11

More File Attributes

Apart from permissions and ownership, a UNIX file has several other attributes, and in this chapter, we look at most of the remaining ones. A file also has properties related to its time stamps and links. It's important to know how these attributes are interpreted when applied to a directory or a device. We need to revisit the **ls** command and see some of its other options that reveal these attributes. We'll also discuss **find**—one of the most versatile attribute-handling tools of the UNIX system.

This chapter also introduces the concept of a *file system*. It also looks at the *inode*, the lookup table that contains almost all file attributes. Though a detailed treatment of file systems is taken up later, knowledge of its basics is essential to our understanding of the significance of some of the file attributes. Moreover, some administration tools act and report on individual file systems separately.

WHAT YOU WILL LEARN

- The concept of the *file system* and how multiple file systems are seen as one.
- The use of the *inode* to store file attributes and **ls** to display the inode number.
- Use **ln** to create a *hard link* by providing a file with a different name.
- The limitations of hard links and how they are overcome by *symbolic links*.
- The concept of the *user mask* and how **umask** changes the default file and directory permissions.
- How to display the listing in order of a file's time stamps.
- Use **touch** to change a file's modification and access times.

TOPICS OF SPECIAL INTEREST

- Three possible situations where hard links can be very useful.
- The significance of directory permissions and how they ultimately impact a file's access rights.
- The elaborate syntax used by **find** to match one or more file attributes, and take action on selected files.
- How It Works: A graphic that shows how **ln** and **rm** affect the inode and directory.

11.1 FILE SYSTEMS AND INODES

We now need to have some idea of the way files are organized in a UNIX system. So far, we have been referring to the UNIX file hierarchy as a "file system" as if all files and directories are held together in one big superstructure. That is seldom the case, and never so in large systems. The hard disk is split into distinct **partitions** (or *slices*), with a separate file system in each partition (or slice).

Every **file system** has a directory structure headed by root. If you have three file systems on one hard disk, then they will have three separate root directories. When the system is up, we see only a single file system with a single root directory.

Of these multiple file systems, one of them is considered to be 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 combined UNIX system. At the time of booting, all secondary file systems *mount* (attach) themselves to the main file system, creating the illusion of a single file system to the user.

Every file is associated with a table that contains all that you could possibly need to know about a file—except its name and contents. This table is called the **inode** (shortened from index node) and is accessed by the **inode number**. The inode contains the following attributes of a file:

- File type (regular, directory, device, etc.).
- File permissions (the nine permissions and three more).
- Number of links (the number of aliases the file has).
- The UID of the owner.
- The GID of the group owner.
- File size in bytes.
- Date and time of last modification.
- Date and time of last access.
- Date and time of last change of the inode.
- An array of pointers that keep track of all disk blocks used by the file.

Observe that *neither the name of the file nor the inode number is stored in the inode*. It's the directory that stores the inode number along with the filename. When you use a command with a filename as argument, the kernel first locates the inode number of the file from the directory and then reads the inode to fetch data relevant to the file.

Every file system has a separate portion set aside for storing inodes, where they are laid out in a contiguous manner. This area is accessible only to the kernel. The inode number is actually the *position* of the inode in this area. The kernel can locate the inode number of any file using simple arithmetic. Since a UNIX machine usually comprises multiple file systems, you can conclude that the inode number for a file is unique in a *single* file system.

The **ls** command reads the inode to fetch a file's attributes, and it can list most of them using suitable options. One of them is the **-i** (inode) option that tells you the inode number of a file:

inode numbers

```
$ ls -il tulec05
9059 -rw-r--r--   1 kumar    metal        51813 Jan 31 11:15 tulec05
```

The file tulec05 has the inode number 9059. No other file *in the same file system* can have the number unless the file is removed. When that happens, the kernel will allocate this inode number to a new file.

Note: The inode contains all attributes of a file except the filename. The filename is stored in the directory housing the file. The inode number is also not stored in the inode, but the kernel can locate any inode by its position since its size is fixed.

HOW IT WORKS: How cat and ls Work

When you run **cat foo**, the kernel first locates the inode number of foo from the current directory. It then reads the inode for foo to fetch the file size and addresses of the disk blocks that contain the file's data. It then goes to each block and reads the data until the number of characters displayed is equal to the file size.

When you execute **ls -l progs** where progs is a directory, the kernel looks up the directory progs and reads all entries. For every entry, the kernel looks up the inode to fetch the file's attributes.

11.2 HARD LINKS

Why is the filename not stored in the inode? So that a file can have multiple filenames. When that happens, we say the file has more than one **link**. We can then access the file by any of its links. All names provided to a single file have one thing in common; they all have the same inode number. The link count is displayed in the second column of the listing. This count is normally 1 (as shown in the previous listing), but the following files have two links:

```
-rwxr-xr--   2 kumar    metal         163 Jul 13 21:36 backup.sh
-rwxr-xr--   2 kumar    metal         163 Jul 13 21:36 restore.sh
```

All attributes seem to be identical, but the "files" could still be copies (which have different inode numbers). It's the link count that seems to suggest that the "files" are linked to each other. But this can only be confirmed by using the **-i** option to **ls**:

```
$ ls -li backup.sh restore.sh
478274 -rwxr-xr--   2 kumar    metal         163 Jul 13 21:36 backup.sh
478274 -rwxr-xr--   2 kumar    metal         163 Jul 13 21:36 restore.sh
```

Both "files" indeed have the same inode number, so there's actually only one file with a single copy on disk. We can't really refer to them as two "files", but only as two "filenames". This file simply has two aliases; changes made in one alias (link) are automatically available in the others. There are two entries for this file in the directory, both having the same inode number.

11.2.1 ln: Creating Hard Links

A file is linked with the **ln** (link) command, which takes two filenames as arguments. The command can create both a *hard* and a *soft* link (discussed later) and has a syntax similar to the one used by **cp**. The following command (hard) links emp.lst with employee:

```
ln emp.lst employee                          employee must not exist
```

The **-i** option to **ls** shows that they have the same inode number, meaning that they are actually one and the same file:

```
$ ls -li emp.lst employee
29518 -rwxr-xr-x   2 kumar    metal         915 May  4 09:58 emp.lst
29518 -rwxr-xr-x   2 kumar    metal         915 May  4 09:58 employee
```

The link count, which is normally one for unlinked files, is shown to be two. You can link a third filename, emp.dat, and increase the number of links to three:

```
$ ln employee emp.dat ; ls -l emp*
29518 -rwxr-xr-x   3 kumar    metal         915 May  4 09:58 emp.dat
29518 -rwxr-xr-x   3 kumar    metal         915 May  4 09:58 emp.lst
29518 -rwxr-xr-x   3 kumar    metal         915 May  4 09:58 employee
```

You can link multiple files (i.e., create a link for each), but then the destination filename must be a directory. Here's how you create links for all the chapters of this text in the directory project8_dir:

```
ln chap?? project8_dir                       project8_dir is a directory
```

If chap?? matches 25 files, there will be 25 linked filenames in project8_dir, i.e., there will be 25 entries in that directory. The **rm** command removes a file by deleting its directory entry, so we expect the same command to remove a link also:

```
$ rm emp.dat ; ls -l emp.lst employee
-rwxr-xr-x   2 kumar    metal         915 May  4 09:58 emp.lst
-rwxr-xr-x   2 kumar    metal         915 May  4 09:58 employee
```

The link count has come down to two. Another **rm** will further bring it down to one. A file is considered to be completely removed from the system when its link count drops to zero.

Tip: **ln** returns an error when the destination file exists. Use the **-f** option to force the removal of the existing link before creation of the new one.

11.2.2 Where to Use Hard Links

Links are an interesting feature of the file system, but where does one use them? We can think of three situations straightaway:

1. Let's consider that you have written a number of programs that read a file foo.txt in $HOME/input_files. Later, you reorganized your directory structure and moved foo.txt to $HOME/data instead. What happens to all the programs that look for foo.txt at its original location? Simple, just link foo.txt to the directory input_files:

HOW IT WORKS: ln and rm Affect the Directory and Inode

The role of **rm** and **ln** are complementary which is evident from the names of their corresponding system calls—link and unlink. The ln command adds an entry in the directory and also increases the link count in the inode by one. The **rm** command simply reverses the action of **ln**. The effect they have on the inode and directory is depicted in Fig. 11.1.

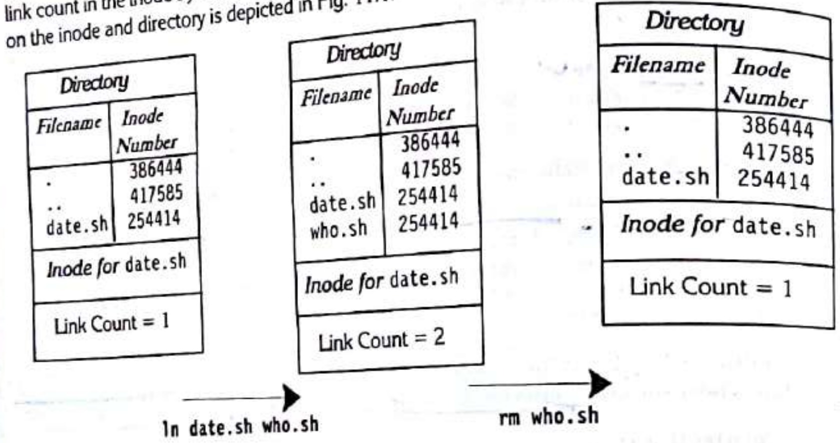

Fig. 11.1 Effect of **ln** and **rm** on Directory and Inode

```
ln data/foo.txt input_files
```
Creates link in directory `input_files`

With this link available, your existing programs will continue to find foo.txt in the input_files directory. It's more convenient to do this that modify all programs to point to the new path.

2. Links provide some protection against accidental deletion, especially when they exist in different directories. Referring to the previous application, even though there's only a single file foo.txt on disk, you have effectively made a backup of this file. If you inadvertently delete input_files/foo.txt, one link will still be available in data/foo.txt; your file is not gone yet.

3. Because of links, we don't need to maintain two programs as two separate disk files if there is very little difference between them. A file's name is available to a C program (as argv[0]) and to a shell script (as $0). A single file with two links can have its program logic make it behave in two different ways depending on the name by which it is called. There's a shell script using this feature in Section 14.10.

Many UNIX commands are linked. Refer to Section 5.18, where we posed the question whether **gzip** and **gunzip** were two separate files. This question can now easily be answered by looking at their inode numbers:

```
13975 -r-xr-xr-x   3 root     bin        60916 Jan  5  2000 gunzip
13975 -r-xr-xr-x   3 root     bin        60916 Jan  5  2000 gzip
```

looking for the entries with the same inode no.

They are, in fact, one and the same file. The listing shows the existence of a third link as well, but how does one locate it? Doing an **ls -li** and then looking for entries with the same inode number isn't a practical approach. It may not also work since links could be spread across multiple directories. For this task, we need the **find** command which is discussed at the end of this chapter.

11.3 SYMBOLIC LINKS AND ln

We have seen how links let us have multiple names for a file. These links are often called **hard links**, and have two limitations:

- You can't have two linked filenames in two file systems. In other words, you can't link a filename in the /usr file system to another in the /home file system.
- You can't link a directory even within the same file system.

This serious limitation was overcome when **symbolic links** made their entry. Until now, we have divided files into three categories (ordinary, directory and device). The symbolic link is the fourth file type considered in this text. Unlike the hard link, a symbolic link doesn't have the file's contents, but simply provides the pathname of the file that actually has the contents. Being more flexible, a symbolic link is also known as a **soft link**. Windows shortcuts are more like symbolic links.

The **ln** command creates symbolic links also, except that you have to use the -s option. This time the listing tells you a different story:

length of the pathname

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

You can identify symbolic links by the character l (el) seen in the permissions field. The pointer notation -> note suggests that note.sym contains the pathname for the filename note. It's note, and not note.sym, that actually contains the data. When you use **cat note.sym**, you don't actually open the symbolic link, note.sym, but the file the link points to. Observe that the size of the symbolic link is 4; this is the length of the pathname it contains (note).

It's important you realize that this time we indeed have two "files", and they are not identical. Removing note.sym won't affect us much because we can easily recreate the link. But if we remove note, we would lose the file containing the data. In that case, note.sym would point to a nonexistent file and become a *dangling* symbolic link.

Symbolic links can also be used with relative pathnames. Unlike hard links, they can also span multiple file systems and also link directories. If you have to link *all* filenames in a directory to another directory, it makes sense to simply link the directories. Like other files, a symbolic link has a separate directory entry with its own inode number. This means that **rm** can remove a symbolic link even if its points to a directory (or even a device).

LINUX: A symbolic link has an inode number separate from the file that it points to. In most cases, the pathname is stored in the symbolic link and occupies space on disk. However, Linux uses a *fast symbolic link* which stores the pathname in the inode itself provided it doesn't exceed 60 characters.

11.4 THE DIRECTORY

A directory has its own permissions, owners and links. However, the significance of the file attributes change a great deal when applied to a directory. For example, the size of a directory is in no way related to the size of files that exist in the directory, but rather to the *number* of files housed by it. The higher the number, the larger is the directory.

Permissions too acquire a different meaning when the term is applied to a directory. Before we take up directory permissions, let's see what its default permissions are on this system:

```
$ ls -l -d progs
drwxr-xr-x   2 kumar    metal         320 May  9 09:57 progs
```

The default permissions are different from those of ordinary files. The user has all permissions, and group and others have read and execute permissions only. The permissions of a directory also impact the security of its files. To understand how that can happen, we must know what permissions for a directory really mean.

11.4.1 Read Permission

Read permission for a directory means that the *list* of filenames stored in that directory is accessible. Since **ls** reads the directory to display filenames, if a directory's read permission is removed, **ls** won't work. Consider removing the read permission first from the directory progs:

```
$ ls -ld progs
drwxr-xr-x   2 kumar    metal         128 Jun 18 22:41 progs
$ chmod -r progs ; ls progs
progs: Permission denied
total 2
```

Being unreadable, the progs directory couldn't be accessed by **ls**, and hence the error message. *However, this doesn't prevent you from reading the files separately if you know their names.*

11.4.2 Write Permission

Be aware that you can't write to a directory file; only the kernel can do that. If that were possible, any user could destroy the integrity of the file system. Write permission for a directory implies that you are permitted to create or remove files in it (that would make the kernel modify the directory entries). To try that out, restore the read permission and remove the write permission from the directory before you try to copy a file to it:

```
$ chmod 555 progs ; ls -ld progs
dr-xr-xr-x   2 kumar    metal         128 Jun 18 22:41 progs
$ cp emp.lst progs
cp: cannot create progs/emp.lst: Permission denied
```

The directory doesn't have write permission; you can't create, copy or delete a file in it. But can you modify its existing files? This question often confuses beginners, and if you are confused too, then follow this line of reasoning:

- The write permission for a directory determines whether you can create or remove files in it because these actions modify the directory.
- Whether you can modify a file depends solely on whether the file itself has write permission. Changing a file doesn't in any way modify its directory entry.

Note: The term "write-protected" has a limited meaning in the UNIX file system. A write-protected file can't be written, but it can be removed if the directory has write permission.

11.4.3 Execute Permission

Executing a directory just doesn't make any sense, so what does its execute privilege mean? It only means that a user can "pass through" the directory in searching for subdirectories. When you use a pathname with any command

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

you need to have execute permission for each of the directories involved in the complete pathname. The directory home contains the entry for kumar, and the directory kumar contains the entry for progs, and so forth. If a single directory in this pathname doesn't have execute permission, then it can't be searched for the name of the next directory. That's why the execute privilege of a directory is often referred to as the *search* permission.

A directory has to be searched for the next directory, so the **cd** command won't work if the search permission for the directory is turned off:

```
$ chmod 666 progs ; ls -ld progs
drw-rw-rw-   2 kumar    metal         128 Jun 18 22:41 progs
$ cd progs
bash: cd: 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, make sure you set them correctly. If you don't, then be assured that an intelligent user could make life miserable for you!

Let's now add a qualifier to the two bulleted observations made regarding write permission for a directory. To be able to create or remove files, write permission for the directory is not enough; the directory must have search (execute) permission as well.

Caution: Danger arises when you mistakenly assign the permissions 775 or 777 to a directory. In the present scenario, 775 allows any user of the metal group to create or remove files in the directory. 777 extends this facility to the world. As a rule, you should never make directories group- or world-writable unless you have definite reasons to do so.

11.5 umask: DEFAULT FILE AND DIRECTORY PERMISSIONS

When you create files and directories, the permissions assigned to them depend on the system's default setting. The UNIX system has the following default permissions for all files and directories:

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

However, you don't see these permissions when you create a file or directory. Actually, this default is transformed by subtracting the **user mask** from it to remove one or more permissions. To understand what this means, let's evaluate the current value of the mask by using **umask** without arguments:

```
$ umask
022
```

This is an octal number which has to be subtracted from the system default to obtain the *actual* default. This becomes 644 (666 - 022) for ordinary files and 755 (777 - 022) for directories. When you create a file on this system, it will have the permissions `rw-r--r--`.

umask is a shell built-in command though it also exists as an external command. A user can also use this command to set a new default. Here's an extreme setting:

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

A umask value of 000 means that you haven't subtracted anything, and this could be dangerous. The system's default then applies (666 for files and 777 for directories). All files and directories are then writable by all; nothing could be worse than that! However, a mask value of 666 or 777 doesn't make much sense either; you'll then be creating files and directories with no permissions.

The important thing to remember is that, no one—not even the administrator—can use **umask** to turn on permissions not specified in the systemwide default settings. However, you can always use **chmod** as and when required. The systemwide umask setting is placed in one of the machine's startup scripts, and is automatically made available to all users.

11.6 MODIFICATION AND ACCESS TIMES

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`

Whenever you write to a file, the time of last modification is updated in the file's inode. A directory can be modified by changing its entries—by creating, removing and renaming files in the directory. Note that changing a file's contents only changes its last modification time but not that of its directory. `ls -l` displays the last modification time.

r 4
w 2
x 1

A file also has an access time, i.e., the last time someone read, wrote or executed the file. This time is distinctly different from the modification time that gets set only when the contents of the file are changed. For a directory, the access time is changed by a read operation only; creating or removing a file or doing a "cd" to a directory doesn't change its access time. The access time is displayed when `ls -l` is combined with the `-u` option.

Even though `ls -l` and `ls -lu` show the time of last modification and access, respectively, the sort order remains standard, i.e. ASCII. However, when you add the `-t` option to `-l` or `-lu`, the files are actually displayed in *order* of the respective time stamps:

```
ls -lt      Displays listing in order of their modification time
ls -lut     Displays listing in order of their access time
```

Knowledge of a file's modification and access times is extremely important for the system administrator. Many of the tools used by him 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, you can use **touch** to reset the times to certain convenient values without actually modifying or accessing the file. **touch** is discussed next.

11.6.1 touch: Changing the Time Stamps

As has just been 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 *expression*, both times are set to the current time. The file is created if it doesn't exist:

```
touch emp.lst                                Creates file if it doesn't exist
```

When **touch** is used without options but with *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:

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

It's also possible to change the two times individually. The -m and -a options change the modification and access times, respectively:

```
$ touch -m 02281030 emp.lst ; ls -l emp.lst
-rw-r--r--   1 kumar    metal        870 Feb 28 10:30 emp.lst
$ touch -a 01261650 emp.lst ; ls -lu emp.lst
-rw-r--r--   1 kumar    metal        870 Jan 26 16:50 emp.lst
```

The system administrator often uses **touch** to "touch up" these times so a file may be included in or excluded from an *incremental backup* (that backs up only changed files). The **find** command can then be used to locate files that have changed or have been accessed after the time set by **touch**. **find** is the last command we discuss in this chapter and is taken up next.

11.7 find: LOCATING FILES

find is one of the power tools of the UNIX system. It *recursively* examines a directory tree to look for files matching some criteria, and then takes some action on the selected files. 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
```

This is how **find** operates:

- First, it recursively examines all files in the directories specified in *path_list*.
- It then matches each file for one or more *selection_criteria*.
- Finally, it takes some *action* on those selected files.

The *path_list* comprises one or more subdirectories separated by whitespace. There can also be a host of *selection_criteria* that you can use to match a file, and multiple *actions* to dispose of the file. 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.

As our first example, let's use **find** to locate all files named **a.out** (the executable file generated by the C compiler):

```
$ find / -name a.out -print
/home/kumar/scripts/a.out
/home/tiwary/scripts/reports/a.out
/home/sharma/a.out
```

The path list (/) indicates that the search should start from the root directory. Each file in the list is then matched against the selection criteria (-name a.out), which always consists of an expression in the form *-operator argument*. If the expression matches the file (i.e., the file has the name a.out), the file is selected. The third section specifies the action (-print) to be taken on the files; in this case, a simple display on the terminal. All **find** operators start with a -, and the *path_list* can never contain one.

LINUX: **find** in UNIX displays the file list only if the -print operator is used. However, Linux doesn't need this option; it prints by default. Linux also doesn't need the path list; it uses the current directory by default. Linux even prints the entire file list when used without any options whatsoever! This behavior is not required by POSIX.

You can also use relative names (like the .) in the path list, and **find** will then output a list of relative pathnames. When **find** is used to match a group of filenames with a wild-card pattern, the pattern should be quoted to prevent the shell from looking at it:

```
find . -name "*.c" -print                  All files with extension .c
find . -name '[A-Z]*' -print               Single quotes will also do
```

The first command looks for all C program source files in the current directory tree. The second one searches for all files whose names begin with an uppercase letter. You must not forget to use the -print option because without it, **find** on UNIX systems will look for files all right but won't print the list.

-name is not the only operator used in framing the selection criteria; there are many others (Table 11.1). The actual list is much longer, and takes into account practically every file attribute. Let's now take a look at some of the important ones. We'll consider the selection criteria first, and then the possible actions we can take on the selected files.

11.7.1 Selection Criteria

Locating a File by Inode Number (-inum) Refer to Section 11.2.2, where we found that **gzip** has three links and **gunzip** was one of them. **find** allows us to locate files by their inode number. Use the -inum option to find all filenames that have the same inode number:

```
$ find / -inum 13975 -print                Inode number obtained from Section 11.2.2
find: cannot read dir /usr/lost+found: Permission denied
/usr/bin/gzip
/usr/bin/gunzip
/usr/bin/gzcat                             "Cats" a compressed file
```

Now we know what the three links are. Note that **find** throws an error message when it can't change to a directory. Sometimes, there will be so many of them on your screen that you would find it difficult to spot the files that actually show up as **find**'s output. To avoid these messages, simply redirect the standard error to /dev/null.

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

```
$ cd; find . -type d -print 2>/dev/null
.                                          Shows the . also
./.netscape                                Displays hidden directories also
./java_progs
./c_progs
./c_progs/include
./c_progs/lib
./shell_scripts
./.ssh
```

Note the relative pathname **find** displays, but that's because the pathname *itself was relative* (.). **find** also doesn't necessarily display an ASCII sorted list. The sequence *in which files are displayed* depends on the internal organization of the file system.

The -perm option specifies the permissions to match. 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 options in combination to restrict the search to only directories:

```
find $HOME -perm 777 -type d -print
```

find uses an AND condition (an implied -a operator between -perm and -type) to select directories that provide all access rights to everyone. It selects files only if both selection criteria (-perm and -type) are fulfilled.

Finding Unused Files (-mtime and -atime) Files tend to build up incessantly on disk. Some of them remain unaccessed or unmodified for months—even years. **find**'s options can easily match a file's modification (-mtime) and access (-atime) times to select them. -mtime helps in backup operations by providing a list of those files that have been modified, say, in less than 2 days:

```
find . -mtime -2 -print
```

-2 here means *less* than 2 days. To select from the /home directory all files that have not been accessed for more than a year, a positive value has to be used with -atime:

```
find /home -atime +365 -print | mailx root
```

Because **find** uses standard output, the list can be stored in a file or used to mail a message.

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

11.7.2 The find Operators (!, -o and -a)

There are three operators that are commonly used with **find**. The ! operator is used before an option to negate its meaning. So,

```
find . ! -name "*.c" -print
```

selects all but the C program files. To look for both shell and **perl** scripts, use the -o operator which represents an OR condition. We need to use an escaped pair of parentheses here:

```
find /home \( -name "*.sh" -o -name "*.pl" \) -print
```

The (and) are special characters that are interpreted by the shell to group commands (*2.7.1*). The same characters are used by **find** to group expressions using the -o and -a operators, the reason why they need to be escaped.

The -a operator represents the AND condition, and is implied by default whenever two selection criteria are placed together.

Table 11.4 Major Expressions Used by **find** (Meaning gets reversed when - is replaced by +, and vice versa)

Selection Criteria	*Selects File*
-inum *n*	Having inode number *n*
-type *x*	If of type *x*; *x* can be f (ordinary file), d (directory) or l (symbolic link)
-type f	If an ordinary file
-perm *nnn*	If octal permissions match *nnn* completely
-links *n*	If having *n* links
-user *usname*	If owned by *usname*
-group *gname*	If owned by group *gname*
-size +*x*[*c*]	If size greater than *x* blocks (characters if *c* is also specified) (Chapter 15)
-mtime -*x*	If modified in less than *x* days
-newer *flname*	If modified after *flname* (Chapter 25)
-mmin -*x*	If modified in less than *x* minutes (Linux only)
-atime +*x*	If accessed in more than *x* days
-amin +*x*	If accessed in more than *x* minutes (Linux only)
-name *flname*	*flname*
-iname *flname*	As above, but match is case-insensitive (Linux only)
-follow	After following a symbolic link
-prune	But don't descend directory if matched
-mount	But don't look in other file systems (Chapter 25)

Action	*Significance*
-print	Prints selected file on standard output
-ls	Executes **ls -lids** command on selected files
-exec *cmd*	Executes UNIX command *cmd* followed by {} \;

11.7.3 Options Available in the Action Component

Displaying the Listing (-ls) The -print option belongs to the *action* component of the **find** syntax. In real life, you'll often want to take some action on the selected files and not just display the filenames. For instance, you may like to view the listing with the -ls option:

```
$ find . -type f -mtime +2 -mtime -5 -ls                    -a option implied
475336    1 -rw-r--r--   1 romeo     users    716 Aug 17 10:31 ./c_progs/fileinout.c
```

find here runs the **ls -lids** command to display a special listing of those regular files that are modified in more than two days and less than five days. In this example, we see two options in the selection criteria (both -mtime) simulating an AND condition. It's the same as using \(-mtime +2 -a -mtime -5 \).

Taking Action on Selected Files (-exec and -ok) The -exec option is the dark horse of the **find** tool. It lets you take any action by running a UNIX command on the selected files. -exec takes the command to execute as its own argument, followed by {} and finally the rather cryptic symbols \; (backslash and semicolon). This is how you can reuse a previous **find** command quite meaningfully:

```
find $HOME -type f -atime +365 -exec rm {} \;
```

Note the usage

This will use **rm** to remove all ordinary files unaccessed for more than a year. This can be a thing to do, so you can consider using **rm**'s -i option. But all commands don't have inte options, in which case you should use **find**'s -ok option:

```
$ find $HOME -type f -atime +365 -ok mv {} $HOME/safe \;
< mv ... ./archive.tar.gz > ? y
< mv ... ./yourunix02.txt > ? n
< mv ... ./yourunix04.txt > ? y
  .......
```

mv turns interactive with -i but only if the destination file exists. Here, -ok seeks confirmatio every selected file to be moved to the $HOME/safe directory irrespective of whether the files e the destination or not. A y deletes the file.

find is the system administrator's tool. You'll see it used for a number of tasks in Chapter 15 specially suitable for backing up files and for use in tandem with the **xargs** command (25.1

Note: The pair of {} is a placeholder for a filename. So, **-exec cp {} {}.bak** provides a .bak exte to all selected files. Don't forget to use the \; symbols at the end of every -exec or -ok keyword

11.8 CONCLUSION

This chapter is a sequel to Chapter 6, and examined practically all the common file attributes. inode is at the heart of it all, but it contains 12 permissions for a file, and not 9, as you mig tempted to think. We'll be examining the three remaining permission bits—the SUID, SGID the sticky bit—in Chapter 15. At this point we have covered most of the concepts, so it's tim applied them. The next chapter is the first of four chapters devoted to those powerful manipulators that we call *filters*.

WRAP UP

The UNIX directory tree is actually a collection of multiple *file systems* that are *mounted* at bo time to appear as a single file system. Every file is identified by the *inode number* and has i attributes stored in the inode. The inode number is unique in a single file system.

A file can have more than one name or *link*, and is linked with **ln**. Two linked filenames have th same inode number. Links provide protection against accidental deletion. A C or shell programme takes advantage of the linking feature to write code that does different things depending on the name by which the file is invoked.

A *symbolic link* is a file which contains the pathname of another file or directory even if it is in another file system. It is created with **ln -s**. Accidental deletion of the file pointed to is dangerous and creates a dangling symbolic link.

Permissions have different significance for directories. Read permission means that the filenames stored in the directory are readable. Write permission implies that you are permitted to create or

remove files in the directory. Execute (or search) permission means that you can change to that directory with the **cd** command.

The UNIX system creates files and directories with 666 and 777 as the default permissions. But this default is generally changed by **umask** in the system's startup scripts. A umask value of 022 changes the system default to 644 and 755 for a file and directory, respectively.

The inode stores the time of last modification and access of a file. You can use **touch** to change both these times to either the current time, by default, or to specific values.

find looks for files by matching one or more file attributes. A file can be specified by type (-type), name (-name), permissions (-perm) or by its time stamps (-mtime and -atime). -print is the action commonly used, but any UNIX command can be run on the selected files (-ls, -exec and -ok) with or without user confirmation.

Test Your Understanding

11.1 Which important file attribute is not maintained in the inode? Where is it stored then?

11.2 What do you mean by saying that a file has three links?

11.3 How do you link all C source files in the current directory and place the links in another directory, bar?

11.4 State whether the following are true or false: (i) The **rm** command removes only a hard link but not a symbolic link. (ii) A symbolic link has the same inode number as the file it is linked to.

11.5 How do you link foo1 to foo2 using (i) a hard link, (ii) a symbolic link? If you delete foo2, does it make any difference?

11.6 How can you make out whether two files are copies or links?

11.7 What do you do to make sure that no one is able see the names of the files you have?

11.8 A file was not writable by group and others and yet could be deleted by users of those categories. How?

11.9 How do you ensure that all files created by you will have the default permissions rw-rw----?

11.10 When you invoke **ls -l foo** the access time of foo changes. True or false?

11.11 How can you find out whether a program has been executed today?

11.12 What does the command **touch foo** do? Why is the command important for the system administrator?

11.13 Copy the file /etc/passwd to your current directory and then observe the listing of the copy. Which attributes have changed?

11.14 How do you change the modification time of a file to Sep 30, 10:30 a.m.?

11.15 Use **find** to locate in /bin and /usr/bin all filenames that (i) begin with z, (ii) have the extension .html or .java.

11.16 How will you count the number of ordinary files in the directory tree, /home/henry?

11.17 Observe the access time of a file with **ls -lu foo**. Next, append the **date** command output to it and observe the access time of foo again. What do you see?

Flex Your Brain

11.1 Explain whether the following statement is true: The UNIX file system has many root directories even though it actually shows one.

11.2 What are the three time stamps maintained in the inode and how do you display two of them for the file foo?

11.3 What change takes place in the inode and directory when a file is linked and later removed?

11.4 What happens when you invoke the command **ln foo bar** if (i) bar doesn't exist, (ii) bar exists as an ordinary file, (iii) bar exists as a directory?

11.5 Explain two application areas of hard links. What are the two main disadvantages of the hard link?

11.6 Explain the significance of fast symbolic links and dangling symbolic links.

11.7 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?

11.8 The command **cd bar** failed. When can that happen even if bar exists?

11.9 If a file has the permissions 000, you may or may not be able to delete the file. Explain how both situations can happen. Does the execute permission have any role to play here?

11.10 If **umask** shows the value (i) 000 (ii) 002, what implications do they have from the security viewpoint?

11.11 Explain the difference between (i) **ls -l** and **ls -lt** (ii) **ls -lu** and **ls -lut**.

11.12 Use **find** to locate from your home directory tree all (i) files with the extension .html or .HTML, (ii) files having the inode number 9076, (iii) directories having permissions 666, (iv) files modified yesterday. Will any of these commands fail?

11.13 Use **find** to (i) move all files modified within the last 24 hours to the posix directory under your parent directory, (ii) locate all files named a.out or core in your home directory tree and remove them interactively, (iii) locate the file login.sql in the /oracle directory tree, and then copy it to your own directory, (iv) change all directory permissions to 755 and all file permissions to 644 in your home directory tree.

1

Simple Fi

This chapter features the simple filters of the system—commands which accept d input, manipulate it and write the results to standard output. Filters are the UNIX tool kit, and each filter featured in this chapter performs a simple fun shows their use both in standalone mode and in combination with other to and piping.

Many UNIX files have lines containing *fields*—strings of characters repr entity. Some commands expect these fields to be separated by a suitable del the data. Typically this delimiter is a **:** (as in /etc/passwd and $PATH), but as delimiter for some of the sample files in this and other chapters. Ma delimited fields, and some simply won't work without them.

What You Will Learn

- Use **pr** to format text to provide margins and headers, doublesp output.
- Pick up lines from the beginning with **head**, and from the end w
- Extract characters or fields with **cut**.
- Join two files laterally, and multiple lines to a single line with
- Sort, merge and remove repeated lines with **sort**.
- Find out the unique and nonunique lines with **uniq**.
- Change, delete or squeeze individual characters with **tr**.