3

Using Your Unix Account

In this chapter:

- *The Unix Filesystem*
- *Looking Inside Files with less*
- *Protecting and Sharing Files*
- *Graphical Filesystem Browsers*
- *Completing File and Directory Names*
- *Changing Your Password*
- *Customizing Your Account*

Once you log in, you can use the many facilities that Unix provides. As an authorized system user, you have an account that gives you:

- A place in the Unix filesystem where you can store your files.
- A username that identifies you, lets you control access to your files, and is an address for your email.
- An environment you can customize.

The Unix Filesystem

A *file* is the unit of storage in Unix, as in most other systems. A file can hold anything: text (a report you're writing, a to-do list), a program, digitally encoded pictures or sound, and so on. All of those are just sequences of raw data until they're interpreted by the right program.

In Unix, files are organized into directories. A *directory* is actually a special kind of file where the system stores information about other files. You can think of a directory as a place, so that files are said to be contained *in* directories and you are said to work *inside* a directory. (If you've used a Macintosh or Microsoft Windows computer, a Unix directory is similar to a folder.)

This section introduces the Unix filesystem. Later sections in this chapter show how you can look in files and protect them. Chapter 4 has more information.

7 January 2002 13:12

Your Home Directory

When you log in to Unix, you're placed in a directory called your *home directory*. This directory, a unique place in the Unix filesystem, contains the files you use almost every time you log in. In your home directory, you can make your own files. As you'll see in a minute, you can also store your own directories within your home directory. Like folders in a file cabinet, this is a good way to organize your files.

Your Working Directory

Your *working directory* (also called your current directory) is the directory you're currently working in. Every time you log in, your home directory is your working directory. You may change to another directory, in which case the directory you move to becomes your working directory.

Unless you tell Unix otherwise, all commands that you enter apply to the files in your working directory. In the same way, when you create files, they're created in your working directory unless you specify another directory. For instance, if you type the command `pico` **report**, the Pico editor is started on a file named *report* in your working directory. But if you type a command such as `pico` **/home/joan/report**, a *report* file is edited in a different directory—without changing your working directory. You'll learn more about this when we cover pathnames later in this chapter.

If you have more than one terminal window open, or you're logged in on several terminals at the same time, each session has its own working directory. Changing the working directory in one session doesn't affect others.

The Directory Tree

All directories on a Unix system are organized into a hierarchical structure that you can imagine as a family tree. The parent directory of the tree (the directory that contains all other directories) is known as the *root directory* and is written as a forward slash (`/`).

The root contains several directories. Figure 3-1 shows a visual representation of the top of a Unix filesystem tree: the root directory and some directories under the root.

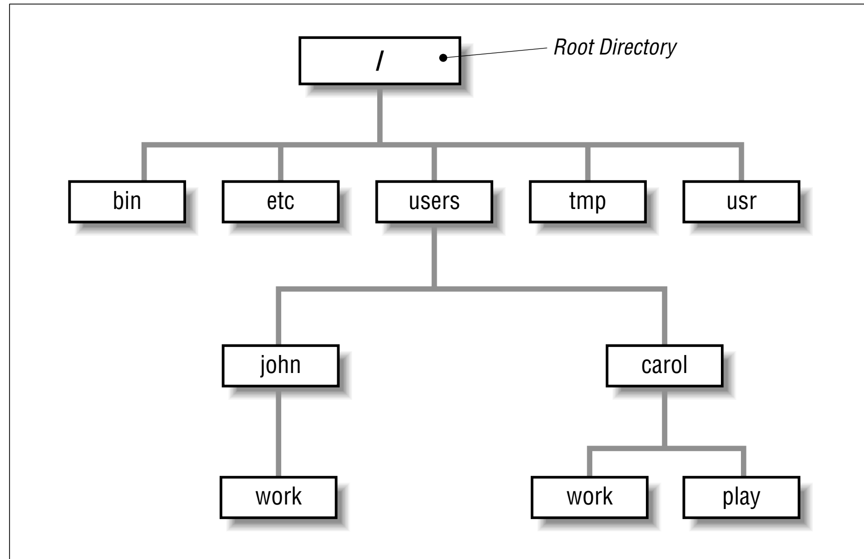


Figure 3-1. Example of a directory tree

bin, *etc*, *users*, *tmp*, and *usr* are some of the *subdirectories* (child directories) of the root directory. These subdirectories are fairly standard directories; they usually contain specific kinds of system files. For instance, *bin* contains many Unix programs. Not all systems have a directory named *users*. It may be called *u* or *home*, and/or it may be located in some other part of the filesystem.

In our example, the parent directory of *users* (one level above) is the root directory. It has two subdirectories (one level below), *john* and *carol*. On a Unix system, each directory has only one parent directory, but it may have one or more subdirectories.* A subdirectory (such as *carol*) can have its own subdirectories (such as *work* and *play*), up to a limitless depth for practical purposes.

To specify a file or directory location, write its *pathname*. A pathname is like the address of the directory or file in the Unix filesystem. We look at pathnames in a moment.

On a basic Unix system, all files in the filesystem are stored on disks connected to your computer. It isn't always easy to use the files on someone else's computer or for someone on another computer to use your files. Your system may have an easier way: a *networked filesystem*. Networked

* On most Unix systems, the root directory, at the top of the tree, is *its own* parent. Some systems have another directory above the root.

filesystems make a remote computer's files appear as if they're part of your computer's directory tree. For instance, a computer in Los Angeles might have a directory named *boston* with some of the directory tree from a company's computer in Boston. Or individual users' home directories may come from various computers, but all be available on your computer as if they were local files. The system staff can help you understand and configure your computer's filesystems to make your work easier.

Absolute Pathnames

As you saw earlier, the Unix filesystem organizes its files and directories in an inverted tree structure with the root directory at the top. An *absolute pathname* tells you the path of directories you must travel to get from the root to the directory or file you want. In a pathname, put slashes (/) between the directory names.

For example, */users/john* is an absolute pathname. It locates one (*only* one!) directory. Here's how:

- The root is the first "/"
- The directory *users* (a subdirectory of *root*)
- The directory *john* (a subdirectory of *users*)

Be sure that you do not type spaces anywhere in the pathname. Figure 3-2 shows this structure.

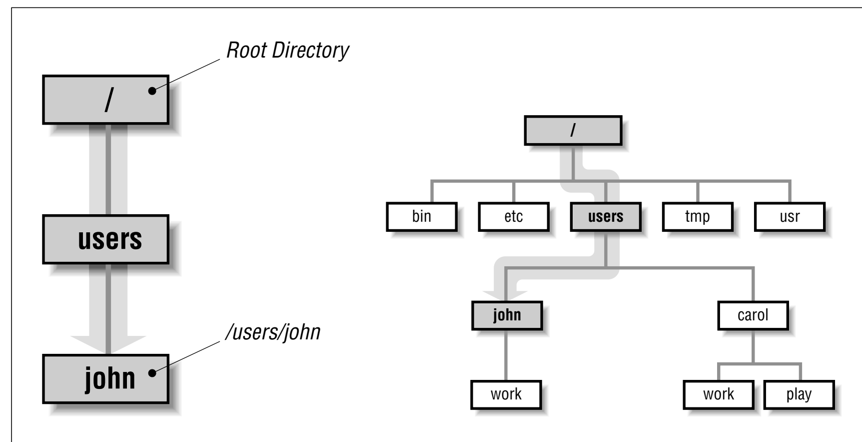


Figure 3-2. Absolute path of directory john

In Figure 3-2, you'll see that the directory *john* has a subdirectory named *work*. Its absolute pathname is */users/john/work*.

The root is always indicated by the slash (/) at the start of the pathname. In other words, *an absolute pathname always starts with a slash.*

Relative Pathnames

You can also locate a file or directory with a *relative pathname*. A relative pathname gives the location relative to your working directory.

Unless you use an absolute pathname (starting with a slash), Unix assumes that you're using a relative pathname. Like absolute pathnames, relative pathnames can go through more than one directory level by naming the directories along the path.

For example, if you're currently in the *users* directory (see Figure 3-2), the relative pathname to the *carol* directory below is simply *carol*. The relative pathname to the *play* directory below that is *carol/play*.

Notice that neither pathname in the previous paragraph starts with a slash. That's what makes them relative pathnames! Relative pathnames start at the working directory, not the root directory. In other words, *a relative pathname never starts with a slash.*

Pathname puzzle

Here's a short but important question. The previous example explains the relative pathname *carol/play*. What do you think Unix would say about the pathname */carol/play*? (Look again at Figure 3-2.)

Unix would say "No such file or directory." Why? (Please think about that before you read more. It's very important and it's one of the most common beginner's mistakes.) Here's the answer. Because it starts with a slash, the pathname */carol/play* is an absolute pathname that starts from the root. It says to look in the root directory for a subdirectory named *carol*. But there is no subdirectory named *carol* one level directly below the root, so the pathname is wrong. The only absolute pathname to the *play* directory is */users/carol/play*.

Relative pathnames up

You can go up the tree with the shorthand ".." (dot dot) for the parent directory. As you saw earlier, you can also go down the tree by using subdirectory names. In either case (up or down), separate each level by a slash (/).

Figure 3-3 shows part of Figure 3-1. If your working directory in the figure is *work*, then there are two pathnames for the *play* subdirectory of *carol*. You already know how to write the absolute pathname, */users/carol/play*. You can also go up one level (with "..") to *carol*, then go down the tree to *play*. Figure 3-3 illustrates this.

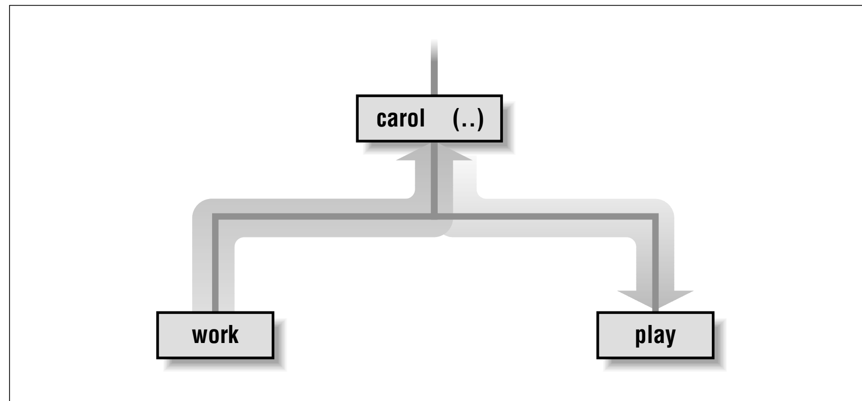


Figure 3-3. Relative pathname from work to play

The relative pathname would be *../play*. It would be wrong to give the relative address as *carol/play*. Using *carol/play* would say that *carol* is a subdirectory of your working directory instead of what it is in this case—the parent directory.

Absolute and relative pathnames are totally interchangeable. Unix programs simply follow whatever path you specify to wherever it leads. If you use an absolute pathname, the path starts from the root. If you use a relative pathname, the path starts from your working directory. Choose whichever is easier at the moment.

Changing Your Working Directory

Once you know the absolute or relative pathname of a directory where you'd like to work, you can move up and down the Unix directory tree to reach it.

pwd

ˆM To find which directory you're currently in, use **pwd** (print working directory). The **pwd** command takes no arguments.

```
$ pwd
/users/john
$
```

pwd prints the absolute pathname of your working directory.

cd

You can change your working directory to any directory (including another user's directory—if you have permission) with the **cd** (change directory) command.

The **cd** command has the form:

cd *pathname*

The argument is an absolute or a relative pathname (whichever is easier) for the directory you want to change to:

```
$ cd /users/carol
$ pwd
/users/carol
$ cd work
$ pwd
/users/carol/work
$
```

Here's a timesaver: the command **cd**, with no arguments, takes you to your home directory from wherever you are in the filesystem.

Note that you can only change to another directory. You cannot **cd** to a filename. If you try, your shell (in this example, **bash**) gives you an error message:

```
$ cd /etc/passwd
bash: /etc/passwd: Not a directory
$
```

/etc/passwd is a file with information about users' accounts.

Files in the Directory Tree

A directory can hold subdirectories. And, of course, a directory can hold files. Figure 3-4 is a close-up of the filesystem around *john*'s home directory. The four files are shown along with the *work* subdirectory.

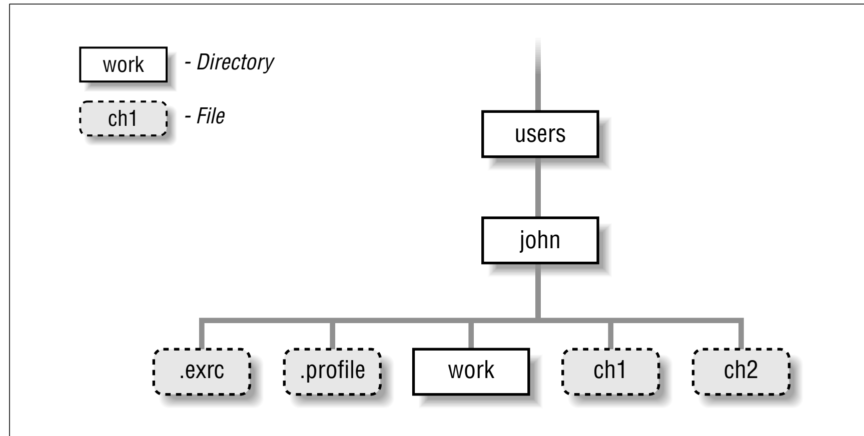


Figure 3-4. Files in the directory tree

Pathnames to files are made the same way as pathnames to directories. As with directories, files' pathnames can be absolute (starting from the root directory) or relative (starting from the working directory). For example, if your working directory is *users*, the relative pathname to the *work* directory below would be *john/work*. The relative pathname to the *ch1* file would be *john/ch1*.

Unix filesystems can hold things that aren't directories or files, such as symbolic links, FIFOs, and sockets (they have pathnames, too). You may see some of them as you explore the filesystem. We don't cover those advanced topics in this little book.

Listing Files with ls

To use the **cd** command, you must decide which entries in a directory are subdirectories and which are files. The **ls** command lists entries in the directory tree and can also show you which is which.

> When you enter the **ls** command, you'll get a listing of the files and subdirectories contained in your working directory. The syntax is:

ls *option(s) directory-and-filename(s)*

If you've just logged in for the first time, entering **ls** without any arguments may seem to do nothing. This isn't surprising because you haven't

made any files in your working directory. If you have no files, nothing is displayed; you'll simply get a new shell prompt:

```
$ ls
$
```

But if you've already made some files or directories in your account, those names are displayed. The output depends on what's in your directory. The screen should look something like this:

```
$ ls
ch1    ch10    ch2    ch3   intro
$
```

(Some systems display filenames in a single column. If yours does, you can make a multicolumn display with the **-C** [uppercase "C"] option or the **-x** option.) **ls** has a lot of options that change the information and display format.

The **-a** option (for *all*) is guaranteed to show you some more files, as in the following example showing a directory like the one in Figure 3-4:

```
$ ls -a
.      .exrc      ch1     ch2     intro
..     .profile   ch10    ch3
$
```

When you use **ls -a**, you'll always see at least two entries with the names "." (dot) and ".." (dot dot). As mentioned earlier, .. is always the relative pathname to the parent directory. A single . always stands for its working directory; this is useful with commands like **cp** (see the section "Copying Files" in Chapter 4). There may also be other files, such as *.profile* or *.exrc*. Any entry whose name begins with a dot is hidden—it's listed only if you use **ls -a**.

To get more information about each item that **ls** lists, add the **-l** option. (That's a lowercase "L" for "long.") This option can be used alone, or in combination with **-a**, as shown in Figure 3-5.

The long format provides the following information about each item:

Total n

n amount of storage used by everything in this directory. (This is measured in *blocks*. On many systems, but not all, a full block holds 1024 bytes. A block can also be partly full.)

```
$ ls -al
total 94
drwxr-xr-x  2  john  doc    512  Jul 10  22:25  .
drwxr-xr-x  4  bin   bin   1024  Jul  8  11:48  ..
-rw-r--r--  1  john  doc    136  Jul  8  14:46  .exrc
-rw-r--r--  1  john  doc    833  Jul  8  14:51  .profile
-rw-rw-rw-  1  john  doc  31273  Jul 10  22:25  ch1
-rw-rw-rw-  1  john  doc      0  Jul 10  21:57  ch2
```


Figure 3-5. Output from ls -al

Type
: Tells whether the item is a directory (`d`) or a plain file (`-`). (There are other less common types that we don't explain here.)

Access modes
: Specifies three types of users (yourself, your group, all others) who are allowed to read (`r`), write (`w`), or execute (`x`) your files. We'll say more about this in a moment.

Links
: The number of files or directories linked to this one. (This isn't the same sort of *link* as in a web page. We don't discuss filesystem links in this little book.)

Owner
: The user who created or owns this file or directory.

Group
: The group that owns the file or directory. (If your version of Unix doesn't show this column, add the `-g` option to see it.)

Size (in bytes)
: The size of the file or directory. (A directory is actually a special type of file. Here, the "size" of a directory is of the directory file itself, not of all the files in that directory.)

Modification date
: When the file was last modified, or the directory contents last changed (when something in the directory was added, renamed, or removed). If an entry was modified more than six months ago, **ls** shows the year instead of the time.

Name
: The name of the file or directory.

Notice especially the columns that list the owner and group of the files, and the access modes (also called permissions). The person who creates a file is its owner; if you've created any files (or system staff did it for you), this column should show your username. You also belong to a group, set by the person who created your account. Files you create are either marked with the name of your group, or in some cases, the group that owns the directory.

The *permissions* show who can read, write, or execute the file or directory; we explain what that means in a moment. The permissions have ten characters. The first character shows the file type (`d` for directory or `-` for a plain file). The other characters come in groups of three. The first group, characters 2–4, show the permissions for the file's owner, which is yourself if you created the file. The second group, characters 5–7, show permissions for other members of the file's group. The third group, characters 8–10, show permissions for all other users.

For example, the permissions for *.profile* are `-rw-r--r--`, so it's a plain file. The owner, *john*, has both read and write permissions. Other users who belong to the file's group *doc*, as well as all other users of the system, can only read the file; they don't have write permission, so they can't change what's in the file. No one has execute (`x`) permission, which should only be used for executable files (files that hold programs).

In the case of directories, `x` means the permission to access the directory—for example, to run a command that reads a file there or to use a subdirectory. Notice that the two directories shown in the example are executable (accessible) by *john*, by users in the *doc* group, and by everyone else on the system. A directory with `w` (write) permission allows deleting, renaming, or adding files within the directory. Read (`r`) permission allows listing the directory with **ls**.

You can use the **chmod** command to change the permissions of your files and directories. See the section "Protecting and Sharing Files," later in this chapter.

If you need to know only which files are directories and which are executable files, you can use the –F option.

If you give the pathname to a directory, **ls** lists the directory but it does *not* change your working directory. The **pwd** command in the following example shows this:

```
$ ls -F /users/andy
calendar    goals    ideas/
ch2         guide/   testpgm*
$ pwd
/etc
$
```

ls -F puts a / (slash) at the end of each directory name. (The directory name doesn't really have a slash in it; that's just the shortcut **ls –F** uses to identify a directory.) In our example, *guide* and *ideas* are directories. You can verify this by using **ls –l** and noting the "d" in the first field of the output. Files with an execute status (**x**), such as programs, are marked with an * (asterisk). The file *testpgm* is an executable file. Files that aren't marked are not executable.

ls –R ("recursive") lists a directory and all its subdirectories. This can make a very long list—especially when you list a directory near the root! (Piping the output of **ls** to a pager program solves this problem. There's an example in the section "Piping to a Pager" in Chapter 1.) You can combine other options with **–R**: for instance, **ls –RF** marks each directory and file type.

On Linux and other systems with the GNU version of **ls**, you may be able to see names in color. For instance, directories could be green and program files could be yellow. Like almost everything on Unix, of course, this is configurable. The details are more than we can cover in an introductory book. Try typing **ls --color** and see what happens. (It's time for our familiar mantra: check your documentation. See Chapter 8—especially the **man** command for reading a command's online manual page.)

Exercise: exploring the filesystem

You're now equipped to explore the filesystem with **cd**, **ls**, and **pwd**. Take a tour of the directory system, hopping one or many levels at a time, with a mixture of **cd** and **pwd** commands.

Go to your home directory.	Enter **cd**
Find your working directory.	Enter **pwd**
Change to new working directory with its absolute pathname.	Enter **cd /etc**
List files in new working directory.	Enter **ls**
Change directory to root and list it in one step. (Use the command separator, a semicolon.)	Enter **cd /; ls**
Find your working directory.	Enter **pwd**
Change to a subdirectory; use its relative pathname.	Enter **cd usr**
Find your working directory.	Enter **pwd**
Change to a subdirectory.	Enter **cd bin**
Find your working directory.	Enter **pwd**
Give a wrong pathname.	Enter **cd xqk**
List files in another directory.	Enter **ls /bin**
Find your working directory (notice that **ls** didn't change it).	Enter **pwd**
Return to your home directory.	Enter **cd**

Looking Inside Files with less

By now, you're probably tired of looking at files from the outside. It's kind of like going to a bookstore and looking at the covers, but never getting to read a word. Let's look at a program for reading files.

If you want to "read" a long file on the screen, your system may have the **less** command to display one "page" (a terminal filled from top to bottom) of text at a time.

If you don't have **less**, you'll probably have similar programs named **more** or **pg**. (In fact, the name **less** is a play on the name of **more**, which came first.) The syntax is:

less *option(s) file(s)*

less lets you move forward or backward in the files by any number of pages or lines; you can also move back and forth between two or more files specified on the command line. When you invoke **less**, the first "page" of the file appears. A prompt appears at the bottom of the terminal (or terminal window), as in the following example:

```
$ less ch03
A file is the unit of storage in Unix, as in most other systems.
A file can hold anything: text (a report you're writing,
        .
        .
        .
:
```

The basic **less** prompt is just a colon (`:`)—although, for the first screenful, **less** displays the file's name as a prompt. The cursor sits to the right of this prompt as a signal for you to enter a **less** command to tell **less** what to do.

Like almost everything about **less**, the prompt can be customized. For example, using the **less -M** option on the **less** command line makes the prompt show the filename and your position in the file. (If you want this to happen every time you use **less**, you can set the LESS environment variable to `M` (without a dash) in your shell setup file. See the section "Customizing Your Account," later in this chapter.)

You can set or unset most options temporarily from the **less** prompt. For instance, if you have the short **less** prompt (a colon), you can enter **-M** while **less** is running. **less** responds "Long prompt (press RETURN)," and for the rest of the session, **less** prompts with the filename, line number, and percentage of the file viewed.

To display the **less** commands and options available on your system, press "h" (for "help") while **less** is running. Table 3-1 lists some simple (but still quite useful) commands.

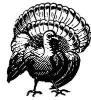
Table 3-1. Useful less commands

Command	Description	Command	Description
SPACE	Display next page.	**v**	v
RETURN	Display next line.	CTRL-L	Redisplay current page.
*n*f	Move forward *n* lines.		Help.
	Move backward one page.	:n	Go to next file on command line.
*n***b**	Move backward *n* lines.	:p	Go back to previous file on command line.
/*word*	Search forward for *word*.	q	Quit **less**.
?*word*	Search backward for *word*.		

Protecting and Sharing Files

Unix makes it easy for users to share files and directories. For instance, everyone in a group can read documents stored in one of their manager's directories without needing to make their own copies—if the manager has allowed access. There might be no need to fill peoples' email inboxes with file attachments if everyone can access those files directly through the Unix filesystem.

Here's a brief introduction to file security and sharing. Networked systems with multiple users, such as Unix, have complex security issues that take tens or hundreds of pages to explain. If you have critical security needs or you just want more information, talk to your system staff or see an up-to-date book on Unix security.



Note that the system's superuser (the system administrator and possibly other users) can do anything to any file at any time, no matter what its permissions are. So, access permissions won't keep your private information safe from *everyone*—although let's hope that you can trust your system staff!

Your system staff should also keep backup copies of users' files. These backup copies may be readable by anyone who has physical access to them. That is, anyone who can take the backup out of a cabinet (or wherever) and mount it on a computer system may be able to read the file copies. The same is true for files stored on floppy disks and any other removable media. (Once you take a file off of a Unix system, that system can't control access to it anymore.)

Directory Access Permissions

A directory's access permissions help to control access to the files and subdirectories in that directory:

- If a directory has read permission, a user can run **ls** to see what's in the directory and use wildcards to match files in it.
- A directory that has write permission allows users to add, rename, and delete files in the directory.

- To access a directory—that is, to read or write the files in the directory or to run the files if they're programs—a user needs execute permission on that directory. Note that to access a directory, a user must *also* have execute permission to all of its parent directories, all the way up to the root!

File Access Permissions

The access permissions on a file control what can be done to the file's *contents*. The access permissions on the *directory* where the file is kept control whether the file can be renamed or removed. (If this seems confusing, think of it this way: the directory is actually a list of files. Adding, renaming, or removing a file changes the contents of the directory. If the directory isn't writable, you can't change that list.)

Read permission controls whether you can read a file's contents. Write permission lets you change a file's contents. A file shouldn't have execute permission unless it's a program.

Setting Permissions with chmod

Once you know what permissions a file or directory needs—and if you're the owner (listed in the third column of **ls -l** output)—you can change the permissions with the **chmod** program.

There are two ways to change permissions: by specifying the permissions to add or delete, or by specifying the exact permissions.* For instance, if a directory's permissions are almost correct, but you also need to make it writable by its group, tell **chmod** to add group-write permission. But if you need to make more than one change to the permissions—for instance, you want to add read and execute permission, but delete write permission—it's easier to set all permissions explicitly instead of changing them one-by-one. The syntax is:

chmod *permissions file(s)*

Let's start with the rules; we see examples next. The *permissions* argument has three parts, which you must give in order with no space between.

* Early versions of **chmod** can't add or delete particular permissions. Instead, you have to give an exact permission as three digits between 0 and 7. If you need to use **chmod** that way, please see a more detailed Unix reference.

1. The category of permission you want to change. There are three: the owner's permission (which **chmod** calls "user," abbreviated u), the group's permission (g), or others' permission (o). To change more than one category, string the letters together, such as go for "group and others," or simply use a to mean "all" (same as ugo).
2. Whether you want to add (+) the permission, delete (-) it, or specify it exactly (=).
3. What permissions you want to affect: read (r), write (w), or execute (x). To change more than one permission, string the letters together—for example, rw for "read and write."

Some examples should make this clearer! In the following command lines, you can replace *dirname* or *filename* with the pathname (absolute or relative) of the directory or file. An easy way to change permissions on the working directory is by using its relative pathname, . (dot), as in "**chmod a-w .**". You can combine two permission changes in the same **chmod** command by separating them with a comma (,), as shown in the final example.

- To protect a file from accidental editing, delete everyone's write permission with the command "**chmod a-w** *filename*". On the other hand, if you own an unwritable file that you want to edit, but you don't want to change other peoples' write permissions, you can add "user" (owner) write permission with "**chmod u+w** *filename*".
- To keep yourself from accidentally removing files (or adding or renaming files) in an important directory of yours, delete your own write permission with the command "**chmod u-w** *dirname*". If other users have that permission, too, you could delete everyone's write permission with "**chmod a-w** *dirname*".
- If you want you and your group to be able to read and write all the files in your working directory—but those files have various permissions now, so adding and deleting the permissions individually would be a pain—this is a good place to use the = operator to set the exact permissions you want. Use the filename wildcard *, which means "everything in this directory" (explained in the section "File and Directory Wildcards" of Chapter 4) and type: "**chmod ug=rw ***".

  If your working directory had any subdirectories, though, that command would be wrong because it takes away execute permission from the subdirectories, so the subdirectories couldn't be accessed

anymore. In that case, you could try a more specific wildcard. Or, instead of a wildcard, you can simply list the filenames you want to change, separated by spaces, as in "**chmod ug=rw afile bfile cfile**".

- To protect the files in a directory and all its subdirectories from everyone else on your system, but still keep the access permissions *you* have there, you could use "**chmod go-rwx** *dirname*" in order to delete all "group" and "others" permission to read, write, and execute. A simpler way is to use the command "**chmod go=** *dirname*" to set "group" and "others" permission to exactly nothing.
- You want full access to a directory. Other people on the system should be able to see what's in the directory—and read or edit the files if the file permissions allow it—but not rename, remove, or add files. To do that, give yourself all permissions, but give "group" and "others" only read and execute permission. Use the command "**chmod u=rwx,go=rx** *dirname*".

After you change permissions, it's a good idea to check your work at first with "**ls -l** *filename*" or "**ls -ld** *dirname*".

More Protection Under Linux

Most Linux systems have a program named **chattr** that gives you more choices on file and directory protection. **chattr** is being developed, and your version may not have all the features that it will have in later Linux versions. For instance, **chattr** can make a Linux file *append-only* (so it can't be overwritten, only added to), *compressed* (to save disk space automatically), *immutable* (so it can't be changed at all), *undeletable*, and more. Check your online documentation (type **man chattr**—see Chapter 8).

Problem checklist

I get the message "chmod: Not owner."
Only the owner of a file or directory—or the superuser—can set its permissions. Use **ls -l** to find the owner, or ask a system staff person to change the permissions.

A file is writable, but my program says it can't be written.
First, check the file permissions with **ls -l** and be sure you're in the category (user, group, or others) that has write permission.

The problem may also be in the permissions of the file's *directory*. Some programs need permission to write more files into the same directory (for example, temporary files), or to rename files (for instance, making a file into a backup) while editing. If it's safe to add write permission to the directory (if other files in the directory don't need protection from removal or renaming) try that. Otherwise, copy the file to a writable directory (with **cp**), edit it there, then copy it back to the original directory.

Changing Group and Owner

Group ownership lets a certain group of users have access to a file or directory. You might need to let a different group have access. The **chgrp** program sets the group owner of a file or directory. You can set the group to any of the groups you belong to. (The system staff control the list of groups you're in.) On most versions of Unix, the **groups** program lists your groups.

For example, if you're an instructor creating a directory named *csc303* for students in a course, the directory's original group owner might be *faculty*. You'd like the students, all of whom are in the group named *csstudnt*, to access the directory; members of other groups should have no access. Use commands such as these:*

```
$ groups
faculty csstudnt wheel research
$ mkdir csc303
$ ls -ld csc303
drwxr-xr-x    2 roberts  faculty      4096 Aug 25 13:35 csc303
$ chgrp csstudnt csc303
$ chmod o= csc303
$ ls -ld csc303
drwxr-x---    2 roberts  csstudnt     4096 Aug 25 13:35 csc303
```

The **chown** program changes the owner of a file or directory. On most Unix systems, only the superuser can use **chown**.†

* Many Unix systems also let you set a directory's group ownership so that any files you later create in that directory will be owned by the same group as the directory. Try the command "**chmod g+s** *dirname*". If this works, the permissions listing from **ls -ld** should show an *s* in place of the second *x*, such as `drwxr-s---`.

† If you have permission to read another user's file, you can make a copy of it (with **cp**; see the section "Copying Files" in Chapter 4). You'll own the copy.

Graphical Filesystem Browsers

Most Unix window systems give you a graphical way to do some of the things you can do with files from the command line. A *filesystem browser*, such as the GNOME File Manager or KDE's Konqueror, lets you see a graphical representation of the filesystem and do a limited number of operations on it. Figure 3-6 shows the GNOME filesystem browser. The left pane has a directory tree. The right pane shows the contents of the directory that's selected (open) in the left pane; here, this is the directory */home/mpeek*. The titlebar shows the pathname of the selected directory.

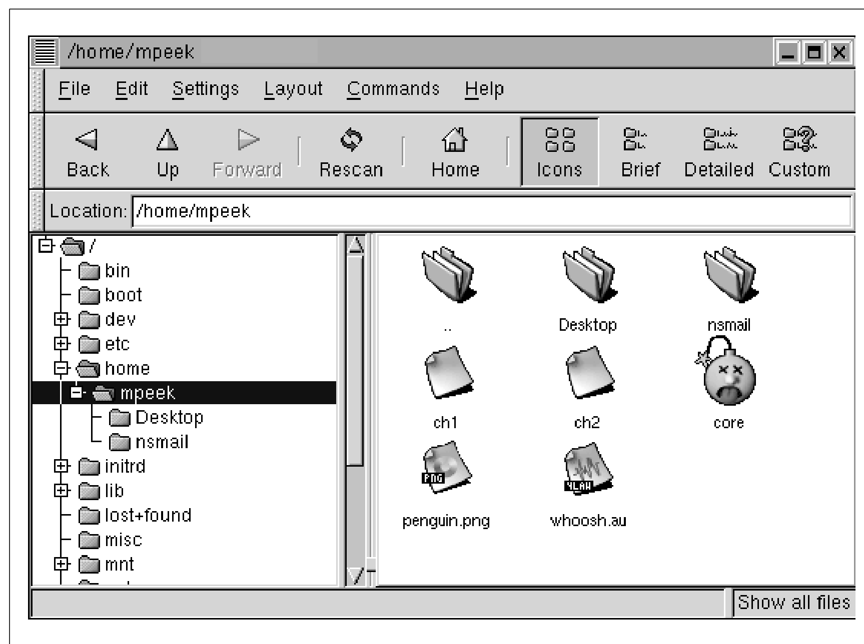


Figure 3-6. GNOME filesystem browser

A filesystem browser can be handy for seeing what's in the filesystem. Unfortunately, because a filesystem browser takes you away from the shell you're using for other work, it can limit what you're able to do with Unix. (You'll see additional information about why this is true when we cover more advanced features such as input-output redirection in Chapter 1.) We recommend learning about your filesystem browser but also learning what you can do at the more powerful Unix command line.

Completing File and Directory Names

Most Unix shells can complete a partly typed file or directory name for you. Different shells have different methods. In many shells, you type the first few letters of the name, then press TAB. If the shell can find just one way to finish the name, it will; your cursor will move to the end of the new name, where you can type more or press RETURN to run the command. (You also can edit or erase the completed name.)

What happens if more than one file or directory name matches what you've typed so far? Again, that depends on the shell you're using. The cursor will probably stay where it is, and the terminal may beep. At this point, the easiest answer could be to type more characters of the name (to make the name unique) and press TAB again to complete the rest of the name. You may also be able to get a list of all possible completions; after the first beep, try pressing TAB again (or CTRL-D, depending on your shell) and you may see a list of all names starting with the characters you've typed so far. Here's an example from the **bash** shell:

```
$ cp /etc/paTAB (beep)TAB
pam.d         paper.config  passwd        passwd-       passwd.OLD
$ cp /etc/pa
```

At this point, I could type another character or two—an *s*, for example—and then press TAB once more to make `/etc/passwd`.

Changing Your Password

On most Unix systems, everyone knows (or can find) your username. When you log in, how does the system decide that you really own your account and aren't an intruder trying to break in? Unix uses your password. If anyone knows both your username and password, they can use your account—including sending email that looks as if you wrote it.* So you should keep your password a secret! Never write it down and leave it anywhere near your terminal.

If you think that someone knows your password, you should probably change it right away—although, if you suspect a computer "cracker" (or

* Unfortunately, it's easy to forge email, without using your computer account at all, so that no one but an expert can tell it was forged.

"hacker") is using your account to break into your system, ask your system administrator for advice first, if possible! You should also change your password periodically; every few months is recommended.

A password should be easy for you to remember but hard for other people (or password-guessing programs!) to guess. Your system should have guidelines for secure passwords. If it doesn't, here are some suggestions. A password should be between six and eight characters long. It should *not* be a word in *any* language, your phone number, your address, or anything anyone else might know or guess that you'd use as a password. It's best to mix upper- and lowercase letters, punctuation, and numbers.

To change your password, you'll probably use either the *passwd* or *yppasswd* program from a shell prompt. After you enter the command, it prompts you to enter your password ("old password"). If the password is correct, it asks you to enter your new password—twice, to be sure there is no typing mistake. For security, neither the old nor new passwords appear as you type them.

On some systems, your password change won't take effect for some time. The change may require between a few minutes to a day.

Customizing Your Account

As we saw earlier, your home directory may have a hidden file called *.profile*. If it doesn't, there'll probably be one or more files named *.login*, *.cshrc*, *.tcshrc*, *.bashrc*, *.bash_profile*, or *.bash_login*. These files are *shell setup files*, and are the key to customizing your account. Shell setup files contain commands that are automatically executed when a new shell starts—especially when you log in.

Let's take a look at these files. Go to your home directory, then use **less** to display the file. Your *.profile* might look something like this:

```
PATH='/bin:/usr/bin:/usr/local/bin:'
LESS='eMq'
export PATH LESS
/usr/games/fortune
date
umask 002
```

A *.login* file could look like this:

```
set path = (/bin /usr/bin /usr/local/bin .)
setenv LESS 'eMq'
/usr/games/fortune
date
umask 002
```

As you can see, these sample setup files contain commands to print a "fortune" and the date—just what happened earlier when we logged in! (*/usr/games/fortune* is a useless but entertaining program that prints a randomly selected saying from its collection. **fortune** isn't available on all systems.)

But what are these other commands?

- The line with `PATH=` or `set path =` tells the shell which directories to search for Unix programs. This saves you the trouble of typing the complete pathname for each program you run. (Notice that */usr/games* isn't part of the path, so we had to use the absolute pathname to get our daily dose of wisdom from the **fortune** program.) The `export PATH` is needed in the *.profile*, but not in *.login*.*
- The line with `LESS=` or `setenv LESS` tells the **less** program which options you want to set every time you use it. This saves you the trouble of typing the options on every **less** command line. The `export LESS` line is needed in the *.profile*, but not in *.login*.
- The **umask** command sets the default file permissions assigned to all files you create. Briefly, a value of 022 sets the permissions `rw-r--r--` (read-write by owner, but read-only by everyone else), and 002 produces `rw-rw-r--` (read-write by owner and group, but read-only by everyone else). If this file is a program or a directory, both **umask** settings also give execute (`x`) permission to all users. For more information, see one of the sources in the section "Documentation" of Chapter 8.

You can change these files with a text editor, such as **pico -w** (see the section "The Pico Text Editor" in Chapter 4). Don't use a word processor that breaks long lines or puts special nontext codes into the file. Any changes you make to those files will take effect the next time you log in (or, in some cases, when you start a new shell—such as opening a new terminal window in your window system). Unfortunately, it's not always easy to know which shell setup file you should change.† And an editing mistake in your shell setup file can keep you from logging in to your account! We suggest that beginners get help from experienced users—and not make changes to these files at all if you're about to do some critical work with your account, unless there's some reason you have to make the changes immediately.

---

* Some shells that read the *.profile* let you set a variable's value on the same line as the **export** command, but not all do. Our two-step method for setting PATH works in all cases.

† Some files are read by *login shells*, and others by *nonlogin shells*. Some are read by *subshells*; others aren't. Some terminal windows open login shells; others don't.

You can execute any of these programs from the command line, as well. In this case, the changes are in effect only until you close that window or log out. If your shell prompt has a `$` character in it, you'll probably use the syntax shown earlier in the *.profile*; if your shell prompt has a `%` or `>` instead, the syntax in the *.login* is probably right.

For example, to change the default options for **less** so it will clear the terminal screen before it shows each new page of text, you'll want to add the **-c** option to the LESS environment variable. The command you'd type at a shell prompt would look something like this:

```
$ LESS='eMqc'
$ export LESS
```

or like this:

```
% setenv LESS 'eMqc'
```

(If you don't want some of the **less** options we've shown, you could leave those letters out.) Unix has many other configuration commands to learn about; the sources listed in the section "Documentation" of Chapter 8 can help.

Just as you can execute the setup commands from the command line, the converse is true: any command that you can execute from the command line can be executed automatically when you log in by placing it in your setup file. (Running interactive commands such as **pine** from your setup file isn't a good idea, though.)