4

The File System

In this chapter, we begin our study of one of the two pillars that support UNIX—the file system. UNIX looks at everything as a file and any UNIX system has thousands of files. If you write a program, you add one more file to the system. When you compile it, you add some more. Files grow rapidly, and if they are not organized properly, you'll find it difficult to locate them. Just as an office has separate file cabinets to group files of a similar nature, UNIX also organizes its own files in *directories* and expects you to do that as well.

The file system in UNIX is one of its simple and conceptually clean features. It lets users access other files not belonging to them, but it also offers an adequate security mechanism so outsiders are not able to tamper with a file's contents. In this chapter, you'll learn to create directories, move around within the system, and list filenames in these directories. We'll deal with file attributes, including the ones related to security, in a later chapter.

What You Will Learn

- The initial categorization of files into three types—*ordinary*, *directory* and *device*.
- The features of a UNIX filename.
- The hierarchical structure containing files and directories, and the parent—child relationship that exists between them.
- The significance of the *home directory* and `HOME` variable.
- Navigate the file system with the **`cd`** and **`pwd`** commands.
- Create and remove directories with **`mkdir`** and **`rmdir`**.
- The significance of *absolute* and *relative* pathnames.
- Use **`ls`** to list filenames in a directory in different formats.

Topics of Special Interest

- The significance of the important directories of the UNIX file system from a functional point of view.
- How It Works: A graphic that shows how **`mkdir`** and **`rmdir`** affect a directory.

4.1 THE FILE

The **file** is a container for storing information. As a first approximation, we can treat it simply as a sequence of characters. If you name a file `foo` and write three characters `a`, `b` and `c` into it, then `foo` will contain only the string `abc` and nothing else. Unlike the old DOS files, a UNIX file doesn't contain the eof (end-of-file) mark. A file's size is not stored in the file, nor even its name. All file attributes are kept in a separate area of the hard disk, not directly accessible to humans, but only to the kernel.

UNIX treats directories and devices as files as well. A directory is simply a folder where you store filenames and other directories. All physical devices like the hard disk, memory, CD-ROM, printer and modem are treated as files. The shell is also a file, and so is the kernel. And if you are wondering how UNIX treats the main memory in your system, it's a file too!

So we have already divided files into three categories:

- *Ordinary file*—Also known as *regular file*. It contains only data as a stream of characters.
- *Directory file*—It's commonly said that a directory contains files and other directories, but strictly speaking, it contains their *names* and a number associated with each name.
- *Device file*—All devices and peripherals are represented by files. To read or write a device, you have to perform these operations on its associated file.

There are other types of files, but we'll stick to these three for the time being. The reason why we make this distinction between file types is that the significance of a file's attributes often depends on its type. Read permission for an ordinary file means something quite different from that for a directory. Moreover, you can't directly put something into a directory file, and a device file isn't really a stream of characters. While many commands work with all types of files, some don't. For a proper understanding of the file system you must understand the significance of these files.

4.1.1 Ordinary (Regular) File

An **ordinary file** or **regular file** is the most common file type. All programs you write belong to this type. An ordinary file itself can be divided into two types:

- Text file
- Binary file

A **text file** contains only printable characters, and you can often view the contents and make sense out of them. All C and Java program sources, shell and **`perl`** scripts are text files. A text file contains lines of characters where every line is terminated with the *newline* character, also known as *linefeed* (LF). When you press *[Enter]* while inserting text, the LF character is appended to every line. You won't see this character normally, but there is a command (**`od`**) which can make it visible.

A **binary file**, on the other hand, contains both printable and unprintable characters that cover the entire ASCII range (0 to 255). Most UNIX commands are binary files, and the object code and executables that you produce by compiling C programs are also binary files. Picture, sound and video files are binary files as well. Displaying such files with a simple **`cat`** command produces unreadable output and may even disturb your terminal's settings.

4.1.2 Directory File

A **directory** contains no data, but keeps some details of the files and subdirectories that it contains. The UNIX file system is organized with a number of directories and subdirectories, and you can also create them as and when you need. You often need to do that to group a set of files pertaining to a specific application. This allows two or more files in separate directories to have the same filename.

A directory file contains an entry for every file and subdirectory that it houses. If you have 20 files in a directory, there will be 20 entries in the directory. Each entry has two components:

- The filename.
- A unique identification number for the file or directory (called the *inode number*).

If a directory bar contains an entry for a file foo, we commonly (and loosely) say that the directory bar contains the file foo. Though we'll often be using the phrase "contains the file" rather than "contains the filename", you must not interpret the statement literally. *A directory contains the filename and not the file's contents.*

You can't write a directory file, but you can perform some action that makes the kernel write a directory. For instance, when you create or remove a file, the kernel automatically updates its corresponding directory by adding or removing the entry (inode number and filename) associated with the file.

Note: The name of a file can only be found in its directory; the file itself doesn't contain its own name or any of its attributes, like its size or access rights.

4.1.3 Device File

You'll also be printing files, installing software from CD-ROMs or backing up files to tape. All of these activities are performed by reading or writing the file representing the device. For instance, when you restore files from tape, you read the file associated with the tape drive. It is advantageous to treat devices as files as some of the commands used to access an ordinary file also work with device files.

Device filenames are generally found inside a single directory structure, /dev. A device file is indeed special; it's not really a stream of characters. *In fact, it doesn't contain anything at all.* You'll soon learn that every file has some attributes that are not stored in the file but elsewhere on disk. The operation of a device is entirely governed by the attributes of its associated file. The kernel identifies a device from its attributes and then uses them to operate the device.

Now that you understand the three types of files, you shouldn't feel baffled by subsequent use of the word in the book. The term "file" will often be used in this book to refer to any of these types, though it will mostly be used to mean an ordinary file. The real meaning of the term should be evident from its context.

4.2 WHAT'S IN A (FILE)NAME?

On most UNIX systems today, a filename can consist of up to 255 characters, though this figure is normally not reached. Files may or may not have extensions, and can consist of practically any ASCII character except the / and the NULL character (ASCII value 0). You are permitted to use control characters or other unprintable characters in a filename. The following are valid filenames in UNIX:

```
.last_time    list.    ^V^B^D-++bcd    -{}[]    @#$%*abcd    a.b.c.d.e
```

The third filename contains three control characters (*[Ctrl-v]* being the first). These characters should definitely be avoided in framing filenames. Moreover, since the UNIX system has a special treatment for characters like $, `, ?, *, & among others, it is recommended that only the following characters be used in filenames:

- Alphabetic characters and numerals.
- The period (.), hyphen (-) and underscore (_).

UNIX imposes no rules for framing filename extensions. A shell script doesn't need to have the .sh extension, even though it helps in identification. In all cases, it's the application that imposes the restriction. Thus the C compiler expects C program filenames to end with .c, Oracle requires SQL scripts to have the .sql extension, and so forth. DOS/Windows users must also keep these two points in mind:

- A file can have as many dots embedded in its name; `a.b.c.d.e` is a perfectly valid filename. A filename can also begin with a dot or end with one.
- UNIX is sensitive to case; `chap01`, `Chap01` and `CHAP01` are three different filenames, and it's possible for them to coexist in the same directory.

Caution: Never use a - at the beginning of a filename. You'll have a tough time getting rid of it! A command that uses a filename as argument often treats it as an option and reports errors. For instance, if you have a file named `-z`, `cat -z` won't display the file but interpret it as an invalid option.

4.3 THE PARENT-CHILD RELATIONSHIP

All files in UNIX are "related" to one another. The file system in UNIX is a collection of all of these related files (ordinary, directory and device files) organized in a hierarchical (an inverted tree) structure. This system has also been adopted by DOS and Windows, and is visually represented in Fig. 4.1.

The implicit feature of every UNIX file system is that there is a top, which serves as the reference point for all files. This top is called *root* and is represented by a / (frontslash). *root is actually a directory.* It is conceptually different from the user-id root used by the system administrator to log in. In this text, we'll be using both the name "root" and the symbol / to represent the root directory.

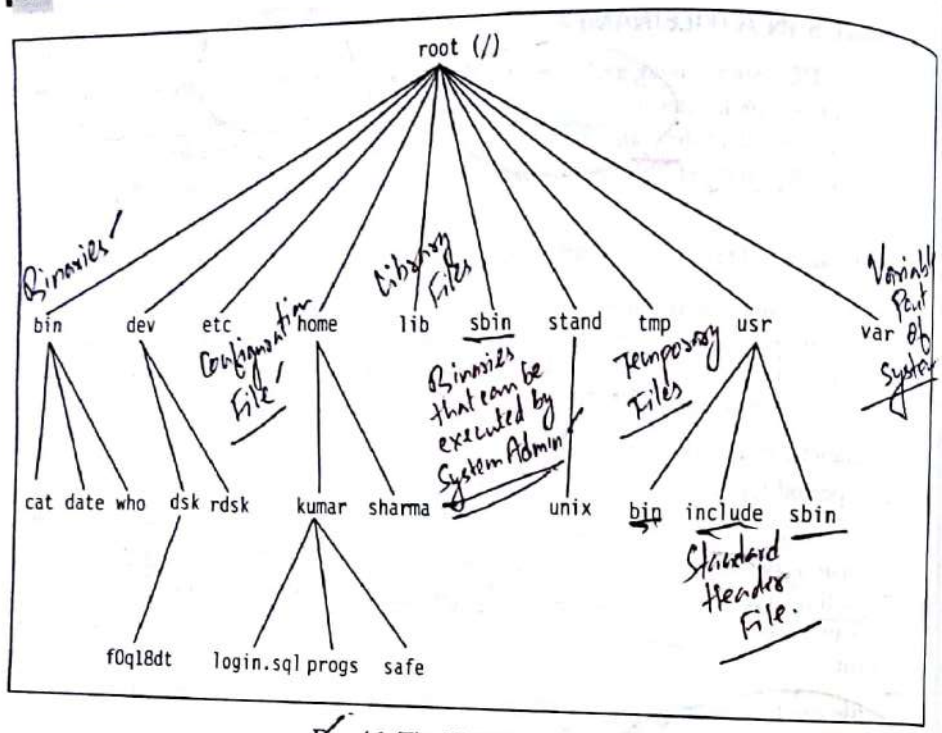

Fig. 4.1 The UNIX File System Tree

The root directory (/) has a number of subdirectories under it. These subdirectories in turn have more subdirectories and other files under them. For instance, bin and usr are two directories directly under /, while a second bin and kumar are subdirectories under usr.

Every file, apart from root, must have a parent, and it should be possible to trace the ultimate parentage of a file to root. Thus, the home directory is the parent of kumar, while / is the parent of home, and the grandparent of kumar. If you create a file login.sql under the kumar directory, then kumar will be the parent of this file.

It's also obvious that, in these parent—child relationships, the parent is always a directory. home and kumar are both directories as they are both parents of at least one file or directory. login.sql is simply an ordinary file; it can't have any directory under it.

4.4 THE HOME VARIABLE: THE HOME DIRECTORY

When you log on to the system, UNIX automatically places you in a directory called the **home directory**. It is created by the system when a user account is opened. If you log in using the login name kumar, you'll land up in a directory that could have the pathname /home/kumar (or something else). You can change your home directory when you like, but you can also effect a quick return to it, as you'll see soon.

The shell variable HOME knows your home directory:

```
$ echo $HOME
/home/kumar
```

First / represents the root directory

What you see above is an **absolute pathname**, which is simply a sequence of directory names separated by slashes. An absolute pathname shows a file's location with reference to the top, i.e., root. These slashes act as delimiters to file and directory names, except that *the first slash is a synonym for root*. The directory kumar is placed two levels below root.

It's often convenient to refer to a file foo located in your home directory as $HOME/foo. Further, most shells (except Bourne) also use the ~ symbol for this purpose. So, $HOME/foo is the same as ~/foo in these shells. The ~ symbol is a little tricky to use because it can refer to any user's home directory and not just your own. If user sharma has the file foo in his home directory, then kumar can access it as ~sharma/foo. The principle is this:

A tilde followed by / (like ~/foo) refers to one's own home directory, but when followed by a string (~sharma) refers to the home directory of that user represented by the string.

Note: The home directory is determined by the system administrator at the time of opening a user account. Its pathname is stored in the file /etc/passwd. On many UNIX systems, home directories are maintained under /home, but your home directory could be different (say, in /export/home). Even if you have moved away from your "home", you can use the cd command to effect a quick return to it, as you'll see soon.

4.5 pwd: CHECKING YOUR CURRENT DIRECTORY

UNIX encourages you to believe that, like a file, a user is *placed* in a specific directory of the file system on logging in. You can move around from one directory to another, but at any point of time, you are located in only one directory. This directory is known as your **current directory**.

At any time, you should be able to know what your current directory is. The **pwd** (print working directory) command tells you that:

```
$ pwd
/home/kumar
```

Like HOME, **pwd** displays the absolute pathname. As you navigate the file system with the **cd** command, you'll be using **pwd** to know your current directory.

Note: It's customary to refer to a file foo located in the home directory as $HOME/foo. Depending on the shell you use, it may be possible to even access foo as ~/foo. One form is shell-dependent but the other isn't, but both naming conventions are followed in this text.

4.6 cd: CHANGING THE CURRENT DIRECTORY

You can move around in the file system by using the **cd** (change directory) command. When used with an argument, it changes the current directory to the directory specified as argument, for instance, progs:

```
$ pwd
/home/kumar
$ cd progs
$ pwd
/home/kumar/progs
```

progs must be in current directory

Though `pwd` displays the absolute pathname, `cd` doesn't need to use one. The command `cd progs` here means this: "Change your subdirectory to progs under the current directory." Using a pathname causes no harm either; use `cd /home/kumar/progs` for the same effect.

When you need to switch to the /bin directory where most of the commonly used UNIX commands are kept, you should use the absolute pathname:

```
$ pwd
/home/kumar/progs
$ cd /bin
$ pwd
/bin
```

Absolute pathname required here because bin isn't in current directory

We can also navigate to /bin (or any directory) using a different type of pathname; we are coming to that shortly.

`cd` can also be used without any arguments:

```
$ pwd
/home/kumar/progs
$ cd
$ pwd
/home/kumar
```

cd used without arguments reverts to the home directory

Attention, DOS users! This command invoked without an argument doesn't indicate the current directory. It simply switches to the home directory, i.e., the directory where the user originally logged into. Therefore, if you wander around in the file system, you can force an immediate return to your home directory by simply using `cd`:

```
$ cd /home/sharma
$ pwd
/home/sharma
$ cd
$ pwd
/home/kumar
```

Returns to home directory

The `cd` command can sometimes fail if you don't have proper permissions to access the directory. This doesn't normally happen unless you deliberately tamper with the permissions of the directory. The technique of doing that is described in Section 6.5.

Note: Unlike in DOS, when `cd` is invoked without arguments, it simply reverts to its home directory. It doesn't show you the current directory!

4.7 mkdir: MAKING DIRECTORIES

Directories are created with the `mkdir` (make directory) command. The command is followed by names of the directories to be created. A directory `patch` is created under the current directory like this:

```
mkdir patch
```

You can create a number of subdirectories with one `mkdir` command:

```
mkdir patch dbs doc
```

Three directories created

So far, simple enough, but the UNIX system goes further and lets you create directory trees with just one invocation of the command. For instance, the following command creates a directory tree:

```
mkdir pis pis/progs pis/data
```

Creates the directory tree

This creates three subdirectories—`pis` and two subdirectories under `pis`. The order of specifying the arguments is important; you obviously can't create a subdirectory before creation of its parent directory. For instance, you can't enter

```
$ mkdir pis/data pis/progs pis
mkdir: Failed to make directory "pis/data"; No such file or directory
mkdir: Failed to make directory "pis/progs"; No such file or directory
```

Note that even though the system failed to create the two subdirectories, `progs` and `data`, it has still created the `pis` directory.

Sometimes, the system refuses to create a directory:

```
$ mkdir test
mkdir: Failed to make directory "test"; Permission denied
```

This can happen due to these reasons:

- The directory `test` may already exist.
- There may be an ordinary file by that name in the current directory.
- The permissions set for the current directory don't permit the creation of files and directories by the user. You'll most certainly get this message if you try to create a directory in /bin, /etc or any other directory that houses the UNIX system's files.

We'll take up file and directory permissions in Chapter 6 featuring file attributes.

4.8 rmdir: REMOVING DIRECTORIES

The `rmdir` (remove directory) command removes directories. You simply have to do this to remove the directory `pis`:

```
rmdir pis
```

Directory must be empty

Like **mkdir**, **rmdir** can also delete more than one directory in one shot. For instance, the three directories and subdirectories that were just created with **mkdir** can be removed by using **rmdir** with a reversed set of arguments:

```
rmdir pis/data pis/progs pis
```

Note that when you delete a directory and its subdirectories, a reverse logic has to be applied. The following directory sequence used by **mkdir** is invalid in **rmdir**:

```
$ rmdir pis pis/progs pis/data
rmdir: directory "pis": Directory not empty
```

Have you observed one thing from the error message? **rmdir** has silently deleted the lowest level subdirectories progs and data. This error message leads to two important rules that you should remember when deleting directories:

- You can't delete a directory *with* **rmdir** unless it is empty. In this case, the pis directory couldn't be removed because of the existence of the subdirectories, progs and data, under it.
- You can't remove a subdirectory unless you are placed in a directory which is hierarchically *above* the one you have chosen to remove.

The first rule follows logically from the example above, but the highlight on **rmdir** has significance that will be explained later. (A directory can also be removed without using **rmdir**.) To illustrate the second cardinal rule, try removing the progs directory by executing the command from the same directory itself:

```
$ cd progs
$ pwd
/home/kumar/pis/progs
$ rmdir /home/kumar/pis/progs
rmdir: directory "/home/kumar/pis/progs": Directory does not exist
```

Trying to remove the current directory

To remove this directory, you must position yourself in the directory above progs, i.e., pis, and then remove it from there:

```
$ cd /home/kumar/pis
$ pwd
/home/kumar/pis
$ rmdir progs
```

The **mkdir** and **rmdir** commands work only in directories *owned* by the user. Generally, a user is the owner of her home directory, and she can create and remove subdirectories (as well as regular files) in this directory or in any subdirectories created by her. However, she normally won't be able to create or remove files and directories in other users' directories. The concept of ownership will be discussed in Section 6.3.

Note: A subdirectory can't be removed with **rmdir** unless it's empty, and one is positioned in its parent directory or above it. But we can remove a directory without using **rmdir** also (discussed later).

HOW IT WORKS: How Files and Directories are Created and Removed

As mentioned in Section 4.1.2, a file (ordinary or directory) is associated with a name and a number, called the *inode number*. When a directory is created, an entry comprising these two parameters is made in the file's parent directory. The entry is removed when the directory is removed. The same holds good for ordinary files also. Figure 4.2 highlights the effect of **mkdir** and **rmdir** when creating and removing the subdirectory progs in /home/kumar.

Filename	*Inode Number*
.	386444
..	417585
foo	499770

mkdir progs ▶

Filename	*Inode Number*
.	386444
..	417585
foo	499770
progs	162112

rmdir progs ▶

Filename	*Inode Number*
.	386444
..	417585
foo	499770

Fig. 4.2 Directory Entry after **mkdir** and **rmdir**

Later in this chapter, we'll discuss the significance of the entries, . and .., that you'll find in every directory. In this chapter and Chapters 5 and 11, we'll be progressively monitoring this directory for changes that are caused by some of the file-handling commands.

4.9 ABSOLUTE PATHNAMES

Many UNIX commands use file and directory names as arguments, which are presumed to exist in the current directory. For instance, the command

```
cat login.sql
```

will work only if the file login.sql exists in your current directory. However, if you are placed in /usr and want to access login.sql in /home/kumar, you can't obviously use the above command, but rather the pathname of the file:

```
cat /home/kumar/login.sql
```

As stated before, if the first character of a pathname is /, the file's location must be determined with respect to root (the first /). Such a pathname, as the one above, is called an **absolute pathname**. When you have more than one / in a pathname, for each such /, you have to descend one level in the file system. Thus, kumar is one level below home, and two levels below root.

When you specify a file by using frontslashes to demarcate the various levels, you have a mechanism of identifying a file uniquely. No two files in a UNIX system can have identical absolute pathnames. You can have two files with the same name, but in different directories; their pathnames will also be different. Thus, the file /home/kumar/progs/c2f.pl can coexist with the file /home/kumar/safe/c2f.pl.

4.9.1 Using the Absolute Pathname for a Command

More often than not, a UNIX command runs by executing its disk file. When you specify the command, the system has to locate the file date from a list of directories specified in the variable, and then execute it. However, if you know the location of a particular command, you precede its name with the complete path. Since **date** resides in /bin (or /usr/bin), you can also the absolute pathname:

```
$ /bin/date
Thu Sep  1 09:30:49 IST 2005
```

Nobody runs the **date** command like that. For any command that resides in the directories specifie in the PATH variable, you don't need to use the absolute pathname. This PATH, you'll recall *(2.4.1)* invariably has the directories /bin and /usr/bin in its list.

If you execute programs residing in some other directory that isn't in PATH, the absolute pathnam then needs to be specified. For example, to execute the program **less** residing in /usr/local/bi you need to enter the absolute pathname:

```
/usr/local/bin/less
```

If you are frequently accessing programs in a certain directory, it's better to include the director itself in PATH. The technique of doing that is shown in Section 10.3.

4.10 RELATIVE PATHNAMES

You would have noted that in a previous example *(4.8)*, we didn't use an absolute pathname t move to the directory progs. Nor did we use one as an argument to **cat** *(4.9)*:

```
cd progs
cat login.sql
```

Here, both progs and login.sql are presumed to exist in the current directory. Now, if progs also contains a directory scripts under it, you still won't need an absolute pathname to change t that directory:

```
cd progs/scripts
```

progs is in current directory

Here we have a pathname that has a /, but it is not an absolute pathname because it doesn't begin with a /. In these three examples, we used a rudimentary form of relative pathnames though they are generally not labeled as such. Relative pathnames, in the sense they are known, are discussed next.

4.10.1 Using . and .. in Relative Pathnames

In a preceding example *(4.8)*, you changed your directory from /home/kumar/pis/progs to its parent directory (/home/kumar/pis) by using **cd** with an absolute pathname:

```
cd /home/kumar/pis
```


Navigation often becomes easier by using a common ancestor (here, /home) as reference. UNIX offers a shortcut—the **relative pathname**—that uses either the current or parent directory as reference, and specifies the path relative to it. A relative pathname uses one of these cryptic symbols:

- . (a single dot)—This represents the current directory.
- .. (two dots)—This represents the parent directory.

We'll now use the .. to frame relative pathnames. Assuming that you are placed in /home/kumar/progs/data/text, you can use .. as an argument to **cd** to move to the parent directory, /home/kumar/progs/data:

```
$ pwd
/home/kumar/progs/data/text
$ cd ..                                   Moves one level up
$ pwd
/home/kumar/progs/data
```

This method is compact and more useful when ascending the hierarchy. The command **cd ..** translates to this: "Change your directory to the parent of the current directory." You can combine any number of such sets of .. separated by /s. However, when a / is used with .. it acquires a different meaning; instead of moving down a level, it moves one level *up*. For instance, to move to /home, you can always use **cd /home**. Alternatively, you can also use a relative pathname:

```
$ pwd
/home/kumar/pis
$ cd ../..                                Moves two levels up
$ pwd
/home
```

Now let's turn to the solitary dot that refers to the current directory. Any command which uses the current directory as argument can also work with a single dot. This means that the **cp** command *(5.2)* which also uses a directory as the last argument can be used with a dot:

```
cp ../sharma/.profile .                   A filename can begin with a dot
```

This copies the file .profile to the current directory (.). Note that you didn't have to specify the filename of the copy; it's the same as the original one. This dot is also implicitly included whenever we use a filename as argument, rather than a pathname. For instance, **cd progs** is the same as **cd ./progs**.

Note: Absolute pathnames can get very long if you are located a number of "generations" away from root. However, whether you should use one depends solely on the number of keystrokes required when compared to a relative pathname. In every case here, the relative pathname required fewer key depressions. Depending on where you are currently placed, an absolute pathname can be faster to type.

4.11 ls: LISTING DIRECTORY CONTENTS

You have already used the **ls** command *(1.4.9)* to obtain a list of all filenames in the current directory. Let's execute it again:

```
$ ls
08_packets.html
TOC.sh
calendar
cptodos.sh
dept.lst
emp.lst
helpdir
progs
usdsk06x
usdsk07x
usdsk08x
```

Numerals first
Uppercase next
Then lowercase

What you see here is a complete list of filenames in the current directory arranged in ASCII collating sequence (numbers first, uppercase and then lowercase), with one filename in each line. It includes directories also, and if you are using Linux, you would probably see the directories and ordinary files in different colors.

LINUX: If your Linux system doesn't show these colors, make sure that you create this *alias* after logging in:

```
alias ls='ls --color=tty'
```

Aliases are discussed in Section 10.4, but note that they are not supported by the Bourne shell.

Directories often contain many files, and you may simply be interested in only knowing whether a particular file is available. In that case, just use `ls` with the filename:

```
$ ls calendar
calendar
```

and if `perl` isn't available, the system clearly says so:

```
$ ls perl
perl: No such file or directory
```

`ls` can also be used with multiple filenames, and has options that list most of the file attributes. In the following sections, you'll see some of these options.

4.11.1 ls Options

`ls` has a large number of options (Table 4.1), but in this chapter, we'll present a handful of them. The other options will be taken up in later chapters. The section numbers are appropriately indicated in the table.

Output in Multiple Columns (-x) When you have several files, it's better to display the filenames in multiple columns. Modern versions of `ls` do that by default (i.e., when used without options), but if that doesn't happen on your system, use the `-x` option to produce a multicolumnar output:

```
$ ls -x
```



```
08_packets.html  TOC.sh      calendar   cptodos.sh
dept.lst         emp.lst     helpdir    progs
usdsk06x         usdsk07x    usdsk08x   ux2nd06
```

If your system needs to use the `-x` option to display multicolumnar output, you can later customize the command to display in this format by default *(10.4)*.

Identifying Directories and Executables (-F) The output of `ls` that you have seen so far merely showed the filenames. You didn't know how many of them, if any, were directory files. To identify directories and executable files, the `-F` option should be used. Combining this option with `-x` produces a multicolumnar output as well:

executable code

directory

```
$ ls -Fx
08_packets.html  TOC.sh*     calendar*  cptodos.sh*
dept.lst         emp.lst     helpdir/   progs/
usdsk06x         usdsk07x    usdsk08x   ux2nd06
```

Note the use of two symbols, * and /, as type indicators. The * indicates that the file contains executable code and the / refers to a directory. You can now identify the two subdirectories in the current directory—`helpdir` and `progs`.

Showing Hidden Files Also (-a) `ls` doesn't normally show all files in a directory. There are certain hidden files (filenames beginning with a dot), often found in the home directory, that normally don't show up in the listing. The `-a` option (all) lists all hidden files as well:

```
$ ls -axF
./               ../              .exrc      .kshrc
.profile         .rhosts          .sh_history .xdtsupCheck
xinitrc          08_packets.html* TOC.sh*    calendar*
.....
```

The file `.profile` contains a set of instructions that are performed when a user logs in. It is conceptually similar to `AUTOEXEC.BAT` of DOS, and you'll know more about it later. The other file, `.exrc`, contains a sequence of startup instructions for the `vi` editor. To display these hidden filenames, you can either use the `-a` option or specify the filenames in the command line.

The first two files (`.` and `..`) are special directories. Recall that we used the same symbols in relative pathnames to represent the current and parent directories *(4.10.1)*. These symbols have the same meaning here. Whenever you create a subdirectory, these "invisible" directories are created automatically by the kernel. You can't remove them, nor can you write into them. They help in holding the file system together.

Note: All filenames beginning with a dot are displayed only when `ls` is used with the `-a` option. The directory `.` represents the current directory and `..` signifies the parent directory.

Listing Directory Contents In the last example, you specified some ordinary filenames to `ls` to have a selective listing. However, the situation will be quite different if you specify the two directory names, `helpdir` and `progs`, instead:

```
$ ls -x helpdir progs
helpdir:
forms.obd    graphics.obd    reports.obd

progs:
array.pl     cent2fah.pl     n2words.pl     name.pl
```

This time the *contents* of the directories are listed, consisting of the Oracle documentation in the helpdir directory and a number of **perl** program files in progs. Note that **ls**, when used with directory names as arguments, doesn't simply show their names as it does with ordinary files.

Recursive Listing (-R) The -R (recursive) option lists all files and subdirectories in a directory tree. Similar to the **DIR /S** command of DOS, this traversal of the directory tree is done recursively until there are no subdirectories left:

```
$ ls -xR
08_packets.html  TOC.sh          calendar        cptodos.sh
dept.lst         emp.lst         helpdir         progs
usdsk06x         usdsk07x        usdsk08x        ux2nd06

./helpdir:
forms.hlp        graphics.hlp    reports.hlp

./progs:
array.pl         cent2fah.pl     n2words.pl      name.pl
```

The list shows the filenames in three sections—the ones under the home directory and those under the subdirectories helpdir and progs. Note the subdirectory naming conventions followed. ./helpdir indicates that helpdir is a subdirectory under . (the current directory). Since /home/kumar happens to be the current directory, the absolute pathname of this file expands to /home/kumar/helpdir.

Table 4.1 Options to **ls**

Option	*Description*
-x	Multicolumnar output
-F	Marks executables with *, directories with / and symbolic links with @
-a	Shows all filenames beginning with a dot including . and ..
-R	Recursive list
-r	Sorts filenames in reverse order (ASCII collating sequence by default)
-l	Long listing in ASCII collating sequence showing seven attributes of a file *(6.1)*
-d *dirname*	Lists only *dirname* if *dirname* is a directory *(6.2)*
-t	Sorts filenames by last modification time *(11.6)*
-lt	Sorts listing by last modification time *(11.6)*
-u	Sorts filenames by last access time *(11.6)*
-lu	Sorts by ASCII collating sequence but listing shows last access time *(11.6)*
-lut	As above but sorted by last access time *(11.6)*
-i	Displays inode number *(11.1)*

4.12 THE UNIX FILE SYSTEM

We have learned to use the basic command set for handling files and directories. Let's conclude this chapter by taking a cursory look at the structure of the UNIX file system. This structure has been changing constantly over the years until AT&T proposed one in its SVR4 release. Though vendor implementations vary in detail, broadly the SVR4 structure has been adopted by most vendors.

Refer to Fig. 4.1 which shows a heavily trimmed structure of a standard UNIX file system. In real life, the root directory has many more subdirectories under it than shown, but for our initial comprehension, we'll stick to the ones presented below. It helps, from the administrative point of view at least, to view the entire file system as comprising two groups of files. The first group contains the files that are made available during system installation:

- /bin and /usr/bin—These are the directories where all the commonly used UNIX commands (binaries, hence the name bin) are found. Note that the PATH variable always shows these directories in its list.
- /sbin and /usr/sbin—If there's a command that you can't execute but the system administrator can, then it would probably be in one of these directories. You won't be able to execute most (some, you can) commands in these directories. Only the system administrator's PATH shows these directories.
- /etc—This directory contains the configuration files of the system. You can change a very important aspect of system functioning by editing a text file in this directory. Your login name and password are stored in files /etc/passwd and /etc/shadow.
- /dev—This directory contains all device files. These files don't occupy space on disk. There could be more subdirectories like pts, dsk and rdsk in this directory.
- /lib and /usr/lib—Contain all library files in binary form. You'll need to link your C programs with files in these directories.
- /usr/include—Contains the standard header files used by C programs. The statement #include <stdio.h> used in most C programs refers to the file stdio.h in this directory.
- /usr/share/man—This is where the man pages are stored. There are separate subdirectories here (like man1, man2, etc.) that contain the pages for each section. For instance, the man page of ls can be found in /usr/share/man/man1, where the 1 in man1 represents Section 1 of the UNIX manual. These subdirectories may have different names on your system (like sman1, sman2, etc. in Solaris).

Over time, the contents of these directories would change as more software and utilities are added to the system. Users also work with their own files; they write programs, send and receive mail and also create temporary files. These files are available in the second group shown below:

- /tmp—The directories where users are allowed to create temporary files. These files are wiped away regularly by the system.
- /var—The variable part of the file system. Contains all your print jobs and your outgoing and incoming mail.

- /home—On many systems users are housed here. kumar would have his home directory /home/kumar. However, your system may use a different location for home directories.

On a busy system, it's in directories belonging to the second group that you could experience rapid depletion of available disk space. You'll learn later to house some of these directory structures in separate *file systems* so that depletion of space in one file system doesn't affect other file systems. File system internals and administration are taken up toward the end of this text.

4.13 CONCLUSION

Though UNIX is known to make little distinction between the various types of files, that was really established in this chapter. You used exclusive commands to handle directories (like **pwd**, **cd**, **mkdir** and **rmdir**). These commands have no relevance when applied to ordinary or device files. It appears that UNIX does care to some extent about the type of file it handles. In the next chapter we look at ordinary files using yet another set of commands meant for them.

WRAP UP

Files have been assumed to be of three types. An *ordinary* file contains what you put into it. A *directory* maintains the filename and its associated *inode number*. A *device* file contains no data but the kernel uses the attributes of the file to operate the device.

Executable files don't need any specific extensions. A file doesn't contain its attributes nor does it contain the end-of-file mark.

The file system is a hierarchical structure, and the top-most directory is called *root*. Files and directories have a parent—child relationship.

pwd tells you the current directory, and **cd** is used to change it. When used by itself, it switches to the *home* directory. The home directory is available in the shell variable HOME. A file foo in the home directory is often referred to as $HOME/foo or ~/foo.

mkdir and **rmdir** are used to create or remove directories. To remove a directory bar with **rmdir**, bar must be empty and you must be positioned above bar.

A pathname is a sequence of directory and filenames separated by slashes. An *absolute* pathname begins with a / and denotes the file's location with respect to root. A *relative* pathname uses the symbols . and .. to represent the file's location relative to the current and parent directory, respectively.

By default, **ls** displays a list of filenames in *ASCII collating sequence*, which accords priority in this sequence—numbers, uppercase, lowercase. It can also display hidden filenames beginning with a dot (-a) and a recursive list (-R). However, when used with a directory name as argument, **ls** displays the filenames in the directory.

Test Your Understanding

4.1 How long can a UNIX filename be? Which characters can't be used in a filename?

4.2 State two reasons for not having a filename beginning with a hyphen.

4.3 Can the files note and Note coexist in the same directory?

4.4 In how many ways can you find out what your home directory is?

4.5 Switch to the root directory with **cd** and then run **cd ..** followed by **pwd**. What do you notice?

4.6 What is the easiest way of changing from /var/spool/lp/admins to /var/spool/mail?

4.7 Explain the significance of these two commands: **ls .. ; ls -d ..**

4.8 Look up the man page of **mkdir** to find out the easiest way of creating this directory structure: share/man/cat1

4.9 If **rmdir c_progs** fails, what could be the possible reasons?

4.10 If the file /bin/echo exists on your system, are the commands **echo** and **/bin/echo** equivalent?

4.11 How do you run **ls** to (i) mark directories and executables separately, (ii) display also hidden files?

4.12 How will you obtain a complete listing of all files and directories in the whole system?

Flex Your Brain

4.1 Name the two types of ordinary files and explain the difference between them. Provide three examples of each type of file.

4.2 How does the device file help in accessing the device?

4.3 Which of these commands will work? Explain with reasons: (i) **mkdir a/b/c** (ii) **mkdir a a/b** (iii) **rmdir a/b/c** (iv) **rmdir a a/b** (v) **mkdir /bin/foo**

4.4 If **mkdir test** fails, what could be the possible reasons?

4.5 Which of these files or directories can you create? Explain with reasons: ., .., ... and ....

4.6 The command **rmdir bar** fails with the message that the directory is not empty. On running **ls bar**, no files are displayed. Why did the **rmdir** command fail?

4.7 Suppose you have to develop a script that refers to a file in charlie's home directory. How will you specify the location of this file in your script to make sure that it works even when charlie's home directory changes?

4.8 Explain the difference between the commands **cd ~charlie** and **cd ~/charlie**. Is it possible for both commands to work?

4.9 Why do we sometimes run a command like this — **./update.sh** instead of **update.sh**?

4.10 What is the sort order prescribed by the ASCII collating sequence?

4.11 Assuming that you are positioned in the directory /home/kumar, what are these commands presumed to do and explain whether they will work at all: (i) **cd ../..** (ii) **mkdir ../bin** (iii) **rmdir ..** (iv) **ls ..**

5 Handling Ordinary Files

The last chapter examined the tools that handle directories. But users actually do most of their work with ordinary (or regular) files, and it's natural that the UNIX system should feature a host of commands to handle them. Although all of these commands use filenames as arguments, they were not designed *only* to read files. In fact, many of them don't need to read a file at all. However, to understand their basic functionality, we'll use them with filenames in this chapter.

We'll first consider the common file-handling commands that the DOS environment also offers, except that the UNIX variety has more features. We'll also discuss those commands that show differences between two files and convert files between DOS and UNIX formats. Finally, we examine the important compression utilities with which we handle documents and software found on the Internet. As we progressively discover the shell's features in later chapters, we'll learn to use the same commands in other ways.

What You Will Learn

- View text files with **cat** and **more** (or **less**)
- Use **cat** to create a file.
- The essential file functions—copy with **cp**, remove with **rm** and rename with **mv**.
- Print a file with **lp** (**lpr** in Linux).
- Classify files with **file**.
- Count the number of lines, words and characters with **wc**.
- Display the ASCII octal value of text with **od**.
- Compare two files with **cmp**, **comm** and **diff**.
- Compress and decompress files with **gzip** and **gunzip**.
- Create an *archive* comprising multiple files with **tar**.
- Perform both functions (compressing and archiving) with **zip** and **unzip**.

Topics of Special Interest

- A discussion on the issues related to file compression and archival.
- Convert between UNIX and DOS files with **unix2dos** and **dos2unix**.
- How It Works: A graphic showing how a directory is affected by **cp**, **mv** and **rm**.

5.1 cat: DISPLAYING AND CREATING FILES

cat is one of the most well-known commands of the UNIX system. It is mainly used to display the contents of a small file on the terminal:

```
$ cat dept.lst
01|accounts|6213
02|progs|5423
03|marketing|6521
04|personnel|2365
05|production|9876
06|sales|1006
```

cat, like several other UNIX commands, also accepts more than one filename as arguments:

```
cat chap01 chap02
```

The contents of the second file are shown immediately after the first file without any header information. In other words, **cat** concatenates the two files—hence its name.

5.1.1 cat Options (-v and -n)

There are two **cat** options that you may find useful, though POSIX doesn't require **cat** to support either one.

Displaying Nonprinting Characters (-v) **cat** is normally used for displaying text files only. Executables, when seen with **cat**, simply display junk. If you have nonprinting ASCII characters in your input, you can use **cat** with the **-v** option to display these characters.

Numbering Lines (-n) The **-n** option numbers lines. C compilers indicate the line number where errors are detected, and this numbering facility often helps a programmer in debugging programs. But then your **vi** editor can show line numbers too, and if your version of **cat** doesn't support **-n**, you can use the **pr** *(12.2)* command to do the same job.

5.1.2 Using cat to Create a File

cat is also useful for creating a file. Though the significance of the following sequence can be appreciated only after reading Section 8.5.2, you should now know how to create small files. Enter the command **cat**, followed by the > (the right chevron) character and the filename (for example, **foo**):

```
$ cat > foo
A > symbol following the command means that the
output goes to the filename following it. cat used
in this way represents a rudimentary editor.
[Ctrl-d]                                    Prompt returns
$ _
```

When the command line is terminated with *[Enter]*, the prompt vanishes. **cat** now waits to take input from the user. Enter the three lines, each followed by *[Enter]*. Finally press *[Ctrl-d]* to signify the end of input to the system. This is the eof character used by UNIX systems and is shown in the

stty output *(3.13)*. When this character is entered, the system understands that no further text input will be made. The file is written and the prompt returned. To verify this, simply "cat" this file:

```
$ cat foo
A > symbol following the command means that the
output goes to the filename following it, cat used
in this way represents a rudimentary editor.
```

Note: The *Ctrl-d* character is used to terminate input not only with **cat**, but with all commands that accept input from the keyboard.

cat is a versatile command. It can be used to create, display, concatenate and append to files. More importantly, it doesn't restrict itself to handling files only; it also acts on a *stream*. You can supply the input to **cat** not only by specifying a filename, but also from the output of another command. You'll learn about all this in Chapter 8.

5.2 cp: COPYING A FILE

The **cp** (copy) command copies a file or a group of files. It creates an exact image of the file on disk with a different name. The syntax requires at least two filenames to be specified in the command line. When both are ordinary files, the first is copied to the second:

```
cp chap01 unit1
```

If the destination file (`unit1`) doesn't exist, it will first be created before copying takes place. If not, it will simply be overwritten without any warning from the system. So be careful when you choose your destination filename. Just check with the `ls` command whether or not the file exists.

If there is only one file to be copied, the destination can be either an ordinary or directory. You then have the option of choosing your destination filename. The following example shows two ways of copying a file to the `progs` directory:

```
cp chap01 progs/unit1        chap01 copied to unit1 under progs
cp chap01 progs              chap01 retains its name under progs
```

cp is often used with the shorthand notation, . (dot), to signify the current directory as the destination. For instance, to copy the file `.profile` from `/home/sharma` to your current directory, you can use either of the two commands:

```
cp /home/sharma/.profile .profile     Destination is a file
cp /home/sharma/.profile .            Destination is the current directory
```

Obviously, the second one is preferable because it requires fewer keystrokes.

cp can also be used to copy more than one file with a single invocation of the command. In that case, the last filename *must* be a directory. For instance, to copy the files `chap01`, `chap02` and `chap03` to the `progs` directory, you have to use **cp** like this:

```
cp chap01 chap02 chap03 progs
```


The files retain their original names in `progs`. If these files are already resident in `progs`, they will be overwritten. For the above command to work, the `progs` directory must exist because **cp** won't create it.

You have already seen *(5.3)* how the UNIX system uses the * to frame a pattern for matching more than one filename. If there were only three filenames in the current directory having the common string `chap`, you can compress the above sequence using the * as a suffix to `chap`:

```
cp chap* progs          Copies all files beginning with chap
```

We'll continue to use the * as a shorthand for multiple filenames. There are other metacharacters too, and they are discussed in complete detail in Section 8.3.

Note: In the previous example, **cp** doesn't look for a file named `chap*`. Before it runs, the shell expands `chap*` to regenerate the command line arguments for **cp** to use.

Caution: **cp** overwrites without warning the destination file if it exists! Run **ls** before you use **cp** unless you are sure that the destination file doesn't exist or deserves to be overwritten.

5.2.1 cp Options

Interactive Copying (-i) The `-i` (interactive) option warns the user before overwriting the destination file. If `unit1` exists, **cp** prompts for a response:

```
$ cp -i chap01 unit1
cp: overwrite unit1 (yes/no)? y
```

A `y` at this prompt overwrites the file, any other response leaves it uncopied.

Copying Directory Structures (-R) Many UNIX commands are capable of **recursive** behavior. This means that the command can descend a directory and examine all files in its subdirectories. The **cp -R** command behaves recurvely to copy an entire directory structure, say, `progs` to `newprogs`:

```
cp -R progs newprogs          newprogs must not exist
```

Attention! How **cp** behaves here depends on whether `newprogs` also exists as a directory. If `newprogs` doesn't exist, **cp** creates it along with the associated subdirectories. But if `newprogs` exists, `progs` becomes a subdirectory under `newprogs`. This means that the command run twice in succession will produce different results!

Caution: Sometimes, it's not possible to copy a file. This can happen if it's read-protected or the destination file or directory is write-protected. File permissions are discussed in Section 6.4.

5.3 rm: DELETING FILES

The **rm** (remove) command deletes one or more files. It normally operates silently and should be used with caution. The following command deletes three files:

rm chap could be dangerous to use!*

```
rm chap01 chap02 chap03
```

A file once deleted can't be recovered. `rm` won't *normally* remove a directory, but it can remove files from one. You can remove two chapters from the progs directory without having to "cd" to it:

Or `rm progs/chap0[12]`

```
rm progs/chap01 progs/chap02
```

You may sometimes need to delete all files in a directory as part of a cleanup operation. The *, when used by itself, represents all files, and you can then use `rm` like this:

All files gone!

```
$ rm *
$ _
```

DOS users, beware! When you delete files in this fashion, the system won't prompt you with the message `All files in directory will be deleted!` before removing the files! The $ prompt will return silently; the work has been done. The * used here is equivalent to *.* used in DOS.

Note: Whether or not you are able to remove a file depends, not on the file's permissions, but on the permissions you have for the *directory*. Directory permissions are taken up in Section 6.6.

5.3.1 rm Options

Interactive Deletion (-i) Like in **cp**, the `-i` (interactive) option makes the command ask the user for confirmation before removing each file:

No response—file not deleted

```
$ rm -i chap01 chap02 chap03
rm: remove chap01 (yes/no)? ?y
rm: remove chap02 (yes/no)? ?n
rm: remove chap03 (yes/no)? [Enter]
```

`y` removes the file, any other response leaves the file undeleted.

Recursive Deletion (-r or -R) With the `-r` (or `-R`) option, `rm` performs a tree walk—a thorough recursive search for all subdirectories and files within these subdirectories. At each stage, it deletes everything it finds. *`rm` won't normally remove directories, but when used with this option, it will.* Therefore, when you issue the command

Behaves partially like `rmdir`

```
rm -r *
```

you'll delete all files in the current directory and all its subdirectories. If you don't have a backup, then these files will be lost forever.

Forcing Removal (-f) `rm` prompts for removal if a file is write-protected. The `-f` option overrides this minor protection and forces removal. When you combine it with the `-r` option, it could be the most risky thing to do:

Deletes everything in the current directory and below

```
rm -rf *
```


Caution: Make sure you are doing the right thing before you use `rm *`. Be doubly sure before you use `rm -rf *`. The first command removes only ordinary files in the current directory. The second one removes everything—files and directories alike. If the root user (the super user) invokes `rm -rf *` in the / directory, the entire UNIX system will be wiped out from the hard disk!

5.4 mv: RENAMING FILES

The `mv` command renames (moves) files. It has two distinct functions:

- It renames a file (or directory).
- It moves a group of files to a different directory.

`mv` doesn't create a copy of the file; it merely renames it. No additional space is consumed on disk during renaming. To rename the file `chap01` to `man01`, you should use

```
mv chap01 man01
```

If the destination file doesn't exist, it will be created. For the above example, `mv` simply replaces the filename in the existing directory entry with the new name. By default, `mv` doesn't prompt for overwriting the destination file if it exists. So be careful again.

Like **cp**, a group of files can be moved to a directory. The following command moves three files to the progs directory:

Directory
Files

```
mv chap01 chap02 chap03 progs
```

`mv` can also be used to rename a directory, for instance, `pis` to `perdir`:

```
mv pis perdir
```

Like in **cp** `-R`, there's a difference in behavior depending on whether `perdir` exists or not. You can check that out for yourself.

There's a `-i` option available with `mv` also, and behaves exactly like in **cp**. The messages are the same and require a similar response.

HOW IT WORKS: How a Directory is Affected by cp, mv and rm

cp, `mv` and `rm` work by modifying the directory entries of the files they access. As shown in Fig. 5.1, **cp** adds an entry to the directory with the name of the destination file and inode number that is allotted by the kernel. `mv` replaces the name of an existing directory entry without disturbing its inode number. `rm` removes from an entry from the directory.

This is a rather simplistic view, and is true only when source and destination are in the same directory. When you "mv" a file to a directory that resides on a separate hard disk, the file is *actually* moved. You'll appreciate this better after you have understood how multiple file systems create the illusion of a single file system on your UNIX machine.

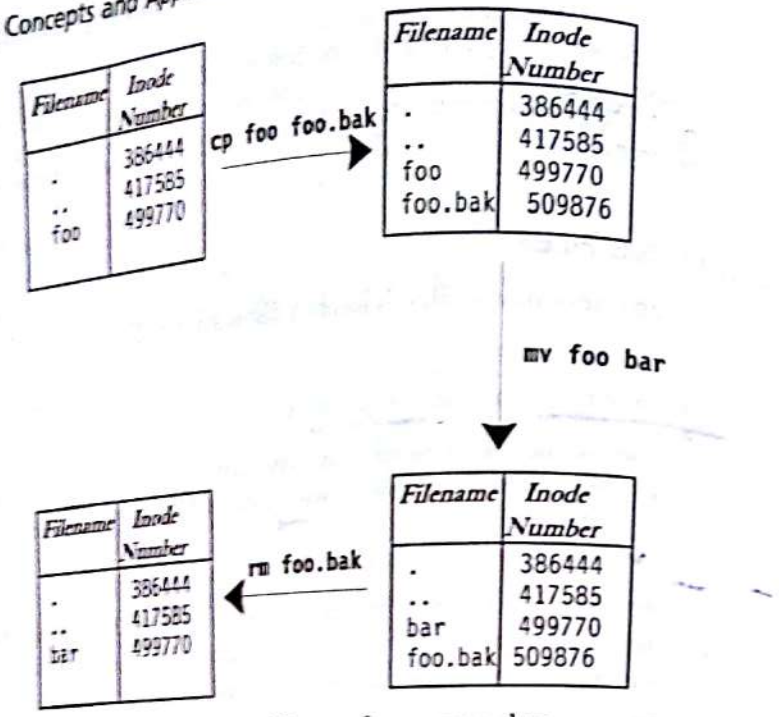

Fig. 5.1 Directory Status after **cp**, **mv** and **rm**

The action of **rm** also needs to be studied further. A file is not actually removed by deleting its direc entry. There could be "similar" entries (ones having the same inode number) for this file in this another directory. We'll examine this directory table again along with its permissions when we take file attributes in Chapter 11.

5.5 more: PAGING OUTPUT

The **man** command displays its output a page at a time. This is possible because it sends its output a pager program. UNIX offers the **more** pager (originally from Berkeley) which has today replac **pg**, the original pager of UNIX. Linux also offers **more** but **less** is its standard pager. We'll discu more in this section and note the exclusive features of **less** separately in the aside on Linux.

To view the file **chap01**, enter the command with the filename:

```
more chap01
```

Press **q** *to exit*

You'll see the contents of chap01 on the screen, one page at a time. At the bottom of the screen, you also see the filename and percentage of the file that has been viewed:

```
--More--(17%)
```

more has a couple of internal commands that don't show up on the screen when you invoke them. the command used to exit **more**, is an internal command.

The AT&T and BSD versions of **more** differ widely in their capabilities and command usage. The POSIX specification on **more** is based on the BSD version. You have to try out the commands shown in Table 5.1, as well as look up the man pages, to know whether they apply to your system. **more** has a fairly useful help screen too; hitting an **h** invokes this screen.

5.5.1 Navigation

Irrespective of version, **more** uses the spacebar to scroll forward a page at a time. You can also scroll by small and large increments of lines or screens. To move forward one page, use

```
f or the spacebar
```

and to move back one page, use

```
b
```

5.5.2 The Repeat Features

The Repeat Factor Many navigation commands in **more**, including **f** and **b**, use a *repeat factor*. This is the term used in **vi** *(7.1.1)* to prefix a number to a **vi** internal command. Use of the repeat factor as a command prefix simply repeats the command that many times. This means you can use **10f** for scrolling forward by 10 pages and **30b** for scrolling back 30 pages. Just remember that the commands themselves are not displayed on the screen—even for a moment.

Repeating The Last Command (.) **more** has a repeat command, the dot (same command used by **vi**), that repeats the last command you used. If you scroll forward with **10f**, you can scroll another 10 pages by simply pressing a dot. This is a great convenience available in **more**!

5.5.3 Searching for a Pattern

You can perform a search for a pattern with the **/** command followed by the string. For instance, to look for the first **while** loop in your program, you'll have to enter this:

Press [Enter] also

```
/while
```

You can repeat this search for viewing the next **while** loop section by pressing **n**, and you can do that repeatedly until you have scanned the entire file. Move back with **b** (using a repeat factor, if necessary) to arrive at the first page.

Note: The search capability in **more** is not restricted to simple strings. Like many UNIX commands (**grep**, **sed** and **vi**), **more** lets you use a *regular expression* to match multiple similar strings. Regular expressions are discussed in several chapters of this text beginning with Chapter 13.

5.5.4 Using more in a Pipeline

The **man** syntax doesn't indicate this (except mention that **more** is a filter), but we often use **more** to page the output of another command. The **ls** output won't fit on the screen if there are too many files, so the command has to be used like this:

No filename with more*!*

```
ls | more
```

We have a *pipeline* here of two commands where the output of one is used as the input of the other. Pipes are a feature of the shell and the topic is taken up in Chapter 8.

Tip: You can start up the **vi** editor from inside **more** by pressing **v**. When you finish editing, exit from **vi** returns you to **more** once again.

LINUX: less—The Standard Pager

Even though every Linux system also offers **more**, **less** is its standard pager. It's ironic that it bears such a name because it's more powerful than **more**. **less** is modeled on the **vi** editor, so learning **less** should be a breeze for **vi** users.

Navigation is **vi**-compatible. You have to work with these keys:

f, *[Ctrl-f]* or spacebar	Scroll forward one screen
b or *[Ctrl-b]*	Scroll backward one screen
j	One line up
k	One line down

Pattern searching techniques are similar. Unlike **more**, **less** can search for a pattern in the reverse direction also using the sequence ?*pattern*. But **less** does have one serious limitation. Unlike **more** (which uses the **.**), it can't repeat the last command.

Table 5.1 Internal Commands of **more** and **less**

more	less	Action
Spacebar or f	Spacebar or f or z	One page forward
20f	-	20 pages forward
b	b	One page back
15b	-	15 pages back
[Enter]	j or *[Enter]*	One line forward
-	k	One line back
-	p or 1G	Beginning of file
-	G	End of file
/pat	*/pat*	Searches forward for expression *pat*
n	n	Repeats search forward
-	?*pat*	Searches back for expression *pat*
. (a dot)	-	Repeats last command
v	v	Starts up vi editor
!*cmd*	!*cmd*	Executes UNIX command *cmd*
q	q	Quit
h	h	View Help

5.6 THE lp SUBSYSTEM: PRINTING A FILE

No user is allowed direct access to the printer. Instead, one has to *spool* (line up) a job along with others in a print queue. Spooling ensures the orderly printing of jobs and relieves the user from the necessity of administering the print resources. The spooling facility in System V is provided by the **lp** (line printing) command. Systems derived from BSD (like Linux) use the **lpr** command instead.

The following **lp** command prints a single copy of the file `rfc822.ps` (a document containing an Internet specification in the form of a Request For Comment):

```
$ lp rfc822.ps                                   A Postscript file
request id is pr1-320 (1 file)
$ _
```

Note that the prompt is returned immediately after the job is submitted. The file isn't actually printed at the time the command is invoked, but later, depending on the number of jobs already lined up in the queue. Several users can print their files in this way without conflict.

lp notifies the request-id—a combination of the printer name (pr1) and the job number (320)—which can later be accessed with other commands. The hard copy of the file is often preceded by a title page mentioning the username, request-id and date.

5.6.1 lp Options

lp accepts the above request because a default printer has been defined by the administrator. If it is not, or if there is more than one printer in the system, you have to use the `-d` option with the printer name (say, laser):

```
lp -dlaser chap01.ps                    Can also provide space after -d
```

The `-t` (title) option, followed by the title string, prints the title on the first page:

```
lp -t"First chapter" chap01.ps
```

After the file has been printed, you can notify the user with the `-m` (mail) option. You can also print multiple copies (`-n`):

```
lp -n3 -m chap01.ps                    Prints three copies and mails user a message
```

Even though we used **lp** with filenames, this will not always be the case. You are aware that the shell's | symbol allows us to use **ls | more**. The same symbol also lets us use **ls | lp**.

5.6.2 Other Commands in the lp Subsystem

The print queue is viewed with the **lpstat** command. By viewing this list, you can use the **cancel** command to cancel any jobs submitted by you. **cancel** uses the request-id or printer name as argument:

```
cancel laser                    Cancels current job on printer laser
cancel pr1-320                  Cancels job with request-id pr1-320
```

You can cancel only those jobs that you own (i.e., you have submitted yourself), but the system administrator can cancel any job. **cancel** is effective only when a job remains in the print queue. If it is already being printed, **cancel** can't do a thing.

Note: Most UNIX printers are of the Postscript variety, i.e., they can properly print files formatted in Postscript, like the files *ch02.ps* and *chap01.ps* used in the examples. (Postscript files are easily identified by the extension *.ps*.) When you select *Print* from the *File* menu of any GUI program, the program converts the data to Postscript which serve as input to the printer. No such conversion takes place when you use **lp** to print a text file like */etc/passwd*. If you have a text file to print, use a Postscript conversion utility before you use **lp**. On Solaris, you can use the program */usr/bin/lp/postscript/postprint* before running **lp**.

LINUX: Printing with lpr, lpq and lprm

Linux uses Berkeley's printing system which supports the **lpr** command for printing. You must have your printer configured properly before you can use **lpr**. The command normally doesn't throw out the job number:

```
lpr /etc/group
```

Linux has a rich set of tools that convert text files to Postscript. Check whether you have the programs a2ps or enscript on your system. Both eventually call up **lpr**; you don't have to do that on your own.

Like in System V, you can print a specific number of copies, choose the title and direct output to a specific printer. You can also mail completion of the job:

```
lpr -P hp4500 foo.ps          Prints on printer hp4500
lpr -J "The List of 3Cs" foo.ps   Uses this title
lpr -#3 foo.ps                Prints 3 copies
lpr -m foo.ps                 Mails message after completion
```

lpq displays the print queue showing job numbers. Using one or more job numbers as arguments to **lprm**, you can remove from the print queue only those jobs owned by you:

```
lprm 32       Removes job number 31
lprm -        Removes all jobs owned by user
```

The **lpc** command is used by the administrator to configure the printer. We'll not discuss printer administration in this text.

5.7 file: KNOWING THE FILE TYPES

Even though we know (so far) that files are of three types, you may often need to know more about these files. For instance, a regular file may contain plain text, a C program or executable code. UNIX provides the **file** command to determine the type of file, especially of an ordinary file. You can use it with one or more filenames as arguments:

```
$ file archive.zip
archive.zip:    ZIP archive
```


file correctly identifies the basic file types (regular, directory or device). For a regular file, it attempts to classify it further. Using the * to signify all files, this is how **file** behaves on this system having regular files of varying types:

```
$ file *
CW6.Eval.exe:   DOS executable (EXE)
User_Guide.ps:  PostScript document
archive.tar.gz: gzip compressed data - deflate method , original file name
create_user.sh: commands text
fatal.o:        ELF 32-bit MSB relocatable SPARC Version 1
fork1.c:        c program text
os_lec03.pdf:   Adobe Portable Document Format (PDF) v1.2
```

This command identifies the file type by examining the **magic number** that is embedded in the first few bytes of the file. Every file type has a unique magic number. **file** recognizes text files, and can distinguish between shell programs, C source and object code. It also identifies DOS executables, compressed files, PDF documents and even empty files. While this method of identifying files is not wholly accurate, it's a reliable indicator.

5.8 wc: COUNTING LINES, WORDS AND CHARACTERS

UNIX features a universal word-counting program that also counts lines and characters. It takes one or more filenames as arguments and displays a four-columnar output. Before you use **wc** on the file **infile**, just use **cat** to view its contents:

```
$ cat infile
I am the wc command
I count characters, words and lines
With options I can also make a selective count
```

You can now use **wc** without options to make a "word count" of the data in the file:

```
$ wc infile
       3      20     103 infile
```

wc counts 3 lines, 20 words and 103 characters. The filename has also been shown in the fourth column. The meanings of these terms should be clear to you as they are used throughout the book:

- A **line** is any group of characters not containing a newline.
- A **word** is a group of characters not containing a space, tab or newline.
- A **character** is the smallest unit of information, and includes a space, tab and newline.

wc offers three options to make a specific count. The **-l** option counts only the number of lines, while the **-w** and **-c** options count words and characters, respectively:

```
$ wc -l infile
       3 infile          Number of lines
$ wc -w infile
      20 infile          Number of words
$ wc -c infile
     103 infile          Number of characters
```

When used with multiple filenames, **wc** produces a line for each file, as well as a total count:

```
$ wc chap01 chap02 chap03
     305    4058   23179 chap01
     550    4732   28132 chap02
     377    4500   25221 chap03
    1232   13290   76532 total
```

A total as a bonus

wc, like **cat**, doesn't work with only files; it also acts on a data stream. You'll learn all about these streams in Chapter 8.

5.9 od: DISPLAYING DATA IN OCTAL

Many files (especially executables) contain nonprinting characters, and most UNIX commands don't display them properly. The file `odfile` contains some of these characters that don't show up normally:

```
$ more odfile
White space includes a
The ^G character rings a bell
The ^L character skips a page
```

The apparently incomplete first line actually contains a tab (entered by hitting *[Tab]*). The **od** command makes these commands visible by displaying the ASCII octal value of its input (here, a file). The **-b** option displays this value for each character separately. Here's a trimmed output:

```
$ od -b odfile
0000000 127 150 151 164 145 040 163 160 141 143 145 040 151 156 143 154
0000020 165 144 145 163 040 141 040 011 012 124 150 145 040 007 040 143
.....Output trimmed....
```

Each line displays 16 bytes of data in octal, preceded by the offset (position) in the file of the first byte in the line. In the absence of proper mapping it's difficult to make sense of this output, but when the **-b** and **-c** (character) options are combined, the output is friendlier:

```
$ od -bc odfile
0000000 127 150 151 164 145 040 163 160 141 143 145 040 151 156 143 154
          W   h   i   t   e       s   p   a   c   e       i   n   c   l
0000020 165 144 145 163 040 141 040 011 012 124 150 145 040 007 040 143
          u   d   e   s       a      \t  \n   T   h   e      007       c
0000040 150 141 162 141 143 164 145 162 040 162 151 156 147 163 040 141
          h   a   r   a   c   t   e   r       r   i   n   g   s       a
0000060 040 142 145 154 154 012 124 150 145 040 014 040 143 150 141 162
              b   e   l   l  \n   T   h   e      \f       c   h   a   r
.....
```

Each line is now replaced with two. The octal representations are shown in the first line. The printable characters and escape sequences are shown in the second line. The first character in the first line is the letter **W** having the octal value 127. You'll recall having used some of the escape sequences with **echo** *(3.3)*. Let's have a look at their various representations:

- The tab character, *[Ctrl-i]*, is shown as \t and the octal value 011.
- The bell character, *[Ctrl-g]*, is shown as 007. Some systems show it as \a.
- The formfeed character, *[Ctrl-l]*, is shown as \f and 014.
- The LF (linefeed or newline) character, *[Ctrl-j]*, is shown as \n and 012. Note that **od** makes the newline character visible too.

Like **wc**, **od** also takes a command's output as its own input, and in Section 5.13, we'll use it to display nonprintable characters in filenames.

5.10 cmp: COMPARING TWO FILES

You may often need to know whether two files are identical so one of them can be deleted. There are three commands in the UNIX system that can tell you that. In this section, we'll have a look at the **cmp** (compare) command. Obviously, it needs two filenames as arguments:

```
$ cmp chap01 chap02
chap01 chap02 differ: char 9, line 1
```

The two files are compared byte by byte, and the location of the first mismatch (in the ninth character of the first line) is echoed to the screen. By default, **cmp** doesn't bother about possible subsequent mismatches but displays a detailed list when used with the **-l** (list) option.

If two files are identical, **cmp** displays no message, but simply returns the prompt. You can try it out with two copies of the same file:

```
$ cmp chap01 chap01
$ _
```

This follows the UNIX tradition of quiet behavior. This behavior is also very important because the comparison has returned a *true* value, which can be subsequently used in a shell script to control the flow of a program.

5.11 comm: WHAT IS COMMON?

Suppose you have two lists of people and you are asked to find out the names available in one and not in the other, or even those common to both. **comm** is the command you need for this work. It requires two *sorted* files, and lists the differing entries in different columns. Let's try it on these two files:

```
$ cat file1
c.k. shukla
chanchal singhvi
s.n. dasgupta
sumit chakrobarty
```

```
$ cat file2
anil aggarwal
barun sengupta
c.k. shukla
lalit chowdury
s.n. dasgupta
```

Both files are sorted and have some differences. When you run **comm**, it displays a three-columnar output:

Comparing file1 and file2

```
$ comm file[12]
                anil aggarwal
                barun sengupta
                        c.k. shukla
chanchal singhvi
        lalit chowdury
                        s.n. dasgupta
sumit chakroborty
```

The first column contains two lines unique to the first file, and the second column shows three lines unique to the second file. The third column displays two lines common (hence its name) to both files.

This output provides a good summary to look at, but is not of much use to other commands that take **comm**'s output as their input. These commands require single-column output from **comm**, and **comm** can produce it using the options **-1**, **-2** or **-3**. To drop a particular column, simply use its column number as an option prefix. You can also combine options and display only those lines that are common:

```
comm -3 fool fool2          Selects lines not common to both files
comm -13 fool fool2         Selects lines present only in second file
```

The last example and one more with the other matching option (**-23**) has more practical value than you may think, but we'll not discuss their application in this text.

5.12 diff: CONVERTING ONE FILE TO OTHER

diff is the third command that can be used to display file differences. Unlike its fellow members, **cmp** and **comm**, it also tells you which lines in one file have to be *changed* to make the two files identical. When used with the same files, it produces a detailed output:

```
$ diff file1 file2          Or diff file[12]
0a1,2                       Append after line 0 of first file
> anil aggarwal             this line
> barun sengupta            and this line
2c4                         Change line 2 of first file
< chanchal singhvi          Replacing this line
---                         with
> lalit chowdury            this line
4d5                         Delete line 4 of first file
< sumit chakrobarty         containing this line
```

diff uses certain special symbols and instructions to indicate the changes that are required to make two files identical. You should understand these instructions as they are used by the **sed** command, one of the most powerful commands on the system.

Each instruction uses an **address** combined with an **action** that is applied to the first file. The instruction **0a1,2** means appending two lines after line 0, which become lines 1 and 2 in the second file. **2c4** changes line 2 which is line 4 in the second file. **4d5** deletes line 4.

Tip: If you are simply interested in knowing whether two files are identical or not, use **cmp** without any options.

Maintaining Several Versions of a File (-e) **diff -e** produces a set of instructions only (similar to the above), but these instructions can be used with the **ed** editor (not discussed in this text) to convert one file to the other. This facility saves disk space by letting us store the oldest file in its entirety, and only the changes between consecutive versions. We have a better option of doing that in the *Source Code Control System* (SCCS), but **diff** remains quite useful if the differences are few. SCCS is discussed in Chapter 22.

5.13 dos2unix AND unix2dos: CONVERTING BETWEEN DOS AND UNIX

Life being the way it is, you'll encounter DOS/Windows files in the course of your work. Sometimes, you'll need to move files between Windows and UNIX systems. Windows files use the same format as DOS, where the end of line is signified by two characters—CR (\r) and LF (\n). UNIX files, on the other hand, use only LF. Here are two lines from a DOS file, **foo**, viewed on a UNIX system with the **vi** editor:

```
Line 1^M                    The [Ctrl-m] character at end
Line 2^M
```

There's a ^M (*[Ctrl-m]*) representing the CR sequence at the end of each line. An octal dump confirms this:

```
$ od -bc foo
0000000 114 151 156 145 040 061 015 012 114 151 156 145 040 062 015 012
          L   i   n   e       1  \r  \n   L   i   n   e       2  \r  \n
```

The CR-LF combination is represented by the octal values 015-012 and the escape sequence \r\n. Conversion of this file to UNIX is just a simple matter of removing the \r. This is often done automatically when downloading a UNIX file from a Windows machine using **ftp**, but sometimes you have to do that job yourself!

For this purpose, some UNIX systems feature two utilities—**dos2unix** and **unix2dos**—for converting files between DOS and UNIX. Sometimes, systems differ in their implementation. This is how we use **dos2unix** to convert this file **foo** to UNIX format on a Solaris system:

```
dos2unix foo foo.dos
```

The output is written to **foo.dos**. When you use **od** again, you'll find that the CR character is gone:

```
$ od -bc foo.dos
0000000 114 151 156 145 040 061 012 114 151 156 145 040 062 012
          L   i   n   e       1  \n   L   i   n   e       2  \n
```

One some systems (like Solaris), the first and second filenames could be the same. Others (like Linux) require only one filename, in which case the command rewrites the input file. Some may even require redirection. Browse through the man page to identify the form that works on your machine:

```
os2unix foo foo          foo rewritten in UNIX format
os2unix foo              Same
os2unix foo > foo.dos    Taking shell's help
```

ix2dos inserts CR before every LF, and thus increases the file size by the number of lines in the le. The syntactical form that works for **dos2unix** also works for **unix2dos**.

ip: You can use **dos2unix** to delete the ^M character that occurs at the end of every line in the file ypescript. This is the file used by **script** to record a login session *(3.6)*.

Note: If you view the UNIX file foo on Windows using Notepad but without performing the conversion, then you'll see a single line: `Line 1Line2`. In fact, whenever you see a single line on a Windows machine that should have been multiple lines, satisfy yourself that you are viewing an unconverted UNIX file.

Caution: Never perform this conversion on a binary file. If you have downloaded a Windows program (say, a .EXE file) on a UNIX machine, the file must be transferred to the Windows machine without conversion. Otherwise, the program simply won't execute.

5.14 COMPRESSING AND ARCHIVING FILES

To conserve disk space you'll need to compress large and infrequently used files. Moreover, before sending a large file as an email attachment, it's good etiquette to compress the file first. Every UNIX system comes with some or all of the following compression and decompression utilities:

- **gzip** and **gunzip** (**.gz**)
- **bzip2** and **bunzip2** (**.bz2**)
- **zip** and **unzip** (**.zip**)

You'll find all of these programs on Solaris and Linux. The extension acquired by the compressed filename is shown in parentheses. The degree of compression that can be achieved depends on the type of file, its size and the compression program used. Large text files compress more, but GIF and JPEG image files (the types used on the World Wide Web) compress very little because they hold data in compressed form.

Apart from compressing, you'll also need to group a set of files into a single file, called an archive. The **tar** and **zip** commands can pack an entire directory structure into an archive. You can send this archive as a single file, either using **ftp** or as an email attachment, to be used on a remote machine. An additional layer of compression helps bring down the file size, the reason why **tar** is often used with **gzip** and **bzip2** for creating a compressed archive. **zip** handles both functions itself. In the next few sections, we'll be discussing these compression and archival utilities.

Note: Modern versions of WINZIP that run on Windows can read all of these formats though it can write only in the ZIP format. So you can choose any of these formats for compressing and archiving files even if you have to restore them on a Windows system. However, do check the WinZip documentation on the other system before you select the format to write an archive.

5.15 gzip AND gunzip: COMPRESSING AND DECOMPRESSING FILES

We'll monitor the size of one HTML and one PostScript file as they go through a number of compression and archival agents. We'll start with **gzip**, a very popular program, that works with one or more filenames. It provides the extension .gz to the compressed filename and removes the original file.

How well do HTML files compress? In the following example, we run **gzip** on the file `libc.html` (documentation for the GNU C library). We also note the file size, both before and after compression, using **wc -c** which counts characters:

```
$ wc -c libc.html
  3875302 libc.html
$ gzip libc.html
$ wc -c libc.html.gz
   788096 libc.html.gz
```

We seem to have achieved very high compression here, but before we go in for the statistics, let's repeat the previous exercise on a Postscript file:

```
$ wc -c User_Guide.ps ; gzip User_Guide.ps ; wc -c User_Guide.ps.gz
 372267 User_Guide.ps                    Before compression
 128341 User_Guide.ps.gz                 After compression
```

How much compression did we actually achieve for both files? Use the **-l** option with the compressed or original filenames as arguments:

```
$ gzip -l libc.html.gz User_Guide.ps.gz          .gz not necessary
compressed  uncompr. ratio uncompressed_name
    788096   3875302 79.6% libc.html
    128341    372267 65.5% User_Guide.ps
    916437   4247569 78.4% (totals)
```

HTML compressed better than Postscript (79.6% vs. 65.5%). This may not always be the case.

5.15.1 gzip Options

Uncompressing a "gzipped" File (-d) To restore the original and uncompressed file, you have two options: Use either **gzip -d** or **gunzip** with one or more filenames as arguments; the .gz extension is optional yet again:

```
gunzip libc.html.gz                      Retrieves libc.html
gzip -d libc.html                        Same—.gz assumed
gunzip libc.html.gz User_Guide.ps.gz     Works with multiple files
```

You can now browse the files with their respective viewers—a browser like Netscape or Mozilla for HTML files, and **gv** for Postscript documents.

Recursive Compression (-r) Like **cp**, you can also descend a directory structure and compress all files found in subdirectories. You need the **-r** option, and the arguments to **gzip** must comprise at least one directory:

```
gzip -r progs                            Compresses all files in progs
```

This option can be used for decompression also. To decompress all files in this directory you need to use **gunzip -r progs** or **gzip -dr progs**.

Tip: To view compressed text files, you really don't need to "gunzip" (decompress) them. Use the **gzcat** and **gzmore** (or **zcat** and **zmore**) commands if they are available on your system. In most cases, the commands run **gunzip -c**.

Note: For some years, **gzip** reigned as the most favored compression agent. Today, we have a better agent in **bzip2** (and **bunzip2**). **bzip2** is slower than **gzip** and creates **.bz2** files. We are beginning to see .bz2 files on the Internet. **bzip2** options are modeled on **gzip**, so if you know **gzip** you also know **bzip2**.

5.16 tar: THE ARCHIVAL PROGRAM

For creating a disk archive that contains a group of files or an entire directory structure, we need to use **tar**. The command is taken up in some detail in Chapter 15 to back up files to tape (or floppy), but in this section we need to know how the command is used to create a disk archive. For this minimal use of **tar** we need to know these *key* options:

`-c`	Create an archive
`-x`	Extract files from archive
`-t`	Display files in archive
`-f` *arch*	Specify the archive *arch*

Only one these key options can be used at a time. We'll also learn to use **gzip** and **gunzip** to compress and decompress the archive created with **tar**.

5.16.1 Creating an Archive (-c)

To create an archive, we need to specify the name of the archive (with `-f`), the copy or write operation (`-c`) and the filenames as arguments. Additionally, we'll use the `-v` (verbose) option to display the progress while **tar** works. This is how we create a file archive, `archive.tar`, from the two uncompressed files used previously:

```
$ tar -cvf archive.tar libc.html User_Guide.ps
a libc.html 3785K
a User_Guide.ps 364K
```

-v (verbose) displays list
a indicates append

By convention, we use the **.tar** extension, so you'll remember to use the same **tar** command for extraction. We created an archive containing two ordinary files, but **tar** also behaves recursively to back up one or more directories. In the following example, **tar** fills the archive `progs.tar` with three directory structures:

```
tar -cvf progs.tar *_progs java_progs shell_scripts
```

We'll soon use the same **tar** command to extract files from this archive. But before we do that, let's see how we can compress this archive.

Using `gzip` *with* `tar` If the created archive is very big, you may like to compress it with **gzip**:

```
gzip archive.tar
```

Archived and compressed

This creates a "tar-gzipped" file, `archive.tar.gz`. This file can now be sent out by FTP or as an email attachment to someone. A great deal of open-source UNIX and Linux software are available as **.tar.gz** files on the Internet. To use the files in this archive, the recipient needs to have both **tar** and **gzip** at her end.

5.16.2 Extracting Files from Archive (-x)

tar uses the `-x` option to extract files from an archive. You can use it right away on a **.tar** file, the one we just used to archive three directories:

```
tar -xvf progs.tar
```

Extracts the three directories

But to extract files from a **.tar.gz** file (like `archive.tar.gz`), you must first use **gunzip** to decompress the archive and then run **tar**:

```
$ gunzip archive.tar.gz
$ tar -xvf archive.tar
x libc.html, 3875302 bytes, 7569 tape blocks
x User_Guide.ps, 372267 bytes, 728 tape blocks
```

Retrieves `archive.tar`
Extracts files
`x` *indicates extract*

You'll now find the two files in the current directory. Selective extraction is also possible. Just follow the above command line with one or more filenames that have to be extracted:

```
tar -xvf archive.tar User_guide.ps
```

Extracts only `User_guide.ps`

This extracts a single file from the archive. If you use a pathname, it must be exactly in the same form that was used during the copying operation. This has been discussed later *(15.9.2—Tip)*.

5.16.3 Viewing the Archive (-t)

To view the contents of the archive, use the `-t` (table of contents) option. It doesn't extract files, but simply shows their attributes in a form that you'll see more often later:

```
$ tar -tvf archive.tar
-rw-r--r-- 102/10  3875302 Aug 24 19:49 2002 libc.html
-rw-r--r-- 102/10  372267 Aug 24 19:48 2002 User_Guide.ps
```

You'll understand the significance of these columns after you have learned to interpret the **ls -l** output *(6.1)*. But you can at least see the individual file size (third column) and their names (last column) in this output.

Both **tar** and **gzip** can be made to behave like *filters* (a group of programs whose input and output are quite flexible). After you have understood how filters work, you'll be able to perform both activities without creating an intermediate file.

LINUX: **tar** and **gzip** are so often used together that GNU **tar** has a **-z** option that compresses and archives together (and decompresses and extracts together). This dual activity is reflected in the following commands:

```
tar -cvzf archive.tar.gz libc.html User_Guide.ps     Compresses also
tar -xvzf archive.tar.gz                             Decompresses also
tar -xvzf archive.tar.gz User_Guide.ps               Extracts one file only
tar -tvzf archive.tar.gz                             Displays list
```

Note that whether you should use the **-z** option with **-x** or **-t** depends on whether the archive was compressed in the first place with **-c**. The archive's extension (**.tar.gz**) should provide this hint, but that's no guarantee. A wrong extension could have been provided by the user at the time of archival.

GNU **tar** also has a **--bzip** option that uses **bzip2** for handling compression. If you decide to use it, provide the extension **.tar.bz2** to the compressed archive so the person at the other end knows how to handle it.

5.17 zip AND unzip: COMPRESSING AND ARCHIVING TOGETHER

Phil Katz's popular PKZIP and PKUNZIP programs are now available as **zip** and **unzip** on UNIX and Linux systems. **zip** generally doesn't compress as much as **bzip2** but it combines the compressing function of **gzip** with the archival function of **tar**. So instead of using two commands to compress a directory structure, you can use only one—**zip**. All the letters of the alphabet are available as its options but we'll consider just a few of them.

zip requires the first argument to be the compressed filename; the remaining arguments are interpreted as files and directories to be compressed. The compression in the previous example could have been achieved with **zip** in the following way:

```
$ zip archive.zip libc.html User_Guide.ps
  adding: libc.html (deflated 80%)
  adding: User_Guide.ps (deflated 66%)
```

The unusual feature of this command is that it doesn't overwrite an existing compressed file. If **archive.zip** exists, files will either be updated or appended to the archive depending on whether they already exist in the archive.

Recursive Compression (-r) For recursive behavior, **zip** uses the **-r** option. It descends the tree structure in the same way **tar** does except that it also compresses files. You can easily compress your home directory in this way:

```
cd ; zip -r sumit_home.zip .          cd is same as cd $HOME
```

Using unzip Files are restored with the **unzip** command, which in its simplest form, uses the compressed filename as argument. **unzip** does a noninteractive restoration if it doesn't overwrite any existing files:

```
$ unzip archive.zip
Archive:  archive.zip
  inflating: libc.html
  inflating: User_Guide.ps
```


But if the uncompressed file exists on disk, **unzip** makes sure that it's doing the right thing by seeking user confirmation:

```
replace libc.html? [y]es, [n]o, [A]ll, [N]one, [r]ename: y
```

You can respond with **y** or **n**. You can also rename the file (**r**) to prevent overwriting or direct **unzip** to perform the decompression on the remaining files noninteractively (**A**).

Viewing the Archive (-v) You can view the compressed archive with the **-v** option. The list shows both the compressed and uncompressed size of each file in the archive along with the percentage of compression achieved:

```
$ unzip -v archive.zip
Archive:  archive.zip
 Length  Method   Size  Ratio   Date    Time   CRC-32    Name
 ------  ------   ----  -----   ----    ----   ------    ----
3875302  Defl:N  788068  80%  08-24-02  19:49  fae93ded  libc.html
 372267  Defl:N  128309  66%  08-24-02  19:48  7839e6b3  User_Guide.ps
 ------          ------  --                             -------
4247569          916377  78%                            2 files
```

Tip: If you have to move files between UNIX and Windows machines, you can take advantage of the end-of-line conversion options offered by **zip**. Use **-l** to convert from LF to CR-LF, and **-ll** to do the opposite. However, use them only when there are no binary files in your archive.

5.18 CONCLUSION

The commands discussed in this chapter don't always take input from files. Some commands (like **more** and **lp**) use, as alternate sources of input, the keyboard or the output of another command. Most of the other commands (like **wc**, **cat**, **cmp**, **od**, **gzip** and **tar**) can also send output to a file or serve as input to another command. Some examples in this chapter (and previous ones) have shown this to be possible with the **>** and **|** symbols. The discussion on these techniques is taken up in Chapter 8.

You would have also noted that we can use **gzip -d** instead of **gunzip** to decompress files. You'll have to understand why two commands have been offered to do the same job when one of them would have sufficed. Are **gzip** and **gunzip** one and the same file? This question is related to file attributes, and we begin our discussion on file attributes in the next chapter.

WRAP UP

cat is not only used to display one or more files but also to create a file.

You can copy files with **cp**, remove them with **rm**, and rename them with **mv**. All of them can be used interactively (**-i**), and the first two can be used to work on a complete directory tree (**-r** or **-R**) i.e., recursively. **rm -r** can remove a directory tree even if is not empty.

more is a pager that supports a repeat factor. You can search for a pattern (**/**) and repeat the search (**n**). Linux offers **less** as a superior pager.

`lp` prints a file and can directly print Postscript documents. You can cancel any submitted job with `cancel`. Linux uses the `lpr` command instead of `lp`.

`file` identifies the file type beyond the normal three categories. `wc` counts the number of lines, words and characters. `od` displays the octal value of each character and is used to display invisible characters.

We discussed three file comparison utilities. `cmp` tells you where the first difference was encountered. `comm` shows the lines that are common and optionally shows you lines unique to either or both sorted files. `diff` lists file differences as a sequence of `sed`-like instructions.

The `dos2unix` and `unix2dos` commands convert files between DOS and UNIX. DOS files use CR-LF as the line terminator, while UNIX uses only LF.

We also discussed several compression and archival tools. `gzip` and `gunzip` compresses and decompresses individual files (extension—`.gz`). `tar` always works recursively to archive a group of files into an archive. `tar` and `gzip` are often used together to create compressed archives (extension—`.tar.gz`).

`zip` and `unzip` can perform all functions that are found in `gzip`, `gunzip` and `tar`. `zip` alone can create a compressed archive from directory structures (`-r`).

Test Your Understanding

5.1 What will `cat foo foo foo` display?

5.2 How will you copy a directory structure bar1 to bar2? Does it make any difference if bar2 exists?

5.3 Run the command `tty` and note the device name of your terminal. Now use this device name (say, /dev/pts/6) in the command, `cp /etc/passwd /dev/pts/6`. What do you observe?

5.4 How will you remove a directory tree even when it's not empty and without using `rmdir`?

5.5 How does the command `mv bar1 bar2` behave, where both bar1 and bar2 are directories, when (i) bar2 exists, (ii) bar2 doesn't exist?

5.6 Use the `file` command on all files in the /dev directory. Can you group these files into two categories?

5.7 How do you print three copies of /etc/passwd on the printer named arjun?

5.8 Run the `script` command and then issue a few commands before you run `exit`. What do you see when you run `cat -v typescript`?

5.9 How will you find out the ASCII octal values of the numerals and alphabets?

5.10 How will you display only the lines common to two files?

5.11 The command `cmp foo1 foo2` displays nothing. What does it indicate?

Flex Your Brain

5.1 Describe the contents of a directory, explaining the mechanism by which its entries are updated. Why is the size of a directory usually small?

5.2 What is the difference between `cat foo` and `cat > foo`? Why do you have to use *[Ctrl-d]* in one and not in the other?

5.3 Will the command `cp foo bar` work if (i) foo is an ordinary file and bar is a directory, (ii) both foo and bar are directories?

5.4 Assuming that bar is a directory, explain what the command `rm -rf bar` does. How is the command different from `rmdir bar`?

5.5 What is the significance of these commands? (i) `mv $HOME/include .` (ii) `cp -r bar1 bar2` (iii) `mv * ../bin`

5.6 The command `cp hosts backup/hosts.bak` didn't work even though all files exist. Name three possible reasons.

5.7 You have a directory structure $HOME/a/a/b/c where the first a is empty. How do you remove it and move the lower directories up?

5.8 Explain the significance of the repeat factor used in `more`. How do you search for the pattern include in a file and repeat the search? What is the difference between this repeat command and the dot command?

5.9 A file with the .ps extension should be a Postscript file, but how can you be sure? What is the importance of this file type in the printing subsystem?

5.10 A file foo contains a list of filenames, with one filename in each line. One of the filenames appears to have an embedded space. How will you (i) count the number of filenames, (ii) check whether there's actually an embedded space in a filename?

5.11 How do DOS and UNIX text files differ? Name the utilities that convert files between these two formats?

5.12 You have two lists, foo1 and foo2, containing names of users. How do you create a third list of those users in foo2 who are absent in foo1? When will the command not work properly?

5.13 How do you use `tar` to add two files, `foo.html` and `bar.html`, to an archive and then compress the archive? How will you reverse the entire process and extract the files in their original uncompressed form?

5.14 Name three advantages `zip` has over `gzip`. How do you use `zip` to send a complete directory structure to someone by email? How does the recipient recreate the directory structure at her end?

5.15 What is meant by *recursive* behavior of a command? Name four commands, along with a suitable example of each, that can operate recursively.

6

Basic File Attributes

In the previous two chapters, you created files and directories, navigated the file system, and copied, moved and removed files without any problem. In real life, however, matters may not be so rosy. You may have problems when handling a file or directory. Your file may be modified or even deleted by others. A restoration from a backup may be unable to write to your directory. You must know why these problems happen and how to rectify and prevent them.

The UNIX file system lets users access other files not belonging to them and without infringing on security. A file also has a number of *attributes* (properties) that are stored in the inode. In this chapter, we'll use the `ls -l` command with additional options to display these attributes. We'll mainly consider the two basic attributes—permissions and ownership—both of which are changeable by well-defined rules. The remaining attributes are taken up in Chapter 11.

What You Will Learn

- Interpret the significance of the seven fields of the `ls -l` output (*listing*).
- How to obtain the listing of a specific directory.
- The importance of *ownership* and *group ownership* of a file and how they affect security.
- The significance of the nine permissions of a file as applied to different *categories* of users.
- Use **chmod** to change all file permissions in a relative and absolute manner.
- Use **chown** and **chgrp** to change the owner and group owner of files on BSD and AT&T systems.

Topics of Special Interest

- The importance of directory permissions (discussion on individual permissions deferred to Chapter 11).
- An introductory treatment of the relationship that exists between file ownership, file permissions and directory permissions.

6.1 ls -l: LISTING FILE ATTRIBUTES

We have already used the **ls** command with a number of options. It's the **-l** (long) option that reveals most. This option displays most attributes of a file—like its permissions, size and ownership

details. The output in UNIX lingo is often referred to as the **listing**. Sometimes we combine this option with other options for displaying other attributes, or ordering the list in a different sequence.

ls looks up the file's inode to fetch its attributes. Let's use `ls -l` to list seven attributes of all files in the current directory:

```
$ ls -l
total 72
-rw-r--r--   1 kumar    metal       19514 May 10 13:45 chap01
-rw-r--r--   1 kumar    metal        4174 May 10 15:01 chap02
-rw-rw-rw-   1 kumar    metal          84 Feb 12 12:30 dept.lst
-rw-r--r--   1 kumar    metal        9156 Mar 12  1999 genie.sh
drwxr-xr-x   2 kumar    metal         512 May  9 10:31 helpdir
drwxr-xr-x   2 kumar    metal         512 May  9 09:57 progs
```

The list is preceded by the words `total 72`, which indicates that a total of 72 blocks are occupied by these files on disk, each block consisting of 512 bytes (1024 in Linux). We'll now briefly describe the significance of each field in the output.

File Type and Permissions The first column shows the type and *permissions* associated with each file. The first character in this column is mostly a `-`, which indicates that the file is an ordinary one. This is, however, not so for the directories `helpdir` and `progs`, where you see a `d` at the same position.

You then see a series of characters that can take the values `r`, `w`, `x` and `-`. In the UNIX system, a file can have three types of permissions—read, write and execute. You'll see how to interpret these permissions and also how to change them in Sections 6.4 and 6.5.

Links The second column indicates the number of *links* associated with the file. This is actually the number of filenames maintained by the system of that file. UNIX lets a file have as many names as you want it to have, even though there is a single file on disk. This attribute will be discussed in Chapter 11.

Note: A link count greater than one indicates that the file has more than one name. That doesn't mean that there are two copies of the file.

Ownership When you create a file, you automatically become its *owner*. The third column shows kumar as the owner of all of these files. The owner has full authority to tamper with a file's contents and permissions—a privilege not available with others except the root user. Similarly, you can create, modify or remove files in a directory if you are the owner of the directory.

Group Ownership When opening a user account, the system administrator also assigns the user to some group. The fourth column represents the *group owner* of the file. The concept of a group of users also owning a file has acquired importance today as group members often need to work on the same file. It's generally desirable that the group have a set of privileges distinct from others as well as the owner. Ownership and group ownership are elaborated in Section 6.7.