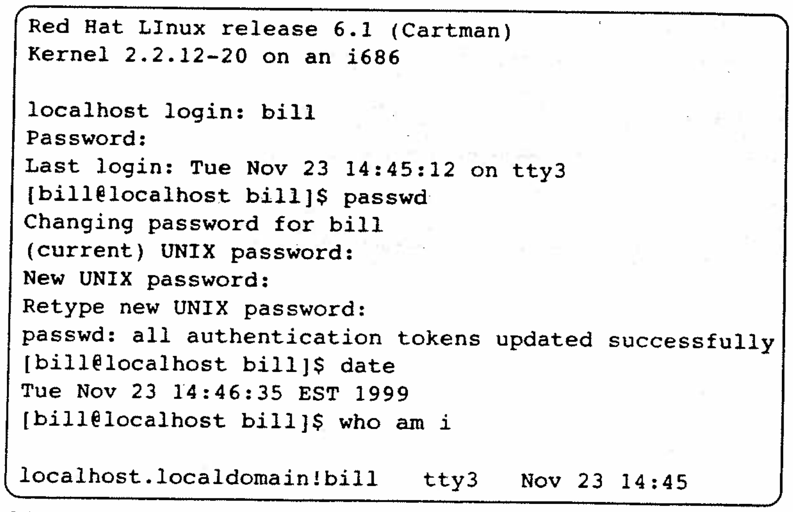
This introduction to UNIX commands and utilities is presented as a tutorial. The examples were run on a Dell computer under Linux and the Bourne shell. Don't just read it. Instead, find a Linux or UNIX system and, as you read about a command, enter it and see for yourself how the computer responds. You'll need a user name and a password; see your instructor or your system's super user. Later, you'll find Appendix C a useful reference.

Logging On

Usually, a system administrator or super user is responsible for such start-up procedures as booting the system and setting the date and time, so the UNIX user can ignore these tasks. When you sit down at a terminal, the system should be up and running.

Every UNIX session begins with a request for a **login name** and a **password** (Figure 10.3). In response to the first prompt, type your login name and press return. Next, you'll be asked for your password. Type it and press return. For security reasons, passwords are never displayed.

On some systems, you'll be expected to select your own password the first time you log on. Use any combination of up to eight keyboard characters. UNIX prefers relatively lengthy passwords and may ask you to try again if you suggest less than six characters.



```
Red Hat LInux release 6.1 (Cartman)
Kernel 2.2.12-20 on an i686

localhost login: bill
Password:
Last login: Tue Nov 23 14:45:12 on tty3
[bill@localhost bill]$ passwd
Changing password for bill
(current) UNIX password:
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully
[bill@localhost bill]$ date
Tue Nov 23 14:46:35 EST 1999
[bill@localhost bill]$ who am i

localhost.localdomain!bill    tty3    Nov 23 14:45
```

Figure 10.3

Normally, your system's super user will assign you a login name and a password. This screen shows a normal log on sequence and illustrates several commands.

Figure 10.3 shows a normal log on sequence and then illustrates several commands. Use the **passwd** utility (Figure 10.4) to change your password. The **date** utility (Figure 10.5) displays the system date and time. To identify users currently logged on your system, type **who** (Figure 10.6). A user working on more than one project may have two or more login names, and that can be confusing. The command

```
who am i
```

displays the user's current login name.

The `write` utility (on some systems, it's `talk`) allows two users to exchange real-time messages, while `mail` sends and receives electronic mail. Many larger UNIX systems feature an on-line reference manual. To obtain a description of any utility, code `man` followed by the utility name. For example,

```
man who
```

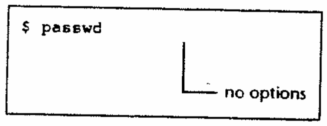
displays a description of the who utility.

Two shells, the Bourne shell and the C shell, are considered UNIX standards; on most systems, one of them is started after logon. While the examples in this text will work under either shell, you can select one. To activate the Bourne shell, type `sh`. To switch to the C shell, type `csh`.

The Bourne Again Shell (bash), which is based on the Bourne shell, is the standard on Linux systems and is used in the examples in this chapter.

Figure 10.4

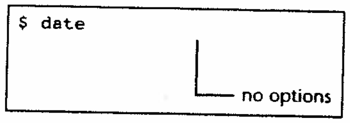
Use the passwd utility to change your password.



```
$ passwd
```

Figure 10.5

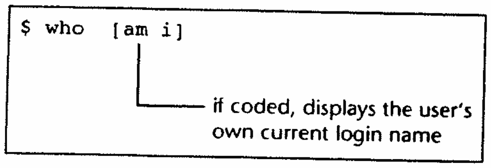
The date utility displays the system date and time.



```
$ date
```

Figure 10.6

who displays the names of users currently logged on the system.



```
$ who  [am i]
```

The File System

The UNIX **file system** allows a user to identify, save, and retrieve files by name. (A program is a type of file.)

File Names

A UNIX **file name** (Figure 10.7) consists of from 1 to 256 characters. (Earlier versions of UNIX had a 14-character limit.) Don't use slashes (/), and avoid starting a file name with a minus sign or hyphen; otherwise, virtually any combination of characters you can type is legal. Note that UNIX distinguishes between upper case and lower case, so "A" and "a" are different. If you include a period, the characters following it are considered the file name **extension.** The extension is significant to some compilers and to the linkage editor; otherwise, it's simply part of the file name. An **invisible file**'s name starts with a period. Invisible file names are not normally displayed when a directory is listed.

Directories

Imagine a user who maintains several different types of files. Letters and other correspondence are generated by a text editor, chapters for a book are output by a word processor, and C programs form another group. Dozens, perhaps even hundreds of different users will have similar needs. Keeping track of all those files in a single directory is almost impossible. Instead, UNIX uses a flexible hierarchical **directory** structure (Figure 10.8).

The structure begins with a **root directory**. "Growing" from the root are several "children." Some hold references to utilities and other system routines. One, `home`, holds all the user directory names. Note that `bill` is `home's` child, a grandchild of the root directory. Under `bill` come subdirectories to hold letters, book chapters, and programs. Incidentally, a directory is a special type of file, so the rules for naming directories and files are the same. The `usr` directory is for files that are shareable across a whole site.

Figure 10.7
The rules for defining UNIX file names.

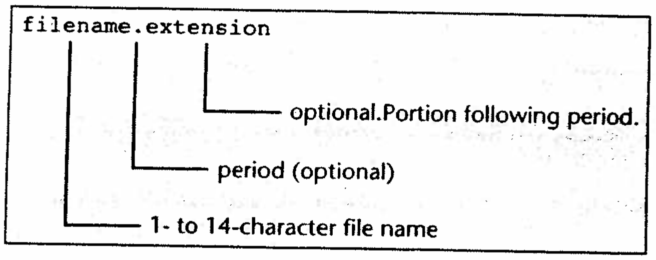

filename.extension
optional.Portion following period.
period (optional)
1- to 14-character file name

Fig
UN
dir

Fig
Be
hie
str
co
un

Path Names

With all these directories, however, you need more than a simple name to find a file. For example, it is possible to have files named `pay` recorded under two different directories. A reference to `pay` would thus be ambiguous—which `pay` do you mean? To uniquely identify a file, you need a complete **path name** (Figure 10.9), for example,

```
/home/bill/letters/pay
```

Look at Figure 10.8, and follow the path name. The first slash (/) indicates the root directory. Move down to `home`, then `bill`, then `letters`, and finally to the file.

At first glance, subdirectories seem to complicate rather than simplify accessing files. In practice, however, people rarely use such lengthy pathnames. Instead, when you log on, UNIX selects your **home directory** (its name usually matches your

Figure 10.8
UNIX uses a hierarchical directory structure.

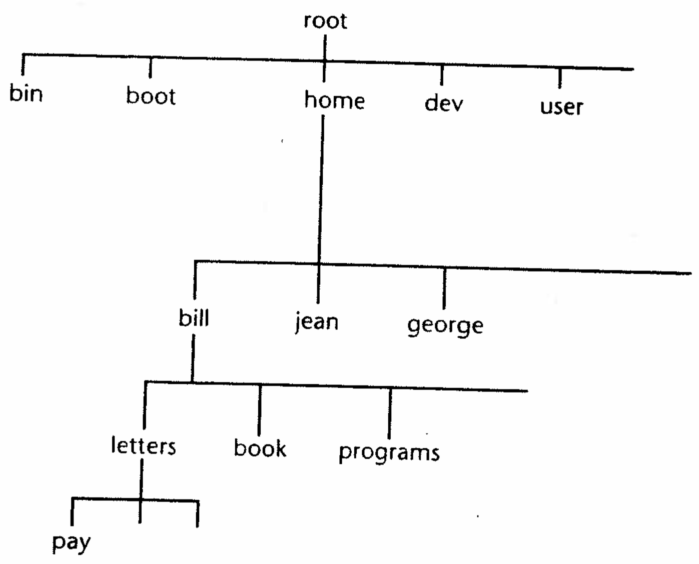
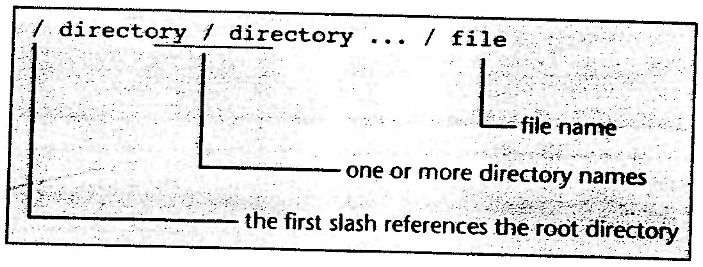


Figure 10.9
Because UNIX uses a hierarchical directory structure, you must specify a complete path name to uniquely identify a file.


/ directory / directory ... / file
file name
one or more directory names
the first slash references the root directory

login name) as your initial **working directory.** Unless it is told otherwise, the operating system searches for files starting with your working directory; thus,

```
letters/pay
```

is all you need to find file `pay`. Later in the chapter, you'll see how to change working directories.

Viewing a Directory

Before you begin creating and manipulating directories, look through some existing ones. Start by printing or displaying your working directory. Just type

```
pwd
```

and then press return; the results are shown in Figure 10.10 (your working directory name will be different).

Even if this is the first time you've logged on, your home directory should contain a few files. To view their names, type an `ls` (list directory) command (Figure 10.11):

```
ls -a
```

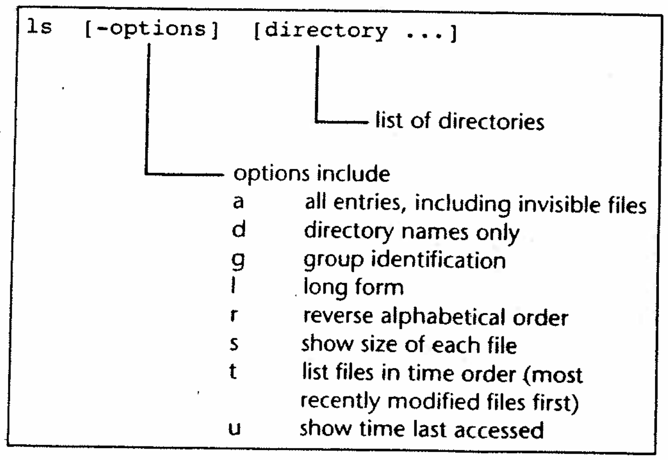
```
[bill@localhost bill]$ pwd
/home/bill
[bill@localhost bill]$
```

Figure 10.10

The pwd (print working directory) command displays the path name of your current working directory. Note: On your UNIX system, your working directory may have a different path name.

Figure 10.11

The list directory (ls) command displays, normally in alphabetical order, the names of the files in the specified directories. If no directories are coded, the contents of the current working directory are listed.



```
ls  [-options]  [directory ...]
```

list of directories

options include

a	all entries, including invisible files
d	directory names only
g	group identification
l	long form
r	reverse alphabetical order
s	show size of each file
t	list files in time order (most recently modified files first)
u	show time last accessed

The output is shown in Figure 10.12 (your output may differ). The file names that begin with a period are usually invisible; had the -a option not been coded, they would not have been listed.

Two files, (.) and (..), are particularly interesting. The single period stands for the working directory; the double period is a synonym for its parent. They are useful shorthand for writing path names.

Try a few variations; some sample results are shown in Figure 10.13. For example, code

```
ls
```

with no options. File names beginning with a period should no longer appear. In fact, on many systems, absolutely nothing is displayed because initially there are no regular files. If you ask for a list of files, and there are none, UNIX displays nothing, not even a "directory empty" message. To experienced programmers, that makes sense. A beginner might find such terseness a bit intimidating, however.

To list only directories, code a d option

```
ls -d
```

Next, take a look at the long form:

```
ls -l
```

Finally, list everything, including invisible files, in long form:

```
ls -la
```

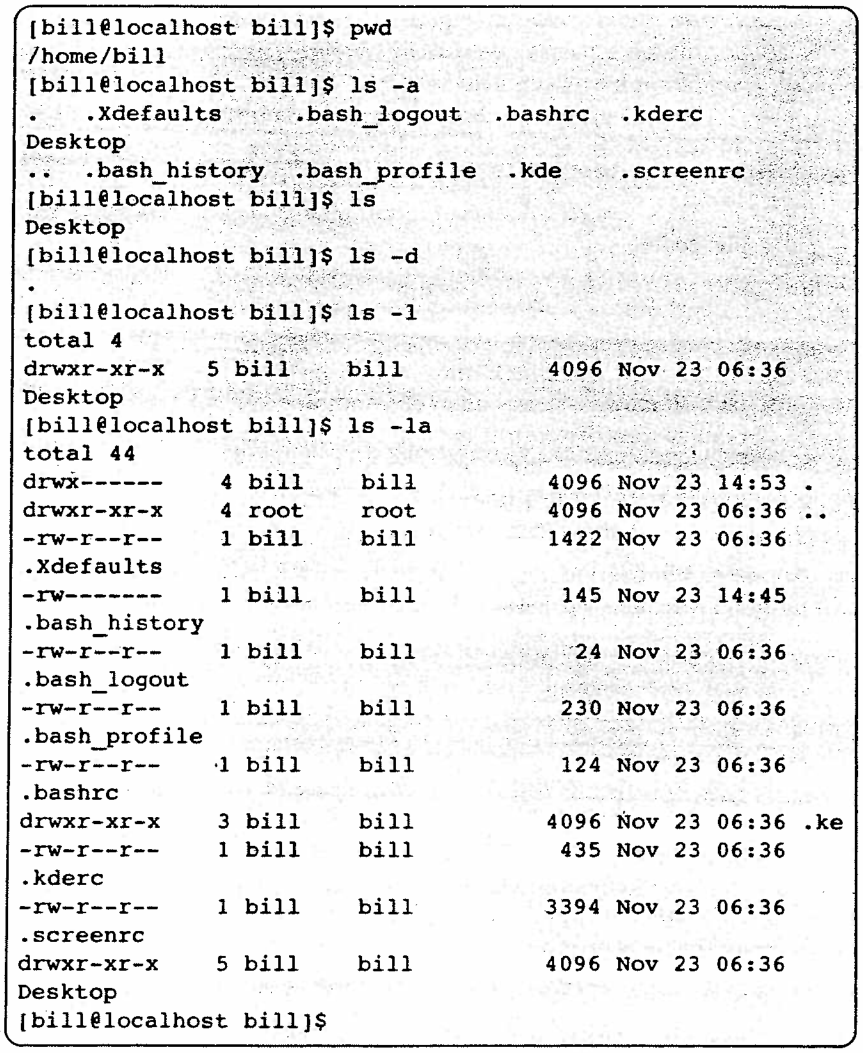
To indicate more than one option, simply code all the option letters one after another.

A "long form" line shows a file's owner, size, and the date and time it was most recently modified. The first 10 characters indicate the file type and its access permissions (Figure 10.14). The file can be an ordinary file (data or a program), a directory, or a special file that corresponds to an input or output device. Three sets of

```
[bill@localhost bill]$ pwd
/home/bill
[bill@localhost bill]$ ls -a
.   .Xdefaults     .bash_logout   .bashrc   .kderc
Desktop
..  .bash_history  .bash_profile  .kde      .screenrc
[bill@localhost bill]$
```

Figure 10.12

Following an ls command, the names of the files in the referenced directory are listed in alphabetical order.



```
[bill@localhost bill]$ pwd
/home/bill
[bill@localhost bill]$ ls -a
.    .Xdefaults    .bash_logout   .bashrc   .kderc
Desktop
..   .bash_history  .bash_profile  .kde      .screenrc
[bill@localhost bill]$ ls
Desktop
[bill@localhost bill]$ ls -d
.
[bill@localhost bill]$ ls -l
total 4
drwxr-xr-x    5 bill     bill          4096 Nov 23 06:36
Desktop
[bill@localhost bill]$ ls -la
total 44
drwx------     4 bill     bill          4096 Nov 23 14:53 .
drwxr-xr-x     4 root     root          4096 Nov 23 06:36 ..
-rw-r--r--     1 bill     bill          1422 Nov 23 06:36
.Xdefaults
-rw-------     1 bill     bill           145 Nov 23 14:45
.bash_history
-rw-r--r--     1 bill     bill            24 Nov 23 06:36
.bash_logout
-rw-r--r--     1 bill     bill           230 Nov 23 06:36
.bash_profile
-rw-r--r--     1 bill     bill           124 Nov 23 06:36
.bashrc
drwxr-xr-x     3 bill     bill          4096 Nov 23 06:36 .ke
-rw-r--r--     1 bill     bill           435 Nov 23 06:36
.kderc
-rw-r--r--     1 bill     bill          3394 Nov 23 06:36
.screenrc
drwxr-xr-x     5 bill     bill          4096 Nov 23 06:36
Desktop
[bill@localhost bill]$
```

Figure 10.13
Experiment with the `ls` command, coding several options.

permissions are included—one for the file's owner, a second for users in the owner's group, and a third for all other users. Based on the recorded values, a given user or group can be granted read (r), write (w), and/or execute (x) permission. A minus sign indicates no permission. To change access permissions, use the `chmod` utility.

Creating Directories

Next, create three subdirectories under your home directory. Type a **`mkdir`** (make directory) command (Figure 10.15)

```
mkdir letters book programs
```

Note that the command is followed by a list of directory names separated by spaces. UNIX will respond by creating the requested directories, but will not display a confirmation message. (You didn't tell it to.)

To find out if the directories actually were created, type

```
ls
```

The output is shown in Figure 10.16. Compare it with Figure 10.13; clearly, the three directories now exist.

The `mkdir` utility creates a directory. Use `rmdir` to remove or delete one.

Figure 10.14

The first 10 characters in a long form directory line indicate the file's type and access permissions. Use chmod to change them.

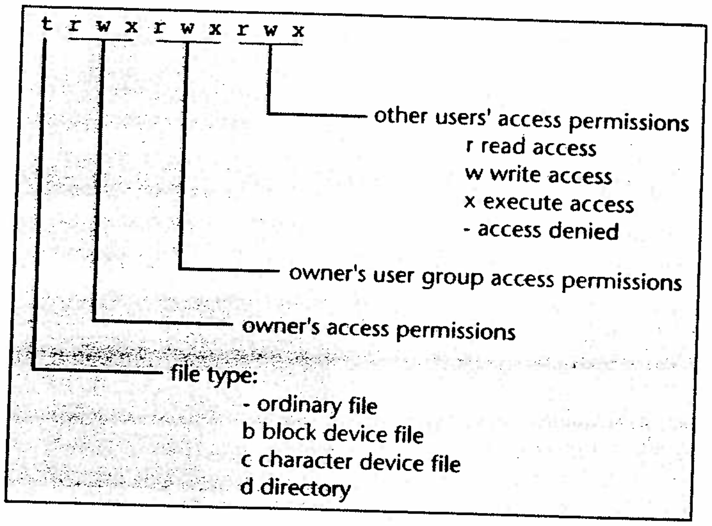
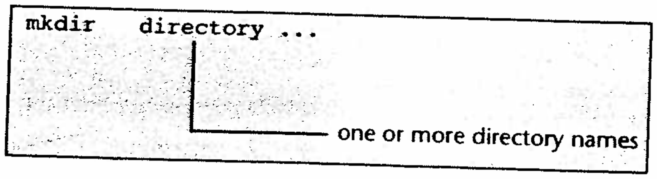


Figure 10.15

Directories are created by the make directory (mkdir) utility.



Changing Working Directories

The `ls` command lists the contents of the current working directory. Use a **cd** command (Figure 10.17) to change the current working directory. For example, code

```
cd /
```

The slash identifies the root directory. UNIX will display no confirmation message, but the root directory will be your new working directory. Now code

```
ls -a
```

The output (on the author's system) is shown in the top half of Figure 10.18.

Next, switch to a lower level, or child, directory. Type

```
cd /home
```

On many UNIX systems, `home` contains each user's home directory. Once again, list your working directory's contents; the output (on the author's system) is shown in the bottom half of Figure 10.18.

Figure 10.16
After a make directory command is executed, a directory list should reveal the new directories' names.

```
[bill@localhost bill]$ mkdir letters book programs
[bill@localhost bill]$ ls
Desktop book letters programs
[bill@localhost bill]$
```

Figure 10.17
Use a cd (change directory) command to switch to a new working directory.

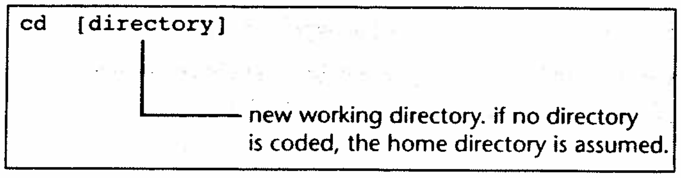

cd [directory]
new working directory. if no directory is coded, the home directory is assumed.

Figure 10.18
This screen shows the contents of the root directory and of directory **home** on the author's UNIX system. You may see a different list of files on your system.

```
[bill@localhost /usr]$ cd /
[bill@localhost /]$ ls -a
.   bin  dev  home  lost+found  opt  root  tmp  var
..  boot etc  lib   mnt         proc sbin  usr
[bill@localhost /]$ cd /home
[bill@localhost /home]$ ls -a
.  ..  bill  raj
[bill@localhost/home]$
```

Finally, move back to your home directory. Type

```
cd
```

with no options or arguments. To verify that you've returned to your home directory, type a pwd (print working directory) command. Once again, list the directory's contents.

Creating Files

Most files are created by programs, such as editors, compilers, interpreters, word processors, spreadsheets, and database managers. Most UNIX systems incorporate both a line editor (ed) and a powerful full screen "visual" editor (**vi**). While an in-depth discussion of its features is beyond the scope of this book, you can use **vi** to create a few simple files. First, however, change the current working directory to `letters`. It's a subdirectory of your home directory. (If you've been following the tutorial, your home directory is your current working directory.) Unless you specify otherwise, UNIX always assumes a file reference starts with the current working directory, so

```
cd letters
```

changes the working directory to "letters."

Start by requesting the visual editor (Figure 10.19). For example, to create (in the working directory) a file named tom, code

```
vi tom
```

Except for a message at the bottom, the screen should go blank. The visual editor has two operating modes: command and insert. As you begin, you're in command mode. (We'll cover only a few essential commands.) To enter insert mode, press the *I* key. You'll get no confirmation, but you should be able to begin entering text. (Note: you may have to tell vi your terminal type—check with your instructor or your system administrator.)

Type anything you want. When you're done, exit insert mode by pressing the escape key (or, on some systems, a function key), and then type :wq (for write quit). Some systems accept a pair of capital Zs as a command to exit vi. You should see

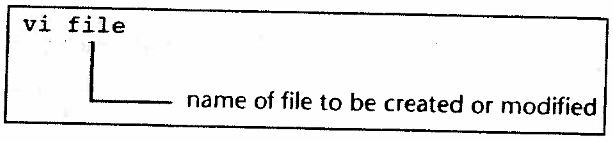


Figure 10.19
The visual editor (vi) can be used to create text files and [illegible] modules.

a system prompt indicating that you're back in the shell. Type an `ls` command to verify that the file is on disk. Repeat this procedure to create two more files named `dick` and `harriet` (Figure 10.20).

Some operating systems require a programmer or user to specify a file's size when creating it. That's not necessary under UNIX. You'll investigate how UNIX dynamically allocates disk space in Chapter 16.

Manipulating Files

Now that you've created some files, let's manipulate them. For example, the concatenate (**`cat`**) utility (Figure 10.21) displays the contents of selected files. To display `tom`, code

```
cat tom
```

The file should appear on your screen. To list the contents of more than one file, code a series of file names; for example,

```
cat tom dick harriet
```

produces something like the output shown in Figure 10.22. (You should see whatever you typed through `vi`.)

Imagine that a file named `resume` contains enough data to fill several screens. If you code

```
cat resume
```

much of the data will scroll off the screen before you can read it. To display the file's contents one screen at a time, code

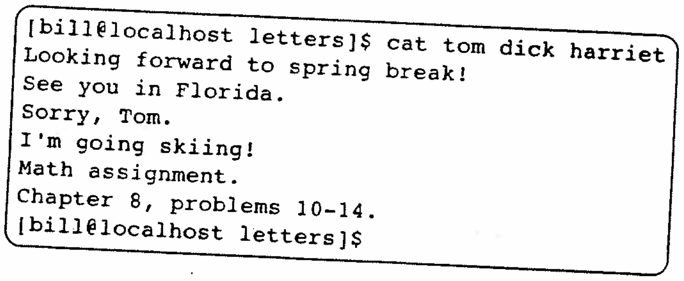
```
more resume
```

Figure 10.20
This screen shows the message displayed by the visual editor after it has created a file.

```
'tom' 2L, 53C written
[bill@localhost letters]$ ls -a
.  ..  dick  harriet  tom
[bill@localhost letters]$
```

Figure 10.21
Use the concatenate (cat) utility to display the contents of one or more files.

```
cat  [file ...]
       |
       └──── one or more file names
```



```
[bill@localhost letters]$ cat tom dick harriet
Looking forward to spring break!
See you in Florida.
Sorry, Tom.
I'm going skiing!
Math assignment.
Chapter 8, problems 10-14.
[bill@localhost letters]$
```

Figure 10.22
The word concatenate means "to join together." Thus, if several file names follow a cat command, the contents of all the files are displayed one after another.

Press the space bar to view another screen, or the delete key to end the program. The more utility is not available on all versions of UNIX. If your system doesn't have more, you might be able to suspend a display by pressing control-s, and resume the display by pressing control-q.

Your current working directory is `letters`. Switch back to your home directory by coding

```
cd
```

Now try

```
cat harriet
```

You should get an error message. Try

```
cat letters/harriet
```

You should get valid output. Why? List your working directory:

```
ls -a
```

Do you see a file named `harriet`? No. However, the directory `letters` does appear. If you follow a path from your current directory, through `letters`, you'll find `harriet`.

To copy a file, use the copy (**cp**) utility (Figure 10.23). For example, code

```
cp letters/dick book/chapt.2
```

Now,

```
cat book/chapt.2
```

should display the contents of the newly created file (Figure 10.24). List the directory

```
ls -a book
```

File `chapt.2` should appear in the list.

The same file can be referenced in more than one directory by creating a link (Figure 10.25). For example, code

```
ln letters/harriet book
```

followed by

```
ls -a book
```

and compare the output to Figure 10.24. You should see a new entry in the list. Copy duplicates a file. Link does not; it merely assigns another name to the same file by creating a new directory entry.

To rename a file, code an `mv` (move) command. To delete a file, code an `rm` (for remove link) command. If a file has multiple links, `rm` will not delete it; you'll still be able to access it through other links. When you remove the last link to a file, however, it's gone.

Wild-card characters make it easy to reference a number of related file names. A question mark (?) represents any single character; for example, the file name

```
term?
```

expands to `term1`, `term2`, `termc`, and any other file named `term` followed by any single character. An asterisk (*) represents multiple characters; for example,

```
term*
```

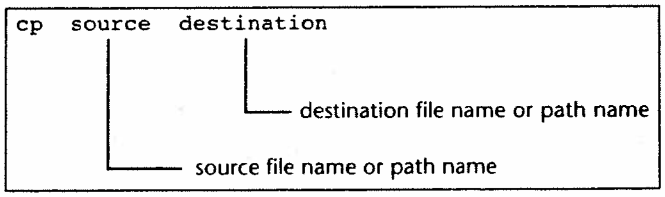

cp source destination
destination file name or path name
source file name or path name

Figure 10.23
To copy a file, use the copy (cp) utility.

Figure 10.24
When a file is copied, its name appears in the new directory.

```
[bill@localhost bill]$ cd
[bill@localhost bill]$ cp letters/tom book/chapt.2
[bill@localhost bill]$ cat book/chapt.2
Looking forward to spring break!
See you in Florida.
[bill@localhost bill]$ ls -a book
.  ..  chapt.2
[bill@localhost bill]$
```

stands for `term1`, `term.v6`, `term.abcdefgh`, and any other file named `term` followed by any combination of from 1 to 250 characters. (The limit, remember, is 255, and the period counts.)

Imagine you've been working on a C program. Your source module is called `mypgm.c`; the object module is `mypgm.o`. You want to copy both. You have two options. One is issuing two `cp` commands. The other is referencing `mypgm.*` or `mypgm.?`. Seeing the wild-card characters, the shell will look for all files that fit; thus a single wild-card name references both.

Pipes, Filters, and Redirection

Many UNIX utilities and commands assume a standard input or output device; for example, `cat` sends its output to the screen, while `vi` gets its input from the keyboard. By using **redirection** operators (Figure 10.26), a user can tell the shell to change those defaults.

Use `cat` to illustrate this feature. You already know that a `cat` command followed by a file name displays the contents of the file. Try coding `cat` with no options,

```
cat
```

Since no inputs or outputs are specified, the shell assumes the standard input and output devices (the keyboard and the screen). Thus, whatever you type is simply echoed back to the screen. Try typing a few lines (Figure 10.27). Press return at the end of each line; when you're finished, press control-D (the end-of-file sentinel value). Some UNIX systems echo line by line; others display all the output only after you press control-D; in either case, data are copied from the standard input to the standard output device.

Figure 10.25
The same file can be referenced in more than one directory by creating a link.

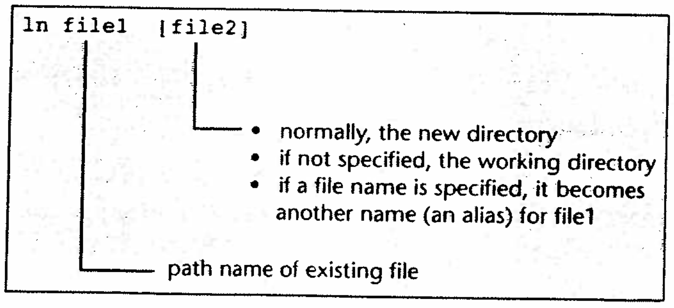

ln file1 [file2]
• normally, the new directory
• if not specified, the working directory
• if a file name is specified, it becomes another name (an alias) for file1
path name of existing file

Figure 10.26

Many UNIX commands and filters deal with the standard input and output devices. Redirection operators and pipes allow a user to change to a specified file or device.

Operator	*Meaning*	*Example*
<	change source to a specified file or device	<myfile
>	change destination to a specified file or device	>tempfile
>>	change destination, usually to an existing file, and append new output to it	>>master.pay
|	pipe standard output to another command or to a filter	cat file1| sort

Figure 10.27

This series of cat commands illustrates redirection.

```
[bill@localhost bill]$ cat
The quick brown fox
The quick brown fox
jumped over the lazy dog.
jumped over the lazy dog.
^D
[bill@localhost bill]$ cat >book/intro
There was a young man from Nantucket
who ...
^D
[bill@localhost bill]$ ls -a book
.  ..  chapt.2  intro
[bill@localhost bill]$
```

Redirect that output. Type

```
cat >book|intro
```

followed by several lines of text (Figure 10.27). Press control-D when you're finished. Now, list your directory

```
ls -a book
```

You should see the new entry, `intro`.

A **filter** accepts input from the standard input device, modifies (or filters) the data in some way, and sends the results to the standard output device. For example, consider **`sort`** (Figure 10.28). It's a utility that reads input from the specified file or files (or the standard input device), sorts them into alphabetical or numerical sequence, and outputs the sorted data to the screen. It can also be used as a filter.

A **pipe** causes one utility's standard output to be used as another utility's standard input. Pipes are designated by a vertical line (|). For example, earlier in the chapter, you coded

```
cat tom dick harriet
```

Repeat that command. Now, try

```
cat tom dick harriet | sort
```

As the bottom half of Figure 10.29 shows, the standard output has been routed through **sort**, and the file contents are displayed, line by line in alphabetical order.

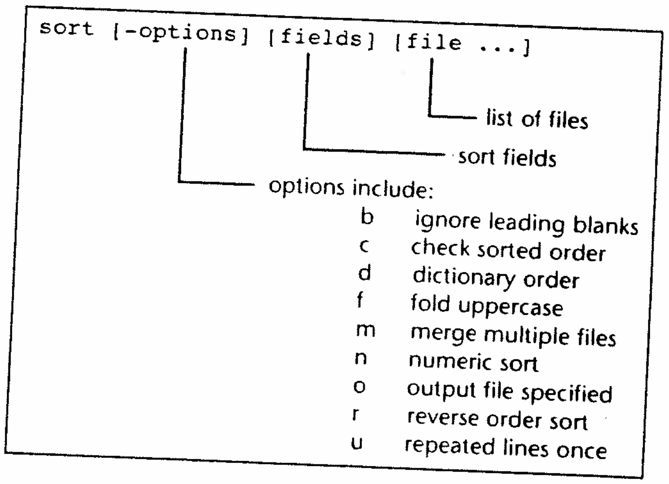
UNIX utilities can be viewed as tools. Each one performs a single function. Instead of writing a massive program to perform a series of functions, it makes sense to use the existing tools. Pipes, filters, and redirection make it easy to link programs and utilities. As you become more experienced with UNIX, you'll find many uses for them.

Incidentally, most utilities send error messages to the standard error device—usually, the screen. The shell does not redirect error messages. Thus, they won't be lost in an output file or down a pipe.

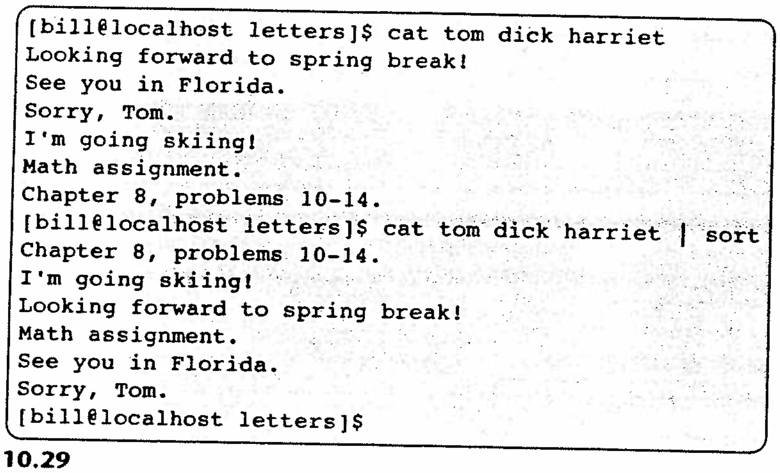
This marks the end of the chapter tutorial. When you're ready, log off the system by pressing control-D. An option is typing a `logout` command.

Figure 10.28

The sort utility can be used by itself or as a filter.


```
sort [-options] [fields] [file ...]
```

b	ignore leading blanks
c	check sorted order
d	dictionary order
f	fold uppercase
m	merge multiple files
n	numeric sort
o	output file specified
r	reverse order sort
u	repeated lines once



```
[bill@localhost letters]$ cat tom dick harriet
Looking forward to spring break!
See you in Florida.
Sorry, Tom.
I'm going skiing!
Math assignment.
Chapter 8, problems 10-14.
[bill@localhost letters]$ cat tom dick harriet | sort
Chapter 8, problems 10-14.
I'm going skiing!
Looking forward to spring break!
Math assignment.
See you in Florida.
Sorry, Tom.
[bill@localhost letters]$
```

Figure 10.29
With pipes, one utility's standard output becomes another utility's standard input.

Shell Scripts

Many data-processing applications are run daily, weekly, or at other regular intervals. Others—for example a program test—are repeated many times. When such applications are run, a set of commands must be issued. Repeating the application means repeating the commands. Retyping the same commands over and over again is annoying and error prone. An option is writing a **shell script**.

A shell script is a file that consists of a series of commands. (It's much like an MS/DOS .BAT file.) The shell is actually a highly sophisticated, interpretive programming language with its own variables, expressions, sequence, decision, and repetitive structures. Writing shell scripts is beyond the scope of this book, but it is a powerful UNIX feature that you will eventually want to learn more about.

Other Useful Commands

You have barely scratched the surface of the UNIX command language. However, assuming you have actually tried the commands described in this chapter, you should be able to read a reference manual or a UNIX textbook and determine how to use additional commands on your own. For example, check `lpr`. It sends the contents of a file to the line printer. Then check `pr`. This filter prepares output for the line printer, adding page headers, page numbers, and so on. Try using pipes to direct the output from `cat`, through `pr`, and then to `lpr`.

You've already encountered the visual editor, `vi`. Several other utilities transform text prepared under `vi` into printable form. For example, `nroff` formats text, `troff` prepares output for a phototypesetter, `eqn` sets up equations, `tbl` formats tables, and `spell` checks for spelling errors. Learn to use them; they are powerful tools.

Some tasks, for example, printing the contents of a file or performing a lengthy compile, can be quite time-consuming. Instead of idly waiting for such a process to finish executing, you can take advantage of UNIX's multiprogramming capability and run it in the **background**. While it runs, the terminal is free to support another program.

To run a program in the background, type an ampersand (&) at the end of the command line; for example,

```
lpr names &
```

Usually, the shell starts a command as soon as you enter it, and then waits for it to terminate before displaying the next prompt. The ampersand tells the shell to start the process, immediately display a process identification number, and give you another prompt (see Chapter 16 for additional details). Respond by typing your next command. To check the status of your background command, type a process status (`ps`) command.

Two other utilities support communications between systems. `Telnet` is a terminal emulator that allows a user to dial a remote computer and access it. UNIX-to-UNIX copy (`uucp`) transfers files between UNIX systems.

Graphic User Interface

If you do not want to use the shell, many Unix systems include a standard graphical interface called X-Windows. Linux systems include KDE or GNOME as interfaces that are built on top of X-Windows. These provide an interface very similar to the Windows interface (Chapter 9). You can do most operations such as changing passwords, creating or changing directories, and copying files using drag and drop via this graphical interface. A snapshot of the KDE interface for a file directory is shown in Figure 10.30.

Summary

UNIX commands are processed by a shell. The basic structure of a command was illustrated, and normal log on procedures introduced. The `passwd` utility can be

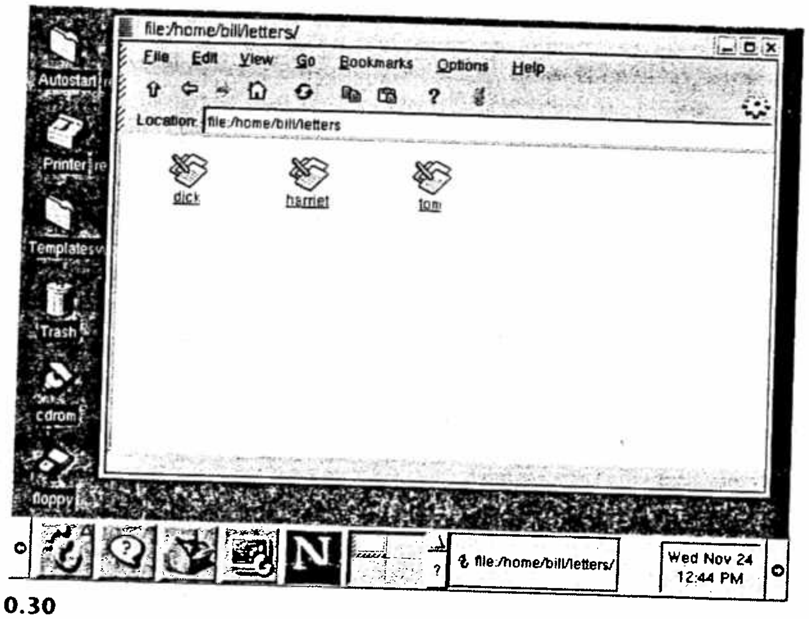


Figure 10.30

A KDE snapshot for a file directory.

used to change a password. The `date` utility displays the system date and time, while `who` displays a list of users logged on the system.

The UNIX file system allows a user to store, retrieve, and manipulate files by name. The rules for defining file names were explained. Because UNIX uses a hierarchical directory structure, a path name must be specified to completely identify a file. The `pwd` command displays the current working directory's path name. You used `ls` to list a directory's contents, trying several different options. Next, you used `mkdir` to create three directories and experimented with the change directory (`cd`) command. Finally, you used the visual editor (`vi`) to create some files, and manipulated those files with `cat`, `cp`, and `ln` commands.

Many UNIX utilities and commands assume the standard input or output device. Redirection tells the shell to change these defaults. A filter accepts data from the standard input device, modifies them in some way, and sends the results to the standard output device. Pipes allow a user to link utilities and other programs, treating the standard output generated by one as the standard input for another. The `sort` utility was used to illustrate pipes and filters.

Many data-processing jobs are run frequently. Thus, the same set of commands must be entered again and again. An option is writing a shell script. The chapter ended with a brief overview of several other UNIX commands and a discussion of background processing.