4

MANIPULATING FILES AND DIRECTORIES

At this point, we are ready for some real work! This chapter will introduce the following commands:

- `cp`—Copy files and directories.
- `mv`—Move/rename files and directories.
- `mkdir`—Create directories.
- `rm`—Remove files and directories.
- `ln`—Create hard and symbolic links.

These five commands are among the most frequently used Linux commands. They are used for manipulating both files and directories.

Now, to be frank, some of the tasks performed by these commands are more easily done with a graphical file manager. With a file manager, we can drag and drop a file from one directory to another, cut and paste files, delete files, and so on. So why use these old command-line programs?

The answer is power and flexibility. While it is easy to perform simple file manipulations with a graphical file manager, complicated tasks can be easier with the command-line programs. For example, how could we copy all the HTML files from one directory to another—but only those that do not exist in the destination directory or are newer than the versions in the destination directory? Pretty hard with a file manager. Pretty easy with the command line:

```
cp -u *.html destination
```

Wildcards

Before we begin using our commands, we need to talk about the shell feature that makes these commands so powerful. Because the shell uses filenames so much, it provides special characters to help you rapidly specify groups of filenames. These special characters are called *wildcards*. Using wildcards (also known as *globbing*) allows you to select filenames based on patterns of characters. Table 4-1 lists the wildcards and what they select.

Table 4-1: Wildcards

Wildcard	Matches
`*`	Any characters
`?`	Any single character
`[`*`characters`*`]`	Any character that is a member of the set *`characters`*
`[!`*`characters`*`]`	Any character that is not a member of the set *`characters`*
`[[:`*`class`*`:]]`	Any character that is a member of the specified *`class`*

Table 4-2 lists the most commonly used character classes.

Table 4-2: Commonly Used Character Classes

Character Class	Matches
`[:alnum:]`	Any alphanumeric character
`[:alpha:]`	Any alphabetic character
`[:digit:]`	Any numeral
`[:lower:]`	Any lowercase letter
`[:upper:]`	Any uppercase letter

Using wildcards makes it possible to construct very sophisticated selection criteria for filenames. Table 4-3 lists some examples of patterns and what they match.

Table 4-3: Wildcard Examples

Pattern	Matches
`*`	All files
`g*`	Any file beginning with *g*
`b*.txt`	Any file beginning with *b* followed by any characters and ending with *.txt*
`Data???`	Any file beginning with *Data* followed by exactly three characters
`[abc]*`	Any file beginning with either *a*, *b*, or *c*
`BACKUP.[0-9][0-9][0-9]`	Any file beginning with *BACKUP.* followed by exactly three numerals
`[[:upper:]]*`	Any file beginning with an uppercase letter
`[![:digit:]]*`	Any file not beginning with a numeral
`*[[:lower:]123]`	Any file ending with a lowercase letter or the numerals *1*, *2*, or *3*

Wildcards can be used with any command that accepts filenames as arguments, but we'll talk more about that in Chapter 7.

CHARACTER RANGES

If you are coming from another Unix-like environment or have been reading some other books on this subject, you may have encountered the `[A-Z]` or the `[a-z]` character range notations. These are traditional Unix notations and worked in older versions of Linux as well. They can still work, but you have to be very careful with them because they will not produce the expected results unless properly configured. For now, you should avoid using them and use character classes instead.

> **WILDCARDS WORK IN THE GUI TOO**
>
> Wildcards are especially valuable, not only because they are used so frequently on the command line but also because they are supported by some graphical file managers.
>
> - In **Nautilus** (the file manager for GNOME), you can select files using Edit ▸ Select Pattern. Just enter a file selection pattern with wildcards, and the files in the currently viewed directory will be highlighted for selection.
> - In some versions of **Dolphin** and **Konqueror** (the file managers for KDE), you can enter wildcards directly on the location bar. For example, if you want to see all the files starting with a lowercase *u* in the */usr/bin* directory, enter **`/usr/bin/u*`** in the location bar, and it will display the result.
>
> Many ideas originally found in the command line interface make their way into the graphical interface, too. It is one of the many things that make the Linux desktop so powerful.

mkdir—Create Directories

The `mkdir` command is used to create directories. It works like this:

```
mkdir directory...
```

A note on notation: In this book, when three periods follow an argument in the description of a command (as above), it means that the argument can be repeated; thus, in this case,

```
mkdir dir1
```

would create a single directory named *dir1*, while

```
mkdir dir1 dir2 dir3
```

would create three directories named *dir1*, *dir2*, and *dir3*.

cp—Copy Files and Directories

The `cp` command copies files or directories. It can be used two different ways:

```
cp item1 item2
```

to copy the single file or directory *item1* to file or directory *item2* and:

```
cp item... directory
```

to copy multiple items (either files or directories) into a directory.

Tables 4-4 and 4-5 list some of the commonly used options (the short option and the equivalent long option) for `cp`.

Table 4-4: cp Options

Option	Meaning
`-a, --archive`	Copy the files and directories and all of their attributes, including ownerships and permissions. Normally, copies take on the default attributes of the user performing the copy.
`-i, --interactive`	Before overwriting an existing file, prompt the user for confirmation. **If this option is not specified, `cp` will silently overwrite files.**
`-r, --recursive`	Recursively copy directories and their contents. This option (or the `-a` option) is required when copying directories.
`-u, --update`	When copying files from one directory to another, copy only files that either don't exist or are newer than the existing corresponding files in the destination directory.
`-v, --verbose`	Display informative messages as the copy is performed.

Table 4-5: cp Examples

Command	Results
`cp file1 file2`	Copy *file1* to *file2*. **If *file2* exists, it is overwritten with the contents of *file1*.** If *file2* does not exist, it is created.
`cp -i file1 file2`	Same as above, except that if *file2* exists, the user is prompted before it is overwritten.
`cp file1 file2 dir1`	Copy *file1* and *file2* into directory *dir1*. *dir1* must already exist.
`cp dir1/* dir2`	Using a wildcard, all the files in *dir1* are copied into *dir2*. *dir2* must already exist.
`cp -r dir1 dir2`	Copy directory *dir1* (and its contents) to directory *dir2*. If directory *dir2* does not exist, it is created and will contain the same contents as directory *dir1*.

mv—Move and Rename Files

The `mv` command performs both file moving and file renaming, depending on how it is used. In either case, the original filename no longer exists after the operation. `mv` is used in much the same way as `cp`:

```
mv item1 item2
```

to move or rename file or directory *item1* to *item2* or

```
mv item... directory
```

to move one or more items from one directory to another.

`mv` shares many of the same options as `cp`, as shown in Tables 4-6 and 4-7.

Table 4-6: mv Options

Option	Meaning
`-i, --interactive`	Before overwriting an existing file, prompt the user for confirmation. **If this option is not specified, `mv` will silently overwrite files.**
`-u, --update`	When moving files from one directory to another, move only files that either don't exist in the destination directory or are newer than the existing corresponding files in the destination directory.
`-v, --verbose`	Display informative messages as the move is performed.

Table 4-7: mv Examples

Command	Results
`mv file1 file2`	Move *file1* to *file2*. **If *file2* exists, it is overwritten with the contents of *file1*.** If *file2* does not exist, it is created. **In either case, *file1* ceases to exist.**
`mv -i file1 file2`	Same as above, except that if *file2* exists, the user is prompted before it is overwritten.
`mv file1 file2 dir1`	Move *file1* and *file2* into directory *dir1*. *dir1* must already exist.
`mv dir1 dir2`	Move directory *dir1* (and its contents) into directory *dir2*. If directory *dir2* does not exist, create directory *dir2*, move the contents of directory *dir1* into *dir2*, and delete directory *dir1*.

rm—Remove Files and Directories

The `rm` command is used to remove (delete) files and directories, like this:

```
rm item...
```

where *item* is the name of one or more files or directories.

> **BE CAREFUL WITH RM!**
>
> Unix-like operating systems such as Linux do not have an undelete command. Once you delete something with `rm`, it's gone. Linux assumes you're smart and you know what you're doing.
>
> Be particularly careful with wildcards. Consider this classic example. Let's say you want to delete just the HTML files in a directory. To do this, you type:
>
> ```
> rm *.html
> ```
>
> which is correct, but if you accidentally place a space between the `*` and the `.html` like so:
>
> ```
> rm * .html
> ```
>
> the `rm` command will delete all the files in the directory and then complain that there is no file called *.html*.
>
> **Here is a useful tip:** Whenever you use wildcards with `rm` (besides carefully checking your typing!), test the wildcard first with `ls`. This will let you see the files that will be deleted. Then press the up arrow key to recall the command and replace the `ls` with `rm`.

Tables 4-8 and 4-9 list some of the common options for `rm`.

Table 4-8: rm Options

Option	Meaning
`-i, --interactive`	Before deleting an existing file, prompt the user for confirmation. If this option is not specified, `rm` will silently delete files.
`-r, --recursive`	Recursively delete directories. This means that if a directory being deleted has subdirectories, delete them too. To delete a directory, this option must be specified.
`-f, --force`	Ignore nonexistent files and do not prompt. This overrides the `--interactive` option.
`-v, --verbose`	Display informative messages as the deletion is performed.

Table 4-9: rm Examples

Command	Results
`rm file1`	Delete *file1* silently.
`rm -i file1`	Before deleting *file1*, prompt the user for confirmation.
`rm -r file1 dir1`	Delete *file1* and *dir1* and its contents.
`rm -rf file1 dir1`	Same as above, except that if either *file1* or *dir1* does not exist, `rm` will continue silently.

ln—Create Links

The `ln` command is used to create either hard or symbolic links. It is used in one of two ways:

```
ln file link
```

to create a hard link and

```
ln -s item link
```

to create a symbolic link where *`item`* is either a file or a directory.

Hard Links

Hard links are the original Unix way of creating links; symbolic links are more modern. By default, every file has a single hard link that gives the file its name. When we create a hard link, we create an additional directory entry for a file. Hard links have two important limitations:

- A hard link cannot reference a file outside its own filesystem. This means a link cannot reference a file that is not on the same disk partition as the link itself.
- A hard link cannot reference a directory.

A hard link is indistinguishable from the file itself. Unlike a directory list containing a symbolic link, a directory list containing a hard link shows no special indication of the link. When a hard link is deleted, the link is removed, but the contents of the file itself continue to exist (that is, its space is not deallocated) until all links to the file are deleted.

It is important to be aware of hard links because you might encounter them from time to time, but modern practice prefers symbolic links, which we will cover next.

Symbolic Links

Symbolic links were created to overcome the limitations of hard links. Symbolic links work by creating a special type of file that contains a text pointer to the referenced file or directory. In this regard they operate in much the same way as a Windows shortcut, though of course they predate the Windows feature by many years. ;-)

A file pointed to by a symbolic link and the symbolic link itself are largely indistinguishable from one another. For example, if you write something to the symbolic link, the referenced file is also written to. However, when you delete a symbolic link, only the link is deleted, not the file itself. If the file is deleted before the symbolic link, the link will continue to exist but will point to nothing. In this case, the link is said to be *broken*. In many implementations, the `ls` command will display broken links in a distinguishing color, such as red, to reveal their presence.

The concept of links can seem confusing, but hang in there. We're going to try all this stuff and it will, hopefully, become clear.

Let's Build a Playground

Since we are going to do some real file manipulation, let's build a safe place to "play" with our file manipulation commands. First we need a directory to work in. We'll create one in our home directory and call it *playground*.

Creating Directories

The `mkdir` command is used to create a directory. To create our *playground* directory, we will first make sure we are in our home directory and then create the new directory:

```
[me@linuxbox ~]$ cd
[me@linuxbox ~]$ mkdir playground
```

To make *playground* a little more interesting, let's create a couple of directories inside it called *dir1* and *dir2*. To do this, we will change our current working directory to *playground* and execute another `mkdir`:

```
[me@linuxbox ~]$ cd playground
[me@linuxbox playground]$ mkdir dir1 dir2
```

Notice that the `mkdir` command will accept multiple arguments, allowing us to create both directories with a single command.

Copying Files

Next, let's get some data into our playground. We'll do this by copying a file. Using the `cp` command, we'll copy the *passwd* file from the */etc* directory to the current working directory.

```
[me@linuxbox playground]$ cp /etc/passwd .
```

Notice how we used the shorthand for the current working directory, the single trailing period. So now if we perform an `ls`, we will see our file:

```
[me@linuxbox playground]$ ls -l
total 12
drwxrwxr-x 2 me  me 4096 2012-01-10 16:40 dir1
drwxrwxr-x 2 me  me 4096 2012-01-10 16:40 dir2
-rw-r--r-- 1 me  me 1650 2012-01-10 16:07 passwd
```

Now, just for fun, let's repeat the copy using the `-v` option (verbose) to see what it does:

```
[me@linuxbox playground]$ cp -v /etc/passwd .
`/etc/passwd' -> `./passwd'
```

The `cp` command performed the copy again, but this time it displayed a concise message indicating what operation it was performing. Notice that `cp` overwrote the first copy without any warning. Again, this is a case of `cp` assuming that you know what you're doing. To get a warning, we'll include the `-i` (interactive) option:

```
[me@linuxbox playground]$ cp -i /etc/passwd .
cp: overwrite `./passwd'?
```

Responding to the prompt by entering a `y` will cause the file to be overwritten; any other character (for example, `n`) will cause `cp` to leave the file alone.

Moving and Renaming Files

Now, the name *passwd* doesn't seem very playful and this is a playground, so let's change it to something else:

```
[me@linuxbox playground]$ mv passwd fun
```

Let's pass the fun around a little by moving our renamed file to each of the directories and back again:

```
[me@linuxbox playground]$ mv fun dir1
```

moves it first to directory *dir1*. Then

```
[me@linuxbox playground]$ mv dir1/fun dir2
```

moves it from *dir1* to *dir2*. Then

```
[me@linuxbox playground]$ mv dir2/fun .
```

finally brings it back to the current working directory. Next, let's see the effect of `mv` on directories. First we will move our data file into *dir1* again:

```
[me@linuxbox playground]$ mv fun dir1
```

and then move *dir1* into *dir2* and confirm it with `ls`:

```
[me@linuxbox playground]$ mv dir1 dir2
[me@linuxbox playground]$ ls -l dir2
total 4
drwxrwxr-x 2 me  me   4096 2012-01-11 06:06 dir1
[me@linuxbox playground]$ ls -l dir2/dir1
total 4
-rw-r--r-- 1 me  me   1650 2012-01-10 16:33 fun
```

Note that because *dir2* already existed, `mv` moved *dir1* into *dir2*. If *dir2* had not existed, `mv` would have renamed *dir1* to *dir2*. Lastly, let's put everything back:

```
[me@linuxbox playground]$ mv dir2/dir1 .
[me@linuxbox playground]$ mv dir1/fun .
```

Creating Hard Links

Now we'll try some links. First the hard links: We'll create some links to our data file like so:

```
[me@linuxbox playground]$ ln fun fun-hard
[me@linuxbox playground]$ ln fun dir1/fun-hard
[me@linuxbox playground]$ ln fun dir2/fun-hard
```

So now we have four instances of the file *fun*. Let's take a look at our *playground* directory:

```
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me   me   4096 2012-01-14 16:17 dir1
drwxrwxr-x 2 me   me   4096 2012-01-14 16:17 dir2
-rw-r--r-- 4 me   me   1650 2012-01-10 16:33 fun
-rw-r--r-- 4 me   me   1650 2012-01-10 16:33 fun-hard
```

One thing you notice is that the second field in the listing for *fun* and *fun-hard* both contain a *4*, which is the number of hard links that now exist for the file. You'll remember that a file will always have at least one link because the file's name is created by a link. So, how do we know that *fun* and *fun-hard* are, in fact, the same file? In this case, `ls` is not very helpful. While we can see that *fun* and *fun-hard* are both the same size (field 5), our listing provides no way to be sure they are the same file. To solve this problem, we're going to have to dig a little deeper.

When thinking about hard links, it is helpful to imagine that files are made up of two parts: the data part containing the file's contents and the name part, which holds the file's name. When we create hard links, we are actually creating additional name parts that all refer to the same data part. The system assigns a chain of disk blocks to what is called an *inode*, which is then associated with the name part. Each hard link therefore refers to a specific inode containing the file's contents.

The `ls` command has a way to reveal this information. It is invoked with the `-i` option:

```
[me@linuxbox playground]$ ls -li
total 16
12353539 drwxrwxr-x 2 me   me   4096 2012-01-14 16:17 dir1
12353540 drwxrwxr-x 2 me   me   4096 2012-01-14 16:17 dir2
12353538 -rw-r--r-- 4 me   me   1650 2012-01-10 16:33 fun
12353538 -rw-r--r-- 4 me   me   1650 2012-01-10 16:33 fun-hard
```

In this version of the listing, the first field is the inode number, and as we can see, both *fun* and *fun-hard* share the same inode number, which confirms they are the same file.

Creating Symbolic Links

Symbolic links were created to overcome the two disadvantages of hard links: Hard links cannot span physical devices, and hard links cannot reference directories, only files. Symbolic links are a special type of file that contains a text pointer to the target file or directory.

Creating symbolic links is similar to creating hard links:

```
[me@linuxbox playground]$ ln -s fun fun-sym
[me@linuxbox playground]$ ln -s ../fun dir1/fun-sym
[me@linuxbox playground]$ ln -s ../fun dir2/fun-sym
```

The first example is pretty straightforward: We simply add the `-s` option to create a symbolic link rather than a hard link. But what about the next two? Remember, when we create a symbolic link, we are creating a text description of where the target file is relative to the symbolic link. It's easier to see if we look at the `ls` output:

```
[me@linuxbox playground]$ ls -l dir1
total 4
-rw-r--r-- 4 me   me   1650 2012-01-10 16:33 fun-hard
lrwxrwxrwx 1 me   me      6 2012-01-15 15:17 fun-sym -> ../fun
```

The listing for *fun-sym* in *dir1* shows that it is a symbolic link by the leading `l` in the first field and the fact that it points to *../fun*, which is correct. Relative to the location of *fun-sym*, *fun* is in the directory above it. Notice too, that the length of the symbolic link file is 6, the number of characters in the string `../fun` rather than the length of the file to which it is pointing.

When creating symbolic links, you can use either absolute pathnames, like this:

```
[me@linuxbox playground]$ ln -s /home/me/playground/fun dir1/fun-sym
```

or relative pathnames, as we did in our earlier example. Using relative pathnames is more desirable because it allows a directory containing symbolic links to be renamed and/or moved without breaking the links.

In addition to regular files, symbolic links can also reference directories:

```
[me@linuxbox playground]$ ln -s dir1 dir1-sym
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me   me   4096 2012-01-15 15:17 dir1
lrwxrwxrwx 1 me   me      4 2012-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me   me   4096 2012-01-15 15:17 dir2
-rw-r--r-- 4 me   me   1650 2012-01-10 16:33 fun
-rw-r--r-- 4 me   me   1650 2012-01-10 16:33 fun-hard
lrwxrwxrwx 1 me   me      3 2012-01-15 15:15 fun-sym -> fun
```

Removing Files and Directories

As we covered earlier, the `rm` command is used to delete files and directories. We are going to use it to clean up our playground a little bit. First, let's delete one of our hard links:

```
[me@linuxbox playground]$ rm fun-hard
[me@linuxbox playground]$ ls -l
total 12
drwxrwxr-x 2 me   me   4096 2012-01-15 15:17 dir1
lrwxrwxrwx 1 me   me      4 2012-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me   me   4096 2012-01-15 15:17 dir2
-rw-r--r-- 3 me   me   1650 2012-01-10 16:33 fun
lrwxrwxrwx 1 me   me      3 2012-01-15 15:15 fun-sym -> fun
```

That worked as expected. The file *fun-hard* is gone and the link count shown for *fun* is reduced from four to three, as indicated in the second field of the directory listing. Next, we'll delete the file *fun*, and just for enjoyment, we'll include the `-i` option to show what that does:

```
[me@linuxbox playground]$ rm -i fun
rm: remove regular file `fun'?
```

Enter `y` at the prompt, and the file is deleted. But let's look at the output of `ls` now. Notice what happened to *fun-sym*? Since it's a symbolic link pointing to a now nonexistent file, the link is *broken*:

```
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 me   me   4096 2012-01-15 15:17 dir1
lrwxrwxrwx 1 me   me      4 2012-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me   me   4096 2012-01-15 15:17 dir2
lrwxrwxrwx 1 me   me      3 2012-01-15 15:15 fun-sym -> fun
```

Most Linux distributions configure `ls` to display broken links. On a Fedora box, broken links are displayed in blinking red text! The presence of a broken link is not in and of itself dangerous, but it is rather messy. If we try to use a broken link, we will see this:

```
[me@linuxbox playground]$ less fun-sym
fun-sym: No such file or directory
```

Let's clean up a little. We'll delete the symbolic links:

```
[me@linuxbox playground]$ rm fun-sym dir1-sym
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 me   me   4096 2012-01-15 15:17 dir1
drwxrwxr-x 2 me   me   4096 2012-01-15 15:17 dir2
```

One thing to remember about symbolic links is that most file operations are carried out on the link's target, not the link itself. However, `rm` is an exception. When you delete a link, it is the link that is deleted, not the target.

Finally, we will remove our *playground*. To do this, we will return to our home directory and use `rm` with the recursive option (`-r`) to delete *playground* and all of its contents, including its subdirectories:

```
[me@linuxbox playground]$ cd
[me@linuxbox ~]$ rm -r playground
```

CREATING SYMLINKS WITH THE GUI

The file managers in both GNOME and KDE provide an easy and automatic method of creating symbolic links. With GNOME, holding the CTRL and SHIFT keys while dragging a file will create a link rather than copying (or moving) the file. In KDE, a small menu appears whenever a file is dropped, offering a choice of copying, moving, or linking the file.

Final Note

We've covered a lot of ground here, and the information may take a while to fully sink in. Perform the playground exercise over and over until it makes sense. It is important to get a good understanding of basic file manipulation commands and wildcards. Feel free to expand on the playground exercise by adding more files and directories, using wildcards to specify files for various operations. The concept of links may be a little confusing at first, but take the time to learn how they work. They can be a real lifesaver.