CHAPTER 7

Managing Files

Much of what you do with a computer involves manipulating files. Most obviously, files hold the correspondence, spreadsheets, digital photos, and other documents you create. Files also hold the configuration settings for Linux—information on how to treat the network interfaces, how to access hard disks, and what to do as the computer starts up. Indeed, even access to most hardware devices and kernel settings is ultimately done through files. Thus, knowing how to manage these files is critically important for administering a Linux computer. This chapter begins with a description of the basic text-mode commands for manipulating files. Directories are files, too, so this chapter covers directories, including the commands you can use to create and manipulate them.

- **Manipulating files**
- **Manipulating directories**

Manipulating Files

If you've used Windows or Mac OS X, chances are you've used a GUI file manager to manipulate files. Such tools are available in Linux, as noted in Chapter 4, "Using Common Linux Programs," and you can certainly use a file manager for many common tasks. Linux's text-mode shells, such as Bash, provide simple but powerful tools for manipulating files, too. These tools can simplify some tasks, such as working with all the files with names that include the string `invoice`. Thus, you should be familiar with these text-mode commands.

To begin this task, I describe some ways you can create files. With files created, you can copy them from one location to another. You may sometimes want to move or rename files, so I explain how to do so. Linux enables you to create *links*, which are ways to refer to the same file by multiple names. If you never want to use a file again, you can delete it. *Wildcards* provide the means to refer to many files using a compact notation, so I describe them. Finally, I cover the case-sensitive nature of Linux's file manipulation commands.

Creating Files

Normally, you create files using the programs that manipulate them. For instance, you might use a graphics program to create a new graphics file. This process varies from one program to another, but GUI programs typically use a menu option called Save or Save As to save a file. Text-mode programs provide similar functionality, but the details of how it's done vary greatly from one program to another.

One program deserves special mention as a way to create files: `touch`. You can type this program's name followed by the name of a file you want to create, such as **`touch newfile.txt`** to create an empty file called `newfile.txt`. Ordinarily, you don't need to do this to create a file of a particular type, since you'll use a specialized program to do the job. Sometimes, though, it's helpful to create an empty file just to have the file itself—for instance, to create a few "scratch" files to test some other command.

Chapter 11, "Editing Files," describes how to create text files with the text-mode `pico`, and `nano`, and Vi editors.

Certification Objective

If you pass `touch` the name of a file that already exists, `touch` updates that file's access and modification time stamps to the current date and time. This can be handy if you're using a command that works on files based on their access times and you want the program to treat an old file as if it were new. You might also want to do this if you plan to distribute a collection of files and you want them all to have identical time stamps.

A programmer's tool known as `make` compiles source code if it's new, so programmers sometimes use `touch` to force `make` to recompile a source code file.

You can use a number of options with `touch` to modify its behavior. The most important of these are as follows:

Don't create a file The `-c` or `--no-create` option tells `touch` to not create a new file if one doesn't already exist. Use this option if you want to update time stamps but not accidentally create an empty file, should you mistype a filename.

Set the time to a specific value You can use `-d` *`string`* or `--date=`*`string`* to set the date of a file to that represented by the specified `string`, which can take any number of forms. For instance, **`touch -d "July 4 2012" afile.txt`** causes the date stamps on `afile.txt` to be set to July 4, 2012. You can achieve the same effect with `-t` `[[`*CC*`]`*YY*`]`*MMDDhhmm*`[.`*ss*`]`, where `[[`*CC*`]`*YY*`]`*MMDDhhmm* `[.`*ss*`]` is a date and time in a specific numeric format, such as 201207041223 for 12:23 PM on July 4, 2012.

Consult the `man` page for `touch` to learn about its more obscure options.

Copying Files

Certification Objective

If you're working in a text-mode shell, the `cp` command copies a file. (Its name is short for *copy*.) Its basic use is to pass it a source filename and a destination filename, a destination directory name, or both. Thus, there are three ways you

can use it, as outlined in Table 7.1. Although the example filenames in Table 7.1 suggest that the original file be in your current working directory, this need not be the case; *orig.txt* could include a directory specification, such as `/etc/fstab` or `../afile.txt`.

Chapter 6, "Getting to Know the Command Line," covers various types of absolute and relative directory references.

TABLE 7.1 Examples of the use of cp

Example command	Effect
`cp orig.txt new.txt`	Copies *orig.txt* to *new.txt* in the current directory.
`cp orig.txt /otherdir`	Copies orig.txt to the */otherdir* directory. The copy will be called *orig.txt*.
`cp orig.txt /otherdir/new.txt`	Copies orig.txt to the */otherdir* directory. The copy will be called *new.txt*.

The critical point to understand is how the destination filename is specified. This can be less than obvious in some cases, since file and directory specifications can look alike. For instance, consider the following command:

```
$ cp outline.pdf ~/publication
```

This command can produce any of three results:

- If `~/publication` is a directory, the result is a file called `~/publication/outline.pdf`.
- If `~/publication` is a file, the result is that this file will be replaced by the contents of `outline.pdf`.
- If `~/publication` doesn't yet exist, the result is a new file, called `~/publication`, which is identical to the original `outline.pdf`.

If you follow a directory name with a slash (/), as in `~/publication/`, `cp` returns an error message if `~/publication` doesn't exist or is a regular file.

Keeping these results straight can be confusing if you're new to command-line file copying. Thus, I encourage you to experiment by creating a test directory using `mkdir` (described later, in "Creating Directories"), creating subdirectories in this directory, and copying files into this test directory tree using all of these methods of referring to files. (This is the type of situation where `touch` can be handy for creating test files.)

The `cp` command provides many options that modify its behavior. Some of the more useful options enable you to modify the command's operation in helpful ways:

Force overwrite The `-f` or `--force` option forces the system to overwrite any existing files without prompting.

Use Interactive mode The `-i` or `--interactive` option causes `cp` to ask you before overwriting any existing files.

Preserve ownership and permissions Normally, a copied file is owned by the user who issues the `cp` command and uses that account's default permissions. The `-p` or `--preserve` option preserves ownership and permissions, if possible.

Perform a recursive copy If you use the `-R` or `--recursive` option and specify a directory as the *source*, the entire directory, including its subdirectories, is copied. Although `-r` also performs a recursive copy, its behavior with files other than ordinary files and directories is unspecified. Most `cp` implementations use `-r` as a synonym for `-R`, but this behavior isn't guaranteed.

Perform an archive copy The `-a` or `--archive` option is similar to `-R`, but it also preserves ownership and copies links as is. The `-R` option copies the files to which symbolic links point rather than the symbolic links themselves. (Links are described in more detail later in this chapter, in "Using Links.")

Perform an update copy The `-u` or `--update` option tells `cp` to copy the file only if the original is newer than the target or if the target doesn't exist.

This list of `cp` options is incomplete but covers the most useful options. Consult `cp`'s `man` page for information about additional `cp` options.

Chapter 13, "Understanding Users and Groups," describes Linux accounts. Chapter 15, "Setting Ownership and Permissions," describes file permissions.

Moving and Renaming Files

Certification Objective

In a text-mode shell, the same command, `mv`, is used to both move and rename files and directories. Its use is very similar to that of `cp`; for instance, if you wanted to move `outline.pdf` to `~/publication`, you would type:

```
$ mv outline.pdf ~/publication
```

If you specify a filename with the destination, the file will be renamed as it's moved. If you specify a filename and the destination directory is the same as the source directory, the file will be renamed but not moved. In other words, `mv`'s effects are much like `cp`'s, except that the new file replaces, rather than supplements, the original.

Behind the scenes, `mv` does the following:

- When the source and target are on the same filesystem, `mv` rewrites directory entries without actually moving the file's data.
- When you move a file from one filesystem to another, `mv` copies the file and then deletes the original file.

The `mv` command takes many of the same options as `cp` does. From the earlier list, `--preserve`, `--recursive`, and `--archive` don't apply to `mv`, but the others do.

Linux uses `mv` for renaming files because the two operations are identical when the source and destination directories are the same.

Using Links

Certification Objective

Sometimes it's handy to refer to a single file by multiple names. Rather than create several copies of the file, you can create multiple *links* to one file. Linux supports two types of links, both of which are created with the `ln` command:

Hard link A *hard link* is a duplicate directory entry. Both entries point to the same file. Because they work by tying together low-level filesystem data structures, hard links can exist only on a single filesystem. In a hard link scenario, neither filename holds any sort of priority over the other; both tie directly to the file's data structures and data. Type **`ln origname linkname`**, where *`origname`* is the original name and *`linkname`* is the new link's name, to create a hard link.

Symbolic link A *symbolic link* (aka a *soft link*) is a file that refers to another file by name. That is, the symbolic link is a file that holds another file's name, and when you tell a program to read to or write from a symbolic link file, Linux redirects the access to the original file. Because symbolic links work by filename references, they can cross filesystem boundaries. Type **`ln -s origname linkname`** to create a symbolic link.

Symbolic links are similar to *shortcuts* on the Windows desktop.

You can identify links in long directory listings (using the `-l` option to `ls`) in a couple of ways. An example will illustrate this:

```
$ ln report.odt hardlink.odt
$ ln -s report.odt softlink.odt
$ ls -l
total 192
-rw-r--r-- 2 rod users 94720 Jan 10 11:53 hardlink.odt
-rw-r--r-- 2 rod users 94720 Jan 10 11:53 report.odt
lrwxrwxrwx 1 rod users    10 Jan 10 11:54 softlink.odt -> report.odt
```

This example began with a single file, `report.odt`. The first two commands created two links, a hard link (`hardlink.odt`) and a symbolic link (`softlink.odt`). Typing `ls -l` shows all three files. The original file and the hard link can be identified as links by the presence of the value `2` in the second column of the `ls -l` output; this column identifies the number of filename entries that point to the file, so a value higher than `1` indicates that a hard link exists. The symbolic link is denoted by an `l` (a lowercase *L*, not a digit *1*) in the first character of the `softlink.odt` file's permissions string (`lrwxrwxrwx`). Furthermore, the symbolic link's filename specification includes an explicit pointer to the linked-to file.

Both types of links are useful for referring to files by multiple names or in multiple directories. For instance, if you write a letter that you send to multiple recipients, you might want to store copies in directories devoted to each recipient. In such a situation, either type of link will probably work fine, but each type

has implications. Most importantly, if you use symbolic links, deleting the original file makes the file completely inaccessible; the symbolic links remain but point to a non-existent file. If you use hard links, by contrast, you must delete *all* the copies of the file to delete the file itself. This is because hard links are duplicate directory entries that point to the same file, whereas symbolic links are separate files that refer to the original file by name.

If you modify a file by accessing its soft link, or by any hard-linked name, you should be sure that the program you use will modify the original file. Some programs create a backup of the original file that you can use to recover the original in case you find that your changes were in error. Most editors do this in such a way that the backup is a new file, and write changes to the original file, thus affecting the original file as well as the link. Some programs, though, rename the original file and then write a new file with the changes. If a program does this and you've accessed the file via a link, the linked-to file will be unaffected by your changes. If in doubt, test your program to be sure it does what you expect.

If you want to create a link to a directory, be aware that you can normally do this only via symbolic links. Hard links between directories are potentially dangerous in terms of low-level filesystem data structures, so the `ln` utility permits only the superuser to create such links. Even then, most filesystems disallow hard links between directories, so in practice even `root` usually can't create them. Any user can create symbolic links to a directory, though.

Linux installations make use of links (mostly symbolic links) in various places. For instance, system startup scripts are often referred to via symbolic links located in directories dedicated to specific startup conditions, known as *runlevels*. Runlevel management is beyond the scope of this book.

The `rm` command's name is (very!) short for *remove*.

Deleting Files

The `rm` command deletes files in a text-mode shell. As you might expect, you pass the names of one or more files to this command:

```
$ rm outline.pdf outline.txt
```

Distributions sometimes set the `-i` option by default for `root`, but not for ordinary users.

This example deletes two files, `outline.pdf` and `outline.txt`. If you want to delete an entire directory tree, you can pass `rm` the `-r`, `-R`, or `--recursive` option along with a directory name:

```
$ rm -r oldstuff/
```

The `-i` option causes `rm` to prompt before deleting each file. This is a useful safety measure. You can use the `-f` (`--force`) option to override this setting, if `-i` is configured as the default. Several other options to `rm` exist; consult its `man` page to learn about them.

It's important to realize that `rm` does *not* implement any functionality like a file manager's "trash can." Once you delete a file with `rm`, it's gone, and you can't recover it except by using low-level filesystem tools—a topic that's well beyond the scope of this book. Thus, you should be careful when using `rm`—and even more careful when using it with its `-r` option or as `root`!

Using Wildcards

Certification Objective

You can use *wildcards* to refer to files. (Using wildcards is also sometimes called *globbing*.) A wildcard is a symbol or set of symbols that stands in for other characters. Three classes of wildcards are common in Linux:

? A question mark (?) stands in for a single character. For instance, `b??k` matches `book`, `balk`, `buck`, or any other four-character filename that begins with `b` and ends with `k`.

***** An asterisk (*) matches any character or set of characters, including no character. For instance, `b*k` matches `book`, `balk`, and `buck` just as does `b??k`. `b*k` also matches `bk`, `bbk`, and `backtrack`.

Bracketed values Characters enclosed in square brackets ([]) normally match any character in the set. For instance, `b[ao][lo]k` matches `balk` and `book` but not `buck`. It's also possible to specify a range of values; for instance, `b[a-z]ck` matches `back`, `buck`, and other four-letter filenames of this form whose second character is a lowercase letter. This differs from `b?ck`—because Linux treats filenames in a case-sensitive way and because `?` matches any character (not just any letter), `b[a-z]ck` doesn't match `bAck` or `b3ck`, although `b?ck` matches both of these filenames.

Wildcards are implemented in the shell and passed to the command you call. For instance, if you type **`ls b??k`**, and that wildcard matches the three files `balk`, `book`, and `buck`, the result is precisely as if you'd typed **`ls balk book buck`**.

The way Bash expands wildcards can lead to unexpected, and sometimes undesirable, consequences. For instance, suppose you want to copy two files, specified via a wildcard, to another directory, but you forget to give the destination directory. The `cp` command will interpret the command as a request to copy the first of the files over the second.

Understanding Case Sensitivity

Certification Objective

Linux's native filesystems are case-sensitive, which means that filenames that differ only in case are distinct files. For instance, a single directory can hold files called `afile.txt`, `Afile,txt`, and `AFILE.TXT`, and each is a distinct file. This

case sensitivity also means that, if you type a filename, you must enter it with the correct case—if a file is called `afile.txt` but you type its name as `Afile.txt`, the program you're using will tell you that the file doesn't exist.

This is different from what happens in Windows or (usually) in Mac OS X, in which filenames that differ only in case are treated identically. This means that, in these OSs, you can't have two files that differ only in case in the same directory, and you can specify a filename using any case variant you like. Windows also creates a short filename (8 characters with an optional 3-character extension) for every file with a longer name, to help out older software that works only with such filenames. Linux doesn't create such alternate filenames.

Apple's Hierarchical File System Plus (HFS+) supports both case-sensitive and case-insensitive variants. Apple uses the case-insensitive mode by default.

Case sensitivity is primarily a function of the filesystem, not of the operating system. Thus, if you access a non-Linux filesystem (on a removable disk, a non-Linux partition on a dual-boot computer, or using a network filesystem), you may find that case-insensitive rules will apply. This is particularly likely when accessing File Allocation Table (FAT) and New Technology File System (NTFS) volumes, which are common on Windows computers, external hard disks, and USB flash drives. A further twist on this rule is that many Linux programs, such as Bash, assume case sensitivity even on case-insensitive filesystems. Features such as command completion, described in Chapter 6, "Getting to Know the Command Line," may work only if you use the case in which filenames are recorded, even on case-insensitive filesystems.

Ordinarily, case sensitivity creates few real problems, particularly if you use GUI programs that enable you to point-and-click to select files. You should be aware of these issues, however, when copying files or directories to FAT, NTFS, HFS+, or other case-insensitive filesystems. If a directory you want to copy contains files with names that differ only in case, you'll end up with a disk that contains just one of the offending files.

Manipulating Directories

No doubt you're familiar with the concept of directories, although you may think of them as "folders," since most GUI file managers represent directories using file folder icons. Naturally, Linux provides text-mode commands to manipulate directories. These include directory-specific commands to create and delete directories, as well as use of some of the file-manipulation commands described earlier to manage directories.

Creating Directories

Certification Objective

You can use the `mkdir` command to create a directory. Ordinarily, you'll use this command by typing the name of one or more directories following the command:

```
$ mkdir newdir
$ mkdir dirone newdir/dirtwo
```

The first example creates just one new directory, `newdir`, which will then reside in the current directory. The second example creates two new directories: `dirone` and `newdir/dirtwo`. In this example, `mkdir` creates `dirtwo` inside the `newdir` directory, which was created with the preceding command.

In most cases, you'll use `mkdir` without options, other than the name of a directory, but you can modify its behavior in a few ways:

Set mode The `-m` *mode* or `--mode=`*mode* option causes the new directory to have the specified permission mode, expressed as an octal number. (Chapter 15, "Setting Ownership and Permissions," describes these topics in more detail.)

Create parent directories Normally, if you specify the creation of a directory within a directory that doesn't exist, `mkdir` responds with a `No such file or directory` error and doesn't create the directory. If you include the `-p` or `--parents` option, though, `mkdir` creates the necessary parent directory. For instance, typing **`mkdir first/second`** returns an error message if `first` doesn't exist, but typing **`mkdir -p first/second`** succeeds, creating both `first` and its subdirectory, `second`.

Chapter 6 includes information on how to specify locations other than the current directory, as well as how to change your current directory with the `cd` command.

Deleting Directories

Certification Objective

The `rmdir` command is the opposite of `mkdir`; it destroys a directory. To use it, you normally type the command followed by the names of one or more directories you want to delete:

```
$ rmdir dirone
$ rmdir newdir/dirtwo newdir
```

These examples delete the three directories created by the `mkdir` commands shown earlier.

Like `mkdir`, `rmdir` supports few options, the most important of which handle these tasks:

Ignore failures on non-empty directories Normally, if a directory contains files or other directories, `rmdir` doesn't delete it and returns an error message.

With the `--ignore-fail-on-non-empty` option, `rmdir` still doesn't delete the directory, but it doesn't return an error message.

Delete tree The `-p` or `--parents` option causes `rmdir` to delete an entire directory tree. For instance, typing **`rmdir -p newdir/dirtwo`** causes `rmdir` to delete `newdir/dirtwo`, then `newdir`. You could use this command rather than the second one shown earlier to delete both of these directories.

You should understand that `rmdir` can delete only *empty* directories; if a directory contains any files at all, it won't work. (You can use the `-p` option, however, to delete a set of nested directories, as long as none of them holds any non-directory file.) Of course, in real life you're likely to want to delete directory trees that hold files. In such cases, you can use the `rm` command, described earlier, in "Deleting Files," along with its `-r` (or `-R` or `--recursive`) option:

```
$ rm -r newdir
```

This command deletes `newdir` and any files or subdirectories it might contain. This fact makes `rm` and its `-r` option potentially dangerous, so you should be particularly cautious when using it.

Linux Security Features

When you log in as an ordinary user, you can accidentally delete your own files if you err in your use of `rm` or various other commands. You cannot, however, do serious damage to the Linux installation itself. This is because Unix was designed as a multi-user OS with multi-user security features in mind, and because Linux is a clone of Unix, Linux has inherited these security features. Among these features are the concepts of file ownership and file permissions. You can only delete your own files—or more precisely, you can only delete files if you have write access to the directories in which they reside. You have such access to your own home directory, but not to the directories in which Linux system files reside. Thus, you can't damage these Linux system files.

Chapter 13, "Understanding Users and Groups," covers these concepts in more detail. Chapter 13 also describes how you can acquire the power to administer the computer. With this power comes the ability to damage the system, though, so you should be careful to do so only when necessary.

Managing Directories

Directories are just special files—they're files that hold other files. Thus, you can use most of the file-manipulation tools described elsewhere in this chapter to manipulate directories. There are some caveats, though:

- You can use `touch` to update a directory's time stamps, but you can't use `touch` to create a directory; `mkdir` handles that task.
- You can use `cp` to copy a directory; however, you must use the `-r`, `-R`, `--recursive`, `-a`, or `--archive` option to copy the directory and all its contents.
- You can use `mv` to move or rename a directory.
- You can use `ln` with its `-s` option to create a symbolic link to a directory. No common Linux filesystem supports hard links to directories, though.

As an example, suppose you have a directory in your home directory called `Music/Satchmo`, which contains Louis Armstrong music files. You want to reorganize this directory so that the files appear under the performer's last name, but you want to retain access to the files under the name `Satchmo`, since your music players refer to them this way. You could type the following commands to achieve this goal:

```
$ cd ~/Music
$ mv Satchmo Armstrong
$ ln -s Armstrong Satchmo
```

Alternatively, you could omit the first command and specify the complete path to each of the directories or links in the `mv` and `ln` commands. As written, the first two of these commands rename the `~/Music/Satchmo` directory to `~/Music/Armstrong`. The final command creates a symbolic link, `~/Music/Satchmo`, that points to `~/Music/Armstrong`.

The Essentials and Beyond

Much of what you do with a computer qualifies as file management. Thus, you must understand the basic tools for managing files in Linux. These include commands to create, delete, copy, move, and rename files, as well as to create links to files. Directories in Linux are just files that contain other files, so most of the same commands you can use on files also work on directories. Special commands to create and delete directories exist, too.

(Continues)

The Essentials and Beyond *(Continued)*

Suggested Exercises

- Create a file with `touch` (or some other program) and then practice copying it with `cp`, renaming it with `mv`, moving it to another directory with `mv`, and deleting it with `rm`.
- Create a directory with `mkdir` and then practice using `cp`, `mv`, and `rm` on it, just as with files. Try copying files into it and then try deleting the directory with both `rmdir` and `rm`. Do both commands work?

Review Questions

1. Which of the following commands would you type to rename `newfile.txt` to `afile.txt`?

   A. `mv newfile.txt afile.txt`

   B. `cp newfile.txt afile.txt`

   C. `ln newfile.txt afile.txt`

   D. `rn newfile.txt afile.txt`

   E. `touch newfile.txt afile.txt`

2. You want to copy a directory, `MyFiles`, to a USB flash drive that uses the FAT filesystem. The contents of `MyFiles` are as follows:

```
$ ls -l MyFiles/
total 276
-rw-r--r-- 1 jen  users 129840 Nov 8 15:13 contract.odt
-rw-r--r-- 1 rod  users  42667 Nov 8 15:12 outline.pdf
-rw-r--r-- 1 sam  users 105979 Nov 8 15:12 Outline.PDF
```

   The USB flash drive is mounted at `/media/usb`, and so you type **`cp -a MyFiles/ /media/usb`**. What problem will occur when you attempt to copy these files?

   A. The command will fail because it tries to create links.

   B. The `MyFiles` directory will be copied, but none of its files will be copied.

   C. One file will be missing on the USB flash drive.

   D. One file's name will be changed during the copy.

   E. Everything will be fine; the command will work correctly.

(Continues)

The Essentials and Beyond *(Continued)*

3. You type **`mkdir one/two/three`** and receive an error message that reads, in part, `No such file or directory`. What can you do to overcome this problem? (Select all that apply.)
   - **A.** Add the `--parents` parameter to the `mkdir` command.
   - **B.** Issue three separate `mkdir` commands: **`mkdir one`**, then **`mkdir one/two`**, and then **`mkdir one/two/three`**.
   - **C.** Type **`touch /bin/mkdir`** to be sure the `mkdir` program file exists.
   - **D.** Type **`rmdir one`** to clear away the interfering base of the desired new directory tree.
   - **E.** Type **`rm -r one`** to clear away the entire interfering directory tree.
4. True or false: You can create a symbolic link from one low-level filesystem to another.
5. True or false: You can easily damage your Linux installation by mistyping an `rm` command when you log into your regular account.
6. True or false: You can set a directory's time stamps with the `touch` command.
7. You want to copy a file (`origfile.txt`) to the `backups` directory, but if a file called `origfile.txt` exists in the `backups` directory, you want to go ahead with the copy only if the file in the source location is newer than the one in `backups`. The command to do this is **`cp ____ origfile.txt backups/`**.
8. You've typed **`rmdir junk`** to delete the `junk` directory, but this command has failed because `junk` contains word processing files. What command might you type to do the job?
9. Which wildcard character matches any one symbol in a filename?