CHAPTER FOUR

BASIC UNIX COMMANDS AND CONCEPTS

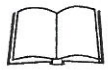
If you've come to Linux from MS-DOS or another non-Unix operating system, you have a steep learning curve ahead of you. We might as well be candid on this point. Unix is a world all its own.

In this chapter, we're going to introduce the rudiments of Unix for those readers who have never had exposure to this operating system. If you are coming from MS-DOS, Microsoft Windows, or other environments, the information in this chapter will be absolutely vital to you. Unlike other operating systems, Unix is not at all intuitive. Many of the commands have seemingly odd names or syntax, the reasons for which usually date back many years to the early days of this system. And, although many of the commands may appear to be similar to their MS-DOS counterparts, there are important differences.

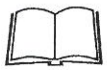
There are dozens of other books that cover basic Unix usage. You should be able to go to the computer section of any chain bookstore and find at least three or four of them on the shelf. (A few we like are listed in the Bibliography.) However, most of these books cover Unix from the point of view of someone sitting down at a workstation or terminal connected to a large mainframe, not someone who is running their own Unix system on a personal computer.

Also, these books often dwell upon the more mundane aspects of Unix: boring text-manipulation commands, such as *awk*, *tr*, and *sed*, most of which you will never need unless you get into doing some serious Unix trickery. In fact, many Unix books talk about the original *ed* line editor, which has long been made obsolete by *vi* and Emacs. Therefore, although many of the Unix books available today contain a great deal of useful information, many of them contain pages upon pages of humdrum material you couldn't probably care less about at this point.

Instead of getting into the dark mesh of text processing, shell syntax, and other issues, in this chapter we strive to cover the basic commands needed to get you up to speed with the system if you're coming from a non-Unix environment. This chapter is far from complete; a real beginner's Unix tutorial would take an entire book. It's our hope that this chapter will give you enough to keep you going in your adventures with Linux, and that you'll invest in one of the aforementioned Unix books once you have a need to do so. We'll give you enough Unix background to make your terminal usable, keep track of jobs, and enter essential commands.


Chapter 5

Chapter 5, *Essential System Management*, contains material on system administration and maintenance. This is by far the most important chapter for anyone running his own Linux system. If you are completely new to Unix, the material found in Chapter 5 should be easy to follow given the tutorial here.


Chapter 9

One big job we don't cover in this chapter is how to edit files. It's one of the first things you need to learn on any operating system. The two most popular editors for Linux, *vi* and Emacs, are discussed at the beginning of Chapter 9, *Editors, Text Tools, Graphics, and Printing.*

Logging In

Let's assume that your installation went completely smoothly, and you are facing the following prompt on your screen:

```
Linux login:
```

Many Linux users are not so lucky; they have to perform some heavy tinkering when the system is still in a raw state or in single-user mode. But for now, we'll talk about logging into a functioning Linux system.

Logging in, of course, distinguishes one user from another. It lets several people work on the same system at once and makes sure that you are the only person to have access to your files.

You may have installed Linux at home and be thinking right now, "Big deal. No one else shares this system with me, and I'd just as soon not have to log in." But logging in under your personal account also provides a certain degree of protection: your account won't have the ability to destroy or remove important system files. The system administration account (covered in the next chapter) is used for such touchy matters.

If you connect your computer to the Internet, even via a modem, make sure you set non-trivial passwords on all of your accounts. Use punctuation and strings that don't represent real words or names.

You were probably asked to set up a login account for yourself when you installed Linux. If you have such an account, type the name you chose at the `Linux login:` prompt. If you don't have an account yet, type `root` because that account

is certain to exist. Some distributions may also set up an account called **install** or some other name for fooling around when you first install the system.

After you choose your account, you see:

```
Password:
```

and you need to enter the correct password. The terminal turns off the normal echoing of characters you enter for this operation, so that nobody looking at the screen can read your password. If the prompt does not appear, you should add a password to protect yourself from other people's tampering; we'll go into this later.

By the way, both the name and the password are case-sensitive. Make sure the Caps Lock key is not set, because typing `ROOT` instead of `root` will not work.

When you have successfully logged in, you will see a prompt. If you're **root**, this may be a simple:

```
#
```

For other users, the prompt is usually a dollar sign. The prompt may also contain the name you assigned to your system or the directory you're in currently. Whatever appears here, you are now ready to enter commands. We say that you are at the "shell level" here and that the prompt you see is the "shell prompt." This is because you are running a program called the shell that handles your commands. Right now we can ignore the shell, but later in this chapter we'll find that it does a number of useful things for us.

As we show commands in this chapter, we'll show the prompt simply as `$`. So if you see:

```
$ pwd
```

it means that the shell prints `$` and that `pwd` is what you're supposed to enter.

Setting a Password

If you don't already have a password, we recommend you set one. Just enter the command *passwd*. The command will prompt you for a password and then ask you to enter it a second time to make sure you enter it without typos.

There are standard guidelines for choosing passwords so that they're hard for other people to guess. Some systems even check your password and reject any that don't meet the minimal criteria. For instance, it is often said that you should have at least six characters in the password. Furthermore, you should mix uppercase and lowercase characters or include characters other than letters and digits.

To change your password, just enter the *passwd* command again. It prompts you for your old password (to make sure you're you) and then lets you change it.

Virtual Consoles

As a multiprocessing system, Linux gives you a number of interesting ways to do several things at once. You can start a long software installation and then switch to reading mail or compiling a program simultaneously. This should be a major part of Linux's appeal to MS-DOS users (although the latest Microsoft Windows has finally come to grips with multiprocessing, too).

Most Linux users, when they want this asynchronous access, will employ the X Window System. But before you get X running, you can do something similar through virtual consoles. This feature appears on a few other versions of Unix, but is not universally available.

To try out virtual consoles, hold down the left Alt key and press one of the function keys, F1 through F8. As you press each function key, you see a totally new screen complete with a login prompt. You can log in to different virtual consoles just as if you were two different people, and you can switch between them to carry out different activities. You can even run a complete X session in each console. The X Window System will use the virtual console 7 by default. So if you start X and then switch to one of the text-based virtual consoles, you can go back again to X by typing Alt-F7. If you discover that the Alt + function key combination brings up an X menu or some other fuction instead of switching virtual consoles, use Ctrl + Alt + function key.

In earlier versions of Linux (until kernel 1.1.54), the number of available virtual consoles was fixed, but could be changed by patching, recompiling and reinstalling the kernel; the default was 8. Nowadays, the Linux kernel creates virtual consoles as needed on the fly. However, this does not mean that you can simply go to virtual console 13 and log in there. You can log in only on virtual consoles where a getty process is running (see the next chapter for more information on this).

Popular Commands

The number of commands on a typical Unix system is enough to fill a few hundred reference pages. And you can add new commands too. The commands we'll tell you about here are just enough to navigate and to see what you have on the system.

Directories

Like MS-DOS, and virtually every modern computer system, Unix files are organized into a hierarchical directory structure. Unix imposes no rules about where files have to be, but conventions have grown up over the years. Thus, on Linux

you'll find a directory called */home* where each user's files are placed. Each user has a subdirectory under */home*. So if your login name is **mdw**, your personal files are located in */home/mdw*. This is called your home directory. You can, of course, create more subdirectories under it.

As you can see, the components of a directory are separated by slashes. The term *pathname* is often used to refer to this slash-separated list.

What directory is */home* in? The directory named / of course. This is called the root directory. We have already mentioned it when setting up file systems.

When you log in, the system puts you in your home directory. To verify this, use the "print working directory" or *pwd* command:

```
$ pwd
/home/mdw
```

The system confirms that you're in */home/mdw*.

You certainly won't have much fun if you have to stay in one directory all the time. Now try using another command, *cd*, to move to another directory:

```
$ cd /usr/bin
$ pwd
/usr/bin
$ cd
```

Where are we now? A *cd* with no arguments returns us to our home directory. By the way, the home directory is often represented by a tilde (~). So the string *~/programs* means that *programs* is located right under your home directory.

While we're thinking about it, let's make a directory called *~/programs*. From your home directory, you can enter either:

```
$ mkdir programs
```

or the full pathname:

```
$ mkdir /home/mdw/programs
```

Now change to that directory:

```
$ cd programs
$ pwd
/home/mdw/programs
```

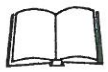
The special character sequence .. refers to "the directory just above the current one." So you can move back up to your home directory by typing the following:

```
$ cd ..
```

The opposite of *mkdir* is *rmdir*, which removes directories:

```
$ rmdir programs
```

Similarly, the *rm* command deletes files. We won't show it here, because we haven't yet shown how to create a file. You generally use the *vi* or Emacs editor for that (see Chapter 9), but some of the commands later in this chapter will create files too. With the *–r* (recursive) option, *rm* deletes a whole directory and all its contents. (Use with care!)


Chapter 9

Listing Files

Enter *ls* to see what is in a directory. Issued without an argument, the *ls* command shows the contents of the current directory. You can include an argument to see a different directory:

```
$ ls /home
```

Some systems have a fancy *ls* that displays special files—such as directories and executable files—in bold, or even in different colors. If you want to change the default colors, edit the file */etc/DIR_COLORS*, or create a copy of it in your home directory named *.dir_colors* and edit that.

Like most Unix commands, *ls* can be controlled with options that start with a hyphen (-). Make sure you type a space before the hyphen. One useful option for *ls* is *–a* for "all," which will reveal to you riches that you never imagined in your home directory:

```
$ cd
$ ls -a
.                   .bashrc             .fvwmrc
..                  .emacs              .xinitrc
.bash_history       .exrc
```

The single dot refers to the current directory, and the double dot refers to the directory right above it. But what are those other files beginning with a dot? They are called hidden files. Putting a dot in front of their names keeps them from being shown during a normal *ls* command. Many programs employ hidden files for user options—things about their default behavior that you want to change. For instance, you can put commands in the file *.Xdefaults* to alter how programs using the X Window System operate. So most of the time you can forget these files exist, but when you're configuring your system you'll find them very important. We'll list some of them later.

Another useful *ls* option is *–l* for "long." It shows extra information about the files. Figure 4-1 shows typical output and what each field means.

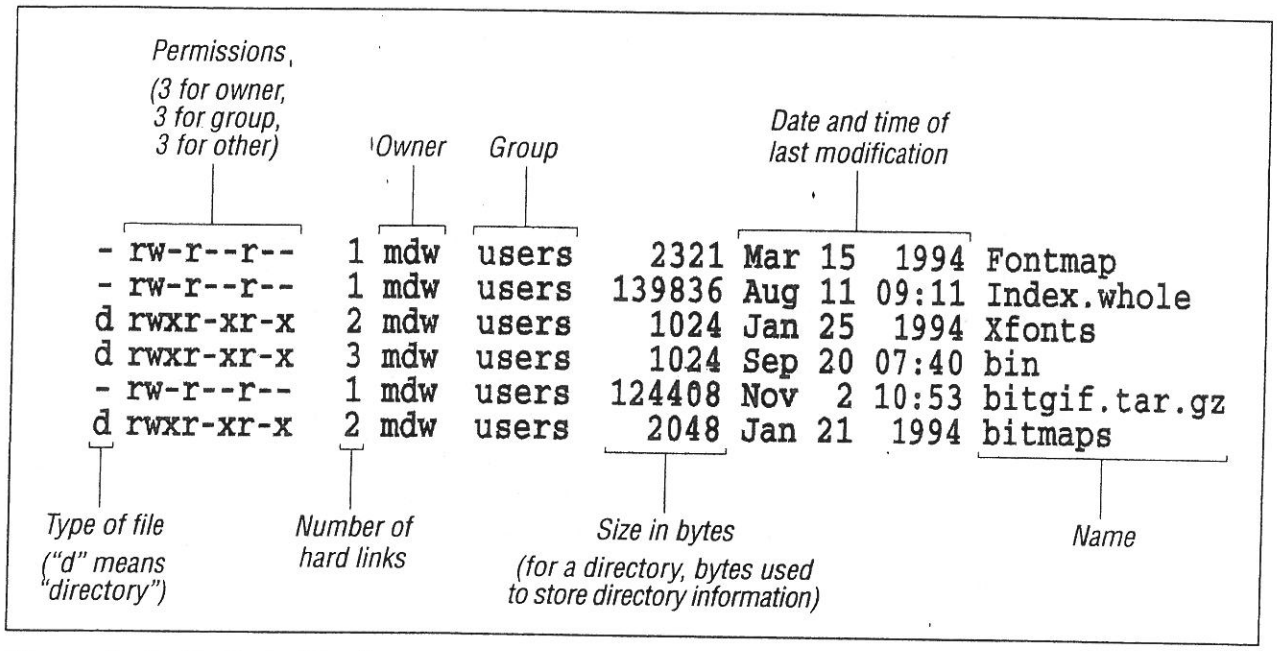


Figure 4–1: Output of ls –l

We'll discuss the permissions, owner, and group fields later in this chapter, in the section "File Ownership and Permissions." The *ls* command also shows the size of each file and when it was last modified.

Viewing Files, More or Less

One way to look at a file is to invoke an editor, such as:

```
$ emacs .bashrc
```

But if you just want to scan a file quickly, rather than edit it, other commands are quicker. The simplest is the strangely named *cat* command (named after the verb *concatenate*, because you can also use it to concatenate several files into one):

```
$ cat .bashrc
```

But a long file will scroll by too fast for you to see it, so most people use the *more* command instead:

```
$ more .bashrc
```

This prints a screenfull at a time and waits for you to press the space bar before printing more. *more* has a lot of powerful options. For instance, you can search for a string in the file: press the slash key (/), type the string, and press Return.

A popular variation on the *more* command is called *less*. It has even more powerful features; for instance, you can mark a particular place in a file and return there later.

Symbolic Links

Sometimes you want to keep a file in one place and pretend it is in another. This is done most often by a system administrator, not a user. For instance, you might keep several versions of a program around, called *prog.0.9*, *prog.1.1*, and so on, but use the name *prog* to refer to the version you're currently using. Or you may have a file installed in one partition because you have disk space for it there, but the program that uses the file needs it to be in a different partition because the pathname is hard-coded into the program.

Unix provides *links* to handle these situations. In this section, we'll examine the *symbolic link*, which is the most flexible and popular type. A symbolic link is a kind of dummy file that just points to another file. If you edit or read or execute the symbolic link, the system is smart enough to give you the real file instead. Symbolic links work a lot like shortcuts under Windows 95/98, but are much more powerful.

Let's take the *prog* example. You want to create a link named *prog* that points to the actual file, which is named *prog.1.1*. Enter the command:

```
$ ln -s prog.1.1 prog
```

Now you've created a new file named *prog* that is kind of a dummy file; if you run it, you're really running *prog.1.1*. Let's look at what *ls -l* has to say about the file:

```
$ ls -l prog
lrwxrwxrwx   2 mdw      users            8 Nov 17 14:35 prog -> prog.1.1
```

The `l` at the beginning of the line shows that the file is a link, and the little `->` indicates the real file the link points to.

Symbolic links are really simple, once you get used to the idea of one file pointing to another. You'll encounter links all the time when installing software packages.

Shells

As we said before, logging in to the system puts you into a shell (or a graphical interface if your system is configured to use a display manager). So does opening an *xterm* window in X. The shell interprets and executes all your commands. Let's look a bit at different shells before we keep going, because they're going to affect some of the material coming up.

If it seems confusing that Unix offers many different shells, just accept it as an effect of evolution. Believe us, you wouldn't want to be stuck using the very first shell developed for Unix, the Bourne shell. While it was a very powerful user interface for its day (the mid-1970s), it lacked a lot of useful features for interactive use—including the ones shown in this section. So other shells have been developed over time, and you can now choose the one that best suits your way of working.

Some of the shells available on Linux are:

bash
: Bourne Again shell. The most commonly used (and most powerful) shell on Linux. POSIX-compliant, compatible with Bourne shell, created and distributed by the GNU project (Free Software Foundation). Offers command-line editing, history substitution, and Bourne Shell compatibility.

csh
: C shell. Developed at Berkeley. Mostly compatible with the Bourne shell for interactive use, but has a very different interface for programming. Does not offer command-line editing, although it does have a sophisticated alternative called history substitution.

ksh
: Korn shell. Perhaps the most popular on Unix systems generally, and the first to introduce modern shell techniques (including some borrowed from the C shell) into the Bourne shell. Compatible with Bourne shell. Offers command-line editing.

sh Bourne shell. The original shell. Does not offer command-line editing.

tcsh
: Enhanced C shell. Offers command-line editing.

zsh
: Z shell. The newest of the shells. Compatible with Bourne shell. Offers command-line editing.

Try the following command to find out what your shell is. It prints out the full pathname where the shell is located. Don't forget to type the dollar sign:

```
$ echo $SHELL
```

You are probably running *bash*, the Bourne Again Shell. If you're running something else, this might be a good time to change to *bash*. It's powerful, POSIX-compliant, well-supported, and very popular on Linux. Use the *chsh* command to change your shell:

```
$ chsh
Enter password: Type your password here—this is for security's sake
Changing the login shell for kalle
Enter the new value, or press return for the default

        Login Shell [/bin/sh]:/bin/bash
```

Before a user can choose a particular shell, that shell must be installed and the system administrator must make it available by entering it in */etc/shells*.

There are a couple of ways to conceptualize the differences between shells. One is to distinguish Bourne-compatible shells from *csh*-compatible shells. This will be of interest to you when you start to program with the shell, also known as writing shell scripts. The Bourne shell and C shell have different programming constructs. Most people now agree that Bourne-compatible shells are better, and there are many Unix utilities that recognize only the Bourne shell.

Another way to categorize shells is to identify those that offer command-line editing (all the newer ones) versus those that do not. *sh* and *csh* lack this useful feature.

When you combine the two criteria—being compatible with the Bourne shell and offering command-line editing—your best choice comes down to *bash*, *ksh*, or *zsh*. Try out several shells before you make your choice; it helps to know more than one, in case someday you find yourself on a system that limits your choice of shells.

Useful Keys and How to Get Them to Work

When you type a command, pressing the Backspace key should remove the last character. Ctrl-U should delete the whole line.* When you have finished entering a command, and it is executing, Ctrl-C should abort it, and Ctrl-Z should suspend it. (When you want to resume the program, enter *fg* for "foreground.")

If any of these keys fail to work, your terminal is not configured correctly for some reason. You can fix it through the *stty* command. Use the syntax:

```
stty function key
```

where *function* is what you want to do, and *key* is the key that you press. Specify a control key by putting a circumflex (^) in front of the key.

Here is a set of sample commands to set up the functions described earlier:

```
$ stty erase ^H
$ stty kill ^U
$ stty intr ^C
$ stty susp ^Z
```

The first control key shown, ^H, represents the ASCII code generated by the Backspace key.

By the way, you can generate a listing of your current terminal settings by entering *stty –a*. But that doesn't mean you can understand the output: *stty* is a complicated command with many uses, some of which require a lot of knowledge about terminals.

---

* Ctrl-U means hold down the Control key and press u.

Typing Shortcuts

If you've been following along this tutorial at the terminal, you may be tired of typing the same things over and over again. It can be particularly annoying when you make a mistake and have to start over again. Here is where the shell really makes life easier. It doesn't make Unix as simple as a point-and-click interface, but it can help you work really fast in a command environment.

This section discusses command-line editing. The tips here work if your shell is *bash*, *ksh*, *tcsh*, or *zsh*. Command-line editing treats the last fifty or so lines you typed as a buffer in an editor. You can move around these lines and change them the way you'd edit a document. Every time you press the Return key, the shell executes the current line.

Word Completion

First, let's try something simple that can save you a lot of time. Type the following, without pressing the Return key:

```
$ cd /usr/inc
```

Now press the Tab key. The shell will add `lude` to complete the name of the directory */usr/include*. Now you can press the Return key, and the command will execute.

The criteria for specifying a filename is "minimal completion." Type just enough characters to distinguish a name from all the others in that directory. The shell can find the name and complete it—up to and including a slash, if the name is a directory.

You can use completion on commands too. For instance, if you type:

```
$ ema
```

and press the Tab key, the shell will add the `cs` to make `emacs` (unless some other command in your path begins with `ema`).

What if there are multiple files that match what you've typed? If they all start with the same characters, the shell completes the word up to the point where names differ. Beyond that, most shells do nothing. *bash* has a neat enhancement: if you press the Tab key twice, it displays all the possible completions. For instance, if you enter:

```
$ cd /usr/l
```

and press the Tab key twice, *bash* prints something like:

```
lib         local
```

Moving Around Among Commands

Press the up arrow, and the command you typed previously appears. The up arrow takes you back through the command history, while the down arrow takes you forward. If you want to change a character on the current line, use the left or right arrow keys.

As an example, suppose you tried to execute:

```
$ mroe .bashrc
bash: mroe: command not found
```

Of course, you typed `mroe` instead of `more`. To correct the command, call it back by pressing the up arrow. Then press the left arrow until the cursor lies over the `o` in `mroe`. You could use the Backspace key to remove the `o` and `r` and retype them correctly. But here's an even neater shortcut: just press Ctrl-T. It will reverse `o` and `r`, and you can then press the Return key to execute the command.

Many other key combinations exist for command-line editing. But the basics shown here will help you quite a bit. If you learn the Emacs editor, you will find that most keys work the same way in the shell. And if you're a *vi* fan, you can set up your shell so that it uses *vi* key bindings instead of Emacs bindings. To do this in *bash*, *ksh*, or *zsh*, enter the command:

```
$ export VISUAL=vi
```

In *tcsh* enter:

```
$ setenv VISUAL vi
```

Filename Expansion

Another way to save time in your commands is to use special characters to abbreviate filenames. You can specify many files at once by using these characters. This feature of the shell is sometimes called "globbing."

MS-DOS provides a few crude features of this type. You can use a question mark to mean "any character" and an asterisk to mean "any string of characters." Unix provides these wildcards too, but in a more robust and rigorous way.

Let's say you have a directory containing the following C source files:

```
$ ls
inv1jig.c   inv2jig.c   inv3jig.c   invinitjig.c   invpar.c
```

To list the three files containing digits in their names, you could enter:

```
$ ls inv?jig.c
inv1jig.c   inv2jig.c   inv3jig.c
```

The shell looks for a single character to replace the question mark. Thus, it displays *inv1jig.c*, *inv2jig.c*, and *inv3jig.c*, but not *invinitjig.c* because that name contains too many characters.

If you're not interested in the second file, you can specify the ones you want using brackets:

```
$ ls inv[13]jig.c
inv1jig.c   inv3jig.c
```

If any single character within the brackets matches a file, that file is displayed. You can also put a range of characters in the brackets:

```
$ ls inv[1-3]jig.c
inv1jig.c   inv2jig.c   inv3jig.c
```

Now we're back to displaying all three files. The hyphen means "match any character from 1 through 3, inclusive." You could ask for any numeric character by specifying `0-9`, and any alphabetic character by specifying `[a-zA-Z]`. In the latter case, two ranges are required because the shell is case-sensitive. The order used, by the way, is that of the ASCII character set.

Suppose you want to see the init file, too. Now you can use an asterisk, because you want to match any number of characters between the `inv` and the `jig`:

```
$ ls inv*jig.c
inv1jig.c   inv2jig.c   inv3jig.c   invinitjig.c
```

The asterisk actually means "zero or more characters," so if a file named *invjig.c* existed, it would be shown, too.

Unlike MS-DOS, the Unix shells let you combine special characters and normal characters any way you want. Let's say you want to look for any source (*.c*) or object (*.o*) file that contains a digit. The resulting pattern combines all the expansions we've studied in this section:

```
$ ls *[0-9]*.[co]
```

Filename expansion is very useful in shell scripts (programs), where you don't always know exactly how many files exist. For instance, you might want to process multiple log files named *log001*, *log002*, and so on. No matter how many there are, the expression *log** will match them all.

One final warning: filename expansions are not the same as regular expressions, which are used by many utilities to specify groups of strings. Regular expressions are beyond the scope of this book, but are described by many books that explain Unix utilities. A taste of regular expressions appears in Chapter 13, *Programming Languages*.

Chapter 13

Saving Your Output

System administrators (and other human beings too) see a lot of critical messages fly by on the computer screen. It's often important to save these messages so you can scrutinize them later, or (all too often) send them to a friend who can figure out what went wrong. So, in this section, we'll explain a little bit about redirection, a powerful feature provided by Unix shells. If you come from MS-DOS, you have probably seen a similar, but more limited, type of redirection.

If you put a greater-than sign (>) and a filename after any command, the output of the command will be sent to that file. For instance, to capture the output of *ls*, you can enter:

```
$ ls /usr/bin > ~/Binaries
```

A listing of */usr/bin* will be stored in your home directory in a file named *Binaries*. If *Binaries* had already existed, the > would wipe out what was there and replace it with the output of the *ls* command. Overwriting a current file is a common user error. If your shell is *csh* or *tcsh*, you can prevent overwriting with the command:

```
$ set noclobber
```

And in *bash* you can achieve the same effect by entering:

```
$ noclobber=1          It doesn't have to be 1; any value will have the same effect.
```

Another (and perhaps more useful) way to prevent overwriting is to append new output. For instance, having saved a listing of */usr/bin*, suppose we now want to add the contents of */bin* to that file. We can append it to the end of the *Binaries* file by specifying two greater-than signs:

```
$ ls /bin >> ~/Binaries
```

You will find the technique of output redirection very useful when you are running a utility many times and saving the output for troubleshooting.

Most Unix programs have two output streams. One is called the standard output, and the other is the standard error. If you're a C programmer you'll recognize these: the standard error is the file named *stderr* to which you print messages.

The > character does not redirect the standard error. It's useful when you want to save legitimate output without mucking up a file with error messages. But what if the error messages are what you want to save? This is quite common during troubleshooting. The solution is to use a greater-than sign followed by an ampersand. (This construct works in every shell except the original Bourne shell.) It redirects both the standard output and the standard error. For instance:

```
$ gcc invinitjig.c >& error-msg
```

This command saves all the messages from the *gcc* compiler in a file named *error-msg*. (Of course, the object code is not saved there. It's stored in *invinitjig.o* as always.) On the Bourne shell and *bash* you can also say it slightly differently:

```
$ gcc invinitjig.c &> error-msg
```

Now let's get really fancy. Suppose you want to save the error messages but not the regular output—the standard error but not the standard output. In the Bourne-compatible shells you can do this by entering the following:

```
$ gcc invinitjig.c 2> error-msg
```

The shell arbitrarily assigns the number 1 to the standard output and the number 2 to the standard error. So the above command saves only the standard error.

Finally, suppose you want to throw away the standard output—keep it from appearing on your screen. The solution is to redirect it to a special file called */dev/null*. (You've heard people say things like "Send your criticisms to /dev/null"? Well, this is where the phrase came from.) The */dev* directory is where Unix systems store special files that refer to terminals, tape drives, and other devices. But */dev/null* is unique; it's a place you can send things so that they disappear into a black hole. For example, the following command saves the standard error and throws away the standard output:

```
$ gcc invinitjig.c 2>error-msg >/dev/null
```

So now you should be able to isolate exactly the output you want.

In case you've wondered whether the less-than sign (<) means anything to the shell: yes, it does. It causes commands to take their input from a file. But most commands allow you to specify input files on their command lines anyway, so this "input redirection" is rarely necessary.

Sometimes you want one utility to operate on the output of another utility. For instance, you can use the *sort* command to put the output of other commands into a more useful order. A crude way to do this would be to save output from one command in a file, and then run *sort* on it. For instance:

```
$ du > du_output
$ sort -n du_output
```

Unix provides a much more succinct and efficient way to do this using a *pipe*. Just place a vertical bar between the first command and the second:

```
$ du | sort -n
```

The shell sends all the input from the *du* program to the *sort* program.

In the previous example, *du* stands for "disk usage" and shows how many blocks each file occupies under the current directory. Normally, its output is in a somewhat random order:

```
$ du
10          ./zoneinfo/Australia
13          ./zoneinfo/US
9           ./zoneinfo/Canada
4           ./zoneinfo/Mexico
5           ./zoneinfo/Brazil
3           ./zoneinfo/Chile
20          ./zoneinfo/SystemV
118         ./zoneinfo
298         ./ghostscript/doc
183         ./ghostscript/examples
3289        ./ghostscript/fonts
          .
          .
          .
```

So we have decided to run it through *sort* with the *–n* and *–r* options. The *–n* option means "sort in numerical order" instead of the default ASCII sort, and the *–r* option means "reverse the usual order" so that the highest number appears first. The result is output that quickly shows you which directories and files hog the most space:

```
$ du | sort -rn
34368       .
16005       ./emacs
16003       ./emacs/20.4
13326       ./emacs/20.4/lisp
4039        ./ghostscript
3289        ./ghostscript/fonts
          .
          .
          .
```

Since there are so many files, we had better use a second pipe to send output through the *more* command (one of the more common uses of pipes):

```
$ du | sort -rn | more
34368       .
16005       ./emacs
16003       ./emacs/20.4
13326       ./emacs/20.4/lisp
4039        ./ghostscript
3289        ./ghostscript/fonts
          .
          .
          .
```

What Is a Command?

We've said that Unix offers a huge number of commands and that you can add new ones. This makes it radically different from most operating systems, which contain a strictly limited table of commands. So what are Unix commands, and how are they stored? On Unix, a command is simply a file. For instance, the *ls* command is a binary file located in the directory *bin*. So, instead of *ls*, you could enter the full pathname, also known as the *absolute pathname*:

```
$ /bin/ls
```

This makes Unix very flexible and powerful. To provide a new utility, a system administrator can simply install it in a standard directory where commands are located. There can also be different versions of a command—for instance, you can offer a new version of a utility for testing in one place while leaving the old version in another place, and users can choose the one they want.

Here's a common problem: sometimes you enter a command that you expect to be on the system, but you receive a message such as "Not found." The problem may be that the command is located in a directory that your shell is not searching. The list of directories where your shell looks for commands is called your path. Enter the following to see what your path is (remember the dollar sign!):

```
$ echo $PATH
/usr/local/bin:/usr/bin:/usr/X11R6/bin:/bin:/usr/lib/java/bin:\
/usr/games:/usr/bin/TeX:.
```

This takes a little careful eyeballing. The output is a series of pathnames separated by colons. The first pathname, for this particular user, is */usr/local/bin*. The second is */usr/bin*, and so on. So if two versions of a command exist, one in */usr/local/bin* and the other in */usr/bin*, the one in */usr/local/bin* will execute. The last pathname in this example is simply a dot; it refers to the current directory. Unlike MS-DOS, Unix does not look automatically in your current directory. You have to tell it to explicitly, as shown here. Some people think it's a bad idea to look in the current directory, for security reasons. (An intruder who gets into your account might copy a malicious program to one of your working directories.) However, this mostly applies to root, normal users generally do not need to worry about this.

If a command is not found, you have to figure out where it is on the system and add that directory to your path. The manual page should tell you where it is. Let's say you find it in */usr/sbin*, where a number of system administration commands are installed. You realize you need access to these system administration commands, so you enter the following (note that the first `PATH` doesn't have a dollar sign, but the second one does):

```
$ export PATH=$PATH:/usr/sbin
```

This command adds */usr/sbin*, but makes it the last directory that is searched. The command is saying, "Make my path equal to the old path plus */usr/sbin*."

The previous command works for some shells but not others. It's fine for most Linux users, who are working in a Bourne-compatible shell like *bash*. But if you use *csh* or *tcsh* you need to issue the following command instead:

```
set path = ( $PATH /usr/sbin )
```

Finally, there are a few commands that are not files; *cd* is one. Most of these commands affect the shell itself, and therefore have to be understood and executed by the shell. Because they are part of the shell, they are called built-in commands.

Putting a Command in the Background

Before the X Window System, which made it easy to run multiple programs at once, Unix users took advantage of Unix's multitasking features by simply putting an ampersand at the end of commands, as shown in this example:

```
$ gcc invinitjig.c &
[1] 21457
```

The ampersand puts the command into the background, meaning that the shell prompt comes back, and you can continue to execute other commands while the *gcc* command is compiling your program. The `[1]` is a job number that is assigned to your command. The `21457` is a process ID, which we'll discuss later. Job numbers are assigned to background commands in order and therefore are easier to remember and type than process IDs.

Of course, multitasking does not come for free. The more commands you put into the background, the slower your system runs as it tries to interleave their execution.

You wouldn't want to put a command in the background if it requires user input. If you do so, you see an error message like:

```
Stopped (tty input)
```

You can solve this problem by bringing the job back into the foreground through the *fg* command. If you have many commands in the background, you can choose one of them by its job number or its process ID. For our long-lived *gcc* command, the following commands are equivalent:

```
$ fg %1
$ fg 21457
```

Don't forget the percent sign on the job number; that's what distinguishes job numbers from process IDs.

To get rid of a command in the background, issue a *kill* command:

```
$ kill %1
```

Manual Pages

The most empowering information you can get is how to conduct your own research. Following this precept, we'll now tell you about the online help system that comes built in to Unix systems. It is called manual pages, or man pages for short.

Actually, manual pages are not quite the boon they ought to be. This is because they are short and take a lot of Unix background for granted. Each one focuses on a particular command and rarely helps you decide why you should use that command. Still, they are critical. Commands can vary slightly on different Unix systems, and the manual pages are the most reliable way to find out what your system does. The Linux Documentation Project deserves a lot of credit for the incredible number of hours they have put into creating manual pages. To find out about a command, enter a command like:

```
$ man ls
```

Manual pages are divided into different sections depending on what they are for. User commands are in section 1, Unix system calls in section 2, and so on. The sections that will interest you most are 1, 5 (file formats), and 8 (system administration commands). When you view man pages online, the section numbers are conceptual; you can optionally specify them when searching for a command:

```
$ man 1 ls
```

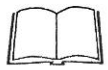
But if you consult a hard-copy manual, you'll find it divided into actual sections according to the numbering scheme. Sometimes an entry in two different sections can have the same name. (For instance, *chmod* is both a command and a system call.) So you will sometimes see the name of a manual page followed by the section number in parentheses, as in *ls(1)*.

There is one situation in which you will need the section number on the command line: when there are several manual pages for the same keyword (e.g., one for a command with that name and one for a system function with the same name). Suppose you want to look up a library call, but the *man* shows you the command because its default search order looks for the command first. In order to see the manual page for the library call, you need to give its section number.

Look near the top of a manual page. The first heading is NAME. Under it is a brief one-line description of the item. These descriptions can be valuable if you're not quite sure what you're looking for. Think of a word related to what you want, and specify it in an *apropos* command:

```
$ apropos edit
```

The previous command shows all the manual pages that have something to do with editing. It's a very simple algorithm: *apropos* simply prints out all the NAME lines that contain the string you ask for.


Chapter 11

An X Window System application, *xman*, also helps you browse manual pages. It is described in the section "xman: A Point-and-Click Interface to Manual Pages" in Chapter 11, *Customizing Your X Environment.*

Like commands, manual pages are sometimes installed in strange places. For instance, you may install some site-specific programs in the directory */usr/local*, and put their manual pages in */usr/local/man*. The *man* command will not automatically look in */usr/local/man*, so when you ask for a manual page you may get the message "No manual entry." Fix this by specifying all the top *man* directories in a variable called MANPATH. For example (you have to put in the actual directories where the manual pages are on your system):

```
$ export MANPATH=/usr/man:/usr/local/man
```

The syntax is like PATH, described earlier in this chapter. Each two directories are separated by a colon. If your shell is *csh* or *tcsh*, you need to say:

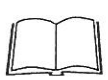
```
$ setenv MANPATH /usr/man:/usr/local/man
```

Have you read some manual pages and still found yourself confused? They're not meant to be introductions to new topics. Get yourself a good beginner's book about Unix, and come back to manual pages gradually as you become more comfortable on the system; then they'll be irreplaceable.

Manual pages are not the only source of information on Unix systems. Programs from the GNU project often have Info pages that you read with the program *info*. For example, to read the Info pages for the command *find*, you would enter:

```
info find
```

The *info* program is arcane and has lots of navigation features; to learn it, your best bet will be to type Ctrl-H in the *info* program and read through the help screen. Fortunately, there are also programs that let you read Info pages more easily, notably *tkinfo* and *kdehelp*. These commands use the X Window System to present a graphical interface. You can also read Info pages from Emacs (see the section "The Emacs Editor" in Chapter 9) or use the command *pinfo* available on some Linux distributions that works more like the Lynx web browser.


Chapter 16

In recent times, more and more documentation is provided in the form of HTML pages. You can read those with any web browser (see Chapter 16, *The World Wide Web and Electronic Mail*). For example, in Netscape Navigator, you select "Open Page . . . " from the "File" menu and press the "Choose File" button, which opens an ordinary file selection dialog where you can select your documentation file.

File Ownership and Permissions

Ownership and permissions are central to security. It's important to get them right, even when you're the only user, because odd things can happen if you don't. For the files that users create and use daily, these things usually work without much thought (although it's still useful to know the concepts). For system administration, matters are not so easy. Assign the wrong ownership or permission, and you might get into a frustrating bind like not being able to read your mail. In general, the message:

```
Permission denied
```

means that someone has assigned an ownership or permission that restricts access more than you want.

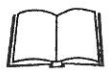
What Permissions Mean

Permissions refer to the ways in which someone can use a file. There are three such permissions under Unix:

- *Read* permission means you can look at the file's contents.
- *Write* permission means you can change or delete the file.
- *Execute* permission means you can run the file as a program.

When each file is created, the system assigns some default permissions that work most of the time. For instance, it gives you both read and write permission, but most of the world has only read permission. If you have a reason to be paranoid, you can set things up so that other people have no permissions at all.

Additionally, most utilities know how to assign permissions. For instance, when the compiler creates an executable program, it automatically assigns executable permission. When you check a file out of the revision control system (RCS) without locking it, you get only read permission (because you're not expected to change the file), but if you lock the file, you get read and write permission (you're expected to edit it and check it back in). We'll discuss RCS in the section "Revision Control Tools—RCS" in Chapter 14, *Tools for Programmers.*


Chapter 14

There are times when defaults don't work, though. For instance, if you create a shell script or Perl program, you'll have to assign executable permission yourself so that you can run it. We'll show how to do that later in this section, after we get through the basic concepts.

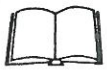
Permissions have different meanings for a directory:

- Read permission means you can list the contents of that directory.

- Write permission means you can add or remove files in that directory.
- Execute permission means you can list information about the files in that directory.

Don't worry about the difference between read and execute permission for directories; basically, they go together. Assign both, or neither.

Note that, if you allow people to add files to a directory, you are also letting them remove files. The two privileges go together when you assign write permission. However, there is a way you can let users share a directory and keep them from deleting each other's files. See the section "Upgrading Other Software" in Chapter 7, *Upgrading Software and the Kernel.*


Chapter 7

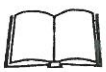
There are more files on Unix systems than the plain files and directories we've talked about so far. These are special files (devices), sockets, symbolic links, and so forth—each type observing its own rules regarding permissions. But you don't need to know the details on each type.

Owners and Groups

Now, who gets these permissions? To allow people to work together, Unix has three levels of permission: owner, group, and other. The "other" covers everybody who has access to the system and who isn't the owner or a member of the group.

The idea behind having groups is to give a set of users, like a team of programmers, access to a file. For instance, a programmer creating source code may reserve write permission to herself, but allow members of her group to have read access through a group permission. As for "other," it might have no permission at all. (You think your source code is *that* good?)

Each file has an owner and a group. The owner is generally the user who created the file. Each user also belongs to a default group, and that group is assigned to every file the user creates. You can create many groups, though, and assign each user to multiple groups. By changing the group assigned to a file, you can give access to any collection of people you want. We'll discuss groups more when we get to the section "The Group File" in Chapter 5.


Chapter 5

Now we have all the elements of our security system: three permissions (read, write, execute) and three levels (user, group, other). Let's looks at some typical files and see what permissions are assigned.

Figure 4-2 shows a typical executable program. We generated this output by executing *ls* with the *–l* option.

Two useful facts stand right out: the owner of the file is an author of this book and your faithful guide, **mdw**, while the group is **lib** (perhaps a group created for programmers working on libraries). But the key information about permissions is encrypted in the set of letters on the left side of the display.

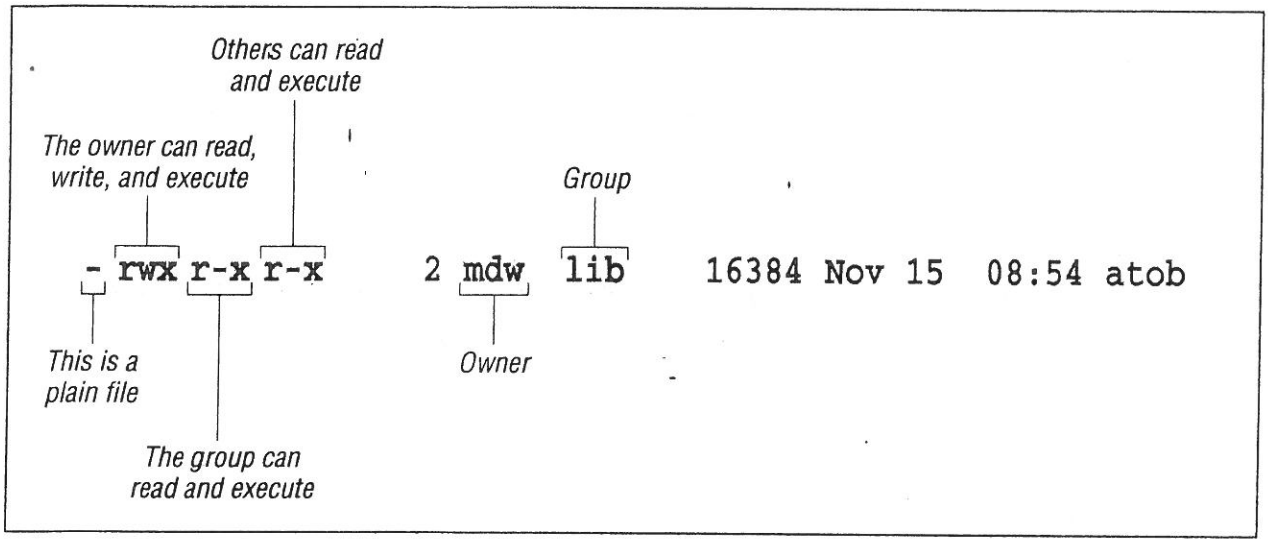


Figure 4–2: Displaying ownership and permissions

The first character is a hyphen, indicating a plain file. The next three bits apply to the owner; as we would expect, **mdw** has all three permissions. The next three bits apply to members of the group: they can read the file (not too useful for a binary file) and execute it, but they can't write to it because the field that should contain a `w` contains a hyphen instead. And the last three bits apply to "other"; they have the same permissions in this case as the group.

As another exercise, here is a file checked out of RCS for editing:

```
-rw-r--r--   2 mdw      lib           878 Aug  7 19:28 tools.tex
```

The only difference between this file and that shown in Figure 4-2 is that the `x` bits in this case have been replaced by hyphens. No one needs to have execute permission because the file is not meant to be executed; it's just text.

One more example—a typical directory:

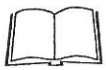
```
drwxr-xr-x   2 mdw      lib           512 Jul 17 18:23 perl
```

The left-most bit is now a `d`, to show that this is a directory. The executable bits are back, because you want people to see the contents of the directory.

Files can be in some obscure states that aren't covered here; see the *ls* manual page for gory details. But now it's time to see how you can change ownership and permissions.

Changing the Owner, Group, and Permissions

As we said, most of the time you can get by with the default security the system gives you. But there are always exceptions, particularly for system administrators.


Chapter 5

To take a simple example, suppose you are creating a directory under */home* for a new user. You have to create everything as **root**, but when you're done you have to change the ownership to the user; otherwise, that user won't be able to use the files! (Fortunately, if you use the *adduser* command discussed in the section "Creating Accounts" in Chapter 5, it takes care of ownership for you.)

Similarly, there are certain utilities such as UUCP and News that have their own users. No one ever logs in as **UUCP** or **News**, but those users and groups must exist so that the utilities can do their job in a secure manner. In general, the last step when installing software is usually to change the owner, group, and permissions as the documentation tells you to do.

The *chown* command changes the owner of a file, and the *chgrp* command changes the group. On Linux, only **root** can use *chown* for changing ownership of a file, but any user can change the group to another group he belongs to.

So after installing some software named *sampsoft*, you might change both the owner and the group to **bin** by executing:

```
# chown bin sampsoft
# chgrp bin sampsoft
```

You could also do this in one step by using the dot notation:

```
# chown bin.bin sampsoft
```

The syntax for changing permissions is more complicated. The permissions can also be called the file's "mode," and the command that changes permissions is *chmod*. Let's start our exploration of this command through a simple example; say you've written a neat program in Perl or Tcl named *header*, and you want to be able to execute it. You would type the following command:

```
$ chmod +x header
```

The plus sign means "add a permission," and the **x** indicates which permission to add.

If you want to remove execute permission, use a minus sign in place of a plus:

```
$ chmod -x header
```

This command assigns permissions to all levels—user, group, and other. Let's say that you are secretly into software hoarding and don't want anybody to use the command but yourself. (No, that's too cruel; let's say instead that you think the script is buggy and want to protect other people from hurting themselves until you've exercised it.) You can assign execute permission just to yourself through the command:

```
$ chmod u+x header
```

Whatever goes before the plus sign is the level of permission, and whatever goes after is the type of permission. User permission (for yourself) is u, group

permission is g, and other is o. So, to assign permission to both yourself and the file's group, enter:

```
$ chmod ug+x header
```

You can also assign multiple types of permissions:

```
$ chmod ug+rwx header
```

There are a few more shortcuts you can learn from the *chmod* manual page in order to impress someone looking over your shoulder, but they don't offer any functionality besides what we've shown you.

As arcane as the syntax of the mode argument may seem, there's another syntax that is even more complicated. We have to describe it though, for several reasons. First of all, there are several situations that cannot be covered by the syntax, called *symbolic mode*, that we've just shown. Second, people often use the other syntax, called *absolute mode,* in their documentation. Third, there are times you may actually find the absolute mode more convenient.

To understand absolute mode, you have to think in terms of bits and octal notation. Don't worry, it's not too hard. A typical mode contains three characters, corresponding to the three levels of permission (user, group, and other). These levels are illustrated in Figure 4-3. Within each level, there are three bits corresponding to read, write, and execute permission.

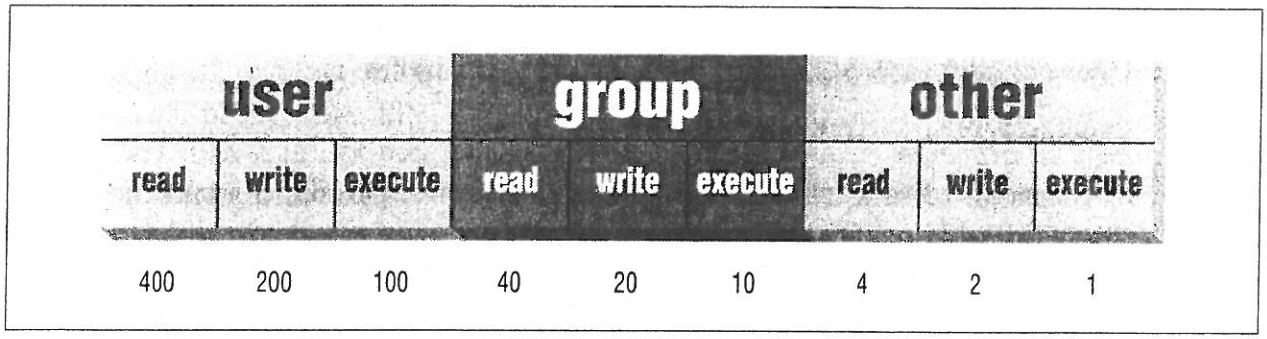


Figure 4-3: Bits in absolute mode

Let's say you want to give yourself read permission and no permission to anybody else. You want to specify just the bit represented by the number 400. So the *chmod* command would be:

```
$ chmod 400 header
```

To give read permission to everybody, choose the correct bit from each level: 400 for yourself, 40 for your group, and 4 for other. The full command is:

```
$ chmod 444 header
```

This is like using a mode `+r`, except that it simultaneously removes any write or execute permission. (To be precise, it's just like a mode of `=r`, which we didn't mention earlier. The equal sign means "assign these rights and no others.")

To give read and execute permission to everybody, you have to add up the read and execute bits. 400 plus 100 is 500, for instance.

So the corresponding command is:

```
$ chmod 555 header
```

which is the same as `=rx`. To give someone full access, you would specify that digit as a 7—the sum of 4, 2, and 1.

One final trick: how to set the default mode that is assigned to each file you create (with a text editor, the > redirection operator, and so on). You do so by executing a *umask* command, or putting one in your shell's start-up file. This file could be called *.bashrc*, *.cshrc*, or something else depending on the shell you use (we'll discuss startup files in the next section).

The *umask* command takes an argument like the absolute mode in *chmod*, but the meaning of the bits is inverted. You have to determine the access you want to grant for user, group, and other, and subtract each digit from 7. That gives you a three-digit mask.

For instance, say you want yourself to have all permissions (7), your group to have read and execute permissions (5), and others to have no permissions (0). Subtract each bit from 7 and you get 0 for yourself, 2 for your group, and 7 for other. So the command to put in your start-up file is:

```
umask 027
```

A strange technique, but it works. The *chmod* command looks at the mask when it interprets your mode; for instance, if you assign execute mode to a file at creation time, it will assign execute permission for you and your group, but will exclude others because the mask doesn't permit them to have any access.

Startup Files

Configuration is a strong element of Unix. This probably stems from two traits commonly found in hackers: they want total control over their environment, and they strive to minimize the number of keystrokes and other hand movements they have to perform. So all the major utilities on Unix—editors, mailers, debuggers, X Window System clients—provide files that let you override their default behaviors in a bewildering number of ways. Many of these files have names ending in *rc* which means *resource configuration*.

Startup files are usually in your home directory. Their names begin with a period, which keeps the *ls* command from displaying them under normal circumstances. None of the files are required; all the affected programs are smart enough to use defaults when the file does not exist. But everyone finds it useful to have the startup files. Here are some common ones:

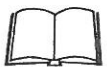
.bashrc

For the *bash* shell. The file is a shell script, which means it can contain commands and other programming constructs. Here's a very short startup file that might have been placed in your home directory by the tool that created your account:

```
PS1='\u$'       # The prompt contains the user's login name.

HISTSIZE=50     # Save 50 commands for when the user presses the up arrow.

# All the directories to search for commands.
PATH=/usr/local/bin:/usr/bin:/bin:/usr/bin/X11

# To prevent the user from accidentally ending a login session,
# disable Ctrl-D as a way to exit.
IGNOREEOF=1

stty erase "^H"         # Make sure the backspace key erases.
```

.bash_profile

For the *bash* shell. Another shell script. The difference between this script and *.bashrc* is that *.bash_profile* runs only when you log in. It was originally designed so you could separate interactive shells from those run by background processors like *cron* (discussed in Chapter 8, *Other Administrative Tasks*). But it is not too useful on modern computers with the X Window System, because when you open a new *xterm* window, only *.bashrc* runs. If you start up a window with the command *xterm –ls*, it will run *.bash_profile*, too.


Chapter 8

.cshrc

For the C shell or *tcsh*. The file is a shell script using C shell constructs.

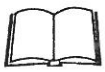
.login

For the C shell or *tcsh*. The file is a shell script, using C shell constructs. Like *.bash_profile* in the *bash* shell, this runs only when you log in. Here are some commands you might find in *.cshrc* or *.login*:

```
set prompt='% '         # Simple % for prompt.

set history=50 # Save 50 commands for when the user presses the up arrow.

# All the directories to search for commands.
set path=(/usr/local/bin /usr/bin /bin /usr/bin/X11)

# To prevent the user from accidentally ending a login session,
```

```
# disable Ctrl-D as a way to exit.
set ignoreeof

stty erase "^H"        # Make sure the backspace key erases.
```

.emacs

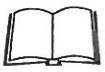

Chapter 9

For the Emacs editor. Consists of LISP functions. See the section "Tailoring Emacs" in Chapter 9.

.exrc

For the *vi* editor (also known as *ex*). Each line is an editor command. See the section "Extending vi" in Chapter 9.

.fvwm2rc


Chapter 10

For the *fvwm2* window manager. Consists of special commands interpreted by *fvwm2*. A sample file is shown in the section "Configuring fvwm" in Chapter 10, *Installing the X Window System.*

.twmrc

For the *twm* window manager. Consists of special commands interpreted by *twm.*

.newsrc

For news readers. Contains a list of all newsgroups offered at the site.

.Xdefaults

For programs using the X Window System. Each line specifies a resource (usually the name of a program and some property of that program) along with the value that resource should take. This file is described in the section "The X Resource Database" in Chapter 10.

.xinitrc

For the X Window System. Consists of shell commands that run whenever you log into an X session. See the section "Basics of X Customization" in Chapter 10 for details on using this file.

Important Directories

You already know about */home*, where user files are stored. As a system administrator and programmer, several other directories will be important to you. Here are a few, along with their contents:

/bin

The most essential Unix commands, such as *ls.*

/usr/bin

Other commands. The distinction between */bin* and */usr/bin* is arbitrary; it was a convenient way to split up commands on early Unix systems that had small disks.

/usr/sbin
: Commands used by the superuser for system administration.

/boot
: Location where the kernel and other files used during booting are sometimes stored.

/etc
: Files used by subsystems such as networking, NFS, and mail. Typically, these contain tables of network services, disks to mount, and so on.

/var
: Administrative files, such as log files, used by various utilities.

/var/spool
: Temporary storage for files being printed, sent by UUCP, and so on.

/usr/lib
: Standard libraries, such as *libc.a*. When you link a program, the linker always searches here for the libraries specified in *–l* options.

/usr/lib/X11
: The X Window System distribution. Contains the libraries used by X clients, as well as fonts, sample resources files, and other important parts of the X package. This directory is usually a symbolic link to */usr/X11R6/lib/X11*.

/usr/include
: Standard location of include files used in C programs, such as *<stdio.h>*.

/usr/src
: Location of sources to programs built on the system.

/usr/local
: Programs and data files that have been added locally by the system administrator.

/etc/skel
: Sample startup files you can place in home directories for new users.

Programs That Serve You

We're including this section because you should start to be interested in what's running on your system behind your back.

Many modern computer activities are too complex for the system simply to look at a file or some other static resource. Sometimes these activities need to interact with another running process.

For instance, take FTP, which you may have used to download some Linux-related documents or software. When you FTP to another system, another program has to be running on that system to accept your connection and interpret your commands. So there's a program running on that system called *ftpd*. The *d* in the name stands for *daemon*, which is a quaint Unix term for a server that runs in the background all the time. Most daemons handle network activities.

You've probably heard of the buzzword *client/server* enough to make you sick, but here it is in action—it has been in action for years on Unix.

Daemons start up when the system is booted. To see how they get started, look in the */etc/inittab* and */etc/inetd.conf* files, as well as distribution-specific configuration files. We won't go into their formats here. But each line in these files lists a program that runs when the system starts. You can find the distribution-specific files either by checking the documentation that came with your system or by looking for pathnames that occur more often than others in */etc/inittab*. Those normally indicate the directory tree where your distribution stores its system startup-files.

To give an example of how your system uses */etc/inittab*, look at one or more lines with the string `getty` or `agetty`. This is the program that listens at a terminal (tty) waiting for a user to log in. It's the program that displays the `login :` prompt we talked about at the beginning of this chapter.

The */etc/inetd.conf* file represents a more complicated way of running programs—another level of indirection. The idea behind */etc/inetd.conf* is that it would waste a lot of system resources if a dozen or more daemons were spinning idly, waiting for a request to come over the network. So, instead, the system starts up a single daemon named *inetd*. This daemon listens for connections from clients on other machines, and when an incoming connection is made, starts up the appropriate daemon to handle it. For example, when an incoming FTP connection is made, *inetd* starts up the FTP daemon (*ftpd*) to manage the connection. In this way, the only network daemons running are those actually in use.

In the next section, we'll show you how to see which daemons are running on your system. There's a daemon for every service offered by the system to other systems on a network: *fingerd* to handle remote *finger* requests, *rwhod* to handle *rwho* requests, and so on. A few daemons also handle nonnetworking services, such as *kerneld*, which handles the automatic loading of modules into the kernel.

Processes

At the heart of Unix lies the concept of a process. Understanding this concept will help you keep control of your login session as a user. If you are also a system administrator, the concept is even more important.

A process is an independently running program that has its own set of resources. For instance, we showed in an earlier section how you could direct the output of a program to a file while your shell continued to direct output to your screen. The reason that the shell and the other program can send output to different places is that they are separate processes.

On Unix, the finite resources of the system, like the memory and the disks, are managed by one all-powerful program called the kernel. Everything else on the system is a process.

Thus, before you log in, your terminal is monitored by a *getty* process. After you log in, the *getty* process dies (a new one is started by the kernel when you log out) and your terminal is managed by your shell, which is a different process. The shell then creates a new process each time you enter a command. The creation of a new process is called *forking*, because one process splits into two.

If you are using the X Window System, each process starts up one or more windows. Thus, the window in which you are typing commands is owned by an *xterm* process. That process forks a shell to run within the window. And that shell forks yet more processes as you enter commands.

To see the processes you are running, enter the command *ps*. Figure 4-4 shows some typical output and what each field means. You may be surprised how many processes you are running, especially if you are using X. One of the processes is the *ps* command itself, which of course dies as soon as the output is displayed.

```
$ ps
  PID   TTY      STAT   TIME     COMMAND
 1663   pp3      S      0:01     -bash
 1672   pp3      T      0:07     emacs
 1676   pp3      R      0:00     ps
```

PID - *Process ID (used to kill a process)*
TTY - *Controlling terminal*
STAT - *State*
TIME - *CPU time used so far*
COMMAND - *Command running*

Figure 4-4: Output of ps command

The first field in the *ps* output is a unique identifier for the process. If you have a runaway process that you can't get rid of through Ctrl-C or other means, you can kill it by going to a different virtual console or X window and entering:

```
$ kill process-id
```

The TTY field shows which terminal the process is running on, if any. (Everything run from a shell uses a terminal, of course, but background daemons don't have a terminal.)

The STAT field shows what state the process is in. The shell is currently suspended, so this field shows an S. An Emacs editing session is running, but it's suspended using Ctrl-Z. This is shown by the T in its STAT field. The last process shown is the *ps* that is generating all this input; its state, of course, is R because it is running.

The TIME field shows how much CPU time the processes have used. Since both *bash* and Emacs are interactive, they actually don't use much of the CPU.

You aren't restricted to seeing your own processes. Look for a minute at all the processes on the system. The *a* option stands for all processes, while the *x* option includes processes that have no controlling terminal (such as daemons started at runtime):

```
$ ps ax | more
```

Now you can see the daemons that we mentioned in the previous section.

And here, with a breathtaking view of the entire Unix system at work, we end this chapter (the lines are cut off at column 76; if you want to see the command lines in their full glory, add the option *-w* to the *ps* command):

```
kalle@tigger: > ps aux
    USER   PID %CPU %MEM    VSZ    RSS  TT STAT  START   TIME COMMAND
at         724  0.0  0.2    824    348  ?  S     Mar 18   0:00 /usr/sbin/
bin        703  0.0  0.2    832    316  ?  S     Mar 18   0:00 /usr/sbin/
kalle      181  0.0  0.6   1512    856   1 S     Mar 18   0:00 -bash
kalle      230  0.0  0.4   1396    596   1 S     Mar 18   0:00 sh /usr/X1
kalle      231  0.0  0.1    808    256   1 S     Mar 18   0:00 tee /home/
kalle      234  0.0  0.4   1952    624   1 S     Mar 18   0:00 xinit /hom
kalle      238  0.0  0.4   1396    616   1 S     Mar 18   0:00 sh /home/k
kalle      242  0.0  3.8   6744   4876   1 S     Mar 18   0:43 kwm
kalle      246  0.0  3.3   6552   4272   1 S     Mar 18   4:48 /usr/local
kalle      255  0.0  0.0      0      0   1 Z     Mar 18   0:00 kaudioserv
kalle      256  0.0  3.0   6208   3844   1 S     Mar 18   0:02 kwmsound
kalle      257  0.0  5.1   8892   6596   1 S     Mar 18   0:11 kfm
kalle      258  0.0  3.3   6292   4320   1 S     Mar 18   0:02 krootwm
kalle      259  0.0  4.6   7848   5988   1 S     Mar 18   0:37 kpanel
kalle      260  0.0  3.6   6764   4688   1 S     Mar 18   0:06 kbgndwm
kalle      273  0.0  3.6   6732   4668   1 S     Mar 18   0:08 kvt -resto
kalle      274  0.0  3.6   6732   4668   1 S     Mar 18   0:11 kvt -resto
kalle      276  0.0  0.6   1536    892  p0 S     Mar 18   0:00 bash
kalle      277  0.0  0.6   1512    864  p1 S     Mar 18   0:00 bash
kalle    11752  0.1  9.8  14056  12604   1 S     Mar 20   3:35 xemacs
kalle    18738  0.2 16.4  26164  21088   1 S      01:14   1:03 netscape
kalle    18739  0.0  2.6  14816   3392   1 S      01:14   0:00 (dns helpe
kalle    29744  0.0  0.3    904    428  p0 R      09:24   0:00 ps -auxw
root         1  0.0  0.2    820    292  ?  S     Mar 18   0:06 init [2]
```

```
root         2  0.0  0.0     0     0  ?  SW   Mar 18   0:00 kflushd
root         3  0.0  0.0     0     0  ?  SW<  Mar 18   0:00 kswapd
root         8  0.0  0.2   804   264  ?  S    Mar 18   0:02 update (bd
root        55  0.0  0.2   816   328  ?  S    Mar 18   0:00 /sbin/kern
root        78  0.0  0.0     0     0  ?  Z    Mar 18   0:00 request-ro
root        96  0.0  0.3   832   408  ?  S    Mar 18   0:00 /usr/sbin/
root        98  0.0  0.3   932   448  ?  S    Mar 18   0:00 /usr/sbin/
root       167  0.0  0.2   824   288  ?  S    Mar 18   0:00 /usr/bin/g
root       182  0.0  0.6  1508   856  2  S    Mar 18   0:00 -bash
root       183  0.0  0.2   808   288  3  S    Mar 18   0:00 /sbin/ming
root       184  0.0  0.2   808   284  4  S    Mar 18   0:00 /sbin/ming
root       185  0.0  0.2   808   284  5  S    Mar 18   0:00 /sbin/ming
root       186  0.0  0.2   808   284  6  S    Mar 18   0:00 /sbin/ming
root       235  0.3 11.8 25292 15196  ?  S    Mar 18  19:19 /usr/X11R6
root       682  0.0  0.4  1076   556  ?  S    Mar 18   0:00 /usr/sbin/
root       684  0.0  0.3   948   484  ?  S    Mar 18   0:00 /usr/sbin/
root       707  0.0  0.3   860   440  ?  S    Mar 18   0:00 /usr/sbin/
root       709  0.0  0.3   896   452  ?  S    Mar 18   0:00 /usr/sbin/
root       712  0.0  0.5  1212   668  ?  S    Mar 18   0:00 /usr/sbin/
root       727  0.0  0.2   840   356  ?  S    Mar 18   0:00 /usr/sbin/
root       733  0.0  0.2   820   304  ?  S    Mar 18   0:00 /usr/sbin/
root       737  0.0  0.2   836   316  ?  S    Mar 18   0:00 /usr/sbin/
root       745  0.0  0.5  1204   708  ?  S    Mar 18   0:00 sendmail:
root       752  0.0  0.4  1772   592  ?  S    Mar 18   0:00 /opt/appli
wwwrun     718  0.0  0.5  1212   668  ?  S    Mar 18   0:00 /usr/sbin/
wwwrun     719  0.0  0.5  1212   652  ?  S    Mar 18   0:00 /usr/sbin/
wwwrun     720  0.0  0.5  1212   644  ?  S    Mar 18   0:00 /usr/sbin/
wwwrun     721  0.0  0.5  1212   644  ?  S    Mar 18   0:00 /usr/sbin/
wwwrun     722  0.0  0.5  1212   644  ?  S    Mar 18   0:00 /usr/sbin/
```