The Operating System Handbook

or, Fake Your Way Through Minis and Mainframes

by Bob DuCharme

UNIX

Table of Contents

This Section, the Rest of the Book

This is one part of the book "Operating Systems Handbook (or, Fake Your Way Through Minis and Mainframes)," which was originally published by McGraw-Hill as a $49.60 hardcover. Once they reverted the rights to me after it went out of print, I converted the original XyWrite files to DocBook XML and then used Norm Walsh's stylesheets (see www.nwalsh.com) and the Apache FOP program (see xml.apache.org) to convert that to Acrobat files. The six parts of the book are all available at http://www.snee.com/bob/opsys.html:

- Part 1: Introduction. Note that this part includes a new section explaining why I didn't update the book. I strongly suggest that, no matter how much or how little of the book you can use, you glance through the whole Introduction as well. The "Comments and Suggestions" part is now obsolete; my home page is now http:/www.snee.com/bob and my e-mail address is bob@snee.com.
- Part 2: UNIX. Everything described here should apply to Linux and its relatives.
- Part 3: OpenVMS. Basically, VMS. DEC was calling it "OpenVMS" at the time.
- Part 4: OS/400. The operating system for IBM's AS/400.
- Part 5: VM/CMS. An IBM mainframe operating system.
- Part 6: MVS. Another IBM mainframe operating system.

See www.snee.com/bob for information on books I've written since. In reverse chronological order, they are:

- *XSLT Quickly* is a tutorial and users guide to XSLT designed to get you writing stylesheets as quickly as possible.
- *XML: The Annotated Specification* is a copy of the official W3C XML specification with examples, terminology, and explanations of any SGML and computer science concepts necessary for a complete understanding of the XML spec.
- *SGML CD* is a users guide to free SGML software, most of which can be used with XML as well. The chapter on Emacs and PSGML, which requires no previous knowledge of Emacs, is available on the web site in English, Russian, and Polish.

One more thing: either I couldn't figure out the XSL `keep-together` property or version 0.18.1 of FOP doesn't implement it yet. Either way I apologize that some screen shots get split across page breaks.

Chapter 2 UNIX: An Introduction

UNIX has been around for over twenty years and many consider it to be the operating system of the future. Why? Because as personal computers become cheaper and more powerful, the original operating systems designed for them are less and less adequate; the portability and multitasking ability of UNIX make it a strong candidate for those who want to upgrade from single-user systems. From PC/DOS 2.0 to the Macintosh's System 7, other operating systems have increasingly reflected the UNIX influence as their manufacturers strive to increase their power and capabilities.

UNIX also has a certain mystique, making it a magnet for would-be hackers. Clifford Stoll's bestselling 1988 book "The Cuckoo's Egg" boosted this mystique with the story of a crunchy-granola Berkeley astronomer who tracks down some German spies employed by the KGB. What made this story different from a John LeCarre novel, besides the fact that it was true, was that the bad guys' spying and the good guy's detective work were all done over a worldwide UNIX network. (You don't need to know any UNIX to enjoy the book, but a basic knowledge—the kind provided by this book—definitely enhances your appreciation of the key characters' maneuverings.)

2.1 History

The mystique of UNIX, however, is much older than Stoll's book. To understand its roots, we must go all the way back to the twenties. Before the invention of computers, IBM realized that people would pay good money for solid, reliable support after they bought IBM's time clocks and tabulating machines. They knew that the relationship between business machines and post-sales support resembled the relationship that Eastman Kodak had found between cameras and film: customers may buy the former only once, but they need to purchase the latter over and over. That's where the real money was.

When IBM started making computers and selling software to go with them, the software's source code was naturally a trade secret. Source code is the program as the programmers wrote it; a program called a compiler translates this into the binary file that is the software you buy and run. The binary file is unintelligible to the eye, while the source code shows how the program really works. Hobbyists show each other their source code, and computer science students hand theirs in to be graded, but no IBM source code went beyond IBM.

In 1969, Ken Thompson of Bell Labs developed the first version of UNIX on a DEC PDP-7 for his own use. (The name and several of the concepts were derived from an unfinished joint venture with General Electric and MIT called MULTICS.) Other Bell Labs programmers liked it, used it, and added to it. It spread rapidly throughout Bell Labs, where it continues to be the dominant operating system today.

Bell Labs' parent company, AT&T, realized that they had something valuable on their hands, but this was before the breakup of AT&T, when government regulations restrained them from get-

ting too far into the computer market. AT&T did license UNIX for inexpensive use by educational institutions, but with some twists to the typical licensing agreements that followed the IBM pattern: instead of selling the operating system and being responsible for supporting it, the deal included the complete source code and the understanding that there would be no support available.

ULTRIX? XENIX? AIX? AUX? POSIX? DYNIX? MACH? SunOS?

AT&T registered "UNIX" as a trademark, so although anyone may create their own version and market it, they may not call it UNIX. As a result, different companies have come up with their own names. We call these slightly different versions "flavors" of UNIX. They often end in the letter "X" so that they sound like the word "UNIX": DEC's ULTRIX, which runs on their DECstation workstations; IBM's AIX, which runs on its RS series of workstations; XENIX, developed for computers with Intel processors (usually machines considered to be powerful PCs that otherwise run DOS); Sequent's Dynix, and Apple's AUX. Sun Microsystems calls the operating system for their workstations "SunOS," and the NeXT computer uses an MIT-developed variant of UNIX called Mach.

POSIX is not an actual operating system, but a developing government standard for a version of UNIX that any vendors must conform to if they want to sell their UNIX products to the government.

The differences between these various flavors, from the user's point of view, are usually slight—for example, an error message might be worded differently. It's safe to say that if you're comfortable with one flavor of UNIX, you can fake it on the others.

The bargain price of UNIX and its ability to run on many different computers quickly made it popular in universities and small companies that were just acquiring their first computer. The universities turned out computer science students who knew UNIX, and its popularity spread further.

The lack of support remained a problem, however, so users banded together to support each other. Some users formed a user group called /usr/group (a pun on the term "user group" and on a UNIX subdirectory name) in order to pool the knowledge they had gained by studying the source code. This could be considered the original UNIX cult—at least the first beyond Bell Labs. Certain Bell Labs names (Kernighan, Ritchie, Aho, and Weinberg, among others) are still the high priests of this cult.

BUZZWORD *The Labs* In addition to UNIX, the C programming language, lasers, communications satellites, and the transistor, Bell Labs is responsible for countless other things that we take for granted in the world of computers and in everyday life. Many consider Bell Labs so important that they don't even need the word "Bell," so you will sometimes hear people refer to "The Labs."

The extreme terseness of UNIX also contributed to its cultiness. Its most important commands are only two or three letters long—for example, the command to list filenames, `ls`, and the command to copy a file, `cp`. (The real fun comes with commands that are abbreviated to look like completely unrelated words. The command `man` has nothing to do with men; it brings up the on-line manual. The command `cat`, which you will find in section 4.1, ("The Eight Most Important Commands") has nothing to do with feline domestic pets. The command `tar` is used for tape archiving, and has nothing to do with road surfaces or Brer Rabbit; the `wall` command is used by system administrators to write a message to all terminals, and has nothing to do with the sides of a building.)

These abbreviated commands, along with the use of symbols like the period, the double period, the slash (`/`), the pipe (`|`), and the greater-than and less-than symbols (`>`, `<`), enable UNIX users to put together flexible, powerful commands with a minimum of typing. People who don't understand these commands and symbols find them intimidating. The combination of terseness, power, and strange symbols in a command like

```
ps -aux | grep ../getty | sort >> gettyproc.txt
```

reminds the uninitiated of the mystical symbols of alchemy, or worse, of assembly language.

2.1.1 Today

When the federal government ordered the breakup of AT&T on January 1, 1984, AT&T did benefit from the deal: restrictions on many of their potential activities were lifted. Some of these restrictions had prevented them from getting too far into the computer industry. With their removal, UNIX became a marketable product for them.

The power and flexibility of UNIX helped it to grow into a big business, but the cultiness was hurting business. /usr/group, whose very name only made sense to the initiated, changed its name to UNIX International in 1989. Complaints about the cryptic nature of UNIX commands and the success of graphical user interfaces on computers like the Macintosh and the Amiga inspired people to create interfaces for UNIX systems with windows and icons that could be controlled with mice.

Computer science students still study UNIX closely at colleges and universities, because when you study the responsibilities and methods of an operating system, the best way to learn is to look at the source code of a real operating system. Although commercial versions of UNIX are more proprietary these days and often too complex for students to understand the source code, simpler versions of UNIX like MINIX and XINU have been developed specifically for students to dissect and study.

Today, the graphical user interface versions of UNIX always have a window where you can type in old-fashioned UNIX commands. In fact, they let you have several of these windows at once.

These commands are not as difficult as their reputation; they're just very abbreviated. DOS and Amiga users in particular will understand more about their PCs' operating systems when they study UNIX, because so much of DOS and AmigaDOS were modeled on UNIX. (Knowing about the UNIX heritage of DOS has earned me some easy money on two occasions—both times, I earned sixty dollars for sending a single paragraph to the "User-to-User" column in the back of "PC Magazine." Each one described a common UNIX trick that also worked on the DOS command line.)

Workstations

Imagine that you had a PC so powerful that no existing personal computer operating system enabled you to take full advantage of that power and that you used some variant of UNIX instead. This is essentially what

a workstation is. Although their multi-tasking ability allows UNIX computers to be used by more than one person at once, workstations are usually used by one person at a time.

Workstations are also designed to communicate easily with each other. Sun Microsystems, the company that first popularized UNIX workstations, is famous for its slogan "The network is the computer."

Workstations usually have large, high-resolution monitors and graphics capabilities far superior to those of other computers. Because of these abilities, they are popular for scientific visualization and computer animation. This makes them far more glamorous (or in computer industry parlance, "sexy") than computers used for mundane tasks such as processing purchase orders. As a result, workstations have become a popular bandwagon. IBM had Hagar the Horrible selling its RS workstations, Steve Jobs pushed his NeXT machine from the cover of Newsweek, Hewlett-Packard bought out the popular workstation manufacturer Apollo in order to gain an entry, and DEC brought out its DECstation.

Meanwhile, as personal computers and their operating systems get more and more powerful, computer trade press journalists each write their annual "We have to redefine what we mean when we say workstation" column.

2.1.2 USENET

When you know UNIX, you not only have the ability to deal with a wide variety of computers from a wide variety of manufacturers (not to mention the many "flavors" of UNIX—see sidebar); you also have the tools necessary to take advantage of USENET.

Some people call USENET a giant computer bulletin board. From the user's perspective, it bears a strong resemblance to a bulletin board; you can download programs and you can send electronic mail and programs to other users. You can read messages from people all over the world and leave them yourself on any topic imaginable. It keeps many scientists and researchers far more up-to-date on news in their fields than any journal published on paper could. In spreading hot stories, USENET has often been known to scoop CNN.

It isn't really a bulletin board, though. USENET is actually much more dynamic than that. Rather than a central computer where people log in to to see what's new, USENET provides a constant flow of information between nodes, or computers designated to receive and send along this information. If your system is hooked into one of these nodes, then your system is itself a node and you have access to whatever portion of USENET is being pulled in to your node.

Chapter 3 Getting Started with UNIX

3.1 Starting Up

When you turn on a terminal connected to a UNIX system, or successfully connect to such a system over a network or phone line, the first thing you see is the login prompt:

```
login:
```

As an authorized user of this system, you should have a login name that represents your identity on the system. Type it in here and press the Enter key. The next prompt asks for your password:

```
password:
```

Type it in and press the Enter key. If all went well, you will be logged in.

A couple of things to remember:

- If you make a typing mistake, press Enter until the system asks you to log in again. Don't try to use your Backspace or cursor movement keys to correct the mistake. Because the computer probably doesn't know what kind of terminal you are using or emulating yet, it may not understand the codes sent by these keys. If Joe User enters `job,` then presses the Backspace key to get rid of the `b` and types `euser,` his screen may show that he has typed `joeuser,` but when he presses Enter the system may receive something that looks more like `job^]euser` as a login name. It won't have a record of such a user and won't give him access to the system regardless of the password that he types with this login name.

- Some systems, if you log in in upper case letters, assume that you are using one of the old-fashioned terminals that cannot type lower case letters. They will then display all text for that session in upper case letters. Make sure you log in in lower case.

- Just because the system asks you for a password doesn't mean that you entered the login name correctly. It always asks. If it only asked when you entered a valid username, then people trying to break in to the system would have an easy way to determine which login names were valid.

- A login name may have no password. This may be the case the first time you log in. As soon as you enter the login name, the system displays the screen indicating that you have logged in.

Once you log in, the system probably displays some information about the particular system that you are logged in to before it displays the prompt where you enter commands. The prompt usually appears as a dollar sign ($) or a percent sign (%), but can be easily changed.

There are ways to set up a UNIX ID so that, when someone logs in, a certain program automati-

cally runs whether that user wants it to or not. You will often find arrangements like this for IDs that have no password—this way, anyone can log in to run one particular program, but they can't have the run of the system. I was once given a UNIX account just to use the mail program. When I logged in, it automatically started up the mail program; when I quit the mail program, it automatically logged me out.

3.1.1 Finishing Your UNIX Session

To show that you want to disconnect from the system, type:

```
exit
```

A shortcut available on most systems is to type Ctrl+D.

> **BUZZWORD** *Box* Many manufacturers produce computers that can run UNIX, or some flavor of it, and users often identify the brand of hardware being used in a given situation as a "box"—For example, "They're using AT&T UNIX, but running it on an NCR box."

3.2 Filenames

Filenames in UNIX can be up to 14 characters long and can consist of just about any characters. Certain characters have special meanings in UNIX and could lead to trouble if used in filenames; for example, you should avoid `<, >, |, -, ?, [, ],` and `*.` Most people use letters, numbers, the underscore and the period. Because spaces are not allowed in filenames, the underscore provides a way to make abbreviated filenames more readable. `jul_budget` is a more understandable filename than `julbudget`. Also, UNIX is case-sensitive—it would treat `BUDGET.TXT, Budget.txt,` and `budget.txt` as three different files. Again, stick to lower case.

Be careful about using a period for a file's first character, because this makes that file hidden. This means that including its names in a list of files on the screen requires you to use a special option when you use the `ls` command to list filenames. As a rule, UNIX users only begin very

specific filenames with a period. Section 6.2, "Command Files," covers two examples: `.profile` and `.login`.

3.2.1 Wildcards

The main wildcards in UNIX are the asterisk and the question mark. Although the examples below demonstrate their use with the `ls` command, remember that you can use them with almost any command that uses a filename as a command line parameter. For more information, see the material on wildcards in section 1.5, "General Advice."

3.2.1.1 The Asterisk

The asterisk at the end of a filename has the same significance in UNIX that it has in most other operating systems. It can represent zero or more characters at that position in the filename or file type.

This is typical of many operating systems. In UNIX, however, the asterisk is much more versatile, because it doesn't have to go at the end of the expression you type. For example,

```
ls *may
```

lists all the filenames that end with the letters "may," and

```
ls *may*
```

lists out all filenames with the letters "may" anywhere in them.

```
ls rpt*94
```

would list out all the filenames that began with the letters "rpt" and ended with the digits "94," regardless of how many characters are between them.

3.2.1.2 The Question Mark

The question mark represents a single character—no more, no less. Several question marks represent that number of characters, so that

```
ls ???94rpt.txt
```

would list out all of the filenames with exactly three characters before the characters "94rpt.txt."

3.3 How Files Are Organized

Like many other aspects of UNIX, its file system has provided a model for many operating systems developed after it, such as PC/DOS and AmigaDOS. We call this system of file organization *tree-structured directories,* which means that the disk is divided into sections called

directories. A directory can be divided into sub-sections called *subdirectories*, which can also be divided. The terms "directory" and "subdirectory" are used almost interchangeably, since every directory—except the root—is a subdirectory of another.

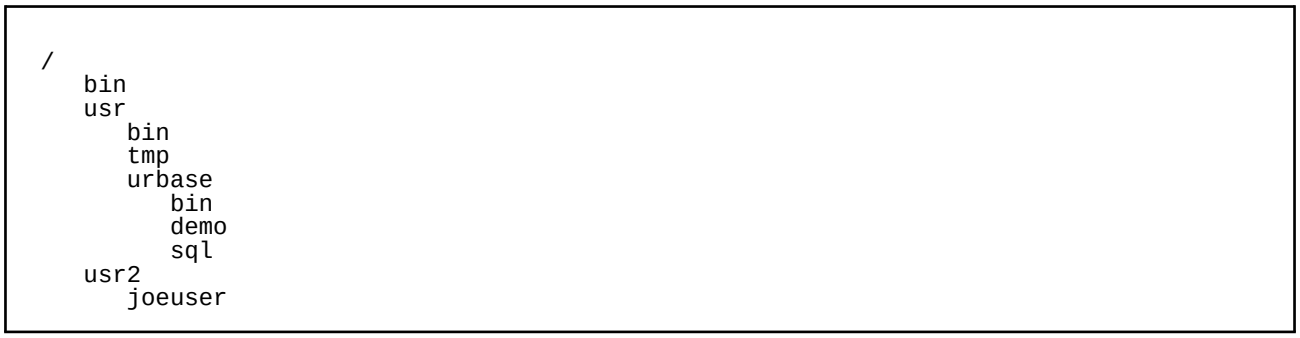
To understand how the main directory, or *root* directory, leads to subdivisions which lead to subdivisions which lead to subdivisions, think of the branches of a tree. The root is like the tree's trunk, which branches into several main branches. These main branches then divide into smaller and smaller branches.

In a typical UNIX system, one of the main branches usually holds most of the programs that come with the operating system. We call this the `/bin` directory. Another main branch could hold the software that was purchased for installation on this UNIX system. This branch might be called `/usr`, and the system administrator would subdivide it into sections to hold each of the software packages. For example, the UpRiteBase database package could be in one of these subdivisions in a directory called `/urbase`. This subdivision's full name would be `/usr/urbase`, because it is a subdirectory of the `/usr` directory and the full name of any directory includes its *pathname*, or the name describing the path up the tree along the various branches it took to get there. Since the UpRiteBase software package consists of quite a few files, it is more efficiently organized if the system administrator divides the `/usr/urbase` directory into subsections when installing it. The binary files sit in a branch called `/usr/urbase/bin`, the files associated with the demonstration database that comes with it are in a subdirectory called `/usr/urbase/demo`, and so forth.

Notice how a slash character separates each component of the pathname. The pathname of the root, or main trunk of the tree, is just a slash by itself. We create a complete pathname by combining the names of the various subdirectories traversed to get to the subdirectory in question, separating each with a slash, and by putting a slash at the very beginning to represent the root.

Figure 3.1 shows a sample UNIX directory tree structure. (Keep in mind that an actual system would have many more branches.) A level of indentation represents a level of the subdirectory structure; for example, the ninth line represents the subdirectory `/usr/urbase/sql`. The first line shows the root directory.

No two directories can have the same name. Although you may see three subdirectories seemingly named `bin` in the directory structure in Figure 3.1, keep in mind that their complete pathnames are different: `/bin`, `/usr/bin` and `/usr/urbase/bin`.



```
         mail
         networking
      maryjones
         mymail
         payroll
      jimcasey
         inventory.dbs
         letters
```

Figure 3.1 Sample UNIX directory structure.

At any given time, one of these directories is your "current" or "default" directory. (An equivalent expression describes you as being "in" that directory.) This matters to many of the UNIX commands—for example, if you enter the command to erase a file but don't specify the directory where the file is located, the system assumes that you want to erase a file in the current directory. Section 3.3.2, "Moving between Directories," shows how to make a new directory the current one.

Each user is assigned his or her own subdirectory known as their *home* directory. The system administrator assigns subdirectories to users as their own disk space in which to keep their personal files. A UNIX system's directory structure has one or more main branches off the root to hold these personal directories for the various users (in Figure 3.1, it's called `/usr2`), just as main branches exist to hold the software that they use.

In the example, `/usr2` leads to the subdirectories `/usr2/joeuser, /usr2/maryjones,` and `/usr2/jimcasey,` which would be the home directories for three different users.

These users can create and maintain subdirectories of their home directories in whatever arrangement they like. Mary Jones might keep her correspondence in a subdirectory called `/usr2/maryjones/mymail` and Joe User might keep his files pertaining to a new networking project in a subdirectory called `/usr2/joeuser/networking.` Section 4.1.9, "Creating Directories," and section 4.1.10, "Removing Directories," show you how to maintain subdivisions of your own home directory.

3.3.1 Relative Pathnames

Because you can divide up subdirectories into so many subdivisions, full pathnames can get long. UNIX provides two shortcuts to make it easier to refer to directories:

- You can substitute of two periods (..) where the system expects a pathname. This means that the you are referring to the *parent* of the current directory, or one level closer to the root. The parent of `/usr2/maryjones` and `/usr2/joeuser` is `/usr2`; the parent of `/usr/urbase` is `/usr`; and the parent of `/usr, /usr2,` and `/bin` is `/`, the root. Section 3.3.2, "Moving between Directories," gives an example of how to use the two dots as a substitute for the parent directory's name.

- Another shortcut makes it easier to refer to the *child* of the current directory (a subdivision of the current directory). Note how all references to directory names up to now begin with the slash (`/`) character. You don't always need this; if you omit the slash, the system assumes that you are referring to a subdivision of the current directory. For example, if Mary wants to copy some mail messages from the `/usr2/maryjones` directory into the `/usr2/maryjones/mymail` directory, she could just enter `mymail` as the destination of her copy command instead of typing out `/usr2/maryjones/mymail`. This works as long as she was in the `/usr2/maryjones` directory at that time. If she was in her `/usr2/maryjones/payroll` directory and entered `mymail` as the destination of her copy command, the system would look for a subdirectory called `/usr2/maryjones/payroll/mymail` and not find it. (Instead of giving you an error message, it would create a file called `mymail` in the `/usr2/maryjones/payroll` subdirectory. See section 4.1.5, "Copying Files," for more information on the logic behind this.)

We call these two shortcuts *relative* pathnames, because the system figures out the directory that you are referring to relative to your current directory. If you enter a command to copy files into a directory called `bin,` without a slash, this would mean the `/bin` directory if you were currently in the root directory, the `/usr/bin` directory if you were in the `/usr` directory, or the `/usr/urbase/bin` directory if you were currently in the `/usr/urbase` directory. (In reality, you would not have permission to alter the contents of subdirectories outside of your home directory unless you were the system administrator). Similarly, the directory that you refer to when you type `..` completely depends on which directory is current when you type it.

3.3.2 Moving between Directories

When you first log in to a UNIX system, your current directory is the one assigned to your login name by the system administrator. If you type `ls,` the command to list out filenames, the system lists out the files in the current directory. (The first time you log in to a given system and enter this command, there may not be any files to list out.)

The command `cd,` followed by the name of a directory, changes your current location into that directory. For example,

```
cd /
```

puts you into the root directory, and

```
cd /bin
```

puts you into the `/bin` directory. If you misspell a directory name so that your command tells the system to change into a non-existent directory—for example, `blin`—it gives a reply similar to this:

```
blin: bad directory
```

When the command executes successfully, the system does not acknowledge that you have a new working directory, but you can easily find out where you are at any given time with the `pwd,` or "print working directory" command. This "prints" the full name of your current directory on the screen. (See section 1.5, "General Advice," if the idea of "printing on the screen" doesn't make sense to you.)

Changing the current working directory provides one example of how the use of relative pathnames can save you a great deal of typing. If your current directory is the `/usr/urbase/bin` directory and you want to change into the `/usr/urbase` directory, you could type

```
cd /usr/urbase
```

but it would be much easier to type

```
cd ..
```

because `/usr/urbase` is the parent directory of `/usr/urbase/bin.` To change back to `/usr/urbase/bin,` just type

```
cd bin
```

because the `bin` directory that you want is a child directory of `/usr/urbase.` Remember, when you type `cd bin,` the system looks for a child of your current directory called `bin.` If you had been in the root directory when you typed the same command, you would have ended up in the `/bin` directory, not `/usr/urbase/bin`.

3.4 Available On-line Help

There are two commands that may give you help after you log in. The first, `help,` is fairly obvious. On many systems, typing `help` by itself starts up a menu-driven program that tells you a great deal about using UNIX. The first screen that it displays explains how to use it.

The `man` command may also assist you. In the great tradition of naming UNIX commands by abbreviating them until they look like completely different words (like `cat tar,` or `wall`) this is an abbreviation of the word "manual." Being more old-fashioned than the `help` command, `man` is used strictly from the command line—there are no menus to help you along. Type in `man` by itself, and it tells you how to use it: you enter `man` followed by the word that you want to look up in the manual. For a start, look up `man` itself by typing:

```
man man
```

(If a screenful of text scrolls up and then stops, press the Enter key each time you want to scroll to a new screen.)

It's possible that nothing happens with either the `help` or `man` commands. Both get the information you request by looking it up in text files stored on the computer's hard disk for this purpose, and some system administrators erase these files from the hard disk to make more room for other files.

BUZZWORD *Gen* (pronounced "jen") When PC users think of putting an operating system onto a computer, they think of copying the operating system files onto their hard disk and maybe running a configuration program to tell the operating system more specific information about the hardware they are using. On a UNIX system, the system administrator must run a program that takes various data and code files and actually creates many of the operating system files. This is known as "generating" the operating system, but if you really know your UNIX slang you refer to "genning" the operating system. For example, "I'm looking forward to checking out the new features of the system upgrade, but I don't know when I'll have the time to gen it."

Chapter 4 Using Files in UNIX

4.1 The Eight Most Important Commands

The *shell* is the part of UNIX that interprets the commands that you type at the UNIX prompt. It passes the instructions along to the *kernel*, the part of UNIX that does the real operating system work. We call the basic operating system commands that you enter at the UNIX prompt "shell commands." If someone in the middle of running a program talks about "accessing the shell" or "shelling out," they're talking about temporarily gaining access to the main system prompt where they can type shell commands.

If you are using a graphical user interface version of UNIX and don't see a window where you can type in commands, never fear—there's one in there somewhere. Either there will be an icon (on a Sun workstation, it's a little picture of a conch shell) or there's a main menu with "Shell" as a choice. (To bring up such a menu, try clicking on the screen background—that is, with your mouse pointer on the background picture, and not on any window or icon—with any buttons available on your mouse.)

There are two basic versions of the shell, with several variations available. All the UNIX commands described here work with both of the most popular ones, the Bourne shell and the C shell . Many systems can run either one, so it's not a dumb question to ask which shell is the default on a given system.

The eight most important shell commands in UNIX are:

`ls`	lists file names.
`cat`	displays the contents of files.
`cp`	copies files.
`mv`	renames and moves files.
`rm`	deletes files.
`chmod`	grants and revokes access to files.
`mkdir`	creates subdirectories.
`rmdir`	removes subdirectories.

4.1.1 Command Options: Switches

UNIX uses a hyphen (`-`) to indicate options, or *switches* that give special instructions about how a command should operate. For example, the `ls` command by itself only lists filenames, but with the `l` switch it lists other information about the file, and with the `t` switch it lists out the files in reverse chronological order instead of alphabetical order. You could enter

```
ls
```

by itself, or you could enter

```
ls -l
```

to indicate that you want to see all the information about the files, or you could enter

```
ls -t
```

to see the filenames in reverse chronological order. You can also combine these switches; you could enter

```
ls -lt
```

or even

```
ls -tl
```

to see all the information about the filenames, listed in reverse chronological order. The order in which you put the switches doesn't matter, as long as you remember to include the hyphen, which means "here come the switches," and to avoid putting any spaces between the letters that denote command-line options.

Section 4.1.3, "Listing Filenames," gives more information on using switches with the `ls` command. When you use UNIX's on-line help system to inquire about a command, it tells you all about the command's various switches and what they do. Knowing that this information is available in on-line help is the main reason to not worry about memorizing a lot of command line switches.

4.1.2 Common Error Messages

When you type anything at the UNIX command prompt, it looks for a program with that name and executes it. If you make a typing mistake, for example

```
max man
```

when you meant to type `man man`, UNIX gives you a message along the lines of:

```
max: not found
```

This means that it looked for a program called `max` and couldn't find it.

Many commands expect you to include some information on the command line after the com-

mand's name. If you omit any, most UNIX systems display a terse explanation of how much information they expected. For example, to make a copy of a file, you must indicate the file you want to copy and the name you want to give to the new copy. If you type the `cp` command by itself without any filenames, the system responds with something similar to this:

```
Usage: cp [-ip] f1 f2; or: cp [-ipr] f1 ... fn d2
```

This shows you that you had to include at least two filenames (represented by `f1` and `f2`) after the `cp` command. The alternative syntax, after the `or:` part, shows that you could also type one or more filenames followed by the name of a destination directory name, if you want to copy the files to another directory. Remember—you can always type `man cp` for more detailed help on the `cp` command.

Another common inspiration for error messages is when you instruct the system to do something to a file that doesn't exist. For example, let's say you want to copy a file called `template.txt` and call the copy `may_bud.txt`, but you make a typo when you enter the command:

```
cp tempalte.txt may_bud.txt
```

The UNIX system responds with

```
cp: tempalte.txt: No such file or directory
```

as if to say "There's a problem executing the `cp` command: I can't find any file or directory named `tempalte.txt`."

Remember, the `cp` command is just used as an example here. Similar mistakes with many other commands will elicit similar error messages. For a full explanation of the use of the `cp` command, see section 4.1.5, "Copying Files."

4.1.3 Listing Filenames

The `ls` command lists out filenames. If you type `ls` by itself, it lists out the names of the files in your current directory in alphabetical order, along with the names of any subdirectories of the current directory. This list might look like the following:

```
061293rr
062093rr.prn
06ifp.txt
082294ts.txt
083194vd.txt
index.txt
mailnotes.txt
prepprn.awk
rptapr94
rptfeb94
rptjan94
rptmar94
rptmay94
s_and_rep.awk
sample.txt
```

```
schedule.txt
text.txt
```

You can put two kinds of parameters after the `ls` command:

- A directory name, which shows that you want to list the files in a directory other than the current one.
- A file specification, which shows that you only want to list files whose names follow a certain pattern.

If you type `ls` followed by a directory name, like this,

```
ls /bin
```

you will see several screenfuls of filenames from the `/bin` directory scroll by alphabetically, without stopping, until it ends with a screenful of filenames similar to the ones shown in Figure 4.1.

```
uuencode
uulog
uuname
uupick
uusend
uustat
uuto
uux
vax
vplot
wall
who
write
xargs
xget
xsend
yacc
ypcat
ypchfn
ypchsh
ypmatch
yppasswd
ypwhich
```

Figure 4.1 End of the output from the command ls /bin.

If you type `ls` followed by a filename, it only lists that file's name. For example, if you type

```
ls .profile
```

it shows you this:

```
.profile
```

This isn't particularly useful unless you include wildcards when you specify the filename. For example, to list all the files that begin with the characters "rpt" and end with the characters "94" you would type

```
ls rpt*94
```

and perhaps see output similar to this:

```
rptapr94
rptfeb94
rptjan94
rptmar94
rptmay94
```

(For more information on using wildcards to specify the filenames you want included in your list, see section 3.2.1, "Wildcards.")

If you want to see specific files in a specific directory, you can add the directory and filename specification after the command. Don't put any spaces between them. For example, to list all the files in the `/bin` directory that begin with the letter "l," type this:

```
ls /bin/l*
```

Your output might look like this:

```
/bin/ld
/bin/line
/bin/ln
/bin/login
/bin/lorder
/bin/ls
```

It just so happens that one of the files beginning with "l" in the `/bin` directory is the "ls" program itself. The `/bin` directory holds many of the most often-used commands in UNIX.

4.1.3.1 Listing More than File Names

The `ls` command may have more switches than other UNIX commands: at least 20, depending on the flavor of UNIX that you are using. There is a switch to put slashes next to the directory names that show up with the filenames, a switch to list the filenames in chronological order instead of alphabetical order, and a switch to reverse the order in which the names appear. Few of these switches are worth memorizing; you can always use the `man` or `help` command to learn about them.

The most important switch gives you the "long" listing of the files. It's not really longer, but ac-

tually wider—if you call it the "long" listing, it's easier to remember that the switch is the letter "l." It tells the `ls` command to give much more information about the files than just their names. If you enter the command

```
ls -l
```

the output would look something like this:

```
 -rw-rw-r--   1 joeuser  marketing    520 Jun 12  1993 061293rr
 -rw-rw-r--   1 joeuser  marketing   3592 Jun 20  1993 062093rr.prn
 -rw-rw-r--   1 joeuser  marketing  22305 Nov  6  1993 06ifp.txt
 -rw-rw-r--   1 joeuser  marketing    660 Aug 23  1993 082294ts.txt
 -rw-rw-r--   1 joeuser  marketing    542 Aug 31  1993 083194vd.txt
 -rw-rw-r--   1 joeuser  marketing    504 Jan  2  1994 index.txt
drw-rw-r--   1 joeuser  marketing    512 Nov 12  1993 mail
 -rw-rw-r--   1 joeuser  marketing     66 Mar 22  1993 notes.txt
 -rwxrwxrwx   1 joeuser  marketing     33 Dec  4  1993 prepprn.awk
 -rw-rw-r--   1 joeuser  marketing     47 Nov 28  1993 rptapr94
 -rwxrwxrwx   1 joeuser  marketing    165 Sep  6  1993 rptfeb94
 -rw-rw-r--   1 joeuser  marketing     98 Jan  2  1994 rptjan94
 -rw-rw-r--   1 joeuser  marketing     73 Dec  4  1993 rptmar94
 -rw-rw-r--   1 joeuser  marketing     44 Nov 28  1993 rptmay94
 -rw-rw-r--   1 joeuser  marketing     46 Dec  4  1993 s_and_rep.awk
 -rw-rw-r--   1 joeuser  marketing    512 Dec  7  1993 sample.txt
 -rw-rw-r--   1 joeuser  marketing    276 Jul  8  1994 schedule.txt
 -rw-rw-r--   1 joeuser  marketing    105 Nov 28  1993 text.txt
```

There's a lot of information here. The last column to the right should look familiar; it's the file's name. To the left of that is the date that the file was last modified, and to the left of that is the current size of the file in bytes. The columns that say `joeuser` and `marketing` show the file's owner (usually the user who created the file) and the group that the user belongs to.

What is a group? UNIX lets the system administrator assign users to groups because it makes the system administrator's job easier when giving or taking away system privileges. For example, let's say that the system administrator Mary Jones has just installed a new spreadsheet program on the system. In order to enable the 23 people in the accounting department to use it, she could just give execution rights to the group "accounting" instead of typing 23 commands to individually give access rights to 23 people.

What are execution rights? And what's the cryptic column all the way to the left? (Ignore the column of ones just to the left of the `joeuser` column—this column shows how many links this file has to substituted names, an advanced UNIX trick.) The first column shows something called the file's mode. The first character in the file's mode is usually either a hyphen (-) or a `d`. A hyphen means that the line describes a normal file, and a `d` means that it's a subdirectory of the directory whose files are being listed. The r's, w's, x's, and other hyphens show who has what rights with that file or directory. Three kinds of rights are available when you access a file:

`r`	The right to read (look at or make copies of) a file.
`w`	The right to write (make changes) to a file.

x	The right to execute a file. If a file is not some kind of program, execution rights are irrelevant.

You can assign one set of rights to the file's owner, another to the other people in the owner's group, and a third set to everybody else. The second through fourth characters (after the one that tells you whether it's a directory) show the owner's rights; the next three, the group's rights; and the last three, everyone else's. Figure 4.2 illustrates this.

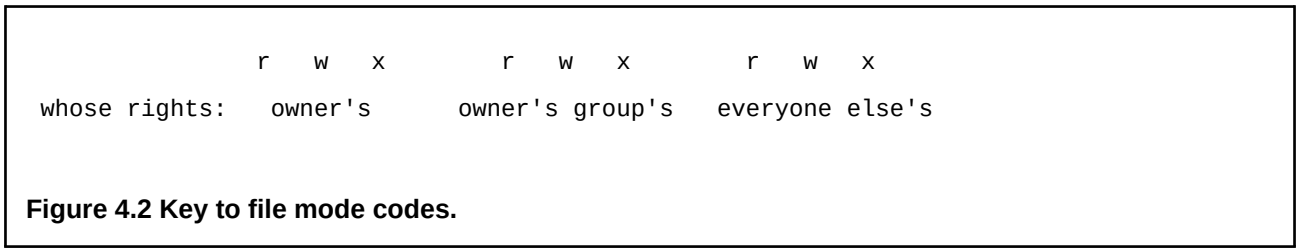


Figure 4.2 Key to file mode codes.

If the `r, w,` or `x` appears, that right exists for that category of user. A hyphen means that right doesn't exist. For example, the following shows a filemode for a file that its owner can read or write, that the owner's group can read, but not write, and that people outside of the owner's group can not even read:

```
-rw-r-----
```

A programmer working on a new program might set its filemode to something like this:

```
-rwxr-xr-x
```

This lets the programmer read, write, or execute the program, and lets everyone else look at it or execute it, but not change it.

As mentioned above, a "d" instead of a hyphen in the first character of the first column means that the name listed is a subdirectory of the current directory, not the name of a file in that directory. The rest of the characters in the file's mode mean the same thing that they do when describing a file, only they describe the privileges that users have when using that directory:

r	The right to read (list the files in) the directory.
w	The right to write to (create files in) the directory.
x	The right to execute the `cd` command to change into that directory.

Try using the `-l` switch with the `ls` command to look at some of the files in the `/bin` directory, like the ones beginning with "c." With the `ls` command, any switches go before the file specification (the part that shows which files you want to see):

```
ls -l /bin/c*
```

The output looks something like this:

```
-rwxr-xr-x   1 bin      bin        28672 Apr 14  1992 /bin/cat
-rwxr-xr-x   1 bin      bin        43008 Apr 14  1992 /bin/cc
-rwxr-xr-x   1 bin      bin        32768 Apr 14  1992 /bin/chgrp
-rwxr-xr-x   1 bin      bin        26624 Apr 14  1992 /bin/chmod
-rwxr-xr-x   1 bin      bin        32768 Apr 14  1992 /bin/chown
-rwxr-xr-x   1 bin      bin        26624 Apr 14  1992 /bin/cmp
-rwxr-xr-x   3 bin      bin        32768 Apr 14  1992 /bin/cp
-rwxr-xr-x   1 bin      bin        73728 Apr 14  1992 /bin/cpio
-rwxr-xr-x   1 bin      bin        28672 Aug 28  1993 /bin/crypt
-rwxr-xr-x   2 bin      bin       110593 Apr 14  1992 /bin/csh
```

It looks like the owner gets to read, write, and execute the files, and everyone else just gets to read them and execute them. Who is the owner? Who is the group? The owner and group columns both say "bin"; this means that the `/bin` directory itself is a kind of user, and a user in its own group.

To change the read, write, and execute privileges of a file, use the `chmod` (change mode) command, which is covered in section 4.1.8, "Controlling Access to a File."

Besides `-l`, the other useful switch for the `ls` command is `-x`. Use it to list several filenames across the screen on each line of output. (Another fine example of computer programmers' novel approach to the English language is the way they abbreviate the word "across" with the letter "x.") If you typed

```
ls -x /bin/c*
```

the output would look like this:

```
/bin/cat    /bin/cc     /bin/chgrp  /bin/chmod  /bin/chown  /bin/cmp
/bin/cp     /bin/cpio   /bin/crypt  /bin/csh
```

(The number of filenames on each line varies from system to system.) This is especially useful when you list out more than 25 filenames; otherwise, these filenames won't all fit on the screen at the same time. If you tried the `ls /bin` command mentioned earlier, you probably saw the names zoom up the screen until it reached the end, at which point you saw the last 25 filenames from the directory. If you added the `x` switch and typed

```
ls -x /bin
```

you would see output like that shown in Figure 4.3.

```
acctcom      adb          ar           as           att          basename
bs           cat          cc           chgrp        chmod        chown
```

```
cmp          cp           cpio         crypt        csh          date
dd           df           diff         dirname      dis          du
echo         ed           env          expr         false        file
find         grep         ipcrm        ipcs         kill         ksh
ld           line         ln           login        lorder       ls
mail         mail.new     mail.newnew  mail.old     make         mesg
mkdir        mv           newgrp       nice         nm           nohup
od           passwd       pdp11        pr           ps           pwd
pyr          red          rm           rmail        rmdir        rsh
sed          sh           sh.new       size         sleep        sort
strip        stty         su           sum          sun          sync
tail         tcsh         tee          telinit      time         touch
true         tty          u370         u3b          u3b10        u3b15
u3b2         u3b5         ucb          uname        universe     vax
wc           who          write
```

Figure 4.3 Sample output of ls -x /bin.

Switches can be easily combined. It wouldn't make much sense to combine the `x` and `l` switches, because there isn't enough room to list out several filenames to a line along with their sizes and the other information that the `l` switch adds to the output. However, you could combine the `l` switch with the `t` switch, which specifies that you want to see the filenames in reverse chronological order, like this:

```
ls -lt
```

and see output like that in Figure 4.4.

```
-rw-rw-r--   1 joeuser  marketing    276 Jul  8  1994 schedule.txt
-rw-rw-r--   1 joeuser  marketing     98 Jan  2  1994 rptjan94
-rw-rw-r--   1 joeuser  marketing    504 Jan  2  1994 index.txt
-rw-rw-r--   1 joeuser  marketing    512 Dec  7  1993 sample.txt
-rw-rw-r--   1 joeuser  marketing     73 Dec  4  1993 rptmar94
-rwxrwxrwx   1 joeuser  marketing     33 Dec  4  1993 prepprn.awk
-rw-rw-r--   1 joeuser  marketing     46 Dec  4  1993 s_and_rep.awk
-rw-rw-r--   1 joeuser  marketing     44 Nov 28  1993 rptmay94
-rw-rw-r--   1 joeuser  marketing    105 Nov 28  1993 text.txt
-rw-rw-r--   1 joeuser  marketing     47 Nov 28  1993 rptapr94
-rw-rw-r--   1 joeuser  marketing  22305 Nov  6  1993 06ifp.txt
-rwxrwxrwx   1 joeuser  marketing    165 Sep  6  1993 rptfeb94
-rw-rw-r--   1 joeuser  marketing    542 Aug 31  1993 083194vd.txt
-rw-rw-r--   1 joeuser  marketing    660 Aug 23  1993 082294ts.txt
-rw-rw-r--   1 joeuser  marketing   3590 Jun 20  1993 062093rr.prn
-rw-rw-r--   1 joeuser  marketing    520 Jun 12  1993 061293rr
-rw-rw-r--   1 joeuser  marketing     66 Mar 22  1993 notes.txt
```

Figure 4.4 Sample output of ls -lt.

The order of the switches doesn't matter in any UNIX command. If you typed

```
ls -tl
```

you would see the same output.

Switches, like the rest of UNIX, are case-sensitive. For example, `-r` means "reverse the listed order" while `-R` means "recursively list subdirectories" (list the contents of any subdirectories along with the names of the files and subdirectories). Because of this, you need to be careful about whether you type switches in upper or lower case. Most of them are in lower case.

Try using `man` or `help` to learn about the other switches to the `ls` command.

4.1.4 Displaying a Text File's Contents

Another source of confusion for beginning UNIX users is the fact that commands used for more than one purpose are not always named after their most popular purpose. The `cat` command, which displays text files on the screen, is also used to combine or "concatenate" files. `cat` is an abbreviation of the word "concatenate," even though it's used far more often to put the contents of a text file on the screen. If you had a file called `schedule.txt` and typed

```
cat schedule.txt
```

the contents of the file would then appear on the screen:

```
October 10
10:30 meet Dave C., Laurie.  call Laurie first--should I bring new diskettes?
12:30 lunch with Benny
2:00 expecting call from Chicago office.  Have page counts ready.
2:30 Anita's presentation--can I get out of going?
4:00 first draft of outline MUST be ready
```

4.1.4.1 Looking at Text Files One Screen at a Time

One of UNIX's greatest strengths is its ability to make several programs work together, all by issuing one command. Although this is usually an advanced technique, combining the `cat` command with the `more` command is so useful that you should learn it as soon as you learn `cat`.

When you display certain files with the `cat` command, you may notice that any files longer than twenty-four lines scroll up and off the screen until the end of the file, at which point you are only looking at the last twenty-four lines.

The `more` command remedies this. (Many systems offer a similar alternative called `pg`. If `more` doesn't work on your system, try `pg`.) It takes what you send it and gives it back to you a screenful at a time (`more` has its own command-line switches that adjust, among other things, how much it outputs at once when you send text to it, but the default value of twenty-four or twenty-five lines is just fine for most uses).

How do you send text to it? UNIX has a special symbol called the pipe (|) that means "take the output of the preceding command and send it to be used as input by the following command." (I told you that UNIX was cryptic—it uses only one symbol to say all that.) Sometimes the pipe symbol appears on screen, on paper, or on a keyboard key as an unbroken vertical line. It may also appear as a vertical line with a gap in the middle.

If the `schedule.txt` file was 100 lines long, you could look at one screenful at a time with this command:

```
cat schedule | more
```

By doing this, you are "piping" the output of the `cat` command to be used as input for the `more` command. After the first screenful appears, the message `–More–` appears at the bottom of the screen, as shown in Figure 4.5.

```
October 9
9:00 Ed may have Knicks tickets for me; bug him when he gets back from Toronto
10:30 office supplies sales rep coming
12:00 lunch with Benny postponed until the 10th
2:30 getting teeth cleaned--call 687-2300 first for address
4:00 Fed Ex new diskettes to Chicago

October 10
10:30 meet Dave C., Laurie.  call Laurie first--should I bring new diskettes?
12:30 lunch with Benny
2:00 expecting call from Chicago office.  Have page counts ready.
2:30 Anita's presentation--can I get out of going?
--More--
```

Figure 4.5 Output from piping schedule file through the more command.

Press the space bar (or, if you piped your output to `pg`, the Enter key) and another screenful appears. Continue this, and you can look at the file at your own pace—unless you want to quit, in which case you type "q" instead of pressing the space bar.

`more` isn't limited to use with the `cat` command; you can also use it with the `ls` command. Typing this

```
ls -l bin | more
```

displays a screen similar to the one shown in Figure 4.6.

```
-rwxr-xr-x   1 bin      bin        63488 Jun 23  1992 acctcom
-rwxr-xr-x   2 bin      bin        73728 May 11  1989 adb
-rwxr-xr-x   1 bin      bin        49252 Apr 14  1992 ar
-rwxr-xr-x   2 bin      bin       110593 Apr 13  1992 as
-rwxr-xr-x   2 bin      bin        30720 Apr 14  1992 att
-rwxr-xr-x   1 bin      bin          147 Apr 14  1992 basename
-rwxr-xr-x   1 bin      bin        77824 Apr 14  1992 bs
-rwxr-xr-x   1 bin      bin        28672 Apr 14  1992 cat
-rwxr-xr-x   1 bin      bin        43008 Apr 14  1992 cc
-rwxr-xr-x   1 bin      bin        32768 Apr 14  1992 chgrp
-rwxr-xr-x   1 bin      bin        26624 Apr 14  1992 chmod
-rwxr-xr-x   1 bin      bin        32768 Apr 14  1992 chown
-rwxr-xr-x   1 bin      bin        26624 Apr 14  1992 cmp
-rwxr-xr-x   3 bin      bin        32768 Apr 14  1992 cp
-rwxr-xr-x   1 bin      bin        73728 Apr 14  1992 cpio
-rwxr-xr-x   1 bin      bin        28672 Aug 28  1993 crypt
-rwxr-xr-x   2 bin      bin       110593 Apr 14  1992 csh
-rwxr-xr-x   1 bin      bin        32768 Apr 14  1992 date
-rwxr-xr-x   1 bin      bin        32768 Apr 14  1992 dd
-rwsr-xr-x   1 root     bin        34816 Apr 14  1992 df
-rwxr-xr-x   1 bin      bin        34816 Apr 14  1992 diff
--More--
```

Figure 4.6 Output from piping ls -l bin through more command.

Any command that sends text to the screen can also send it to `more`. This is a good example of the real beauty of UNIX: instead of giving you a couple of big utilities that claim to do everything you need, UNIX gives you many small ones that you can combine any way you like. If you like a particular combination so much that you'll want to use it repeatedly, you can store those commands in a shell script file and give this file any name you like. When you want to use your shell script, you only need to remember the name you made up rather than the spelling and syntax of the combination of commands. Section 6.2, "Command Files," shows you how to do this.

4.1.5 Copying Files

Copying files in UNIX is simple. The command is clearly an abbreviation of the word "copy": `cp`. To make a copy of your file with a different name, type

```
cp sourcefile destfile
```

where `sourcefile` is the file that you are copying and `destfile` is the name of the copy that you are making. (See section 3.2, "Filenames," for information on valid filenames.) For example, if you plan to edit a file called `proposal` and you're going to make so many edits that you want a backup of your original before you start changing it around, you would type:

```
cp proposal proposal.bak
```

This is the simplest form of the `cp` command. It assumes that the file you want to copy is in your current directory and that you want to put the copy in the same directory. To get fancier, you can use the syntax

```
 cp /pathname/sourcefile /pathname/destfile
```

where `/pathname` specifies the pathname, or full directory name, of the source and destination files. Let's say Mary Jones tells Joe User that Herb sent her some electronic mail that he should look at and add comments to. She says, "I saved it in the subdirectory of my home directory called `mymail` in a file called `aug20.herb`. I'll see you later. I'm flying to Phoenix in an hour, and I want your comments when I get back." Joe calmly sits at his terminal and types

```
cp /usr2/maryjones/mymail/aug20.herb /usr2/joeuser
```

Notice that he included the source file's directory, the name of the source file, and the destination file's directory, but not a new name for the destination file. If you omit the name of the destination file, UNIX gives it the same name as the source file—in this case, `aug20.herb`. (When making a copy of a file in the same directory as the source file, you must specify a new name—you can't have two files in the same directory with the same name.)

In section 3.3, "How Files Are Organized," we saw that you can use two periods (..) as shorthand to refer to the parent of the current directory. You can also use a single period to refer to the current directory. This doesn't come up when using the `cd` command, because typing

```
cd .
```

would be useless; it means "change my current directory to the one that I'm currently in." The single period does come in handy, however, with the copy command. If Joe is in the `/usr2/joeuser` directory and wants to copy the `aug20.herb` file from the `/usr2/maryjones/mymail` directory into his current directory, he types:

```
cp /usr2/maryjones/mymail/aug20.herb .
```

One more comment about copying that file from Mary's directory: Joe needs read privileges to make a copy of it. If he got a message along the lines of "cannot unlink" or the slightly more comprehensible "permission denied," then he would use the `ls` command with the `-l` switch, as

described in section 4.1.3.1, "Listing More than File Names," to see what kind of privileges were assigned to that file. He doesn't need to look at all the filenames in Mary's `mymail` subdirectory, so he types:

```
ls -l /usr2/maryjones/mymail/aug20.herb
```

If he saw something like this,

```
-rw-------   1 maryjones marketing   147 Aug 20  1994 aug20.herb
```

he would see that the mode of that file was set so that Mary, its owner, could read it or write to it, but no one else could read it, not even other people in her group (like Joe). Secure in the knowledge that it's Mary's fault that he can't add the comments she's expecting, he sends her electronic mail tactfully explaining why he couldn't do as she had asked.

If he could read the file and make a copy of it, he would then own the copy and be able to do anything he wanted to it.

What happens if you name your new copy after an existing file? There may or may not be a warning, depending on the UNIX system that you are using. The copy operation might take place as if the existing file didn't exist, making a new copy over the existing file. Try copying over an unimportant file on your system to see what happens. If there is no warning, you'll have to be careful about destination filenames when using the `cp` command on your system.

What happens if you try to make a copy of a file that doesn't exist? Like for example, if you misspell the filename of the source file:

```
cp /usr2/maryjones/mymail/aug20.hreb /usr2/joeuser
```

UNIX would display a message telling you that it "cannot access (the source file)," which implies that the file was there, but it couldn't get to it. In reality, it means that no such file exists.

4.1.6 Renaming Files

Like the command to look at a text file, the command to rename a file is named after one of its less common uses. Since it's used to move files from one directory to another, the command is `mv`. Just as the copy command can make a new copy of a file in the same directory as the original file, but with a new name, the `mv` command can "move" a file within its current directory, but with a new name—in other words, rename it. For example, if you typed

```
mv aug20.herb herbfile.txt
```

you would take the file called `aug20.herb` in the current directory and give it a new name: `herbfile.txt`. If you did want to move the file to another directory, perhaps from your home directory to your `/usr2/joeuser/networking` directory, the syntax is similar to copying a file from one directory to another:

```
mv aug20.herb /usr2/joeuser/networking
```

Unlike copying, after this command executes, the original `aug20.herb` file will no longer be in your home directory. You will find it in its new home, `/usr2/joeuser/networking.` If you want to move it and give it a new name at the same time, it's easy:

```
mv aug20.herb /usr2/joeuser/networking/herbfile.txt
```

When you refer to a file but don't specify its directory location, UNIX assumes that it's in the current directory. If you want to do something with a file that isn't in the current directory, insert its pathname in front of the filename. For example, if Mary had told you to move `aug20.herb` out of her directory, instead of just making a copy, you would use syntax similar to when you copied it out of her directory into your own:

```
mv /usr2/maryjones/mymail/aug20.herb /usr2/joeuser
```

Of course, you could have assigned a new name to it when you specified where it should end up.

Just as you can use the single period to specify the destination directory when you copy a file to your current directory, you can also use the single period to specify the destination when you move a file to the current directory:

```
mv /usr2/maryjones/mymail/aug20.herb .
```

Section 3.3, "How Files Are Organized," shows you other ways to avoid typing out complete pathnames.

A file's mode has no effect on your permission to rename a file, as it does with the `cp` command. Regardless of the privileges assigned to a file, only the owner (the user who created the file) may rename it. And remember: if you make a copy of someone else's file, you become the owner of the copy. How do you find out who owns a file? You use the `ls` command with the `-l` switch to list out that file's name and then look at the third column of information listed with the filename.

If you rename a file with a name that already applies to an existing file, the renaming takes place with no problem. Or rather, it takes place with no problem for your renamed file—the previously existing file with the same name is lost. For example, if you have files called `schedule.txt` and `dec13.txt` and rename `dec13.txt` to be called `schedule.txt`, your original `schedule.txt` will be lost.

If you try to rename a file that doesn't exist, UNIX gives you the same error message as when you try to copy a file that didn't exist: "Cannot access (filename)."

4.1.7 Deleting Files

Think of deleting files as removing them, because that helps you to remember the command: `rm`. The syntax is simple; `rm` followed by the filename or filenames that you wish to remove. For example,

```
rm schedule.txt
```

removes the file called `schedule.txt`. Typing

```
rm schedule.txt junememo.txt
```

removes `schedule.txt` and `junememo.txt`.

To remove a file, you need write permission in the directory in which the file is located. If you do not have write permission for the specific file you want to erase, UNIX displays a cryptic message:

```
(filename): 444 mode?
```

This means "Are you sure you want to remove this file, which has a mode of 444?" Sometimes a numbering system is used as a shorthand for the -rwxrwxrwx notation to describe the permissions that make up a file's mode. Without explaining which numbers mean what, it's enough to say that if you have write permission on a file that you're trying to erase, the system won't give you the warning message. Answer the warning message with either a "y" for "yes" or an "n" for "no," and press the Enter key. To be on the safe side, "n" is probably a better idea; you can then use the `ls -l` command to double-check the file's mode and then enter the `rm` command again if you're sure that you want to erase that file.

Why would someone not have write permission of a file that they own? You might use the `chmod` command to take away write permission from yourself for an important file to protect yourself from accidentally erasing it. You'll still own that file, so you can always grant yourself write permission for it with the `chmod` command. (For more on the `chmod` command, see section 4.1.8, "Controlling Access to a File.")

The `rm` command accepts wildcard characters in its argument. Be careful, though, because this ability to remove more than one file at a time can lead to big mistakes. If you wanted to remove all of the files that ended with ".bak" you would type this:

```
rm *.bak
```

Imagine that you made the simple typing mistake of adding a space after the asterisk:

```
rm * .bak
```

Just as the command `rm schedule.txt junememo.txt` removed the `schedule.txt` and the `junememo.txt` files, this command also specifies two things to remove: first, all the files that match `*` and second, the file named `.bak`, if it exists. All the files that match `*` would be all the files in the current directory, so you could get yourself into big trouble.

What if you typed

```
rm maymemos
```

and received the following message:

```
rm: maymemos directory
```

Sometimes UNIX is not much more eloquent than Tarzan. "Me UNIX, maymemos directory." `maymemos` is a directory, and you can't remove it with the command that removes files. Section 4.1.10, "Removing Directories," explains how to use the `rmdir` command for this.

4.1.8 Controlling Access to a File

Use the `chmod` command to change a file's mode. There are two possible ways to specify the access rights to your file: first, by a three-digit "octal" number (which means that each digit is lower than 8, because the number is written in "base 8" notation); second, by initials representing whose rights are being controlled, whether those rights are being added or removed, and what the rights are. The latter way is easier to remember, so that's the best one for beginners to start with.

Use these initials to specify whose rights are being controlled:

`u`	You, the file's owner, the *u*ser.
`g`	Other users in your *g*roup.
`o`	*O*thers outside of your group.

As you saw in section 4.1.3.1, "Listing More than File Names," the letters `r`, `w`, and `x` indicate read, write, and execute permission.

To show that you want to add or take away permission, use the plus (+) and minus (-) characters.

The complete format of the `chmod` command is:

```
chmod [ugo]+/-rwx filename
```

This command has the following parts:

- The `[ugo]` is where you put the combination of the letters `u`, `g`, and `o` showing whose permissions you are specifying. The square braces mean that you can leave this out. If you do, the system assumes that you mean `ugo`—in other words, everybody.
- Next, you put a plus sign when adding permission or a minus sign when removing it.
- After the plus or minus symbol, you put the combination of the letters `r, w,` and `x` that indicate the permission or permissions being added or taken away.
- Finally, after a space, you type the name of the file for which you are specifying permissions. You can use wildcards if you want to change the mode of several files at once.

Make sure that the string of characters showing the users, action, and permissions have no

spaces. The only spaces in the whole command should be right after the word `chmod` and just before the filename.

For example, let's say you created a file with your resume in it. You cleverly give the file a boring name that won't attract attention, like "budget.old." You then realize, however, that maybe a clever name isn't enough; maybe your file needs more protection than that, so you check on its permissions with the `ls -l` command, and see the following:

```
-rw-rw-r--   1 joeuser  marketing   3590 Jun 17  1994 budget.old
```

You and the people in your group may change it, and everyone may read it. This is not good, so you first take away your group's permission to write to your file. To specify rights, you enter "g-w" for "group-remove-write privileges."

```
chmod g-w budget.old
```

When you type "ls -l" to see if it worked, you should see this:

```
-rw-r--r--   1 joeuser  marketing   3590 Jun 17  1994 budget.old
```

Next, you want to take away the permission of your group and the others outside of your group to read the file. Instead of doing this in two separate commands, you can combine the `g` and the `o` with the following command:

```
chmod go-r budget.old
```

You can also combine the permissions being given or taken away. In fact, the two preceding commands could have been combined with the following command:

```
chmod go-rw budget.old
```

Until you feel comfortable with the `chmod` command, always use the `ls -l` command afterwards to make sure that you did exactly what you intended to the file's mode.

You can also grant or revoke permissions from more than one file at a time by using wildcards in the filename. For example,

```
chmod go+rw *.txt
```

would set the mode of all the files that end with ".txt" so that your group and everyone else could read them and write to them.

Try taking permissions away from yourself by entering `u` as the user whose rights are being controlled. Then, try to read or write the file with the `cat` command or the `vi` editor. Then try giving permission back to yourself. (When fooling around with a new command like this, make sure to use a file that means nothing to you!)

In section 6.2, "Command Files," you'll see an example of execution permission being added to a file.

4.1.9 Creating Directories

The commands to create and remove directories are both simple: `mkdir` (make directory) followed by a directory name creates a new directory and `rmdir` followed by a directory name removes a directory.

The rules governing valid subdirectory names are the same as those that govern valid filenames. To create a subdirectory of `/usr2/maryjones/mail` called `oldmail,` Mary could type the following:

```
mkdir /usr2/maryjones/mail/oldmail
```

Relative pathnames also work; if Mary is already in the `usr2/maryjones/mail` directory (and she can always use the `pwd` command to check which directory she's in) then she only needs to type this:

```
mkdir oldmail
```

If she tried to create a subdirectory of one that she didn't own, like `/bin,` the system wouldn't allow her to. Typing

```
mkdir /bin/wahoo
```

would cause an error message similar to the following:

```
mkdir: cannot access /bin
```

which means that she doesn't have enough access to the `/bin` directory to allow her to create something new there. (In other words, she doesn't have "write" access, which would allow her to create something in that directory.) The system administrator can create directories anywhere. In fact, that's an important part of the system administrator's job—to create and maintain directories to hold the system and application files.

One other word of caution: because of the similarities between names of files and names of child directories, it's possible to try to create one of these when you already have used the same name for the other. For example, if Mary had a file called `oldmail` and entered the command

```
mkdir /usr2/maryjones/oldmail
```

she would get a message similar to this:

```
mkdir: cannot make directory /usr2/maryjones/oldmail
```

Since the error message doesn't tell you why it couldn't make the directory, you'll have to watch out for this yourself.

4.1.10 Removing Directories

The syntax and restrictions on removing subdirectories is similar to that of creating them. If Mary had successfully created her `oldmail` subdirectory and she wanted to get rid of it, she could type:

```
rmdir /usr2/maryjones/mail/oldmail
```

If she was already in the `usr2/maryjones/mail` directory then she can use the relative pathname:

```
rmdir oldmail
```

Just as she cannot create subdirectories of directories that she does not own, she cannot remove directories that she does not own. Only the system administrator can remove any subdirectory on the system.

One other obstacle could prevent someone from removing a directory: if it has either files or subdirectories in it, UNIX won't let you remove it. This is really a safety feature to protect you from yourself. If Mary had gotten the message

```
rmdir: oldmail not empty
```

then she would use the `cd` command to change into `oldmail` to see what was there and either erase what she found, move it somewhere else, or change her mind about deleting `oldmail.`

We saw what happens when you mistake a subdirectory name for a filename and try to remove it with the `rm` command, which we normally use to remove files. The reverse is also a common mistake; look what happens when you use the `rmdir` command to try to remove a file. After typing

```
rmdir schedule.txt
```

the system responds with

```
schedule.txt: not a directory
```

to let you know that you can't use this command with `schedule.txt`, because it is a file and not a directory.

Chapter 5 The UNIX vi Text Editor

There are two commonly used editors on UNIX systems. The older one, known as `ed`, is a line editor. (Most systems also have a more advanced version of `ed` called `ex`.)

The most popular editor on UNIX is a full-screen editor called `vi`. (Some people pronounce it as a one-syllable word rhyming with "eye" and others pronounce it as the two letters that spell it—"vee eye." I couldn't even find a consensus when I asked a roomful of Bell Labs employees.) The name is an abbreviation of "visual editor." `vi` has much more in common with modern word processors than it does with `ed`. You can move your cursor anywhere on the screen and correct the text under the cursor. You can scroll the text and search for specific strings of text. You can use `vi` to create a new text file, as well as to edit an existing text file.

`vi` is a command-driven editor. You don't use function keys and menus to tell it what you want, as with other text editors and word-processors; you type in commands, many of which are only one letter long, and it carries them out. The advantage to this arrangement is that you can do a lot of different things with very little typing. The disadvantage is that many systems do not indicate when you are in command mode and when you are entering text in insert or replace mode. This leads to two common mistakes:

- You might accidentally enter a command when the system thinks that you are entering text, so that you enter `d3w` to delete three words and the characters "d3w" show up in the middle of your memo, program, or whatever you are writing.
- The opposite problem also occurs: you type a word onto you document, such as "Hello," and the system thinks that you are doing whatever the `H` command means, followed by the `e` command, followed by the `l` command twice, and so on.

The best way to avoid this problem is to double-check your current mode when you are unsure by pressing the Escape key. The Escape key puts you into command mode, and if you press Escape when you are already in command mode, the terminal beeps at you, as if to tell you, "Enter command mode? We're already in command mode."

Because many programs and operating systems require you to press the Enter key after you enter a command, it is tempting to do so with `vi`, but unnecessary with most commands. In fact, if you enter an `i` command (which puts you into insert mode) and then press Enter, you insert a carriage return into your document. If you press Enter in command mode when it didn't make sense in the context of what `vi` thought you were doing, it would just beep at you. As a `vi` beginner, get used those beeps!

5.1 Entering vi

To enter `vi`, type `vi` followed by the name of the file that you want to edit at the UNIX shell

prompt. If a file with that name does not exist, `vi` creates an empty, new file with that name. If it does exist, `vi` displays that file on your screen and waits for you to edit it. If you do not include a filename, you will still enter `vi`, but you must assign a filename later. See "Saving Your File" below.

When you first enter `vi`, you are in command mode. You can use your cursor keys to move your cursor around the screen to any place with text at any time in command mode. (Sometimes you can move your cursor like this in insert mode, but there's a greater chance that `vi` will act flaky, particularly if you use a PC running a terminal emulation program and not a real terminal.) If a file is too long to completely fit on the screen at once, move your cursor to the bottom of the screen and then continue to press the Cursor Down key to scroll the file up, revealing more text. If there are more lines above the one visible at the top of your screen, move the cursor to the top of the screen and continue to press the Cursor Up key to scroll the file down, revealing the text above the line that was at the top of your screen.

If your file is not long enough to fill up a screen, `vi` represents lines that have no text with a tilde symbol (~). If you enter

```
vi johngay.txt
```

and that file has only seven lines, you will see a screen like the one in Figure 5.1.

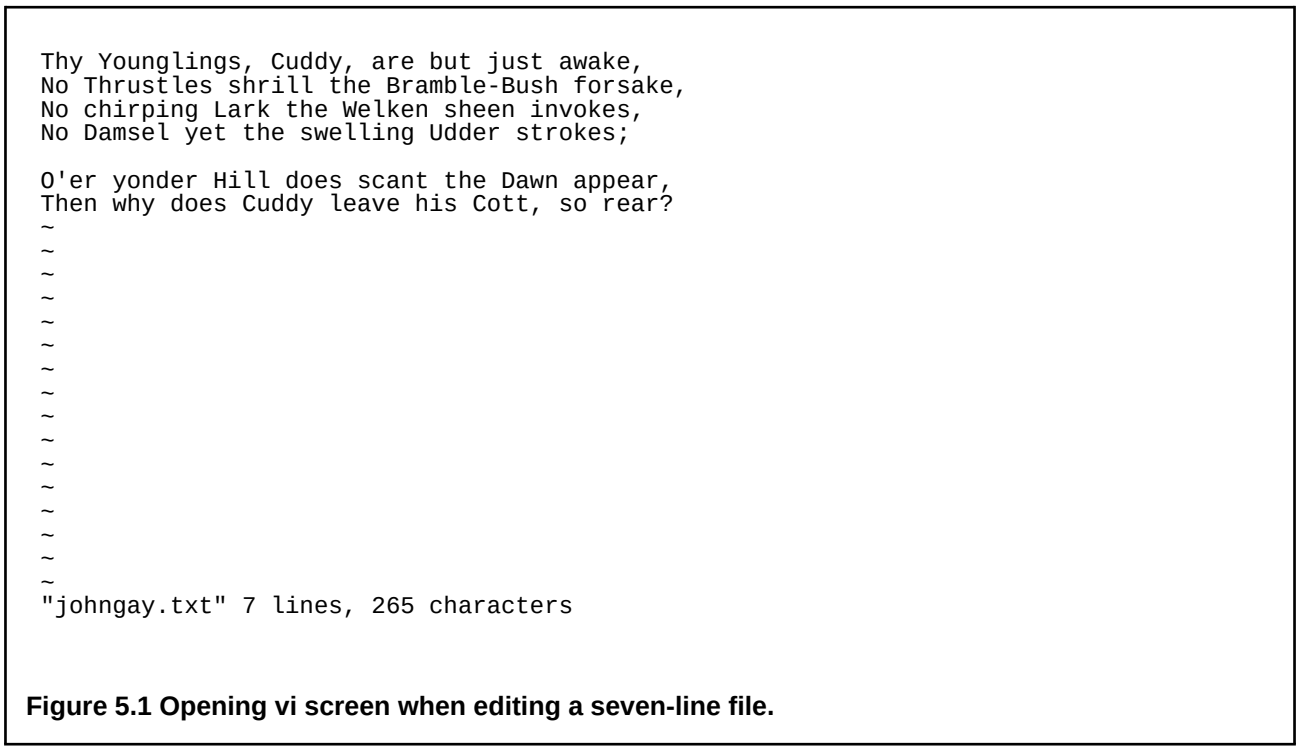


Figure 5.1 Opening vi screen when editing a seven-line file.

Note also that it tells you at the bottom of the screen the name of the file you are editing and how many lines (including blank ones) and characters it has. If you had created a new file with the `vi` command, it would say "New File" at the bottom.

5.2 Inserting Text

To insert text, first move your cursor to the place where you want the new text to begin. Make sure you are in command mode (as mentioned above, if you're not sure whether you're in command mode, press the Escape key first). Type a lower case `i` to put `vi` into insert mode. If you are lucky (if you are using or emulating a more sophisticated terminal), the word "INSERT" or something similar appears somewhere on your screen to indicate that you are in insert mode. If not, you won't see anything happen, but all the text you type until the next time you press Escape appears at the cursor as part of your file.

If you type to the end of the line, the cursor jumps to the next line, but only as an alternative to running off the right of the screen—it didn't really insert a carriage return character at that position in your file, so make sure to press Enter when you are inserting text and your cursor nears the right side of your screen. When you finish entering new text, press Escape to return to command mode.

5.3 Deleting Text

The lower case `x` has the same effect in `vi` as the delete key on many keyboards: it deletes the character at the cursor. Press it as many times as you like to get rid of more than one character.

To delete more than one character, it is often easier to use the `d` command. Pressing `d` by itself does nothing; `vi` waits to find out what to delete. The `d` is used in combination with other letters and numbers to delete words, lines, the rest of a sentence, or the rest of a paragraph. The most important of these for a beginner is the `dw` command, which deletes from the cursor to the beginning of the next word. You can stick a number in there to delete more than one word; for example, `d4w` deletes the next four words.

The `dw` command can also delete a blank line, like the one between "No Damsel yet" and "O'er yonder Hill" in Figure 5.1.

When you use `vi` commands that consist of more than one character, you may occasionally enter a character or two without being sure of how many you just entered. Again, the Escape key always puts you back to a fresh start in command mode. If you're unsure whether you typed `d` or `d4`, press Escape and type `d4` again. (If you accidentally typed an extra "4" after your `d4`, you could end up deleting 44 words!)

5.4 Typing Over Existing Text

All the `vi` commands that we have seen so far have been lower case letters. To enter overstrike mode, you'll use your first upper case `vi` command. Type `R` to enter Replace mode, and everything you type writes over the characters at the cursor until the next time you press Escape. (A lower case `r` has a related function: it means you only want to type over one character, so the next character you type appears at your cursor, but `vi` then puts you right back into command mode. This is useful for making very minor corrections.)

If you reach the end of a line while typing in replace mode, you can continue typing. `vi` will add your new text to the end of the line at the cursor position.

5.5 Searching for Text

To search for a string of characters in your document, first press the slash (/) key while in command mode. The slash appears in the lower-left hand corner of your screen, waiting for you to type in the characters to search for. After you type them and press Enter, `vi` will search for the characters and display that part of the document if the string is found. The search is case-sensitive, so make sure you type the letters in upper case and lower case exactly as you want to search for them.

5.6 Saving Your Changes

`vi` inherited many commands from older UNIX text editors such as `ed` and `ex`. High on the list of these commands are those that save your work, indicate another file to edit, and exit from the editor.

This category of commands has its own prompt. To display it, type a colon (:) while in command mode. The prompt, which is also a colon, appears in the lower-left corner of your screen. If you type a command at this prompt that `vi` does not understand (which includes upper case versions of commands that should be in lower case), it displays an error message telling you that it doesn't recognize the command. For example, if you enter "potrzebie" at the colon prompt, `vi` responds with "potrzebie: Not an editor command" or "potrzebie: No such command from open/visual."

To save your work, type a `w` for "write" at this prompt. If you put a filename after the `w`, `vi` saves the file with that name. If you type `w` alone, `vi` saves your file with its current name and displays a message telling you the file's name, the number of lines in it, and the number of characters. The first time you save a particular file, it also says "[New file]."

If you've created a new file but try to save it with the name of an existing file, `vi` displays a message like this:

```
File exists - use "w! filename.txt" to overwrite
```

This means that you tried to save a file with a name that already exists and that you must put an exclamation point (a "bang") after the "w" to override the warning. Although the warning message shows the filename in the syntax, you don't need to include it if the file already has a name; just type `w!` at the colon prompt.

> **BUZZWORD** *Bang* The exclamation point comes up a lot in UNIX. In addition to its use in `vi`, it's sometimes used to distinguish the components of an electronic mail address when sending mail through a big network to another UNIX machine. Because "exclamation point" can be a real mouthful if you have to say it often, the term "bang" is often used. (Another cute one is "Ballbat.") If someone tells you to "type w bang space filename dot txt," they're telling you to type `w! filename.txt`.

If you entered `vi` by merely typing `vi` at the UNIX shell prompt and then created a new file from scratch, it won't have a name yet, so entering `w` by itself at the colon prompt causes `vi` to display the message "No current filename." In other words, type the `w` command again, but include a filename this time.

You can't edit just any file, or create a new one in just any directory. If you only have read permission for a file, you can still bring it up into `vi`; when you do, the editor displays the message "[Read only]" at the bottom of your screen. You can make all the changes you like, but when you type the `w` command at the colon prompt and press Enter, vi doesn't save the file; it displays the message "File is read only" instead. You may save the changed file with a new name by adding a filename after the `w` command. This is based on the same principle as copying a file that you only have permission to read—if you make a copy, you own the copy, and you can do anything you want to that copy, but you aren't allowed to make any changes to the original.

If you don't even have read permission for a file, `vi` starts up but displays a message that says "Permission denied." It shows you an empty file, just as if you had typed `vi` by itself at the command line.

5.7 Quitting vi

To quit `vi` and return to the UNIX shell prompt, type "q" at the colon prompt. If you made any changes without saving them, `vi` gives you the message

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

instead of quitting. This serves as a reminder that you might want to save your file before you quit, and it also reminds you that if you really want to quit without saving your changes, put an exclamation point after the "q." (It really suggests that you spell out the word "quit" and then put an exclamation point, but spelling out an entire English word would be very un-UNIX, so just enter a "q.")

You can combine commands at the colon prompt. To save your file and quit `vi` in one command, type "wq" at the colon. `vi` displays a message telling you that it is writing the file to the disk, and it then returns you to the UNIX shell prompt.

5.8 Other vi commands

The commands described here are the bare minimum that you need to get by in `vi`. There are many, many more; they are powerful and often confusing. `vi` has a reputation among people accustomed to normal word processors as being more cryptic and confusing than UNIX itself. This often stems from the fact that someone once tried to teach them all of `vi` at once, instead of showing them the basics, letting them get comfortable, and then showing them a little more.

To learn more about `vi`, check the quick reference cards that are available. Also, nearly every book on UNIX devotes a chapter to it.

There are two more tricks that you should know in case you find `vi` acting flaky. Usually it's not `vi` itself that's causing the trouble, but a lack of cooperation between a terminal emulation program on a personal computer and the system running `vi`.

The first trick sometimes makes cursor control easier. Before computer users took it for granted that every keyboard had special keys devoted to cursor movement (yes kids, there was such a time), the `h, j, k,` and `l` keys were used as the `vi` commands to move the cursor left, up, down, and right, respectively. If you can remember which four letters are used (they're lined up next to each other on the keyboard), then you can easily remember the purpose of the `h` and the `l` because they sit at the left and right of these four keys. `j` for up and `k` for down are a little more difficult to remember, and I always need to press them a couple of times to remember which does what. As with any `vi` command, make sure that you're in command mode when you press these four keys, or you'll add their letters to your file in places where you don't want them.

Many touch typists with cursor keys on their keyboard actually prefer the use of these four letter keys over the cursor keys when they move their cursor around, because they can find them without looking down and moving their hands away from the middle of the keyboard. If this doesn't apply to you, but your cursor keys don't behave correctly when you edit a file with `vi`, try using

these alternate keys for cursor movement.

The other trick for people who find that `vi` does not cooperate with their terminal emulation program as much as they would like is the redraw command. If you delete characters without seeing them deleted from the screen, or come across other situations where the words on the screen don't reflect the commands you just entered, press `Ctrl+L` to tell `vi` to resend the whole screen to your terminal. Your terminal emulation program may have retained or deleted a couple more characters on the screen than it was supposed to. Although the wrong characters may be on the screen, this doesn't necessarily reflect what's in the copy of the file being edited. `Ctrl+L` straightens out your terminal emulation program by redisplaying the true contents of that portion of the file that you are editing.

Don't worry about the need for this unless you notice `vi` acting strangely. It's still a good idea to use the `cat` command to look over any file created or edited with `vi` after you've saved the file to disk, just to make sure that your terminal emulation program didn't play any tricks on you.

Chapter 6 Using a UNIX System

6.1 Printing Text Files

The original term for the machine that printed your file on hard copy was "line printer," because it printed text a line at a time. The command to send text to the printer is an abbreviation of "line printer": `lp`, pronounced "el-pee." Don't worry if you're attached to a laser printer, which is actually a page printer; it's still the same command.

To print a file, just type:

```
lp filename
```

It's that simple. If the file exists, the system sends it to the printer and UNIX displays a message similar to this:

```
request id is hp1-1151 (1 file)
```

The *request id* is the name by which the system remembers your file. It comes in handy when you want to list out the jobs that are waiting to print and when you want to cancel a print job. If you get an error message (other than the kind indicating that the system didn't understand the filename you typed) contact your system administrator. Various details about printing can be specific to each UNIX system (like the name the system uses to refer to each printer) and you can't be expected to know them on a strange system.

6.1.1 Checking the Print Queue

To find out the status of jobs waiting to print on the "line printer," the command is `lpstat`. If Joe User just printed a very short job and no other jobs were waiting to print, it may be too late, so `lpstat` may display no information. This means that the system has already sent his file to the printer. On the other hand, he may see something like this:

```
$ hp1-1151            maryjones        5,232   Jul 19 09:50 on hp1
$ hp1-1158            joeuser          1,491   Jul 19 09:53 on hp1
$ hp1-1159            jimcasey         2,781   Jul 19 09:53 on hp1
```

This means that three print jobs are waiting to print on the printer that the system administrator named "hp1"—a 5,232 byte job that Mary Jones sent at 9:50, then your job, and finally Jim Casey's job.

6.1.2 Canceling Your Print Job

Perhaps Joe realizes, looking at the print queue, that he accidentally sent a draft of a memo about why he can't stand Jim Casey. Since he doesn't want it to pop out of the printer while Jim stands there waiting for his 2,781 byte job, he cancels his with the `cancel` command followed by the

request id:

```
cancel hp1-1158
```

Only you and the system administrator are allowed to cancel your jobs.

6.2 Command Files

In UNIX, a file full of commands that you can execute as a program is called a *shell script*, because it's a script of commands for the shell to execute one after the other. Shell scripts can be complex, but simple ones can also be useful—especially for users who have trouble remembering a lot of UNIX syntax.

We saw in section 4.1.2 ("Common Error Messages") that when you type anything at the shell prompt, UNIX looks for a program to execute with that name. When you create a shell script, you are essentially adding a new command to your UNIX environment. If you write a shell script that contains valid shell commands and store them in a file called "wahoo," then typing "wahoo" starts up your new program.

For example, let's say you're a DOS or a VAX/VMS user and accustomed to typing `dir` to see a list of filenames. You're also used to seeing the size and age of files along with their names, and you don't want them to zoom off your screen if there are more than twenty-four of them. The following UNIX command lists files this way:

```
ls -l | more
```

This is kind of a pain for the new user to remember. So, to make things easier for yourself, you use `vi` to create a file called "dir" which only has one line in it:

```
ls -l | more
```

After you save your one-line text file and return to the shell prompt, you can't wait to try your new program, so you type

```
dir
```

and UNIX displays a message telling you "execute permission denied" or worse, "not found." No execute permission? Not found? But you own it! You just created it! Check out the mode of your `dir` file by using `ls -l`, the very command that you planned to avoid by creating the `dir` shell script:

```
ls -l dir
```

You'll see that it has a mode of something like `-rw-r--r--`. (You don't need the `| more` when you check out the `dir` file's mode because you only want to list one filename, so there's no need to use the `more` program to make the `ls` output appear a page at a time.) The system administrator sets the default file mode for everyone's new files. This default usually doesn't include execute permission, because system administrators assume that most of the files you create

will not be shell scripts.

As you saw in section 4.1.8, "Controlling Access to a File," we use the `chmod` command to add privileges to a file. In this case, you want to give yourself execution privileges for your `dir` program:

```
chmod u+x dir
```

Now when you type `dir`, it should have the same effect as typing `ls -l | more`. You've written your first useful, working shell script!

At this point, your shell script only works for you if you are in the same directory as the shell script file. Ask your system administrator for help modify your *search path,* which determines where UNIX looks for programs to execute when you type a command name at the shell prompt. If you store your shell scripts in their own subdirectory of your home directory and add the name of that directory to your path, you can use your scripts no matter which directory you are in.

6.2.1 The Automatic Login Command File

If you type

```
ls -al
```

you may see one or more files that you didn't see before in the list of files in your home directory. The `a` means "all," and tells `ls` to include the "hidden" files from that directory in the list. As you can see, they're not hidden very well; they all have a period (.) as the first character in their filename. Typing

```
ls -l .*
```

is another way to list these files—you're telling UNIX to list the files whose names begin with a period.

One of these files is called either `.profile` or `.login` (pronounced "dot profile" and "dot login"). Both are shell scripts; whenever you log in, the system looks for `.profile` if you are using the Bourne shell or `.login` if you are using the C shell and executes it automatically. This means that, if there's any commands that you want executed every time you log in, you only need to add them to that file. (If you don't have one, you can create it the same way you create any other shell script. Just remember to store it in your home directory and to give yourself permission to execute it with the `chmod` command.)

For example, you could add the lines

```
mail
cat schedule.txt | more
```

to your `.profile` or `.login` file. The first line, as you will see in the next section, checks to see if you have mail (although some systems automatically check for this when you log in any-

way) and the next displays a file called `schedule.txt` one page at a time. If you keep your appointments in this file, it's handy to have them listed like this whenever you log in.

Some systems automatically create a `.profile` or `.login` file when the system administrator creates a new user ID. If this happens with your ID, the shell script probably has some strange-looking commands in it. Try looking these up with the `man` or `help` commands.

When you add new commands, add them at the end of the file. To test your revised automatic login script to see if it works, you don't have to log off and log back in again; as with any shell script, you can just type its name to start it up.

6.3 Communicating with Other Users

You can send mail to someone by merely typing the word `mail` followed by the login name of the person getting the mail. For example,

```
mail maryjones
```

indicates that you want to send mail to Mary Jones. The mail program then waits for you to type in your message, but it doesn't give you any proper editing capabilities. It's better to use the `vi` editor to create a text file with the message that you want to send, and then "send" that file to the mail program the same way that you sent a file to the `more` command when you wanted to see the file's text one screen at a time: with the pipe (|) symbol. The following command sends the file `072194mj.txt` to Mary Jones' mailbox:

```
cat 072194mj.txt | mail maryjones
```

You can send any text file to someone with this trick, not just one that you created yourself for this purpose.

6.3.1 Receiving Mail

To check whether you have any mail, simply type

```
mail
```

all by itself at the UNIX prompt. If you have no mail waiting for you, it tells you something like

```
No mail.
```

and returns you to the UNIX prompt.

If you do have mail, this enters the mail program. It first shows you either the message most recently sent to you or a list of headers that describe who sent the messages in your mailbox and when they sent them. Then, the mail program's prompt—usually either a question mark or an ampersand—appears. The mail program has at least twenty commands that you can type at this

prompt, but you only need a couple to get by. As with most programs, the most important command is the one that tells you about the others: the question mark.

The system numbers your messages and shows these numbers in the list of mail headers. You can refer to these messages by number; for example, entering `d 3` at the mail program's prompt means "delete message number 3."

You don't have to use the numbers, especially if you're a beginner. Keep in mind that when you don't include a number in a command entered at the mail program prompt, the system treats the command as if you're referring to the "current" message. The simplest way to see which message is current is to enter the `p` command without any number. This displays ("prints") the current message. It's a good idea to do this before you delete a message with the `d` command, to make sure that you're deleting the right one.

Among the available commands, here are the important ones:

`?`	List the available commands in the mail program with a brief description of each.
`p`	Print the current message on the screen. The message will have a header describing, at the very least, the date and time that it was sent and the sender's login ID. It may also include other information with lots of numbers, initials, and punctuation. This is information that the mail program uses to route the message; it may be useful to your system administrator if you have any problems sending or receiving mail.
`+`	Move forward to the next message and print it. If there is no next message, this may return you to the UNIX prompt. The letter `n`, for "next," also works.
`-`	Print the message before the current one.
`1`	(Or any other number) When you enter a number without any other characters, you're telling the mail program to print the message with that number. See the `h` command to find out how to see the messages' numbers.
`s [filename]`	Save the current message with the filename shown. The filename is optional; if you don't include it, the mail program saves the message in your current directory with the name `mbox`. (If `mbox` already exists, the mail program will add the message to the end of the file.) Saving a message deletes it from your mailbox, which makes sense, because once you save it in a file, you don't need it in your mailbox anymore.

`d [1]` — Delete the current message. If you specify a number after the `d` command, the mail program deletes the message with that number. After you enter this command, the `mail` program shows you the message in question. The system doesn't actually delete it until you leave the mail program; the `u` command unmarks a message that you've marked for deletion.

`h` — Print out active message headers. (Not all mail programs provide this command.) The message headers show one line of information about each message in your mailbox. This line will include the login name of the person who sent you the message, the date and time it was sent, and the size of the message in bytes. (If the size appears as two numbers with a slash, like 11/266, these numbers represent the number of lines in the message and the number of bytes.) A greater-than symbol (>) on the left of the screen shows which message is current; many mail programs also have a column with a one-letter code that tells you more about the status of each message: "u" for unread, "s" for saved, "d" if it was marked for deletion, or "n" for new.

`r` — Reply to the current message. After you enter `r` and press Enter, the mail program may prompt you for the subject of your message, and then you will have a seemingly blank line. Enter the first line of your message, press Enter, and continue to type in your reply. Type it carefully, because each time you press Enter there's no going back to edit that line. When you are done, type a period (.) as the first character of a new line and press Enter.

If this sounds like a pain, it is. If you want to send a reply of more than two or three lines, jot down the login ID of the person that sent you this message, quit out of the mail program, and send them mail the normal way by first composing it in a text file with `vi` and then sending it to the mail program with the pipe symbol.

`q` — Quit out of the mail program and return to the UNIX prompt.

6.4 A Sample UNIX Session

You just received the following fax from your boss Mary in Phoenix:

```
Joe - I just remembered that there's some confidential stuff in the
aug20.herb file, so I may have set it so that only I could read it.
If you had any trouble, most of the info from it is in another file
in my home directory called seppromo.txt.  Everyone else has made
copies of that, so you can't have any problems copying it to your own
directory.  Look it over, and then give me a status report on your
responsibilities in the special promotion we're doing in September.
```

```
Just e-mail me your status report; I'll log in from Phoenix to read it.
Make sure it's there by Wednesday at 9AM, Arizona time.
-MJ
```

No problem, except that you haven't done half the work that you're supposed to on the September promotion. First step: you'll print out your own notes on the promotion, which you've stored in a file called `promo1.txt`. You type the following command:

```
lp porno1.txt
```

The system responds with the message

```
request id is hp1-1343 (1 file)
```

so you know that your file is queued for the laser printer. But wait! You mistyped "promo1.txt" as "porno1.txt," accidentally sending the first chapter of your erotic work-in-progress to the printer! Not only is your eventual masterpiece not ready for a publisher yet, it's definitely not ready for the secretaries who hang around the laser printer waiting for their memos to come out.

You don't panic. Instead, you type

```
lpstat
```

to see the print queue, and you see the following:

```
$ hp1-1318            mlopez          3,275   Aug 22 11:31 on hp1
$ hp1-1325            jimcasey        1,381   Aug 22 11:33 on hp1
$ hp1-1343            joeuser         6,923   Aug 22 11:34 on hp1
```

It's not too late; you type

```
cancel hp1-1434
```

and see the message

```
cancel: request hp1-1434 non-existent
```

Your palms start to sweat. You look back at the list of waiting print jobs, and realize that you typed the wrong request id number. You try again:

```
cancel hp1-1343
```

and press Enter. This time you type it very carefully. No error message appears. To make sure that it worked, you type

```
lpstat
```

again, and this time you see this:

```
$ hp1-1325            jimcasey        1,381   Aug 22 11:33 on hp1
```

It looks like you were just in time. Maria Lopez's job finished printing, and Jim's is about to start. The important thing is, yours will not print. Now you try printing `promo1.txt` again, typing

more carefully this time:

```
lp promo1.txt
```

Next, you check the print queue with the `lpstat` command. This time, you see

```
 $ hp1-1325            jimcasey         1,381   Aug 22 11:33 on hp1
 $ hp1-1348            joeuser          2,348   Aug 22 11:36 on hp1
```

so you know that you'll have it soon.

While you wait for it, you copy the file Mary told you about into your own directory. She said that it was in her home directory, not her mail directory, so you copy `seppromo.txt` into your home directory with the following command:

```
cp /usr2/maryjones/seppromo.txt /usr2/joeuser
```

After printing it, you check to make sure that it's there. You could type `ls` and list every file in the current directory, but you only want to check this single file. On the other hand, you're too lazy to type out `ls seppromo.txt`, so you take advantage of UNIX's wildcards and type:

```
ls -l sep*
```

You include the `-l` because you're curious about the file's size. The system responds with the following:

```
-rw-rw-r--   1 maryjones marketing  19853 Sep 18  1994 seppromo.txt
```

Almost 20 kilobytes, which is about 10 pages. You hope it's useful.

After you pick up your `promo1.txt` printout from the printer, you look it over and it's not too bad. If you trim it down and add some stuff from `seppromo.txt`, it should keep Mary happy. To be safe, you want to keep the original file, so you make a copy of it and work on the copy. This copy will end up as a summary of your work on the September promotion, so you enter the following copy command:

```
cp promo1.txt seppromo.txt
```

After you press the Enter key, you realize: Mary's file was called `seppromo.txt`, and you just copied over it. Dumb, but not a disaster—you can copy her original file into your directory again. This time, when you copy it, you give the copy a new name:

```
cp /usr2/maryjones/seppromo.txt /usr2/joeuser/mjpromo.txt
```

Then you print it out with the command

```
lp mjpromo.txt
```

You read through your printouts of `promo1.txt` and `mjpromo.txt` and mark them up with ideas for your own `seppromo.txt` file. After bringing up `seppromo.txt` into the `vi` text editor, you delete a bit, turn some phrases into complete sentences, and add some quotes from

`mjpromo.txt`. You also use the `vi :w` command to save your work every few minutes while you're editing. When you're done, you use the `:wq` to save your final changes and quit out of `vi`.

After printing out your edited file with the lp command and proofreading it, it looks good enough to send to Mary. You mail it to her with the following command:

```
cat seppromo.txt | mail mjones
```

You receive this message:

```
mjones... User unknown
```

Because you entered Mary's login name incorrectly, the system didn't recognize it. You try again:

```
cat seppromo.txt | mail maryjones
```

This time there's no error message, so you know it worked.

That's enough work for the morning, and lunch calls. Because you don't want to leave your desk with your terminal logged in, you type

```
exit
```

to log out.