

# File Attributes

In the previous chapter, you created 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 prevent and rectify them.

The UNIX file system lets users access other files not belonging to them—without infringing on security. A file also has a number of attributes which are changeable by certain well-defined rules. We'll be using the **ls** command in all possible ways to display these attributes. We'll also use other commands to change these attributes. Finally, we'll discuss **find**—one of the most versatile attribute handling tools of the UNIX system.

## Objectives

- Use the **ls** command to list files in all possible ways. (7.1)
- Understand the significance of the seven fields of the **ls -l** (listing) output. (7.2)
- Learn the significance of file and directory permissions and how to change them with **chmod**. (7.4 to 7.6)
- Know how the **umask** setting determines the default permissions. (7.7)
- Learn the concept of file *ownership* and how to change it with **chown** and **chgrp**. (7.8 and 7.9)
- Use **ls** to display a file's last *modification* and *access times* and **touch** to change them. (7.10 and 7.11)
- Become familiar with *file systems* and the *inode* as a file identifier. (7.12)
- Create *hard links* to a file with **ln** and use **ls -li** to display the *inode number*. (7.13)
- Learn the practical applications of links. (7.13.1)
- Understand why *symbolic links* are superior to hard links. (7.14)
- Locate files by matching one or more file attributes with **find**. (7.15)

## 7.1 ls: Listing Files

In the introductory chapter, you used the **ls** command (1.10.2) to display some files in the current directory. The command displays practically every attribute of a file—at

least the ones that concern us in this chapter. Let's execute it again in a directory which contains a good number of files:

```
$ ls
08_packets.html           Numerals first
TOC.sh                    Uppercase next
calendar                  Then lowercase
cptodos.sh
dept.lst
emp.lst
helpdir
progs
usdsk06x
usdsk07x
usdsk08x
```

What you see here is a list of filenames arranged in **ASCII collating sequence**, with one filename in each line. This sequence accords priority in the following order:

```
Whitespace (spaces and tabs)
Numerals
Uppercase letters
Lowercase letters
```

The list also includes directories which you'll be able to identify after using **ls** with suitable options. When the list is long, you won't be able to see all files. 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 in `/usr/bin`, **ls** clearly says so:

```
$ ls /usr/bin/perl
ls: /usr/bin/perl: No such file or directory
```

**ls** can also be used with multiple filenames. It has a host of options, not matched by many commands in number (over 20 in most versions). We'll discuss the important ones, especially those that reveal a file's or directory's attributes. Many of these options are discussed in the next section, but some are placed in the topical discussion of the attributes. The options are summarized in Table 7.1.

### 7.1.1 ls Options

**Output in Multiple Columns (-x)** When you have several files, it is better to display the filenames in multiple columns. Use the **-x** option to produce a multicolumnar output:

TABLE 7.1 Options to `ls`

Option	Description
<code>-x</code>	Displays multicolumnar output
<code>-F</code>	Marks executables with <code>*</code> , directories with <code>/</code> and symbolic links with <code>@</code>
<code>-a</code>	Shows all files including <code>.</code> , <code>..</code> and those beginning with a dot
<code>-R</code>	Recursive listing of all files in subdirectories
<code>-L</code>	Lists files pointed to by symbolic links
<code>-d</code>	Forces listing of a directory
<code>-l</code>	Long listing showing 7 attributes of a file
<code>-n</code>	Displays numeric user-id (UID) and group-id (GID) instead of their names
<code>-t</code>	Sorts by last file modification time
<code>-lt</code>	Displays listing sorted by last modification time
<code>-u</code>	Sorts by last access time
<code>-lu</code>	Displays last access time in listing but sorts by ASCII collating sequence (by access time in Linux)
<code>-lut</code>	Displays and sorts by last access time (same as <code>-lu</code> in Linux)
<code>-i</code>	Shows inode number
<code>-c</code>	Sorts by last inode change time
<code>-s</code>	Shows file size in 512 K-byte blocks (1024 in Linux)
<code>-r</code>	Sorts files in reverse order (ASCII collating sequence by default)

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

The display makes full use of the terminal width and is so convenient that many people have customized the command to display in this format by default (i.e., when used without options). You can do that too after you have learned to use aliases (17.4) and shell functions (19.10).

**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 the `-x` option produces a multicolumnar output:

```
$ 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 that tag some of the filenames. The `*` indicates that the file contains executable code; the `/` refers to a directory. You can now identify the two subdirectories in the current directory—`helpdir` and `progs`.

**Showing Hidden Files (-a)** By default, `ls` doesn't show all files in a directory. There are certain hidden files (those beginning with a dot) in every directory, especially in the home directory, that normally don't show up in the listing. The `-a` (all) option lists all the hidden files as well:

```
$ ls -axF
./                ../              .cshrc           .emacs
.exrc             .fetchmailrc    .netscape/      .profile
.rhosts          .sh_history     .xinitrc         08_packets.html*
TOC.sh*          calendar*
.....
```

Generally, these “dot” files determine the behavior of their corresponding commands. **emacs** reads `.emacs` when invoked, and there are a lot of files in the `.netscape` directory which **netscape** reads on startup. The X Window system executes the instructions in `.xinitrc`.

The files `.cshrc` and `.profile` need special mention. They contain a set of instructions that are performed when a user logs in. They are conceptually similar to `AUTOEXEC.BAT` of Windows, and you'll know more about them later (17.9). We'll discuss the significance of the various hidden files in some of the chapters of this text.

The first two files (`.` and `..`) are special directories that should remind you of the symbols you used to represent the current and parent directories (6.8). These symbols have the same meaning here, and whenever you create a subdirectory, these “invisible” directories are created automatically by the system. 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, while `..` signifies the parent directory. You can also list a hidden file by specifying it as an argument to `ls`.

**Listing Directory Contents** We are not discussing an option here but just a way to list a directory's contents. In a previous example (`ls calendar`), you specified an ordinary filename to `ls` to check for its existence. However, the situation will be quite different if you specify a directory name `progs` instead:

```
$ ls -x progs
array.pl   cent2fah.pl  n2words.pl   name.pl
```

This time the *contents* of the directories are listed, consisting of a number of **perl** program files. Note that the output doesn't indicate the directory whose contents are displayed.



Note

If **ls** displays a list of files when used with a single filename as argument, you can conclude that the file is actually a directory. **ls** then shows the contents of the directory. The **-d** option suppresses this behavior. We'll discuss this option when we discuss directory attributes.

**Recursive Listing (-R)** The **-R** (recursive) option lists all files and subdirectories in a directory tree. This traversal of the directory tree is done recursively till 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 files in three sections—the ones under the home directory and those under the subdirectories `helpdir` and `progs`. You should now find the subdirectory naming conventions familiar; `./helpdir` is under the current directory. If this current directory is `/home/romeo`, the absolute pathname of `helpdir` expands to `/home/romeo/helpdir`.



Note

Any of these options can be used with the **-r** option to reverse the order of display. Here, this option would simply present files in reverse ASCII sequence.

## 7.2 ls -l: Listing File Attributes

The **-l** (long) option of **ls** reveals most attributes of a file—like its permissions, size and ownership details. The output in UNIX lingo is often referred to as the **listing**, and a typical listing is shown in Fig. 7.1.

The list is preceded by the words `total 35`; a total of 35 blocks are occupied by these files in the disk. (Each block contains 512 or 1024 bytes.) The **-l** option can be combined with other options to display more attributes—and in different ordering sequences. We'll now discuss the significance of each of the seven fields of the listing.

**Type and Permissions** The first column shows the type and **permissions** associated with each file. The first character in this column shows a **-** for the first four files, which indicates that the file is an ordinary one. This is, however, not so for the directories `helpdir` and `progs` where there's 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 Section 7.5.

FIGURE 7.1 Long Listing of Files with the `ls -l` Command

```

$ ls -l
total 35
-rw-r--r--  1 romeo  metal 19514  May 10 13:45  chap01
-rw-r--r--  1 romeo  metal  4174  May 10 15:01  chap02
-rw-rw-rw-  1 romeo  metal   84  Feb 12 12:30  dept.lst
-rw-r--r--  3 juliet  metal  9156  Mar 12 1999  genie.sh
drwxr-xr-x  2 romeo  metal   64  May  9 10:31  helpdir
drwxr-xr-x  2 romeo  metal  320  May  9 09:57  progs

```

Permissions	Links	Owner	Group Owner	Size	Last File Modification	Filename
-rw-r--r--	1	romeo	metal	19514	May 10 13:45	chap01
-rw-r--r--	1	romeo	metal	4174	May 10 15:01	chap02
-rw-rw-rw-	1	romeo	metal	84	Feb 12 12:30	dept.lst
-rw-r--r--	3	juliet	metal	9156	Mar 12 1999	genie.sh
drwxr-xr-x	2	romeo	metal	64	May 9 10:31	helpdir
drwxr-xr-x	2	romeo	metal	320	May 9 09:57	progs

**Links** The second column indicates the number of **links** associated with the file. This is actually the number of names 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. Here, `genie.sh` has three links; the other links could be in different directories. This attribute will be discussed in Section 7.13.



#### Note

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

**Ownership and Group Ownership** When you create a file, you automatically become its **owner**. The third column shows `romeo` as the owner of all files except `genie.sh`. The owner has full authority to tamper with a file's contents, permissions, and even ownership—a privilege not available with others except the root user. Here, `romeo` can alter the attributes of all files except `genie.sh`, which can be changed only by user `juliet` (and by root).

When opening a user account, the system administrator also assigns the user to some group. The fourth column stands for 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 also elaborated in Section 7.8.

**Size** The fifth column shows the size of the file in bytes—a measure of the character count and not the disk space occupied by the file. The space used on disk is usually larger than this figure since files are written to disk in blocks of 1024 bytes (normally). In other words, even though the file `dept.lst` contains 84 bytes, it occupies 1024 bytes on the disk.

The two directories show smaller file sizes. A directory maintains a list of files along with an identification number (the *inode*) for each file. The size of the directory file depends on the size of this list—whatever the size of the files themselves.

**Last Modification Time** The next set of three columns indicate the last **modification time** of the file—a “time stamp” that is stored to the nearest second. A file is considered to be modified if its contents have changed in any way. The year is displayed if the file is more than a year old since its last modification date. This is true for the file `genie.sh`.

You’ll often need to run automated tools that make decisions based on a file’s modification time. This column shows two other time stamps when `ls` is used with certain options. The time stamps are discussed in Section 7.10.

**Filename** The last column displays the filenames arranged alphabetically. You already know (6.2) that UNIX filenames can be up to 255 characters long. We’ll also stick to the alphanumeric characters, the hyphen (-), the underscore (\_) and the dot (.) for framing all filenames. If you would like to see an important file at the top of the listing, then choose its name in uppercase—at least, its first letter.



Linux

Linux shows the year in the output of the listing if the file is just more than six months old. It also reports in 1024-byte blocks. For instance, the first line of the `ls -l` output (containing the word **total**) uses a block size of 1024.

### 7.3 Listing Directory Attributes (`ls -d`)

`ls` lists files in the directory when used with a directory name. If you want to force `ls` to list the directory attributes rather than its contents, you require the `-d` (directory) option:

```
$ ls -ld helpdir progs
drwxr-xr-x  2 romeo  metal      48 Jan 30 14:21 helpdir
drwxr-xr-x  2 romeo  metal      96 Mar 12 14:50 progs
```

The `-l` option has been combined here to show that the listing for a directory is somewhat different from an ordinary file. Directories are easily identified in the listing by the first character of the first column; it’s always a `d`. For ordinary files, this place always shows a `-` (hyphen), and for device files, either a `b` or a `c`. Directory attributes are discussed in Section 7.6.



Note

If you wish to see the attributes of a directory rather than the files contained in it, use `ls -d` with the directory name. Note that simply using `ls -d` will *not* list all subdirectories in the current directory. Strange though it may seem, `ls` has no option to list only directories!

### 7.4 File Permissions

UNIX has a simple, but well-defined system of assigning permissions to files. To understand them, you must understand the significance of ownership and group ownership as well. Issue the `ls -l` command once again to list some files in a directory:

```
$ ls -l chap02 dept.lst dateval.sh
-rwxr-xr--  1 romeo  metal      20500 May 10 19:21 chap02
-rwxr-xr-x   1 romeo  metal       890 Jan 29 23:17 dateval.sh
-rw-rw-rw-   1 romeo  metal       84 Feb 12 12:30 dept.lst
```

Observe the first column that represents the file permissions. These permissions are different for the three files. UNIX follows a three-tiered file protection system that determines the access rights that you have for a file. Each tier represents a **category**, and comprises a string of rs, ws and xs to represent three types of permissions.

r indicates read permission, which means **cat** will display the file. w indicates write permission; you can edit such a file with an editor. x indicates execute permission; the file can be executed as a program. To understand the significance of the entire string of rs, ws and xs, we must break it up into three groups as shown in Fig. 7.2.

In this figure representing the permissions of the file chap02, the first group (rwx) has all three permissions. The file is readable, writable and executable by the owner of the file. That's fine, but do we know who the owner is? Yes, we do; the third column of the listing shows romeo as the owner. You have to log in with the username romeo for these privileges to apply to you.

The second group (r-x) has a hyphen in the middle slot, and applies to the group owner of the file. This group owner is metal, and all users belonging to the metal group have read and executable permissions only.

The third group (r--) has the write and executable bits absent. This set of permissions is applicable for *others*—those who are neither the owner romeo nor belong to the metal group.

The group and others category is often collectively referred to as the **world**. If you are told that a file is “world-writable,” it means that there is a w in that part of the permissions string that applies to group and others (like r--rw-rw-).



#### Note

The group permissions don't apply to romeo (the owner) even if romeo is a member of the metal group. The owner has its own set of permissions that override the group owner's permissions. However, when romeo renounces the ownership of the file, the group permissions then apply to him.

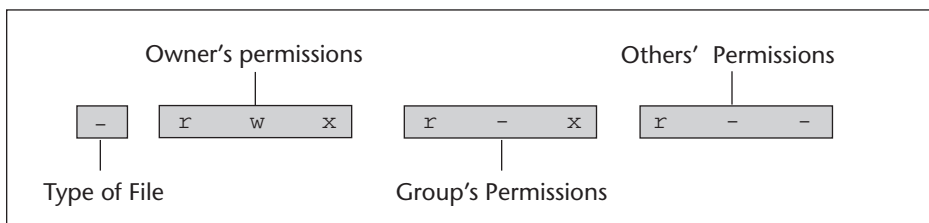
You can change all these permissions, but you'll have to be careful. If you are generous enough (and careless, too) to have read, write and executable permissions for all categories of users, then the permissions field will look like this:

```
rwxrwxrwx
```

*All permissions for everyone!*

You shouldn't be able to read, write or execute every file; you can then never have a secure system. The UNIX system, by default, never creates files with these permissions. But don't get overly cautious and remove all permissions either:

**FIGURE 7.2** Structure of a File's Permissions String



-----

*No permissions for anyone!*

If chap01 had no permissions as shown above, we could test them quite easily:

```
$ cat chap01
cat: chap01: Permission denied           File can't be read
$ echo "Come to the Web" > chap01
chap01: cannot create                   File can't be written
$ chap01
chap01: cannot execute                   File can't be executed
```

UNIX offers a well-defined protection mechanism by which you can set different permissions for the three categories of users—owner, group and others. This is applicable for all three types of files, and it's important that you understand them. A little learning here can be a dangerous thing; a faulty file permission is a sure recipe for disaster.

## 7.5 chmod: Changing File Permissions

The default security feature provided by UNIX write-protects a file from all except the owner of the file. Generally, all users have read access by default, but it could be different on your system. The default file permissions are easily revealed by creating a file `small`:

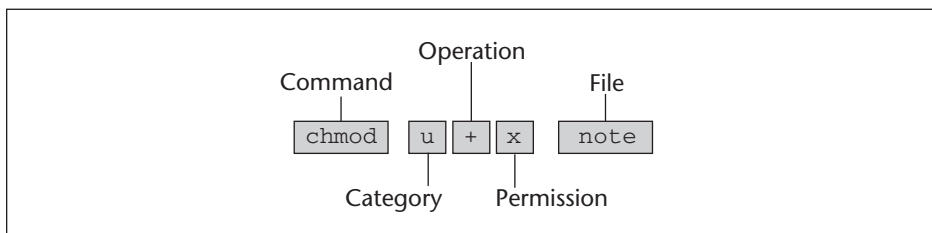
```
$ echo hello web > small
$ ls -l small
-rw-r--r-- 1 romeo  metal    10   May 10 20:30 small
```

Observe that a file doesn't have executable permission by default—not even for the owner. So how does one execute such a file? Just change its permissions with the **chmod** (change mode) command. Only the owner of the file can use this command. **chmod** sets a file's permissions (read, write and execute) for all three categories of users (owner, group and others) and uses the following syntax:

```
chmod category operation permission file(s)
```

**chmod** is a rather unusual command, not quite fitting into the general command structure. The structure of a **chmod** command with its arguments is shown in Fig. 7.3. The expression used by the command contains three components:

**FIGURE 7.3** *The Structure of a chmod Command*



- *Category* of user (owner, group owner or others).
- *Operation* to be performed (assign or remove a permission).
- *Permission* type (read, write or execute).

By using suitable abbreviations for each of these components, you can frame a compact string and then use the string as an argument to **chmod**. The abbreviations used for these three components are shown in Table 7.2.

Now, let's take an example. We'll assign executable permission to the owner (referred to as *user* in Table 7.2) of the file `small` without changing the other permissions. Look at the table closely; the expression required should be `u+x`:

```
$ chmod u+x small Executable permission for owner
$ ls -l small
-rwxr--r-- 1 romeo metal 10 May 10 20:30 small
```

The command assigns (+) executable permission (x) to the user (u). (User in **chmod** is to be understood as owner.) romeo as owner can now execute the file, but the other categories (group and others) still can't. To enable execution by all, you should use more than one character for the category:

```
$ chmod ugo+x small ; ls -l small x for all categories
-rwxr-xr-x 1 romeo metal 10 May 10 20:30 small
```

`ugo` combines all three categories—user, group and others. **chmod** also offers a shorthand symbol `a` (all) as a synonym of `ugo`. As if that's not enough, there's an even shorter form—one that doesn't specify the category at all! So, the previous sequence can be replaced by either of the following:

```
chmod a+x small a implies ugo
chmod +x small By default, a is implied
```

Permissions are removed with the `-` operator. To remove the read permission from both group and others, use the expression `go-r`:

```
$ ls -l small
-rwxr--r-- 1 romeo metal 10 May 10 20:30 small
$ chmod go-r small ; ls -l small
-rwx----- 1 romeo metal 10 May 10 20:30 small
```

**TABLE 7.2** *Abbreviations Used by chmod*

Category	Operation	Permission
u—User	+—Assigns permission	r—Read permission
g—Group	--Removes permission	w—Write permission
o—Others	=—Assigns absolute permission	x—Execute permission
a—All		

To remove all permissions from this file, you have to use

```
$ chmod u-rwx small ; ls -l small
----- 1 romeo  metal          10   May 10 20:30 small
```

What can you do with this file? You can't read, write or execute it, but can you remove it? Let's try that out:

```
$ rm small
rm: small: override protection 0 (yes/no)? yes
```

**rm** turns interactive when a user attempts to delete a file that doesn't have write permission for the owner. On this Solaris system a **yes** at the prompt deletes the file; any other response leaves it alone. Some systems expect just a **y**. If you dislike **rm**'s prompting, then use **rm -f**.



**Caution**

An undocumented feature of **chmod** prevents **+w** from setting write permission for all categories. You must use **a+w** or **ugo+w** explicitly. Similarly, **-w** removes write permission only from owner; you have to use **a-w** or **ugo-w** to remove the permission from all users.

### 7.5.1 Using Multiple Expressions

Using multiple expressions and the comma as delimiter, **chmod** can set more than one permission. For instance, to restore the original permissions (**rw-r--r--**) to the file **small** (currently, **-----**), you have to assign read permission to all and write permission to the owner:

```
$ chmod a+r,u+w small ; ls -l small
-rw-r--r-- 1 romeo  metal          10   May 10 20:30 small
```

Setting write and executable permissions for others requires use of multiple permissions for the same category. This can also be done very easily:

```
$ chmod o+wx small ; ls -l small
-rw-r--rwx 1 romeo  metal          10   May 10 20:30 small
```

The permissions assigned so far were **relative**. When you use the expression **u+r**, you are assigning read permission to the owner without disturbing the other permissions—including those for group and others. **chmod** also uses an **absolute** assignment feature, which is taken up in the next topic.

### 7.5.2 Absolute Assignment

Absolute assignment by **chmod** is done with the **=** operator. Unlike the **+** or **-** operators, it assigns only those permissions that are specified along with it and *removes other permissions*. Thus, if you want to assign only read permission to all three categories and remove all other permissions, you can do it either by

```
chmod u-w,o-wx small
```

or simply by using the = operator in any of these ways:

```
chmod ugo=r small
chmod a=r small
chmod =r small
```

Sometimes, you don't need to know what a file's current permissions are but want to set all the nine permission bits *explicitly*. The absolute assignment method is then preferable and easier to use.



A file's permissions can be changed only by its owner (understood by **chmod** as user). One user can't change the protection modes of files belonging to another user. However, the system administrator can tamper with all file attributes including permissions, irrespective of their ownership.

### 7.5.3 The Octal Notation

There's a shorthand notation available for representing file permissions. **chmod** also takes a numeric argument that describes both the category and the permission. The notation uses octal numbers (numbers using the base 8 contrasted with the standard decimal system which uses the base 10). Each type of permission is assigned a number as shown:

- Read permission—4
- Write permission—2
- Execute permission—1

When one category has multiple permissions, the respective numbers are added. For instance, if the owner has read and write permissions, the permissions for this category are represented by the number 6 (4 + 2). When this exercise is repeated for the other categories, you have a three-character octal number following this sequence: user, group and others. Like the = operator, the assignment is absolute.

You can use this method to assign read and write permission to all the three categories. Without octal numbers, you should normally be using **chmod ugo+rw small** to achieve the task. Now you can use a different method:

```
$ chmod 666 small ; ls -l small
-rw-rw-rw- 1 romeo metal          10   May 10 20:30 small
```

The 6 indicates read and write permissions (4 + 2). To restore the original permissions to the file, you need to remove the write permission (2) from group and others:

```
$ chmod 644 small ; ls -l small
-rw-r--r-  1 romeo metal          10   May 10 20:30 small
```

To assign all permissions to the owner, read and write permissions to the group, and only executable permission to the others, you can use either of the two following forms:

```
chmod u=rwx,g=rw,o=x small
chmod 761 small
```

*This is much simpler*

The highest number possible for each category is 7 (4 + 2 + 1) and the lowest is 0. So, 777 signifies all permissions for all categories, while 000 indicates absence of all permissions for all categories. We know the consequences of using 000, but what happens if a file has all permissions?

You probably wouldn't have expected this answer, but a file writable by all can be edited, overwritten and appended to by all. But only the owner can delete the file. This really makes little difference since anyone can remove every byte from this file without actually deleting it. Table 7.3 shows the use of **chmod** both with and without using octal notation.

#### 7.5.4 Recursive Operation (-R)

Even though we used **chmod** with only a single file, it works with multiple files also. You can assign the same set of permissions to a group of files using a single **chmod** command:

```
chmod u+x note note1 note3
```

It's now possible to apply the **chmod** command recursively to all files and subdirectories. This is done with the **-R** (recursive) option and needs only the directory name or the metacharacter **\*** as argument:

```
chmod -R a+x progs
chmod -R a+x *
```

*Acts recursively on all files in progs directory*

*Acts recursively on all files in current directory*

You have used the **\*** before (6.12.1) to match all files in the current directory. These two **chmod** commands make all files and subdirectories (in the progs directory or the current directory) executable by all users. If you want to use **chmod** for your home directory tree, then "cd" to it and use it like this:

```
chmod -R 644 .
```

*The dot is the current directory*

**TABLE 7.3** **chmod** Usage

Initial Permissions	Symbolic Expression	Octal Expression	Final Permissions
rw-r----	o+rw	646	rw-r--rw-
rw-r--r--	u-w,go-r	600	r-----
rwX-----	go+rwx	777	rwXrwxrwx
rwXrW--wX	u-rwx,g-rw,o-wX	000	-----
-----	+r	666	r--r--r--
r--r--r--	+w	644	rw-r--r-- (Note this)
rw-rw-rw-	-w	466	r--rw-rw- (Note this)
rw-rw-rw-	a-w	444	r--r--r--
-----	u+w,g+rX,o+x	251	-w-r-x--x
rwXrwxrwx	a=r	444	r--r--r--

There's an important observation to be made here. So far, we've been changing permissions of ordinary files. The commands used here with the `-R` option also change the permissions of all directories found in the tree walk. What do permissions mean when they are applied to a directory? Just read on.

## 7.6 Directory Permissions

A directory stores the list of filenames along with an identification number for each file it houses. It's possible that a file can't be read even though it has read permission, and can be removed by others even when it's write-protected from them. This may come as a shock to you, but it's true; a file's access rights are also influenced by the permissions of its directory. Let's first check its default permissions:

```
$ ls -ld progs
drwxr-xr-x 2 romeo metal 128 Jun 18 22:41 progs
```

Every directory shows a `d` in the first column of the permissions string. By default, all users are allowed read and execute access to a directory—something not allowed for ordinary files. The significance of a directory's permissions are also different.



Note

You can create a directory with `mkdir` and then view its permissions just to make sure that the default permissions are the same as shown here.

**Read Permission** Read permission for a directory means that `ls` can read the list of filenames stored in that directory. If you remove its read permission, `ls` won't work, and standard utilities won't be able to read the directory information:

```
$ chmod -r progs ; ls -ld progs
d-wx--x--x 2 romeo metal 128 Jun 18 22:41 progs
$ ls -l progs
ls: can not access directory progs: Permission denied (error 13)
```

However, if you remember the filenames, you can still read the files separately. It's just that `ls` won't display their names.

**Write Permission** Write permission for a directory implies that you are permitted to create or remove filenames in it. Restore the read permission, remove the write permission and then try to copy a file to this directory:

```
$ chmod 555 progs ; ls -ld progs
dr-xr-xr-x 2 romeo metal 128 Jun 18 22:41 progs
$ cp emp.lst progs
cp: unable to create file progs/emp.lst: Permission denied
```

The directory doesn't have write permission; you can't create or copy a file in it. Can you overwrite the existing files, or append to them? Does the directory file get changed in this way?

What happens if you allow all users to write to a directory?

```
$ chmod 777 progs ; ls -ld progs
drwxrwxrwx  2 romeo  metal           128 Jun 18 22:41 progs
```

This is the most dangerous thing you can ever do. Irrespective of the permissions that individual files may have, *every* user can remove *every* file in this directory!



Note

Write permission for the owner of a directory doesn't imply that the owner can directly edit the directory file; that power is reserved only for the kernel. If that was possible, then any user could destroy the integrity of the file system.

**Execute Permission** Execution privilege of a directory means that a user can “pass through” the directory in searching for subdirectories. When you issue the command

```
cat /home/romeo/progs/emp.sh
```

you need to have executable permission for each of the directories involved in the complete pathname. If a single directory doesn't have this permission, it can't be searched for the name of the next directory. It also means that you can't even switch to that directory with **cd**:

```
$ chmod 666 progs ; ls -ld progs
drw-rw-rw-  2 romeo  metal           128 Jun 18 22:41 progs
$ cd progs
progs: permission denied
```

Like for regular files, directory permissions are extremely important because system security is heavily dependent upon them. If you tamper with the permissions of your directories, then make sure you set them correctly. If you don't, be assured that an intelligent user could make life miserable for you!



Caution

If a directory is writable by group and others, then any user from these categories can delete every file in that directory. It doesn't matter who owns the file or whether the file itself has write permission for that user. As a rule, you should never make directories world-writable unless you have definite reasons to do so.

## 7.7 umask: Default File Permissions

When you create files and directories, the default permissions that are assigned to them depend on the system's default setting. These default permissions are inherited by files and directories created by *all* users:

- `rw-rw-rw-` (octal 666) for regular files
- `rwxrwxrwx` (octal 777) for directories

However, these are not the permissions you see when you create a file or a directory. Actually, this default is transformed by subtracting the **user mask** from it to remove one or more permissions. This mask is evaluated by using **umask** without arguments:

```
$ umask
022
```

This is an octal number, and subtracting this value from the file default yields  $666 - 022 = 644$ . This represents the default permissions that you normally see when you create a regular file (`rw-r--r--`). Similarly, the default directory permissions are also `rwx-r-xr-x` ( $777 - 022 = 755$ ).

The **umask** setting can be changed only by the administrator who has to ensure that the setting is proper. Two extreme instances are shown below:

```
umask 666 All permissions off
umask 000 All read-write permissions on
```

The important thing to remember is that no one—not even the administrator—can turn on permissions not specified in the systemwide default settings. However, you can always use **chmod** as and when required. The effect of the **umask** settings on file and directory permissions is shown in Table 7.4.

## 7.8 File Ownership

The third and fourth fields of the listing show a file's owner and group owner. By default, the owner of a file is its creator. Consider this listing:

```
-rwxrw-r-- 1 julie dialout 717 Sep 14 20:37 wall.html
```

Only julie can change this file's attributes. But is julie a member of the dialout group? That depends on the user who created the file. If the creator is julie, her group automatically becomes the group owner. If she has not changed the default ownership pattern, dialout must be julie's group.

If you now copy this file to your directory, you become the owner of the copy and your group acquires group ownership (if your group is different). You also inherit the owner's file permissions, so `rwx` should now apply to you. By virtue of being the owner, you can manipulate the attributes of the copy at will.

**TABLE 7.4** Effect of **umask** Settings on Default Permissions

<b>umask</b> Value	Default File Permissions	Default Directory Permissions
000	<code>rw-rw-rw-</code>	<code>rwxrwxrwx</code>
666	<code>-----</code>	<code>--x--x--x</code>
777	<code>-----</code>	<code>-----</code>
022	<code>rw-r--r--</code>	<code>rwxr-xr-x</code>
046	<code>rw--w----</code>	<code>rwx-wx--x</code>
066	<code>rw-----</code>	<code>rwx--x--x</code>
222	<code>r--r--r--</code>	<code>r-xr-xr-x</code>
002	<code>rw-rw-r--</code>	<code>rwxrwxr-x</code>
026	<code>rw-r----</code>	<code>rwxr-x--x</code>
062	<code>rw----r--</code>	<code>rwx--xr-x</code>
600	<code>--rw-rw-</code>	<code>--xrw-rwx</code>

Here, any member of the dialout group can write to this file. This is an important requirement in group projects where you want group members to be able to read and write a group of files. Note, however, that none of the members of dialout (apart from julie) can change these permissions.



Tip

How do you know whether a file belongs to you? View the ownership column of its listing and then check with the **who am i** command. If they match, then you are the owner of the file.

### 7.8.1 /etc/passwd and /etc/group: How Ownership Details Are Stored

The usernames and the group names that you normally see in the listing and output of many UNIX commands are provided purely for your convenience. The system understands only numbers. Your user-id is actually a number (the **UID**) and this number is stored in the file `/etc/passwd`. Your group-id is a number too (the **GUID**) which is stored in both `/etc/passwd` and `/etc/group`. Here's a typical entry from `/etc/passwd`:

```
julie:x:508:100:julie andrews:/home/julie:/bin/csh
```

This is a line of seven fields showing the username in the first field. julie has 508 as the UID and 100 as the GUID. The name of this group-id can be found in `/etc/group`:

```
dialout:x:100:henry, image, enquiry
```

The first column shows the group name and the third column has the numeric group-id (the GUID). A user can belong to multiple groups, but the GUID shown in `/etc/passwd` is the **primary group**. `/etc/group` shows the usernames for **secondary groups**. Here, dialout is the secondary group for henry.

Using these two files, most commands translate these numbers to names before displaying ownership details. However, **ls** can be used with the `-n` option to display numbers instead of names.

### 7.8.2 An Intruder in the Listing

Sometimes, you'll see a set of numbers rather than the owner and group owner in the ownership fields of the listing:

```
-rwxrw-r--  1  30  204          717 Sep 14 20:37 wall.html
```

Problems of this sort are often encountered when files are transferred from another system. This user probably has an identical account on the other system but with different values of the UID and GUID. The number-name translation wasn't done here because these numbers don't exist in `/etc/passwd` and `/etc/group` of this system. System administrators should draw a lesson from here.

Observe the listing of romeo's home directory in Fig. 7.1, and you'll see another intruder:

```
-rw-r--r--  3  juliet  metal  9156  Mar 12 1999  genie.sh
```

There's a file owned by juliet in romeo's directory. This can happen for a number of reasons:

- The directory was world-writable so juliet created a file in this directory.
- romeo copied a file from juliet's directory with **cp -p**—the command that preserves a file's attributes (*7.10 Tip*).
- The file was transferred from a different system where romeo has the same UID that juliet has in this machine.

Now, what does a user like romeo do with these files if he finds them in his directory? Observe that neither file is writable by romeo since it is not owned by him. Even though `wall.html` is writable by members of the 204 group, romeo is not a member of this group. Even if romeo belongs to the metal group, `genie.sh` is not group-writable.

romeo can delete these files or copy them and then change their permissions. Alternatively, he can ask the system administrator to transfer the ownership to him. This has to be done with the **chown** and **chgrp** commands which are discussed next.



Tip

If you set up accounts of a user in two machines, then make sure that their numeric user-ids (the UIDs) match, as well as their group-ids (the GUIDs). Files can then be transferred freely between the two systems.

## 7.9 **chown** and **chgrp**: Changing File Ownership

There are two commands meant to manipulate the ownership of a file or directory—**chown** and **chgrp**. They can be used only by the owner of the file. Here's the syntax for both:

```
chown options new_user file(s)
chgrp options new_group file(s)
```

Before we proceed with **chown**, let's view the listing of the file `note` which will be used with these commands:

```
$ ls -l note
-rwxr---x  1 romeo  metal      347 May 10 20:30 note
```

Here, we assume that the user executing the **chown** and **chgrp** commands is also romeo—the owner of the file. Now, let's say that romeo wants to renounce the ownership of this file to juliet. To do that, he has to use **chown** (change ownership) which takes the new user's user-id as argument followed by one or more files:

```
$ chown juliet note ; ls -l note
-rwxr---x  1 juliet  metal      347 May 10 20:30 note
```

This does the job, but once the ownership of the file is transferred to juliet, the original ownership can't be reinstated:

```
$ chown romeo note
chown: cannot change owner ID of note: Operation not permitted
```

The file permissions pertaining to the owner are now applicable to juliet only. Thus, romeo can no longer edit the file `note` nor delete it since there is no write privilege for group and others. (He can copy it, of course.)

The **chgrp** (change group) command changes the group owner of a file. In the following example, romeo changes the group ownership of `dept.lst` to `dba`:

```
$ chgrp dba dept.lst ; ls -l dept.lst
-rw-r--r-- 1 romeo dba          139 Jun  8 16:43 dept.lst
```

That's done, but what happens when he tries to get it back?

```
$ chgrp metal dept.lst ; ls -l dept.lst
-rw-r--r-- 1 romeo metal        139 Jun  8 16:43 dept.lst
```

He does get back the group membership that he had surrendered. That's possible because he retains all rights related to the file as he's still the owner.

Like **chmod**, both **chown** and **chgrp** also work with the **-R** option to perform their operations in a recursive manner. All three commands place no restrictions whatsoever when used by the super user. In fact, the super user can change *every* file attribute that is discussed in this chapter.



Tip

If you want members of a project to be able to read and write a set of files, ask the system administrator to have a common group for them and then set the permissions of the group to `rwX`. There's a better way of doing this (with the sticky bit), and it is discussed in Chapter 22.



Linux

## Restrictions on Use of **chown** and **chgrp**

In Linux, the **chown** command can only be used by the super user. When romeo attempts to use the command, the system refuses:

```
$ chown henry note
chown: note: Operation not permitted
```

A Linux user can use **chgrp**, but she can use it only for changing the group to one to which she also belongs. If juliet belongs to the `dialout` and `uucp` groups, she is confined to these two groups only for using **chgrp**. She can't give away ownership to root, which the System V user can do easily.

## 7.10 File Modification and Access Times

Apart from permissions and ownership, a UNIX file has three time stamps associated with it. In this section, we'll be discussing just two of them (the first two):

- Time of last file modification *Shown by `ls -l`*
- Time of last access *Shown by `ls -lu`*
- Time of last inode modification *Shown by `ls -lc`*

When a file's contents are changed, its modification time is updated by the kernel. Even though `ls -l` shows this time for a file, the `-t` option actually presents files in order of their modification time; the last modified file is placed first:

```
$ ls -lt
total 278
-rw-r--r-- 1 romeo metal 10411 May 10 15:56 chap02
-rw-r--r-- 1 romeo metal 19514 May 10 13:45 chap01
drwxr-xr-x 2 romeo metal 64 May 9 10:31 helpdir
-rw-rw-rw- 1 romeo metal 84 Feb 12 12:30 dept.lst
-rw-r--r-- 1 romeo metal 9156 Mar 12 1998 genie.sh
```

A file also has an **access time**—the last time someone read, wrote or executed the file. This time is also maintained by the system, and is distinctly different from the modification time that gets set only when the contents of the file are changed.

The `-u` option of `ls` displays a file's access time. When you use `ls -lu`, the access time is displayed against each file, but the sort order remains standard (the ASCII collating sequence). But when the `-t` option is coupled with `-u`, files are actually sequenced in order of their access time:

```
$ ls -lut unit02 unit03 unit04
-rw-r--r-- 1 romeo metal 48527 Mar 20 23:17 unit04
-rw-r--r-- 1 romeo metal 39480 Feb 28 16:35 unit02
-rw-r--r-- 1 romeo metal 27183 Feb 13 14:57 unit03
```

If you now view the contents of the file `unit03` with `cat`, you'll update its access time but not the modification time. This will be evident when you use the `ls -lu` command again:

```
$ ls -lu unit03 Check access time again
l-rw-r--r-- 1 romeo metal 27183 Sep 30 23:30 unit03
$ date
Wed Sep 30 23:30:20 EST 1999
```

You can also use the `-r` option with both `-t` and `-ut`, in which case the files will be sorted in reverse order by the respective time (modification or access).

Knowledge of a file's modification and access times is extremely important for the system administrator. Many of the tools used by her have to look at these time stamps to decide whether a particular file will participate in a backup or not. A file is often incorrectly stamped when extracting it (using an option) from a backup with a file restoration utility (like `tar` or `cpio`). If that has happened to you, then you can use `touch` to reset the times to certain convenient values without actually modifying or accessing the file. `touch` is discussed next.



#### Note

It's possible to change the access time of a file without changing its modification time. In an inverse manner, when you modify a file, you generally change its access time as well. However, there is an exception; when you redirect command output (with the `>` and `>>` symbols) to a file, you do change the contents of the file, but this leaves the last access time unchanged. This feature is found on many UNIX and Linux systems.



Tip

What happens when you copy a file with **cp**? By default, the copy has both time stamps set to the time of copying. Sometimes, you may not like this to happen. In that case, use **cp -p** (preserve) to retain both time stamps. The option also preserves the existing ownership pattern.

## 7.11 touch: Changing the Time Stamps

As just discussed, you may sometimes need to set the modification and access times to predefined values. The **touch** command changes these times and is used in the following manner:

```
touch options expression filename(s)
```

When **touch** is used without options or expressions, both times are set to the current time. The file is created if it doesn't exist, but not overwritten if it does:

```
touch note Creates file if it doesn't exist
```

When **touch** is used without options but with an expression, it changes both times. The expression consists of an eight-digit number—using the format *MMDDhhmm* (month, day, hour and minute). Optionally, you can suffix a two- or four-digit year string.

Now, let's use the command on `emp.lst`, but only after seeing its initial time stamps:

```
$ ls -l emp.lst Modification time
-rw-r--r-- 1 romeo metal 870 May 11 12:49 emp.lst
$ ls -lu emp.lst Access time
-rw-r--r-- 1 romeo metal 870 Jun 9 09:10 emp.lst
```

The first command shows the modification time, while the second one shows the access time of the file. **touch** here changes both times:

```
$ touch 03161430 emp.lst ; ls -l emp.lst
-rw-r--r-- 1 romeo metal 870 Mar 16 14:30 emp.lst
$ ls -lu emp.lst
-rw-r--r-- 1 romeo metal 870 Mar 16 14:30 emp.lst
```

It's also possible to change the two times individually. With the **-m** (modification) option, you can alter the modification time alone:

```
$ touch -m 02281030 emp.lst ; ls -l emp.lst
-rw-r--r-- 1 romeo metal 870 Feb 28 10:30 emp.lst
```

The **-a** (access) option changes the access time:

```
$ touch -a 01261650 emp.lst ; ls -lu emp.lst
-rw-r--r-- 1 romeo metal 870 Jan 26 16:50 emp.lst
```

The system administrator often uses **touch** to “touch up” these times so that a file may be included in or excluded from an **incremental backup** (that backs up only changed files). When the **find** command goes about locating files that have changed after a certain time, it may or may not locate these files depending on how the times have been set.

## 7.12 File Systems and Inodes

Before we take up links, we need some idea of the way files are organized in a UNIX system. A certain area of the disk is always set aside to store the attributes of files. All the attributes that we have discussed so far (along with links) are stored in a table called the **inode**. Every file has one inode and is accessed by a number called the **inode number**. The inode number for a file is unique in a single **file system**.

This brings us to the concept of the file system. We have casually used this term as if it referred to a single superstructure holding all files and directories together. That seldom is the case, and never so in large systems. The hard disk is split up into separate file systems, and each file system has its own root directory. If you have three file systems in one hard disk, then they will have three separate root directories.

Of these multiple file systems, one of them is considered the main one, and contains most of the essential files of the UNIX system. This is the **root file system**, which is more equal than others in at least one respect: its root directory is also the root directory of the UNIX system. At the time of booting, all the secondary file systems attach themselves to the main file system, creating the illusion of a single file system to the user.

Every file system has a separate area earmarked for holding inodes. **ls** fetches the attributes of files from their inodes. A file's inode number is unique only in that file system that holds the file. This means that if you see two files with the same inode number, then they must be on two different file systems.

This is all the knowledge we need to understand links. We won't go any further at this point as inodes and other file system concepts are covered in some detail in Chapter 21.

## 7.13 1n: Links

UNIX allows a file to have more than one name and yet maintain a single copy in the disk. The file is then said to have more than one **link**. A file can have as many names as you want to give it, but the only thing common to all of them is that they all have the same inode number.

You can easily know the number of links of a file from the second column of the listing. This number is normally 1, but this file has two links:

```
-rw-r--r--  2 sumit  dialout      504 Oct  4 16:29 alias.sam
```

A file is linked with the **ln** (link) command which takes two filenames as arguments. The first filename is the one that actually exists; the other filename is the alias we provide. The following command links the existing file `display.sh` with `print.sh`:

```
ln display.sh print.sh
```

The `-i` option of **ls** displays the inode number of a file. Here, the first column shows that both links have the same inode number, proving conclusively that they are actually one and the same file:

```
$ ls -li display.sh print.sh
29518 -rwxr-xr-x  2 romeo  metal      915 May  4 09:58 display.sh
29518 -rwxr-xr-x  2 romeo  metal      915 May  4 09:58 print.sh
```

The number of links is shown to be two. You can provide another link, say `show.sh` and increase the number to three:

```
$ ln print.sh show.sh ; ls -li display.sh print.sh show.sh
29518 -rwxr-xr-x  3 romeo  metal    915 May  4 09:58 display.sh
29518 -rwxr-xr-x  3 romeo  metal    915 May  4 09:58 print.sh
29518 -rwxr-xr-x  3 romeo  metal    915 May  4 09:58 show.sh
```

Note that even though `ls` shows three files, they are actually one. You can say that the file has two aliases. Links are equal in all respects; they have the same inode number, permissions and time stamps. Changes made to one link are automatically available in the others.

Links often occur in different directories, which makes it difficult to locate them—unless you are using the `find` command. For instance, the following command provides a link with the same name but in a different directory:

```
ln toc.txt ../project5_safe/toc.txt
```

We use `rm` to remove files. Technically speaking, `rm` actually removes a link from the file, so the following command removes the link `show.sh` and brings down the link count by one:

```
$ rm show.sh ; ls -li display.sh print.sh
-rwxr-xr-x  2 romeo  metal    915 May  4 09:58 display.sh
-rwxr-xr-x  2 romeo  metal    915 May  4 09:58 print.sh
```

Another `rm` will further bring the link count down to one. A file is considered to be completely removed from the system when its link count drops to zero. Links thus provide some protection against accidental deletion. If a file is linked and you have inadvertently used `rm` to delete the file, at least one link will still be available; your file is not gone yet.

### 7.13.1 Where to Use Links

When do you need to link a file? Well, we can think of three applications straightaway:

1. You can use a link to “notionally place” a file in a specific directory where many programs expect to find it. Say, you have a number of programs which invoke `perl` using the absolute pathname `/usr/local/bin/perl`. When you upgraded your system, you found `perl` has moved to `/usr/bin`. Would you change all your programs to point to this new path? Not necessary, just leave a link like this in `/usr/local/bin`:

```
ln /usr/bin/perl /usr/local/bin/perl
```

2. C and the shell programming language support a feature which lets a program know the name by which it is called. Consider a program `display.sh` which formats its input and displays it on the terminal. Now, take another program, `print.sh`, which does a similar job but prints on the printer. Since both programs will have a substantial common portion, it makes sense to have all the logic in a single file.

A section of code in this file checks the name by which the file is called. It then executes the device-specific code to print the job either on the terminal or the printer. Once the program is developed, you can link the file to have two different names. Maintaining the program also becomes easy because there's only one file that does both jobs. There's a shell script using this feature in Section 18.12.

3. There's another useful application for links. Suppose you have a number of files that you use often but are placed in different subdirectories:

```
/home/henry/.profile
/home/henry/.elm/elmrc
/home/henry/Mail/received
```

All these are important files but have pathnames which are either long or difficult to remember. There's no need to use these pathnames at all; just place links of these files in a working directory, say \$HOME/work:

```
cd $HOME/work
ln ../.profile prof.lnk
ln ../.elm/elmrc elmrc.lnk
ln ../Mail/received recd.lnk
```

Note how relative pathnames reduce your typing load. Once you have linked these files, you can consider them available in the `work` directory. To edit the file `elmrc` in the directory `.elm`, you no longer need to use `vi ../.elm/elmrc`; you can simply use `vi elmrc.lnk`. No more pathnames to access these files!

## 7.14 Symbolic Links

The preceding discussions throw up a couple of questions. Creating a link like the one we created for `perl` is fine as long as you are not linking too many files. But what if `/usr/local/bin` had originally contained a hundred programs, and all of them have now moved to `/usr/bin`? Use the `ln` command a hundred times? And what if the directory `/usr/local/bin` is on a different hard disk?

It's here that one encounters the limitations of links—at least the ones of the type described. These links have two serious drawbacks:

- You can't link a file across two file systems. In other words, if you have a file in the `/usr` file system, you can't provide a link for it in the `/home` file system.
- You can't link a directory even within the same file system.

A **symbolic link** overcomes both these problems. Unlike the other link, a symbolic link is an ordinary file that points to the file or directory that actually has the contents. Being more flexible, it is also known as a **soft link**. The link that we discussed in the previous section has today come to be known as a **hard link**.

To return to the problem of linking the hundred programs in `/usr/local/bin`, you have to use a symbolic link to link these two directories. The command is the same, except that it uses the `-s` option. Let `/usr/local/bin` now point to `/usr/bin`:

```
cd /usr/local
ln -s /usr/bin bin
```

*The second file must not exist*

Even though a symbolic link is an ordinary file, it doesn't occupy space on disk. For the `ln` command to work, make sure that the destination filename doesn't exist. When you run `ls -l` now, two columns of the listing show up differently:

```
$ pwd
/usr/local
$ ls -l
lrwxrwxrwx  1 root  root           8 Mar 1 23:53 bin -> /usr/bin
```

You can identify symbolic links by the character `l` in the permissions field. The notation `bin -> /usr/bin` signifies that `bin` contains the pathname to the directory `/usr/bin`. You can now run all the hundred programs that once belonged to `/usr/local/bin`.

Many systems use symbolic rather than hard links to link regular files. The following `ln` command links two files in the same directory, and the `-i` (inode) option of `ls` has yet another story to tell:

```
$ ln -s note note.sym
$ ls -li note note.sym
 9948 -rw-r--r--  1 henry  group           80 Feb 16 14:52 note
 9952 lrwxrwxrwx  1 henry  group           4 Feb 16 15:07 note.sym -> note
```

The two files have different inode numbers (and hence a single link count) and file sizes; they are two separate files. However, it is `note`, and not `note.sym`, that actually contains the data. You can use relative pathnames here as well:

```
$ ln -s ../jscript/search.htm search.htm
$ ls -l search.htm
lrwxrwxrwx  1 sumit  dial  21   Mar 2 00:17 search.htm-> ../jscript/search.htm
```

Why should we use symbolic links for ordinary files? Unlike hard links, a symbolic link can link files across file systems. This means that two files in two hard disks can be connected with a symbolic link.

Even though a symbolic link can link directories, don't forget that it is an ordinary file. This means that the `rm` command removes a symbolic link—even if it points to a directory:

```
$ rm /usr/local/bin An ordinary file
$_
```

What happens when you delete the file pointed to rather than the symbolic link? The listing gives no indication, and it's only when you try to access the file that you'll find that it's no longer there:

```
$ rm ../jscript/search.htm
$ ls -l search.htm
lrwxrwxrwx  1 sumit  dial  21   Mar 2 00:17 search.htm-> ../jscript/search.htm
$ cat ../jscript/search.htm
cat: ../jscript/search.htm: No such file or directory
```

Symbolic links are used extensively in the UNIX system. System files constantly change locations with version enhancements. Yet, it must be ensured that all programs still find the files where they originally were.



Note

When you use **cd** with a symbolic link, the built-in **pwd** command shows you the path you used to *get* to the directory. This is not necessarily the same as the actual directory you are in. To know the “real” location, you should use the external command **/bin/pwd**.

## 7.15 find: Locating Files

**find** is one of the power tools of the system. It recursively examines a directory tree to look for files either by name or by matching one or more file attributes. It has a difficult command line, and if you have ever wondered why UNIX is hated by many, then you should look up the cryptic **find** documentation. However, **find** is easily tamed if you break up its arguments into three components:

**find** *path\_list selection\_criteria action*

Fig. 7.4 shows the structure of a typical **find** command. The command completely examines a directory tree in this way:

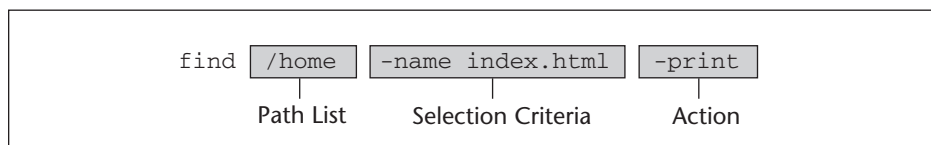
- First, it recursively examines all files in the directories specified in *path\_list*. Here, it begins the search from `/home`.
- It then matches each file for one or more *selection\_criteria*. This always consists of an expression in the form *-operator argument* (`-name index.html`). Here, **find** selects the file if it has the name `index.html`.
- Finally, it takes some *action* on those selected files. The action `-print` simply displays the **find** output on the terminal.

All **find** operators start with a `-` but the path list can never contain one. You can provide one or more subdirectories to act as the path list and multiple selection criteria to match one or more files. This makes the command difficult to use initially, but it is a program that every user must master since it lets her make file selection under practically any condition.

Let’s run our first **find** command to locate all files named `core` (the process image that is dumped on disk):

```
# find / -name core -print
/home/romeo/scripts/core
/home/andrew/scripts/reports/core
/home/juliet/core
```

**FIGURE 7.4** Structure of a **find** Command



Since the search starts from the root directory, **find** displays the absolute pathnames of the files. You can also use metacharacters in the name (like the **\***) to match a group of filenames, but then you need to enclose the pattern within quotes:

```
find . -name "*.lst" -print Don't forget the quotes!
```

This matches all files having the `.lst` extension from the current directory tree (the `.` is current directory). Once you know how to use the special characters like the **\*** (8.2), you can use **find** as a real power tool. You can search for all filenames beginning with an uppercase letter in this way:

```
find . -name '[A-Z]*' -print Single quotes will also do
```

Up to this point, **find** is already more powerful than its Windows counterpart. There are many features to follow.

**find** accepts a set of metacharacters with the `-name` argument; the **\*** is one of them. The shell also uses the same set (8.2), and after you have learned to use them, you'll find pattern matching with **find** quite easy and useful. For instance, you can use **find** to locate all the `.doc` files irrespective of their case:



Note

```
find -name ".*[Dd][Oo][Cc]" -print
```

**find** is probably the only UNIX command that supports this pattern matching feature in exactly the same way it's used in the shell.

### 7.15.1 **find** Options

`-name` is not the only operator used in framing the selection criteria; there are many others. Table 7.5 displays **find**'s selection criteria in the same sequence **ls -lids** presents a file's attributes. Many of these options are intuitive enough, so let's see some of the ways you can put **find** to good use.

**File Type and Permissions (`-type` and `-perm`)** The `-type` operator followed by the letter `f`, `d` or `l` selects files of the ordinary, directory and symbolic link type. Here's how you locate all the directories of your home directory tree:

```
$ find /home/henry -type d -print
/home/henry
/home/henry/.elm
/home/henry/Mail
/home/henry/.netscape
```

You can also use the `-perm` operator to locate files having a specified set of permissions. For instance, `-perm 666` selects files having read and write permission for all categories of users. Such files are security hazards. You'll often want to use two operators in combination to restrict the search to only directories:

```
find /home/henry -perm 777 -type d -print
```

TABLE 7.5 Expressions Used by **find** (+ can be replaced with - for reverse meaning)

Selection Criteria	Significance
-type <i>x</i>	Selects file if of type <i>x</i> ; <i>x</i> can be f (ordinary file), d (directory) and l (symbolic link)
-perm <i>nnn</i>	Selects file if octal permissions match <i>nnn</i> completely
-perm -400	Selects file where owner has read permission irrespective of what the other bits may be
-links <i>n</i>	Selects files if having <i>n</i> links
-inum <i>n</i>	Selects file having inode number <i>n</i>
-user <i>uname</i>	Selects file if owned by <i>uname</i>
-group <i>gname</i>	Selects file if owned by group <i>gname</i>
-size + <i>x</i> [ <i>c</i> ]	Selects file if size greater than <i>x</i> blocks (characters if <i>c</i> is also specified)
-mtime - <i>x</i>	Selects file if modified in less than <i>x</i> days
-mmin - <i>x</i>	Selects file if modified in less than <i>x</i> minutes (Linux only)
-newer <i>fname</i>	Selects file if modified after <i>fname</i>
-atime + <i>x</i>	Selects file if accessed in more than <i>x</i> days
-amin + <i>x</i>	Selects file if accessed in more than <i>x</i> minutes (Linux only)
-name <i>fname</i>	Selects file <i>fname</i>
-iname <i>fname</i>	As above, but match is case-insensitive (Linux only)
-follow	Selects file after following a symbolic link
-prune <i>dir</i>	Don't descend directory <i>dir</i>
-mount	Don't look in other file systems (Chapter 22)
Action	Significance
-exec <i>cmd</i>	Executes UNIX command <i>cmd</i> followed by { } \;
-ok <i>cmd</i>	Like -exec, except that command is executed after user confirmation
-print	Prints selected file on standard output
-ls	Executes <b>ls -lids</b> command on selected files

Here, we are looking for directories that have all access rights for everyone—something even more hazardous than `-perm 666` used with ordinary files. It's an AND condition that is implemented above; **find** selects the files only if both selection criteria (`-perm` and `-type`) are fulfilled.

**Following Symbolic Links (-follow)** If a file is a symbolic link pointing to a directory, then **find** by default won't look in that directory if it is not in the path list. To illustrate this point, let's first symbolically link the directory `../jscrip` which contains a file `night.gif`:

```
ln -s ../jscript jscript
```

We now have a symbolic link `jscript` which points to a directory of the same name in the parent directory. When we run **find** to look for `night.gif`, it won't look in that directory:

```
$ find . -name night.gif -print
$_
```

But when we combine this with the `-follow` operator, **find** locates the file:

```
$ find . -name night.gif -follow -print
./jscript/night.gif
```

Note the relative pathname **find** displays, but that is because the path list itself was relative (`.`).

**Matching Modification Times (-mtime)** You can use the `-mtime` operator to match a file's modification time. (`-atime` matches the access time.) This command shows the files that have been modified in less than two days—starting from the current directory:

```
# find . -mtime -2 -print
.
./unit13
./unit15
./unit14
```

*Includes the current directory also*

As can be seen above, **find** doesn't necessarily display an ASCII sorted list. The sequence in which the files are displayed depends on the internal organization of the file system. Further, by using the same option twice, you can narrow down your search condition to represent a range:

```
$ find . -mtime +2 -mtime -5 -ls
37353  2 -rwxr-xr-x  1 sumit  dialout      26 Apr 21 21:38 ./toc.sh
37392 296 -rw-r--r--  1 sumit  dialout    150426 Apr 20 23:09 ./session4
37393 146 -rw-r--r--  1 sumit  dialout     73728 Apr 20 23:10 ./session5
37394  38 -rw-r--r--  1 sumit  dialout     17946 Apr 20 23:10 ./session6
```

**find** here uses the `-ls` action component to display a special listing of those files that were modified in more than two days *and* less than five days. `-ls`—an action component—runs the `ls -lids` command and shows the inode number (first column) and the file size in 512-Kbyte blocks (second column). Apart from `-print` and `-ls`, **find** uses two other action components that are discussed shortly.



**Note**

+365 means greater than 365 days, -365 means less than 365 days. For specifying exactly 365, use 365.

**find** also uses the operators `-a` and `-o` to signify the AND and OR conditions. Use of two selection criteria generally represents the AND condition; you know that already. But you need to use `-o` to locate both the **perl** and shell version of your script:

```
find /home -name binary.pl -o -name binary.sh -print
```

When a series of selection criteria are specified without using the `-a` or `-o` operators, they translate into an AND condition. A previous example used `-mtime` twice without using `-a`; it is implied. Just to make sure, try both these commands and see if you notice any difference:



#### Note

```
find . -mtime +2 -mtime -5 -ls
find . -mtime +2 -a -mtime -5 -ls
```

**Taking Action on Selected Files (-exec and -ok)** All these operators produce a list of files on the terminal (which you can save in a file using the operator `>`). In one instance, we displayed the listing of the files as well (with `-ls`). In real life, however, you'll want to take some action on them—like deleting them. This is done with the `-exec` operator, followed by the command to be executed and terminated with the sequence `{}` `\;`.

The following command removes all temporary files in `/var/preserve` that are more than a month old:

```
find /var/preserve -mtime +30 -exec rm -f {} \;           {} represents filename
```

`-exec` permits execution of any UNIX command. The filename here is specified with the curly braces `{}` and **rm** forces removal. Note that the `exec` sequence is terminated with `\;`. The usage of this operator is quite cryptic, but it's one thing you can't afford to forget.

Removing a file without confirmation can be quite risky, so you can use the `-ok` operator instead of `-exec` for interactive deletion. This is how **find** selectively removes files that are larger than 2000 blocks and have not been accessed in 180 days:

```
# find /home -size +2000 -atime +180 -ok rm {} \;
< rm ... /home/romeo/README >?  y
< rm ... /home/juliet/README >?  n
.....
```

Each file is presented for your decision. A `y` executes the command; any other response leaves the file undeleted.

**find** is the system administrator's tool and in Chapter 22, you'll see it used for a host of tasks. It is especially suitable for backing up files.



#### Note

When using `-exec` or `-ok` with **find**, represent the filename with `{}` but don't forget to terminate the UNIX command line with `\;`.



Linux

UNIX **find** displays the filenames only if the `-print` operator is used. However, GNU **find** doesn't need this option; it prints by default. It also doesn't need the path list; it uses the current directory by default. In other words, the command line **find -name "\*.java"** matches and prints all `.java` files found in the current subdirectory tree. To “simplify” matters further, GNU **find** doesn't need any arguments at all; **find** used by itself displays recursively all files in the current directory tree!

The `-iname` operator makes the match case-insensitive. The options `-mmin` and `-amin` use the minute as the unit for matching rather than the hour used by `-mtime` and `-atime`.

## SUMMARY

You can use the **ls** command to list files in any manner—in multiple columns (`-x`), to identify directories and executables (`-F`) and display hidden files beginning with a dot (`-a`). You can reverse the sort order (`-r`) and get a recursive list (`-R`).

By default, **ls** sorts files in *ASCII collating sequence*, which means numbers precede uppercase, which in turn precede lowercase.

The UNIX file has a number of attributes, and seven of them are listed with **ls -l**. They are the permissions, links, owner and group owner, size, date and time of last modification and the filename. **ls -ld** lists directory attributes.

The size of the file in bytes is not the actual space the file occupies on disk, but the number of bytes it contains. A file containing 1 byte will actually occupy 1024 bytes on disk.

A file can have read, write or executable permission, and there can be three sets of such permissions for the owner, the group owner of the file, as well as others.

**chmod** is used to alter these permissions, and can be used only by the owner of the file. The permissions can be *relative* when used with the `+` or `-` symbols, or *absolute* when used with octal numbers.

The significance of directory permissions differs from ordinary files. Read permission means that the filenames stored in the directory are readable. **ls** works by reading the directory file. Write permission allows you to create or remove files in the directory. Executable permission lets you “cd” to the directory.

The **umask** setting determines the default permissions that will be used when creating a file or a directory.

A file has an *owner*, usually the name of the user who creates the file. A file is also owned by a *group*, by default, the group to which the user belongs. Only the owner can change the file attributes. **chown** and **chgrp** are used to transfer ownership and group ownership, respectively.

A file has both a *modification* and an *access time*. **ls** can sort a file by its modification time (`-t`) or access time (`-ut`). **touch** changes these times to any arbitrary values.

Multiple *file systems*, each with its own root directory, merge at system startup to form a single file system. A file is identified by the *inode number*, and its attributes are stored in the *inode*. The inode number is displayed by **ls -li**.

A file can have more than one name (*hard link*). You can link files with **ln** and remove a link with **rm**. Links have the same inode number. A linked file can behave as two separate commands depending on the name by which it is invoked.

A *symbolic link* (soft link) is an ordinary file that points to the location of another file even if it is in another file system. Unlike hard links, the soft link can also link directories. Symbolic links are created with **ln -s** but removed with **rm**.

**find** looks for files satisfying certain criteria which can be any file attribute. A file can be specified by type (**-type**), name (**-name**), size (**-size**), permissions (**-perm**) or by its time stamps (**-mtime** and **-atime**). Any UNIX command can be run on the selected files (**-exec** and **-ok**) with or without confirmation.

## SELF-TEST

---

- 7.1 What is the sort order prescribed by the ASCII collating sequence?
- 7.2 Which **ls** option marks directories and executables separately?
- 7.3 What are hidden files?
- 7.4 What do you mean by the *listing* of a file?
- 7.5 How will you obtain a complete listing of all files and directories in the whole system?
- 7.6 How will you list the files of the parent directory?
- 7.7 How do you identify directories from the listing?
- 7.8 Who can change the attributes of a file or directory?
- 7.9 Does **ls -l** show all files?
- 7.10 When is the time (but not date) of modification not shown in the listing?
- 7.11 For a group member to be able to remove a file what does she require?
- 7.12 First create a file. How will you now assign all permissions to the owner and remove all permissions from others assuming that the default file permissions are **rw-r--r--**?
- 7.13 You removed the write permission of a file from group and others, and yet they could delete your file. How could that happen?
- 7.14 How do you display the inode number of a file?
- 7.15 Where are the ownership and group ownership details stored?
- 7.16 Transfer recursively the ownership of all files in the current directory to henry.
- 7.17 Change the modification time of a file to Sep 30, 10:30 a.m.
- 7.18 What will the command **touch foo** do?
- 7.19 What does the inode store?
- 7.20 What do you mean by saying that a file has three *links*?
- 7.21 How do you remove a linked file?
- 7.22 Frame a command to locate all the **.html** and **.java** files in the system.

## EXERCISES

---

- 7.1 On executing `ls bar`, you see a list of 10 files of which `bar` is one of them. How can that happen?
- 7.2 How do you display all files recursively (including the hidden ones) in multiple columns with distinguishing marks on executables and directories?
- 7.3 Does the owner belong to the same group as the group owner of a file?
- 7.4 A file contains 1026 bytes. How many bytes of disk space does it occupy?
- 7.5 How do you list the attributes of the current directory?
- 7.6 Why is the size of a directory usually small?
- 7.7 If the file doesn't have write permission for the owner, can she remove it?
- 7.8 Show the octal representation of these permissions:  
(i) `rxr-xrw-` (ii) `rw-r-----`
- 7.9 What will the permissions string look like for these octal values?—(i) 567  
(ii) 623
- 7.10 You tried to copy a file `foo` from another user's directory, but you got the error message `cannot create file foo`. You have write permission in your own directory. What could be the reason and how do you copy the file?
- 7.11 What does `chmod -w foo` do?
- 7.12 If a directory has the permissions `777` and a file in it has the permissions `000`, what are the important security implications?
- 7.13 What do you do to make sure that no one can see the names of the files you have?
- 7.14 If you are not able to change to a directory, what could be the likely cause?
- 7.15 Which file attributes change when you copy a file from another user account?
- 7.16 How can you copy a file while preserving the attributes?
- 7.17 If the owner doesn't have write permission on a file but her group has, can she edit it?
- 7.18 How is `chown` different from `chgrp` when it comes to renouncing ownership?
- 7.19 If you make one change to a file, undo it, and then exit the editor after saving, is the file considered to be modified?
- 7.20 How can you find out whether a program has been executed today?
- 7.21 What's the difference between `ls -l` and `ls -lt`?
- 7.22 What's the difference between `ls -lu` and `ls -lut`?
- 7.23 How can you make out whether two files are copies or links?
- 7.24 What are the two main disadvantages of the hard link?
- 7.25 You have a number of programs in `$HOME/progs` which are called by other programs. You have now decided to move these programs to `$HOME/internet/progs`. How can you ensure that users don't notice this change?
- 7.26 How does the administrator ensure that all files created by users will have the default permissions `rw-rw--`?
- 7.27 Find out from the `/bin` and `/usr/bin` directories all files that begin with `z`.

- 7.28 Use *only* **find** to locate the file `login.sql` in the `/oracle` directory tree, and then copy it to your own directory.
- 7.29 Use **find** to move all files modified within the last 24 hours to the `posix` directory under your parent directory.

## KEY TERMS

---

**absolute permission** (7.5.2)

**ASCII collating sequence** (7.1)

**file access time** (7.10)

**file group owner** (7.2)

**file modification time** (7.2)

**file owner** (7.2)

**file permission** (7.2)

**file system** (7.12)

**GUID** (7.8.1)

**hard link** (7.14)

**incremental backup** (7.11)

**inode** (7.12)

**inode number** (7.12)

**link** (7.2 and 7.13)

**listing** (7.2)

**others category** (7.4)

**primary group** (7.8.1)

**relative permission** (7.5.1)

**root file system** (7.12)

**secondary group** (7.8.1)

**soft link** (7.14)

**symbolic link** (7.14)

**UID** (7.8.1)

**user mask** (7.7)

**world category** (7.4)