PART 1

LEARNING THE SHELL

1

WHAT IS THE SHELL?

When we speak of the command line, we are really referring to the shell. The *shell* is a program that takes keyboard commands and passes them to the operating system to carry out. Almost all Linux distributions supply a shell program from the GNU Project called `bash`. The name *bash* is an acronym for *Bourne Again Shell*, a reference to the fact that `bash` is an enhanced replacement for `sh`, the original Unix shell program written by Steve Bourne.

Terminal Emulators

When using a graphical user interface, we need another program called a *terminal emulator* to interact with the shell. If we look through our desktop menus, we will probably find one. KDE uses `konsole` and GNOME uses `gnome-terminal`, though it's likely called simply "terminal" on our menu. A

number of other terminal emulators are available for Linux, but they all do basically the same thing: give us access to the shell. You will probably develop a preference for one or another based on the number of bells and whistles it has.

Your First Keystrokes

So let's get started. Launch the terminal emulator! Once it comes up, you should see something like this:

```
[me@linuxbox ~]$
```

This is called a *shell prompt,* and it appears whenever the shell is ready to accept input. While it may vary in appearance somewhat, depending on the distribution, it will usually include your *username@machinename*, followed by the current working directory (more about that in a little bit) and a dollar sign.

If the last character of the prompt is a hash mark (#) rather than a dollar sign, the terminal session has *superuser* privileges. This means that either we are logged in as the root user or we've selected a terminal emulator that provides superuser (administrative) privileges.

Assuming that things are good so far, let's try some typing. Enter some gibberish at the prompt like so:

```
[me@linuxbox ~]$ kaekfjaeifj
```

Since this command makes no sense, the shell tells us so and gives us another chance:

```
bash: kaekfjaeifj: command not found
[me@linuxbox ~]$
```

Command History

If we press the up-arrow key, we see that the previous command `kaekfjaeifj` reappears after the prompt. This is called *command history*. Most Linux distributions remember the last 500 commands by default. Press the down-arrow key, and the previous command disappears.

Cursor Movement

Recall the previous command with the up-arrow key again. Now try the left- and right-arrow keys. See how we can position the cursor anywhere on the command line? This makes editing commands easy.

A FEW WORDS ABOUT MICE AND FOCUS

While the shell is all about the keyboard, you can also use a mouse with your terminal emulator. A mechanism built into the X Window System (the underlying engine that makes the GUI go) supports a quick copy-and-paste technique. If you highlight some text by holding down the left mouse button and dragging the mouse over it (or double-clicking a word), it is copied into a buffer maintained by X. Pressing the middle mouse button will cause the text to be pasted at the cursor location. Try it.

Don't be tempted to use CTRL-C and CTRL-V to perform copy and paste inside a terminal window. They don't work. For the shell, these control codes have different meanings that were assigned many years before Microsoft Windows came on the scene.

Your graphical desktop environment (most likely KDE or GNOME), in an effort to behave like Windows, probably has its *focus policy* set to "click to focus." This means for a window to get focus (become active), you need to click it. This is contrary to the traditional X behavior of "focus follows mouse," which means that a window gets focus when the mouse just passes over it. The window will not come to the foreground until you click it, but it will be able to receive input. Setting the focus policy to "focus follows mouse" will make using terminal windows easier. Give it a try. I think if you give it a chance, you will prefer it. You will find this setting in the configuration program for your window manager.

Try Some Simple Commands

Now that we have learned to type, let's try a few simple commands. The first one is `date`. This command displays the current time and date:

```
[me@linuxbox ~]$ date
Thu Oct 25 13:51:54 EDT 2012
```

A related command is `cal`, which, by default, displays a calendar of the current month:

```
[me@linuxbox ~]$ cal
    October 2012
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

To see the current amount of free space on your disk drives, enter **df**:

```
[me@linuxbox ~]$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda2             15115452   5012392   9949716  34% /
/dev/sda5             59631908  26545424  30008432  47% /home
/dev/sda1               147764     17370    122765  13% /boot
tmpfs                   256856         0    256856   0% /dev/shm
```

Likewise, to display the amount of free memory, enter the **free** command:

```
[me@linuxbox ~]$ free
             total       used       free     shared    buffers     cached
Mem:        513712     503976       9736          0       5312     122916
-/+ buffers/cache:     375748     137964
Swap:      1052248     104712     947536
```

Ending a Terminal Session

We can end a terminal session by either closing the terminal emulator window or entering the `exit` command at the shell prompt:

```
[me@linuxbox ~]$ exit
```

THE CONSOLE BEHIND THE CURTAIN

Even if we have no terminal emulator running, several terminal sessions continue to run behind the graphical desktop. Called *virtual terminals* or *virtual consoles*, these sessions can be accessed on most Linux distributions by pressing CTRL-ALT-F1 through CTRL-ALT-F6 on most systems. When a session is accessed, it presents a login prompt into which we can enter our username and password. To switch from one virtual console to another, press ALT and F1–F6. To return to the graphical desktop, press ALT-F7.

2

NAVIGATION

The first thing we need to learn (besides just typing) is how to navigate the filesystem on our Linux system. In this chapter we will introduce the following commands:

- `pwd`—Print name of current working directory.
- `cd`—Change directory.
- `ls`—List directory contents.

Understanding the Filesystem Tree

Like Windows, a Unix-like operating system such as Linux organizes its files in what is called a *hierarchical directory structure*. This means that they are organized in a tree-like pattern of directories (sometimes called folders in other systems), which may contain files and other directories. The first directory in the filesystem is called the *root directory*. The root directory contains files and subdirectories, which contain more files and subdirectories, and so on.

Note that unlike Windows, which has a separate filesystem tree for each storage device, Unix-like systems such as Linux always have a single filesystem tree, regardless of how many drives or storage devices are attached to the computer. Storage devices are attached (or more correctly, *mounted*) at various points on the tree according to the whims of the *system administrator*, the person (or persons) responsible for the maintenance of the system.

The Current Working Directory

Most of us are probably familiar with a graphical file manager, which represents the filesystem tree, as in Figure 2-1. Notice that the tree is usually shown upended, that is, with the root at the top and the various branches descending below.

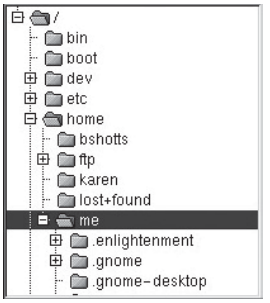


Figure 2-1: Filesystem tree as shown by a graphical file manager

However, the command line has no pictures, so to navigate the filesystem tree, we need to think of it in a different way.

Imagine that the filesystem is a maze shaped like an upside-down tree and we are able to stand in the middle of it. At any given time, we are inside a single directory and we can see the files contained in the directory and the pathway to the directory above us (called the *parent directory*) and any subdirectories below us. The directory we are standing in is called the *current working directory*. To display the current working directory, we use the `pwd` (print working directory) command:

```
[me@linuxbox ~]$ pwd
/home/me
```

When we first log in to our system (or start a terminal emulator session), our current working directory is set to our *home directory*. Each user account is given its own home directory, which is the only place the user is allowed to write files when operating as a regular user.

Listing the Contents of a Directory

To list the files and directories in the current working directory, we use the `ls` command:

```
[me@linuxbox ~]$ ls
Desktop  Documents  Music  Pictures  Public  Templates  Videos
```

Actually, we can use the `ls` command to list the contents of any directory, not just the current working directory, and it can do many other fun things as well. We'll spend more time with `ls` in Chapter 3.

Changing the Current Working Directory

To change your working directory (where we are standing in our tree-shaped maze) we use the `cd` command: Type **`cd`** followed by the pathname of the desired working directory. A *pathname* is the route we take along the branches of the tree to get to the directory we want. Pathnames can be specified in one of two ways, as absolute pathnames or as relative pathnames. Let's deal with absolute pathnames first.

Absolute Pathnames

An *absolute pathname* begins with the root directory and follows the tree branch by branch until the path to the desired directory or file is completed. For example, there is a directory on your system in which most of your system's programs are installed. The pathname of that directory is */usr/bin*. This means from the root directory (represented by the leading slash in the pathname) there is a directory called *usr* that contains a directory called *bin*.

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
[me@linuxbox bin]$ ls

...Listing of many, many files ...
```

Now we can see that we have changed the current working directory to */usr/bin* and that it is full of files. Notice how the shell prompt has changed? As a convenience, it is usually set up to automatically display the name of the working directory.

Relative Pathnames

Where an absolute pathname starts from the root directory and leads to its destination, a *relative pathname* starts from the working directory. To do this, it uses a couple of special symbols to represent relative positions in the file-system tree. These special symbols are `.` (dot) and `..` (dot dot).

The `.` symbol refers to the working directory and the `..` symbol refers to the working directory's parent directory. Here is how it works. Let's change the working directory to */usr/bin* again:

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

Okay, now let's say that we wanted to change the working directory to the parent of */usr/bin*, which is */usr*. We could do that two different ways, either with an absolute pathname:

```
[me@linuxbox bin]$ cd /usr
[me@linuxbox usr]$ pwd
/usr
```

or with a relative pathname:

```
[me@linuxbox bin]$ cd ..
[me@linuxbox usr]$ pwd
/usr
```

Two different methods produce identical results. Which one should we use? The one that requires the least typing!

Likewise, we can change the working directory from */usr* to */usr/bin* in two different ways, either by using an absolute pathname:

```
[me@linuxbox usr]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

or with a relative pathname:

```
[me@linuxbox usr]$ cd ./bin
[me@linuxbox bin]$ pwd
/usr/bin
```

Now, there is something important that I must point out here. In almost all cases, you can omit the `./` because it is implied. Typing

```
[me@linuxbox usr]$ cd bin
```

does the same thing. In general, if you do not specify a pathname to something, the working directory will be assumed.

Some Helpful Shortcuts

In Table 2-1 we see some useful ways the current working directory can be quickly changed.

Table 2-1: cd Shortcuts

Shortcut	Result
`cd`	Changes the working directory to your home directory.
`cd -`	Changes the working directory to the previous working directory.
`cd ~`*`username`*	Changes the working directory to the home directory of *`username`*. For example, `cd ~bob` changes the directory to the home directory of user *bob*.

IMPORTANT FACTS ABOUT FILENAMES

- Filenames that begin with a period character are hidden. This only means that `ls` will not list them unless you say `ls -a`. When your account was created, several hidden files were placed in your home directory to configure things for your account. Later on we will take a closer look at some of those files to see how you can customize your environment. In addition, some applications place their configuration and settings files in your home directory as hidden files.
- Filenames and commands in Linux, as in Unix, are case sensitive. The filenames *File1* and *file1* refer to different files.
- Linux has no concept of a "file extension" like some other operating systems. You may name files any way you like. The contents and/or purpose of a file is determined by other means. Although Unix-like operating systems don't use file extensions to determine the contents/purpose of files, some application programs do.
- Though Linux supports long filenames that may contain embedded spaces and punctuation characters, limit the punctuation characters in the names of files you create to period, dash (hyphen), and underscore. *Most importantly, do not embed spaces in filenames.* Embedding spaces in filenames will make many command line tasks more difficult, as we will discover in Chapter 7. If you want to represent spaces between words in a filename, use underscore characters. You will thank yourself later.

3

EXPLORING THE SYSTEM

Now that we know how to move around the filesystem, it's time for a guided tour of our Linux system. Before we start, however, we're going to learn some more commands that will be useful along the way:

- `ls`—List directory contents.
- `file`—Determine file type.
- `less`—View file contents.

More Fun with ls

`ls` is probably the most used command and for good reason. With it, we can see directory contents and determine a variety of important file and directory attributes. As we have seen, we can simply enter `ls` to see a list of files and subdirectories contained in the current working directory:

```
[me@linuxbox ~]$ ls
Desktop  Documents  Music  Pictures  Public  Templates  Videos
```

Besides the current working directory, we can specify the directory to list, like so:

```
me@linuxbox ~]$ ls /usr
bin  games    kerberos  libexec  sbin   src
etc  include  lib       local    share  tmp
```

or even specify multiple directories. In this example we will list both the user's home directory (symbolized by the ~ character) and the */usr* directory:

```
[me@linuxbox ~]$ ls ~ /usr
/home/me:
Desktop  Documents  Music  Pictures  Public  Templates  Videos
/usr:
bin  games    kerberos  libexec  sbin   src
etc  include  lib       local    share  tmp
```

We can also change the format of the output to reveal more detail:

```
[me@linuxbox ~]$ ls -l
total 56
drwxrwxr-x 2 me me 4096 2012-10-26 17:20 Desktop
drwxrwxr-x 2 me me 4096 2012-10-26 17:20 Documents
drwxrwxr-x 2 me me 4096 2012-10-26 17:20 Music
drwxrwxr-x 2 me me 4096 2012-10-26 17:20 Pictures
drwxrwxr-x 2 me me 4096 2012-10-26 17:20 Public
drwxrwxr-x 2 me me 4096 2012-10-26 17:20 Templates
drwxrwxr-x 2 me me 4096 2012-10-26 17:20 Videos
```

By adding `-l` to the command, we changed the output to the long format.

Options and Arguments

This brings us to a very important point about how most commands work. Commands are often followed by one or more *options* that modify their behavior and, further, by one or more *arguments*, the items upon which the command acts. So most commands look something like this:

```
command -options arguments
```

Most commands use options consisting of a single character preceded by a dash, such as `-l`. But many commands, including those from the GNU Project, also support *long options*, consisting of a word preceded by two dashes. Also, many commands allow multiple short options to be strung together. In this example, the `ls` command is given two options, the `l` option to produce long format output, and the `t` option to sort the result by the file's modification time:

```
[me@linuxbox ~]$ ls -lt
```

We'll add the long option `--reverse` to reverse the order of the sort:

```
[me@linuxbox ~]$ ls -lt --reverse
```

The `ls` command has a large number of possible options. The most common are listed in Table 3-1.

Table 3-1: Common ls Options

Option	Long Option	Description
`-a`	`--all`	List all files, even those with names that begin with a period, which are normally not listed (i.e., hidden).
`-d`	`--directory`	Ordinarily, if a directory is specified, `ls` will list the contents of the directory, not the directory itself. Use this option in conjunction with the `-l` option to see details about the directory rather than its contents.
`-F`	`--classify`	This option will append an indicator character to the end of each listed name (for example, a forward slash if the name is a directory).
`-h`	`--human-readable`	In long format listings, display file sizes in human-readable format rather than in bytes.
`-l`		Display results in long format.
`-r`	`--reverse`	Display the results in reverse order. Normally, `ls` displays its results in ascending alphabetical order.
`-S`		Sort results by file size.
`-t`		Sort by modification time.

A Longer Look at Long Format

As we saw before, the `-l` option causes `ls` to display its results in long format. This format contains a great deal of useful information. Here is the `Examples` directory from an Ubuntu system:

```
-rw-r--r-- 1 root root 3576296 2012-04-03 11:05 Experience ubuntu.ogg
-rw-r--r-- 1 root root 1186219 2012-04-03 11:05 kubuntu-leaflet.png
-rw-r--r-- 1 root root   47584 2012-04-03 11:05 logo-Edubuntu.png
-rw-r--r-- 1 root root   44355 2012-04-03 11:05 logo-Kubuntu.png
-rw-r--r-- 1 root root   34391 2012-04-03 11:05 logo-Ubuntu.png
-rw-r--r-- 1 root root   32059 2012-04-03 11:05 oo-cd-cover.odf
-rw-r--r-- 1 root root  159744 2012-04-03 11:05 oo-derivatives.doc
-rw-r--r-- 1 root root   27837 2012-04-03 11:05 oo-maxwell.odt
-rw-r--r-- 1 root root   98816 2012-04-03 11:05 oo-trig.xls
```

```
-rw-r--r-- 1 root root  453764 2012-04-03 11:05 oo-welcome.odt
-rw-r--r-- 1 root root  358374 2012-04-03 11:05 ubuntu Sax.ogg
```

Let's look at the different fields from one of the files and examine their meanings in Table 3-2.

Table 3-2: ls Long Listing Fields

Field	Meaning
`-rw-r--r--`	Access rights to the file. The first character indicates the type of file. Among the different types, a leading dash means a regular file, while a `d` indicates a directory. The next three characters are the access rights for the file's owner, the next three are for members of the file's group, and the final three are for everyone else. The full meaning of this is discussed in Chapter 9.
`1`	File's number of hard links. See the discussion of links at the end of this chapter.
`root`	The user name of the file's owner.
`root`	The name of the group that owns the file.
`32059`	Size of the file in bytes.
`2012-04-03 11:05`	Date and time of the file's last modification.
`oo-cd-cover.odf`	Name of the file.

Determining a File's Type with file

As we explore the system, it will be useful to know what files contain. To do this, we will use the `file` command to determine a file's type. As we discussed earlier, filenames in Linux are not required to reflect a file's contents. For example, while a filename like *picture.jpg* would normally be expected to contain a JPEG compressed image, it is not required to in Linux. We can invoke the `file` command this way:

```
file filename
```

When invoked, the `file` command will print a brief description of the file's contents. For example:

```
[me@linuxbox ~]$ file picture.jpg
picture.jpg: JPEG image data, JFIF standard 1.01
```

There are many kinds of files. In fact, one of the common ideas in Unix-like operating systems such as Linux is that "everything is a file." As we proceed with our lessons, we will see just how true that statement is.

While many of the files on your system are familiar, for example MP3 and JPEG files, many kinds are a little less obvious, and a few are quite strange.

Viewing File Contents with less

The `less` command is a program to view text files. Throughout our Linux system, there are many files that contain human-readable text. The `less` program provides a convenient way to examine them.

Why would we want to examine text files? Because many of the files that contain system settings (called *configuration files*) are stored in this format, being able to read them gives us insight about how the system works. In addition, many of the actual programs that the system uses (called *scripts*) are stored in this format. In later chapters, we will learn how to edit text files in order to modify system settings and write our own scripts, but for now we will just look at their contents.

WHAT IS "TEXT"?

There are many ways to represent information on a computer. All methods involve defining a relationship between the information and some numbers that will be used to represent it. Computers, after all, understand only numbers, and all data is converted to numeric representation.

Some of these representation systems are very complex (such as compressed video files), while others are rather simple. One of the earliest and simplest is called *ASCII text*. ASCII (pronounced "As-Key") is short for American Standard Code for Information Interchange. This simple encoding scheme was first used on Teletype machines.

Text is a simple one-to-one mapping of characters to numbers. It is very compact. Fifty characters of text translate to fifty bytes of data. It is not the same as text in a word processor document such as one created by Microsoft Word or OpenOffice.org Writer. Those files, in contrast to simple ASCII text, contain many non-text elements that are used to describe their structure and formatting. Plain ASCII text files contain only the characters themselves and a few rudimentary control codes like tabs, carriage returns, and linefeeds.

Throughout a Linux system, many files are stored in text format, and many Linux tools work with text files. Even Windows recognizes the importance of this format. The well-known Notepad program is an editor for plain ASCII text files.

The less command is used like this:

```
less filename
```

Once started, the less program allows you to scroll forward and backward through a text file. For example, to examine the file that defines all the system's user accounts, enter the following command:

```
[me@linuxbox ~]$ less /etc/passwd
```

Once the less program starts, we can view the contents of the file. If the file is longer than one page, we can scroll up and down. To exit less, press the Q key.

Table 3-3 lists the most common keyboard commands used by less.

Table 3-3: less Commands

Command	Action
PAGE UP or b	Scroll back one page.
PAGE DOWN or Spacebar	Scroll forward one page.
Up Arrow	Scroll up one line.
Down Arrow	Scroll down one line.
G	Move to the end of the text file.
1G or g	Move to the beginning of the text file.
/*characters*	Search forward to the next occurrence of *characters*.
n	Search for the next occurrence of the previous search.
h	Display help screen.
q	Quit less.

LESS IS MORE

The less program was designed as an improved replacement of an earlier Unix program called more. Its name is a play on the phrase "less is more"—a motto of modernist architects and designers.

less falls into the class of programs called *pagers*, programs that allow the easy viewing of long text documents in a page-by-page manner. Whereas the more program could only page forward, the less program allows paging both forward and backward and has many other features as well.

A Guided Tour

The filesystem layout on your Linux system is much like that found on other Unix-like systems. The design is actually specified in a published standard called the *Linux Filesystem Hierarchy Standard*. Not all Linux distributions conform to the standard exactly, but most come pretty close.

Next, we are going to wander around the filesystem ourselves to see what makes our Linux system tick. This will give you a chance to practice your navigation skills. One of the things we will discover is that many of the interesting files are in plain, human-readable text. As we go about our tour, try the following:

1. `cd` into a given directory.
2. List the directory contents with `ls -l`.
3. If you see an interesting file, determine its contents with `file`.
4. If it looks as if it might be text, try viewing it with `less`.

Note: *Remember the copy-and-paste trick! If you are using a mouse, you can double-click a filename to copy it and middle-click to paste it into commands.*

As we wander around, don't be afraid to look at stuff. Regular users are largely prohibited from messing things up. That's the system administrator's job! If a command complains about something, just move on to something else. Spend some time looking around. The system is ours to explore. Remember, in Linux, there are no secrets!

Table 3-4 lists just a few of the directories we can explore. Feel free to try more!

Table 3-4: Directories Found on Linux Systems

Directory	Comments
/	The root directory, where everything begins.
/bin	Contains binaries (programs) that must be present for the system to boot and run.
/boot	Contains the Linux kernel, initial RAM disk image (for drivers needed at boot time), and the boot loader. Interesting files: • */boot/grub/grub.conf* or *menu.lst*, which are used to configure the boot loader • */boot/vmlinuz*, the Linux kernel

(*continued*)

Table 3-4 (*continued*)

Directory	Comments
/dev	This is a special directory that contains *device nodes*. "Everything is a file" also applies to devices. Here is where the kernel maintains a list of all the devices it understands.
/etc	The */etc* directory contains all of the system-wide configuration files. It also contains a collection of shell scripts that start each of the system services at boot time. Everything in this directory should be readable text. Interesting files: While everything in */etc* is interesting, here are some of my all-time favorites: • */etc/crontab*, a file that defines when automated jobs will run • */etc/fstab*, a table of storage devices and their associated mount points • */etc/passwd*, a list of the user accounts
/home	In normal configurations, each user is given a directory in */home*. Ordinary users can write files only in their home directories. This limitation protects the system from errant user activity.
/lib	Contains shared library files used by the core system programs. These are similar to DLLs in Windows.
/lost+found	Each formatted partition or device using a Linux file-system, such as ext3, will have this directory. It is used in the case of a partial recovery from a filesystem corruption event. Unless something really bad has happened to your system, this directory will remain empty.
/media	On modern Linux systems the */media* directory will contain the mount points for removable media such as USB drives, CD-ROMs, etc. that are mounted automatically at insertion.
/mnt	On older Linux systems, the */mnt* directory contains mount points for removable devices that have been mounted manually.
/opt	The */opt* directory is used to install "optional" software. This is mainly used to hold commercial software products that may be installed on your system.

Table 3-4 (*continued*)

Directory	Comments
/proc	The */proc* directory is special. It's not a real filesystem in the sense of files stored on your hard drive. Rather, it is a virtual filesystem maintained by the Linux kernel. The "files" it contains are peepholes into the kernel itself. The files are readable and will give you a picture of how the kernel sees your computer.
/root	This is the home directory for the root account.
/sbin	This directory contains "system" binaries. These are programs that perform vital system tasks that are generally reserved for the superuser.
/tmp	The */tmp* directory is intended for storage of temporary, transient files created by various programs. Some configurations cause this directory to be emptied each time the system is rebooted.
/usr	The */usr* directory tree is likely the largest one on a Linux system. It contains all the programs and support files used by regular users.
/usr/bin	*/usr/bin* contains the executable programs installed by your Linux distribution. It is not uncommon for this directory to hold thousands of programs.
/usr/lib	The shared libraries for the programs in */usr/bin*.
/usr/local	The */usr/local* tree is where programs that are not included with your distribution but are intended for system-wide use are installed. Programs compiled from source code are normally installed in */usr/local/bin*. On a newly installed Linux system, this tree exists, but it will be empty until the system administrator puts something in it.
/usr/sbin	Contains more system administration programs.
/usr/share	*/usr/share* contains all the shared data used by programs in */usr/bin*. This includes things like default configuration files, icons, screen backgrounds, sound files, etc.
/usr/share/doc	Most packages installed on the system will include some kind of documentation. In */usr/share/doc*, we will find documentation files organized by package.

(*continued*)

Table 3-4 (*continued*)

Directory	Comments
/var	With the exception of */tmp* and */home*, the directories we have looked at so far remain relatively static; that is, their contents don't change. The */var* directory tree is where data that is likely to change is stored. Various databases, spool files, user mail, etc. are located here.
/var/log	*/var/log* contains *log files*, records of various system activity. These are very important and should be monitored from time to time. The most useful one is */var/log/messages*. Note that for security reasons on some systems, you must be the superuser to view log files.

Symbolic Links

As we look around, we are likely to see a directory listing with an entry like this:

```
lrwxrwxrwx 1 root root   11 2012-08-11 07:34 libc.so.6 -> libc-2.6.so
```

Notice how the first letter of the listing is `l` and the entry seems to have two filenames? This is a special kind of a file called a *symbolic link* (also known as a *soft link* or *symlink*). In most Unix-like systems it is possible to have a file referenced by multiple names. While the value of this may not be obvious now, it is really a useful feature.

Picture this scenario: A program requires the use of a shared resource of some kind contained in a file named *foo*, but *foo* has frequent version changes. It would be good to include the version number in the filename so the administrator or other interested party could see what version of *foo* is installed. This presents a problem. If we change the name of the shared resource, we have to track down every program that might use it and change it to look for a new resource name every time a new version of the resource is installed. That doesn't sound like fun at all.

Here is where symbolic links save the day. Let's say we install version 2.6 of *foo*, which has the filename *foo-2.6*, and then create a symbolic link simply called *foo* that points to *foo-2.6*. This means that when a program opens the file *foo*, it is actually opening the file *foo-2.6*. Now everybody is happy. The programs that rely on *foo* can find it, and we can still see what actual version is installed. When it is time to upgrade to *foo-2.7*, we just add the file to our system, delete the symbolic link *foo*, and create a new one that points to the new version. Not only does this solve the problem of the version upgrade, but it also allows us to keep both versions on our machine. Imagine that *foo-2.7* has a bug (damn those developers!) and we need to revert to the old

version. Again, we just delete the symbolic link pointing to the new version and create a new symbolic link pointing to the old version.

The directory listing above (from the */lib* directory of a Fedora system) shows a symbolic link called *libc.so.6* that points to a shared library file called *libc-2.6.so*. This means that programs looking for *libc.so.6* will actually get the file *libc-2.6.so*. We will learn how to create symbolic links in the next chapter.

HARD LINKS

While we are on the subject of links, we need to mention that there is a second type of link called a *hard link*. Hard links also allow files to have multiple names, but they do it in a different way. We'll talk more about the differences between symbolic and hard links in the next chapter.

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.

5

WORKING WITH COMMANDS

Up to this point, we have seen a series of mysterious commands, each with its own mysterious options and arguments. In this chapter, we will attempt to remove some of that mystery and even create some of our own commands. The commands introduced in this chapter are these:

- `type`—Indicate how a command name is interpreted.
- `which`—Display which executable program will be executed.
- `man`—Display a command's manual page.
- `apropos`—Display a list of appropriate commands.
- `info`—Display a command's info entry.
- `whatis`—Display a very brief description of a command.
- `alias`—Create an alias for a command.

What Exactly Are Commands?

A command can be one of four things:

- **An executable program** like all those files we saw in */usr/bin*. Within this category, programs can be *compiled binaries,* such as programs written in C and C++, or programs written in *scripting languages,* such as the shell, Perl, Python, Ruby, and so on.
- **A command built into the shell itself.** `bash` supports a number of commands internally called *shell builtins*. The `cd` command, for example, is a shell builtin.
- **A shell function.** *Shell functions* are miniature shell scripts incorporated into the *environment*. We will cover configuring the environment and writing shell functions in later chapters, but for now just be aware that they exist.
- **An alias.** An *alias* is a command that we can define ourselves, built from other commands.

Identifying Commands

It is often useful to know exactly which of the four kinds of commands is being used, and Linux provides a couple of ways to find out.

type—Display a Command's Type

The `type` command is a shell builtin that displays the kind of command the shell will execute, given a particular command name. It works like this:

```
type command
```

where *command* is the name of the command you want to examine. Here are some examples:

```
[me@linuxbox ~]$ type type
type is a shell builtin
[me@linuxbox ~]$ type ls
ls is aliased to `ls --color=tty'
[me@linuxbox ~]$ type cp
cp is /bin/cp
```

Here we see the results for three different commands. Notice that the `ls` command (taken from a Fedora system) is actually an alias for the `ls` command with the `--color=tty` option added. Now we know why the output from `ls` is displayed in color!

which—Display an Executable's Location

Sometimes more than one version of an executable program is installed on a system. While this is not very common on desktop systems, it's not unusual on large servers. To determine the exact location of a given executable, the `which` command is used:

```
[me@linuxbox ~]$ which ls
/bin/ls
```

`which` works only for executable programs, not builtins or aliases that are substitutes for actual executable programs. When we try to use `which` on a shell builtin (for example, `cd`), we get either no response or an error message:

```
[me@linuxbox ~]$ which cd
/usr/bin/which: no cd in (/opt/jre1.6.0_03/bin:/usr/lib/qt-3.3/bin:/usr/kerber
os/bin:/opt/jre1.6.0_03/bin:/usr/lib/ccache:/usr/local/bin:/usr/bin:/bin:/home
/me/bin)
```

This is a fancy way of saying "command not found."

Getting a Command's Documentation

With this knowledge of what a command is, we can now search for the documentation available for each kind of command.

help—Get Help for Shell Builtins

`bash` has a built-in help facility for each of the shell builtins. To use it, type **`help`** followed by the name of the shell builtin. For example:

```
[me@linuxbox ~]$ help cd
cd: cd [-L|-P] [dir]
Change the current directory to DIR. The variable $HOME is the default DIR.
The variable CDPATH defines the search path for the directory containing DIR.
Alternative directory names in CDPATH are separated by a colon (:). A null
directory name is the same as the current directory, i.e. `.'. If DIR begins
with a slash (/), then CDPATH is not used. If the directory is not found, and
the shell option `cdable_vars' is set, then try the word as a variable name.
If that variable has a value, then cd to the value of that variable. The -P
option says to use the physical directory structure instead of following
symbolic links; the -L option forces symbolic links to be followed.
```

A note on notation: When square brackets appear in the description of a command's syntax, they indicate optional items. A vertical bar character indicates mutually exclusive items. An example is the `cd` command above: `cd [-L|-P] [dir]`.

This notation says that the command `cd` may be followed optionally by either a `-L` or a `-P` and further, optionally followed by the argument `dir`.

While the output of `help` for the `cd` command is concise and accurate, it is by no means a tutorial, and as we can see, it also seems to mention a lot of things we haven't talked about yet! Don't worry. We'll get there.

--help—Display Usage Information

Many executable programs support a `--help` option that displays a description of the command's supported syntax and options. For example:

```
[me@linuxbox ~]$ mkdir --help
Usage: mkdir [OPTION] DIRECTORY...
Create the DIRECTORY(ies), if they do not already exist.

  -Z, --context=CONTEXT (SELinux) set security context to CONTEXT
Mandatory arguments to long options are mandatory for short options too.
  -m, --mode=MODE   set file mode (as in chmod), not a=rwx - umask
  -p, --parents     no error if existing, make parent directories as
                    needed
  -v, --verbose     print a message for each created directory
      --help        display this help and exit
      --version     output version information and exit
Report bugs to <bug-coreutils@gnu.org>.
```

Some programs don't support the `--help` option, but try it anyway. Often it results in an error message that will reveal the same usage information.

man—Display a Program's Manual Page

Most executable programs intended for command-line use provide a formal piece of documentation called a *manual* or *man page*. A special paging program called `man` is used to view them, like this:

```
man program
```

where *program* is the name of the command to view.

Man pages vary somewhat in format but generally contain a title, a synopsis of the command's syntax, a description of the command's purpose, and a listing and description of each of the command's options. Man pages, however, do not usually include examples, and they are intended as a reference, not a tutorial. As an example, let's try viewing the man page for the `ls` command:

```
[me@linuxbox ~]$ man ls
```

On most Linux systems, `man` uses `less` to display the manual page, so all of the familiar `less` commands work while displaying the page.

The "manual" that `man` displays is broken into sections and covers not only user commands but also system administration commands, programming interfaces, file formats, and more. Table 5-1 describes the layout of the manual.

Table 5-1: Man Page Organization

Section	Contents
1	User commands
2	Programming interfaces for kernel system calls
3	Programming interfaces to the C library
4	Special files such as device nodes and drivers
5	File formats
6	Games and amusements such as screensavers
7	Miscellaneous
8	System administration commands

Sometimes we need to look in a specific section of the manual to find what we are looking for. This is particularly true if we are looking for a file format that is also the name of a command. If we don't specify a section number, we will always get the first instance of a match, probably in section 1. To specify a section number, we use `man` like this:

```
man section search_term
```

For example:

```
[me@linuxbox ~]$ man 5 passwd
```

will display the man page describing the file format of the */etc/passwd* file.

apropos—Display Appropriate Commands

It is also possible to search the list of man pages for possible matches based on a search term. Though crude, this approach is sometimes helpful. Here is an example of a search for man pages using the search term *floppy*:

```
[me@linuxbox ~]$ apropos floppy
create_floppy_devices (8)  - udev callout to create all possible
                             floppy device based on the CMOS type
fdformat             (8)  - Low-level formats a floppy disk
floppy               (8)  - format floppy disks
gfloppy              (1)  - a simple floppy formatter for the GNOME
mbadblocks           (1)  - tests a floppy disk, and marks the bad
                            blocks in the FAT
mformat              (1)  - add an MSDOS filesystem to a low-level
                            formatted floppy disk
```

The first field in each line of output is the name of the man page, and the second field shows the section. Note that the `man` command with the `-k` option performs exactly the same function as `apropos`.

whatis—Display a Very Brief Description of a Command

The whatis program displays the name and a one-line description of a man page matching a specified keyword:

```
[me@linuxbox ~]$ whatis ls
ls                   (1)  - list directory contents
```

THE MOST BRUTAL MAN PAGE OF THEM ALL

As we have seen, the manual pages supplied with Linux and other Unix-like systems are intended as reference documentation and not as tutorials. Many man pages are hard to read, but I think that the grand prize for difficulty has to go to the man page for bash. As I was doing my research for this book, I gave it a careful review to ensure that I was covering most of its topics. When printed, it's over 80 pages long and extremely dense, and its structure makes absolutely no sense to a new user.

On the other hand, it is very accurate and concise, as well as being extremely complete. So check it out if you dare, and look forward to the day when you can read it and it all makes sense.

info—Display a Program's Info Entry

The GNU Project provides an alternative to man pages called *info pages*. Info pages are displayed with a reader program named, appropriately enough, info. Info pages are *hyperlinked* much like web pages. Here is a sample:

```
File: coreutils.info,  Node: ls invocation,  Next: dir invocation, Up:
Directory listing

10.1 `ls': List directory contents
==================================

The `ls' program lists information about files (of any type, including
directories). Options and file arguments can be intermixed arbitrarily, as
usual.

   For non-option command-line arguments that are directories, by default `ls'
lists the contents of directories, not recursively, and omitting files with
names beginning with `.'. For other non-option arguments, by default `ls'
lists just the filename. If no non-option argument is specified, `ls' operates
on the current directory, acting as if it had been invoked with a single
argument of `.'.

   By default, the output is sorted alphabetically, according to the
--zz-Info: (coreutils.info.gz)ls invocation, 63 lines --Top----------
```

The info program reads *info files*, which are tree-structured into individual *nodes*, each containing a single topic. Info files contain hyperlinks that can move you from node to node. A hyperlink can be identified by its leading asterisk and is activated by placing the cursor upon it and pressing the ENTER key.

To invoke info, enter **info** followed optionally by the name of a program. Table 5-2 lists commands used to control the reader while displaying an info page.

Table 5-2: info Commands

Command	Action
?	Display command help.
PAGE UP or BACKSPACE	Display previous page.
PAGE DOWN or Spacebar	Display next page.
n	Next—Display the next node.
p	Previous—Display the previous node.
u	Up—Display the parent node of the currently displayed node, usually a menu.
ENTER	Follow the hyperlink at the cursor location.
q	Quit.

Most of the command-line programs we have discussed so far are part of the GNU Project's coreutils package, so you can find more information about them by typing

```
[me@linuxbox ~]$ info coreutils
```

which will display a menu page containing hyperlinks to documentation for each program provided by the coreutils package.

README and Other Program Documentation Files

Many software packages installed on your system have documentation files residing in the */usr/share/doc* directory. Most of these are stored in plaintext format and can be viewed with less. Some of the files are in HTML format and can be viewed with a web browser. We may encounter some files ending with a *.gz* extension. This indicates that they have been compressed with the gzip compression program. The gzip package includes a special version of less called zless, which will display the contents of gzip-compressed text files.

Creating Your Own Commands with alias

Now for our very first experience with programming! We will create a command of our own using the `alias` command. But before we start, we need to reveal a small command-line trick. It's possible to put more than one command on a line by separating each command with a semicolon character. It works like this:

```
command1; command2; command3...
```

Here's the example we will use:

```
[me@linuxbox ~]$ cd /usr; ls; cd -
bin  games    kerberos  lib64    local  share  tmp
etc  include  lib       libexec  sbin   src
/home/me
[me@linuxbox ~]$
```

As we can see, we have combined three commands on one line. First we change directory to */usr*, then we list the directory, and finally we return to the original directory (by using `cd -`) so we end up where we started. Now let's turn this sequence into a new command using `alias`. The first thing we have to do is dream up a name for our new command. Let's try `test`. Before we do that, it would be a good idea to find out if the name `test` is already being used. To find out, we can use the `type` command again:

```
[me@linuxbox ~]$ type test
test is a shell builtin
```

Oops! The name `test` is already taken. Let's try `foo`:

```
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

Great! `foo` is not taken. So let's create our alias:

```
[me@linuxbox ~]$ alias foo='cd /usr; ls; cd -'
```

Notice the structure of this command:

```
alias name='string'
```

After the command `alias` we give the alias a name followed immediately (no whitespace allowed) by an equal sign, which is followed immediately by a quoted string containing the meaning to be assigned to the name. After we define our alias, it can be used anywhere the shell would expect a command.

Let's try it:

```
[me@linuxbox ~]$ foo
bin  games    kerberos  lib64    local  share  tmp
etc  include  lib       libexec  sbin   src
/home/me
[me@linuxbox ~]$
```

We can also use the `type` command again to see our alias:

```
[me@linuxbox ~]$ type foo
foo is aliased to `cd /usr; ls ; cd -'
```

To remove an alias, the `unalias` command is used, like so:

```
[me@linuxbox ~]$ unalias foo
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

While we purposely avoided naming our alias with an existing command name, it is sometimes desirable to do so. This is often done to apply a commonly desired option to each invocation of a common command. For instance, we saw earlier how the `ls` command is often aliased to add color support:

```
[me@linuxbox ~]$ type ls
ls is aliased to `ls --color=tty'
```

To see all the aliases defined in the environment, use the `alias` command without arguments. Here are some of the aliases defined by default on a Fedora system. Try to figure out what they all do:

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
```

There is one tiny problem with defining aliases on the command line. They vanish when your shell session ends. In a later chapter we will see how to add our own aliases to the files that establish the environment each time we log on, but for now, enjoy the fact that we have taken our first, albeit tiny, step into the world of shell programming!

Revisiting Old Friends

Now that we have learned how to find the documentation for commands, go and look up the documentation for all the commands we have encountered so far. Study what additional options are available and try them out!

6

REDIRECTION

In this lesson we are going to unleash what may be the coolest feature of the command line: *I/O redirection*. The *I/O* stands for *input/output*, and with this facility you can redirect the input and output of commands to and from files, as well as connect multiple commands to make powerful command *pipelines*. To show off this facility, we will introduce the following commands:

- `cat`—Concatenate files.
- `sort`—Sort lines of text.
- `uniq`—Report or omit repeated lines.
- `wc`—Print newline, word, and byte counts for each file.
- `grep`—Print lines matching a pattern.
- `head`—Output the first part of a file.
- `tail`—Output the last part of a file.
- `tee`—Read from standard input and write to standard output and files.

Standard Input, Output, and Error

Many of the programs that we have used so far produce output of some kind. This output often consists of two types. First, we have the program's results; that is, the data the program is designed to produce. Second, we have status and error messages that tell us how the program is getting along. If we look at a command like `ls`, we can see that it displays its results and its error messages on the screen.

Keeping with the Unix theme of "everything is a file," programs such as `ls` actually send their results to a special file called *standard output* (often expressed as *stdout*) and their status messages to another file called s*tandard error* (*stderr*). By default, both standard output and standard error are linked to the screen and not saved into a disk file.

In addition, many programs take input from a facility called *standard input* (*stdin*), which is, by default, attached to the keyboard.

I/O redirection allows us to change where output goes and where input comes from. Normally, output goes to the screen and input comes from the keyboard, but with I/O redirection we can change that.

Redirecting Standard Output

I/O redirection allows us to redefine where standard output goes. To redirect standard output to another file instead of the screen, we use the `>` redirection operator followed by the name of the file. Why would we want to do this? It's often useful to store the output of a command in a file. For example, we could tell the shell to send the output of the `ls` command to the file *ls-output.txt* instead of the screen:

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Here, we created a long listing of the */usr/bin* directory and sent the results to the file *ls-output.txt*. Let's examine the redirected output of the command:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me   167878 2012-02-01 15:07 ls-output.txt
```

Good—a nice, large, text file. If we look at the file with `less`, we will see that the file *ls-output.txt* does indeed contain the results from our `ls` command:

```
[me@linuxbox ~]$ less ls-output.txt
```

Now, let's repeat our redirection test but this time with a twist. We'll change the name of the directory to one that does not exist:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt
ls: cannot access /bin/usr: No such file or directory
```

We received an error message. This makes sense because we specified the nonexistent directory */bin/usr*, but why was the error message displayed on the screen rather than being redirected to the file *ls-output.txt*? The answer is that the `ls` program does not send its error messages to standard output. Instead, like most well-written Unix programs, it sends its error messages to standard error. Since we redirected only standard output and not standard error, the error message was still sent to the screen. We'll see how to redirect standard error in just a minute, but first, let's look at what happened to our output file:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me   0 2012-02-01 15:08 ls-output.txt
```

The file now has zero length! This is because, when we redirect output with the `>` redirection operator, the destination file is always rewritten from the beginning. Since our `ls` command generated no results and only an error message, the redirection operation started to rewrite the file and then stopped because of the error, resulting in its truncation. In fact, if we ever need to actually truncate a file (or create a new, empty file) we can use a trick like this:

```
[me@linuxbox ~]$ > ls-output.txt
```

Simply using the redirection operator with no command preceding it will truncate an existing file or create a new, empty file.

So, how can we append redirected output to a file instead of overwriting the file from the beginning? For that, we use the `>>` redirection operator, like so:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
```

Using the `>>` operator will result in the output being appended to the file. If the file does not already exist, it is created just as though the `>` operator had been used. Let's put it to the test:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me   503634 2012-02-01 15:45 ls-output.txt
```

We repeated the command three times, resulting in an output file three times as large.

Redirecting Standard Error

edirecting standard error lacks the ease of using a dedicated redirection operator. To redirect standard error we must refer to its *file descriptor*. A program can produce output on any of several numbered file streams. While

we have referred to the first three of these file streams as standard input, output, and error, the shell references them internally as file descriptors 0, 1, and 2, respectively. The shell provides a notation for redirecting files using the file descriptor number. Since standard error is the same as file descriptor 2, we can redirect standard error with this notation:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> ls-error.txt
```

The file descriptor 2 is placed immediately before the redirection operator to perform the redirection of standard error to the file *ls-error.txt*.

Redirecting Standard Output and Standard Error to One File

There are cases in which we may wish to capture all of the output of a command to a single file. To do this, we must redirect both standard output and standard error at the same time. There are two ways to do this. First, here is the traditional way, which works with old versions of the shell:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt 2>&1
```

Using this method, we perform two redirections. First we redirect standard output to the file *ls-output.txt*, and then we redirect file descriptor 2 (standard error) to file descriptor 1 (standard output) using the notation 2>&1.

Note: *Notice that the order of the redirections is significant. The redirection of standard error must always occur after redirecting standard output or it doesn't work. In the example above,* `> ls-output.txt 2>&1` *redirects standard error to the file* ls-output.txt, *but if the order is changed to* `2>&1 > ls-output.txt`, *standard error is directed to the screen.*

Recent versions of bash provide a second, more streamlined method for performing this combined redirection:

```
[me@linuxbox ~]$ ls -l /bin/usr &> ls-output.txt
```

In this example, we use the single notation &> to redirect both standard output and standard error to the file *ls-output.txt*.

Disposing of Unwanted Output

Sometimes silence really is golden, and we don't want output from a command—we just want to throw it away. This applies particularly to error and status messages. The system provides a way to do this by redirecting output to a special file called */dev/null*. This file is a system device called a *bit bucket*, which accepts input and does nothing with it. To suppress error messages from a command, we do this:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> /dev/null
```

/DEV/NULL IN UNIX CULTURE

The bit bucket is an ancient Unix concept, and due to its universality it has appeared in many parts of Unix culture. So when someone says he is sending your comments to "dev null," now you know what it means. For more examples, see the Wikipedia article at *http://en.wikipedia.org/wiki/Dev/null*.

Redirecting Standard Input

Up to now, we haven't encountered any commands that make use of standard input (actually we have, but we'll reveal that surprise a little bit later), so we need to introduce one.

cat—Concatenate Files

The `cat` command reads one or more files and copies them to standard output like so:

```
cat [file...]
```

In most cases, you can think of `cat` as being analogous to the `TYPE` command in DOS. You can use it to display files without paging. For example,

```
[me@linuxbox ~]$ cat ls-output.txt
```

will display the contents of the file *ls-output.txt*. `cat` is often used to display short text files. Since `cat` can accept more than one file as an argument, it can also be used to join files together. Say we have downloaded a large file that has been split into multiple parts (multimedia files are often split this way on Usenet), and we want to join them back together. If the files were named

```
movie.mpeg.001  movie.mpeg.002 ... movie.mpeg.099
```

we could rejoin them with this command:

```
[me@linuxbox ~]$ cat movie.mpeg.0* > movie.mpeg
```

Since wildcards always expand in sorted order, the arguments will be arranged in the correct order.

This is all well and good, but what does this have to do with standard input? Nothing yet, but let's try something else. What happens if we enter `cat` with no arguments?

```
[me@linuxbox ~]$ cat
```

Nothing happens—it just sits there like it's hung. It may seem that way, but it's really doing exactly what it's supposed to.

If `cat` is not given any arguments, it reads from standard input, and since standard input is, by default, attached to the keyboard, it's waiting for us to type something!

Try this:

```
[me@linuxbox ~]$ cat
The quick brown fox jumped over the lazy dog.
```

Next, type CTRL-D (i.e., hold down the CTRL key and press D) to tell cat that it has reached *end-of-file (EOF)* on standard input:

```
[me@linuxbox ~]$ cat
The quick brown fox jumped over the lazy dog.
The quick brown fox jumped over the lazy dog.
```

In the absence of filename arguments, cat copies standard input to standard output, so we see our line of text repeated. We can use this behavior to create short text files. Let's say that we wanted to create a file called *lazy_dog.txt* containing the text in our example. We would do this:

```
[me@linuxbox ~]$ cat > lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Enter the command followed by the text we want to place in the file. Remember to type CTRL-D at the end. Using the command line, we have implemented the world's dumbest word processor! To see our results, we can use cat to copy the file to standard output again:

```
[me@linuxbox ~]$ cat lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Now that we know how cat accepts standard input in addition to filename arguments, let's try redirecting standard input:

```
[me@linuxbox ~]$ cat < lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Using the < redirection operator, we change the source of standard input from the keyboard to the file *lazy_dog.txt*. We see that the result is the same as passing a single filename argument. This is not particularly useful compared to passing a filename argument, but it serves to demonstrate using a file as a source of standard input. Other commands make better use of standard input, as we shall soon see.

Before we move on, check out the man page for cat, as it has several interesting options.

Pipelines

The ability of commands to read data from standard input and send to standard output is utilized by a shell feature called *pipelines*. Using the pipe operator | (vertical bar), the standard output of one command can be *piped* into the standard input of another.

```
command1 | command2
```

To fully demonstrate this, we are going to need some commands. Remember how we said there was one we already knew that accepts standard input? It's `less`. We can use `less` to display, page by page, the output of any command that sends its results to standard output:

```
[me@linuxbox ~]$ ls -l /usr/bin | less
```

This is extremely handy! Using this technique, we can conveniently examine the output of any command that produces standard output.

Filters

Pipelines are often used to perform complex operations on data. It is possible to put several commands together into a pipeline. Frequently, the commands used this way are referred to as *filters*. Filters take input, change it somehow, and then output it. The first one we will try is `sort`. Imagine we want to make a combined list of all of the executable programs in */bin* and */usr/bin*, put them in sorted order, and then view the list:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | less
```

Since we specified two directories (*/bin* and */usr/bin*), the output of `ls` would have consisted of two sorted lists, one for each directory. By including `sort` in our pipeline, we changed the data to produce a single, sorted list.

uniq—Report or Omit Repeated Lines

The `uniq` command is often used in conjunction with `sort`. `uniq` accepts a sorted list of data from either standard input or a single filename argument (see the `uniq` man page for details) and, by default, removes any duplicates from the list. So, to make sure our list has no duplicates (that is, any programs of the same name that appear in both the */bin* and */usr/bin* directories) we will add `uniq` to our pipeline:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | less
```

In this example, we use `uniq` to remove any duplicates from the output of the `sort` command. If we want to see the list of duplicates instead, we add the `-d` option to `uniq` like so:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq -d | less
```

wc—Print Line, Word, and Byte Counts

The `wc` (word count) command is used to display the number of lines, words, and bytes contained in files. For example:

```
[me@linuxbox ~]$ wc ls-output.txt
 7902  64566 503634 ls-output.txt
```

In this case it prints out three numbers: lines, words, and bytes contained in *ls-output.txt*. Like our previous commands, if executed without command-line arguments, `wc` accepts standard input. The `-l` option limits its output to only report lines. Adding it to a pipeline is a handy way to count things. To see the number of items we have in our sorted list, we can do this:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | wc -l
2728
```

grep—Print Lines Matching a Pattern

`grep` is a powerful program used to find text patterns within files, like this:

```
grep pattern [file...]
```

When `grep` encounters a "pattern" in the file, it prints out the lines containing it. The patterns that `grep` can match can be very complex, but for now we will concentrate on simple text matches. We'll cover the advanced patterns, called *regular expressions*, in Chapter 19.

Let's say we want to find all the files in our list of programs that have the word *zip* in the name. Such a search might give us an idea of which programs on our system have something to do with file compression. We would do this:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

There are a couple of handy options for `grep`: `-i`, which causes `grep` to ignore case when performing the search (normally searches are case sensitive) and `-v`, which tells `grep` to print only lines that do not match the pattern.

head/tail—Print First/Last Part of Files

Sometimes you don't want all the output from a command. You may want only the first few lines or the last few lines. The `head` command prints the first 10 lines of a file, and the `tail` command prints the last 10 lines. By default, both commands print 10 lines of text, but this can be adjusted with the `-n` option:

```
[me@linuxbox ~]$ head -n 5 ls-output.txt
total 343496
-rwxr-xr-x 1 root root       31316 2011-12-05 08:58 [
```

```
-rwxr-xr-x 1 root root        8240 2011-12-09 13:39 411toppm
-rwxr-xr-x 1 root root      111276 2011-11-26 14:27 a2p
-rwxr-xr-x 1 root root       25368 2010-10-06 20:16 a52dec
[me@linuxbox ~]$ tail -n 5 ls-output.txt
-rwxr-xr-x 1 root root        5234 2011-06-27 10:56 znew
-rwxr-xr-x 1 root root         691 2009-09-10 04:21 zonetab2pot.py
-rw-r--r-- 1 root root         930 2011-11-01 12:23 zonetab2pot.pyc
-rw-r--r-- 1 root root         930 2011-11-01 12:23 zonetab2pot.pyo
lrwxrwxrwx 1 root root           6 2012-01-31 05:22 zsoelim -> soelim
```

These can be used in pipelines as well:

```
[me@linuxbox ~]$ ls /usr/bin | tail -n 5
znew
zonetab2pot.py
zonetab2pot.pyc
zonetab2pot.pyo
zsoelim
```

tail has an option that allows you to view files in real time. This is useful for watching the progress of log files as they are being written. In the following example, we will look at the *messages* file in */var/log*. Superuser privileges are required to do this on some Linux distributions, because the */var/log/messages* file may contain security information.

```
[me@linuxbox ~]$ tail -f /var/log/messages
Feb  8 13:40:05 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 13:40:05 twin4 dhclient: bound to 192.168.1.4 -- renewal in 1652
seconds.
Feb  8 13:55:32 twin4 mountd[3953]: /var/NFSv4/musicbox exported to both
192.168.1.0/24 and twin7.localdomain in 192.168.1.0/24,twin7.localdomain
Feb  8 14:07:37 twin4 dhclient: DHCPREQUEST on eth0 to 192.168.1.1 port 67
Feb  8 14:07:37 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 14:07:37 twin4 dhclient: bound to 192.168.1.4 -- renewal in 1771
seconds.
Feb  8 14:09:56 twin4 smartd[3468]: Device: /dev/hda, SMART Prefailure
Attribute: 8 Seek_Time_Performance changed from 237 to 236
Feb  8 14:10:37 twin4 mountd[3953]: /var/NFSv4/musicbox exported to both
192.168.1.0/24 and twin7.localdomain in 192.168.1.0/24,twin7.localdomain
Feb  8 14:25:07 twin4 sshd(pam_unix)[29234]: session opened for user me by
(uid=0)
Feb  8 14:25:36 twin4 su(pam_unix)[29279]: session opened for user root by
me(uid=500)
```

Using the -f option, tail continues to monitor the file and when new lines are appended, they immediately appear on the display. This continues until you type CTRL-C.

tee—Read from Stdin and Output to Stdout and Files

In keeping with our plumbing analogy, Linux provides a command called tee which creates a "T" fitting on our pipe. The tee program reads standard input and copies it to both standard output (allowing the data to continue down the pipeline) and to one or more files. This is useful for capturing a pipeline's contents at an intermediate stage of processing. Here we repeat

one of our earlier examples, this time including `tee` to capture the entire directory listing to the file *ls.txt* before `grep` filters the pipeline's contents:

```
[me@linuxbox ~]$ ls /usr/bin | tee ls.txt | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

Final Note

As always, check out the documentation of each of the commands we have covered in this chapter. We have seen only their most basic usage, and they all have a number of interesting options. As we gain Linux experience, we will see that the redirection feature of the command line is extremely useful for solving specialized problems. Many commands make use of standard input and output, and almost all command-line programs use standard error to display their informative messages.

LINUX IS ABOUT IMAGINATION

When I am asked to explain the difference between Windows and Linux, I often use a toy analogy.

Windows is like a Game Boy. You go to the store and buy one all shiny new in the box. You take it home, turn it on, and play with it. Pretty graphics, cute sounds. After a while, though, you get tired of the game that came with it, so you go back to the store and buy another one. This cycle repeats over and over. Finally, you go back to the store and say to the person behind the counter, "I want a game that does this!" only to be told that no such game exists because there is no "market demand" for it. Then you say, "But I only need to change this one thing!" The person behind the counter says you can't change it. The games are all sealed up in their cartridges. You discover that your toy is limited to the games that others have decided that you need and no more.

Linux, on the other hand, is like the world's largest Erector Set. You open it up, and it's just a huge collection of parts—a lot of steel struts, screws, nuts, gears, pulleys, and motors and a few suggestions on what to build. So you start to play with it. You build one of the suggestions and then another. After a while you discover that you have your own ideas of what to make. You don't ever have to go back to the store, because you already have everything you need. The Erector Set takes on the shape of your imagination. It does what you want.

Your choice of toys is, of course, a personal thing, so which toy would you find more satisfying?

7

SEEING THE WORLD AS THE SHELL SEES IT

In this chapter we are going to look at some of the "magic" that occurs on the command line when you press the ENTER key. While we will examine several interesting and complex features of the shell, we will do it with just one new command:

- `echo`—Display a line of text.

Expansion

Each time you type a command line and press the ENTER key, `bash` performs several processes upon the text before it carries out your command. We've seen a couple of cases of how a simple character sequence, for example *, can have a lot of meaning to the shell. The process that makes this happen is called *expansion*. With expansion, you enter something, and it is expanded into something else before the shell acts upon it. To demonstrate what we

mean by this, let's take a look at the `echo` command. `echo` is a shell builtin that performs a very simple task: It prints out its text arguments on standard output.

```
[me@linuxbox ~]$ echo this is a test
this is a test
```

That's pretty straightforward. Any argument passed to `echo` gets displayed. Let's try another example:

```
[me@linuxbox ~]$ echo *
Desktop Documents ls-output.txt Music Pictures Public Templates Videos
```

So what just happened? Why didn't `echo` print *? As you recall from our work with wildcards, the * character means "match any characters in a filename," but what we didn't see in our original discussion was how the shell does that. The simple answer is that the shell expands the * into something else (in this instance, the names of the files in the current working directory) before the `echo` command is executed. When the ENTER key is pressed, the shell automatically expands any qualifying characters on the command line before the command is carried out, so the `echo` command never saw the *, only its expanded result. Knowing this, we can see that `echo` behaved as expected.

Pathname Expansion

The mechanism by which wildcards work is called *pathname expansion*. If we try some of the techniques that we employed in our earlier chapters, we will see that they are really expansions. Given a home directory that looks like this:

```
[me@linuxbox ~]$ ls
Desktop    ls-output.txt  Pictures  Templates
Documents  Music          Public    Videos
```

we could carry out the following expansions:

```
[me@linuxbox ~]$ echo D*
Desktop Documents
```

and

```
[me@linuxbox ~]$ echo *s
Documents Pictures Templates Videos
```

or even

```
[me@linuxbox ~]$ echo [[:upper:]]*
Desktop Documents Music Pictures Public Templates Videos
```

And looking beyond our home directory:

```
[me@linuxbox ~]$ echo /usr/*/share
/usr/kerberos/share /usr/local/share
```

PATHNAME EXPANSION OF HIDDEN FILES

As we know, filenames that begin with a period character are hidden. Pathname expansion also respects this behavior. An expansion such as

```
echo *
```

does not reveal hidden files.

It might appear at first glance that we could include hidden files in an expansion by starting the pattern with a leading period, like this:

```
echo .*
```

It almost works. However, if we examine the results closely, we will see that the names `.` and `..` will also appear in the results. Since these names refer to the current working directory and its parent directory, using this pattern will likely produce an incorrect result. We can see this if we try the command

```
ls -d .* | less
```

To correctly perform pathname expansion in this situation, we have to employ a more specific pattern. This will work correctly:

```
ls -d .[!.]?*
```

This pattern expands into every filename that begins with a period, does not include a second period, contains at least one additional character, and may be followed by any other characters.

Tilde Expansion

As you may recall from our introduction to the `cd` command, the tilde character (~) has a special meaning. When used at the beginning of a word, it expands into the name of the home directory of the named user or, if no user is named, the home directory of the current user:

```
[me@linuxbox ~]$ echo ~
/home/me
```

If user *foo* has an account, then

```
[me@linuxbox ~]$ echo ~foo
/home/foo
```

Arithmetic Expansion

The shell allows arithmetic to be performed by expansion. This allows us to use the shell prompt as a calculator:

```
[me@linuxbox ~]$ echo $((2 + 2))
4
```

Arithmetic expansion uses the following form:

```
$((expression))
```

where *expression* is an arithmetic expression consisting of values and arithmetic operators.

Arithmetic expansion supports only integers (whole numbers, no decimals) but can perform quite a number of different operations. Table 7-1 lists a few of the supported operators.

Table 7-1: Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division (But remember, because expansion supports only integer arithmetic, results are integers.)
%	Modulo, which simply means *remainder*
**	Exponentiation

Spaces are not significant in arithmetic expressions, and expressions may be nested. For example, multiply 5^2 by 3:

```
[me@linuxbox ~]$ echo $(($((5**2)) * 3))
75
```

Single parentheses may be used to group multiple subexpressions. With this technique, we can rewrite the example above and get the same result using a single expansion instead of two:

```
[me@linuxbox ~]$ echo $(((5**2) * 3))
75
```

Here is an example using the division and remainder operators. Notice the effect of integer division:

```
[me@linuxbox ~]$ echo Five divided by two equals $((5/2))
Five divided by two equals 2
```

```
[me@linuxbox ~]$ echo with $((5%2)) left over.
with 1 left over.
```

Arithmetic expansion is covered in greater detail in Chapter 34.

Brace Expansion

Perhaps the strangest expansion is called *brace expansion*. With it, you can create multiple text strings from a pattern containing braces. Here's an example:

```
[me@linuxbox ~]$ echo Front-{A,B,C}-Back
Front-A-Back Front-B-Back Front-C-Back
```

Patterns to be brace expanded may contain a leading portion called a *preamble* and a trailing portion called a *postscript*. The brace expression itself may contain either a comma-separated list of strings or a range of integers or single characters. The pattern may not contain embedded whitespace. Here is an example using a range of integers:

```
[me@linuxbox ~]$ echo Number_{1..5}
Number_1 Number_2 Number_3 Number_4 Number_5
```

Here we get a range of letters in reverse order:

```
[me@linuxbox ~]$ echo {Z..A}
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

Brace expansions may be nested:

```
[me@linuxbox ~]$ echo a{A{1,2},B{3,4}}b
aA1b aA2b aB3b aB4b
```

So what is this good for? The most common application is to make lists of files or directories to be created. For example, if we were photographers and had a large collection of images that we wanted to organize by years and months, the first thing we might do is create a series of directories named in numeric year-month format. This way, the directory names will sort in chronological order. We could type out a complete list of directories, but that's a lot of work and it's error prone too. Instead, we could do this:

```
[me@linuxbox ~]$ mkdir Pics
[me@linuxbox ~]$ cd Pics
[me@linuxbox Pics]$ mkdir {2009..2011}-0{1..9} {2009..2011}-{10..12}
[me@linuxbox Pics]$ ls
2009-01  2009-07  2010-01  2010-07  2011-01  2011-07
2009-02  2009-08  2010-02  2010-08  2011-02  2011-08
2009-03  2009-09  2010-03  2010-09  2011-03  2011-09
2009-04  2009-10  2010-04  2010-10  2011-04  2011-10
2009-05  2009-11  2010-05  2010-11  2011-05  2011-11
2009-06  2009-12  2010-06  2010-12  2011-06  2011-12
```

Pretty slick!

Parameter Expansion

We're only going to touch briefly on parameter expansion in this chapter, but we'll be covering it extensively later. It's a feature that is more useful in shell scripts than directly on the command line. Many of its capabilities have to do with the system's ability to store small chunks of data and to give each chunk a name. Many such chunks, more properly called *variables*, are available for your examination. For example, the variable named `USER` contains your username. To invoke parameter expansion and reveal the contents of `USER`, you would do this:

```
[me@linuxbox ~]$ echo $USER
me
```

To see a list of available variables, try this:

```
[me@linuxbox ~]$ printenv | less
```

You may have noticed that with other types of expansion, if you mistype a pattern, the expansion will not take place and the `echo` command will simply display the mistyped pattern. With parameter expansion, if you misspell the name of a variable, the expansion will still take place but will result in an empty string:

```
[me@linuxbox ~]$ echo $SUER

[me@linuxbox ~]$
```

Command Substitution

Command substitution allows us to use the output of a command as an expansion:

```
[me@linuxbox ~]$ echo $(ls)
Desktop Documents ls-output.txt Music Pictures Public Templates Videos
```

One of my favorites goes something like this:

```
[me@linuxbox ~]$ ls -l $(which cp)
-rwxr-xr-x 1 root root 71516 2012-12-05 08:58 /bin/cp
```

Here we passed the results of `which cp` as an argument to the `ls` command, thereby getting the listing of the `cp` program without having to know its full pathname. We are not limited to just simple commands. Entire pipelines can be used (only partial output shown):

```
[me@linuxbox ~]$ file $(ls /usr/bin/* | grep zip)
/usr/bin/bunzip2:      symbolic link to `bzip2'
/usr/bin/bzip2:        ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV
), dynamically linked (uses shared libs), for GNU/Linux 2.6.9, stripped
/usr/bin/bzip2recover: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.9, stripped
/usr/bin/funzip:       ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV
```

```
), dynamically linked (uses shared libs), for GNU/Linux 2.6.9, stripped
/usr/bin/gpg-zip:      Bourne shell script text executable
/usr/bin/gunzip:       symbolic link to `../../bin/gunzip'
/usr/bin/gzip:         symbolic link to `../../bin/gzip'
/usr/bin/mzip:         symbolic link to `mtools'
```

In this example, the results of the pipeline became the argument list of the `file` command.

There is an alternative syntax for command substitution in older shell programs that is also supported in `bash`. It uses *back quotes* instead of the dollar sign and parentheses:

```
[me@linuxbox ~]$ ls -l `which cp`
-rwxr-xr-x 1 root root 71516 2012-12-05 08:58 /bin/cp
```

Quoting

Now that we've seen how many ways the shell can perform expansions, it's time to learn how we can control it. For example, take this:

```
[me@linuxbox ~]$ echo this is a     test
this is a test
```

Or this:

```
[me@linuxbox ~]$ echo The total is $100.00
The total is 00.00
```

In the first example, *word splitting* by the shell removed extra whitespace from the `echo` command's list of arguments. In the second example, parameter expansion substituted an empty string for the value of `$1` because it was an undefined variable. The shell provides a mechanism called *quoting* to selectively suppress unwanted expansions.

Double Quotes

The first type of quoting we will look at is *double quotes*. If you place text inside double quotes, all the special characters used by the shell lose their special meaning and are treated as ordinary characters. The exceptions are `$` (dollar sign), `\` (backslash), and `` ` `` (back tick). This means that word splitting, pathname expansion, tilde expansion, and brace expansion are suppressed, but parameter expansion, arithmetic expansion, and command substitution are still carried out. Using double quotes, we can cope with filenames containing embedded spaces. Say we were the unfortunate victim of a file called *two words.txt*. If we tried to use this on the command line, word splitting would cause this to be treated as two separate arguments rather than the desired single argument:

```
[me@linuxbox ~]$ ls -l two words.txt
ls: cannot access two: No such file or directory
ls: cannot access words.txt: No such file or directory
```

By using double quotes, we stop the word splitting and get the desired result; further, we can even repair the damage:

```
[me@linuxbox ~]$ ls -l "two words.txt"
-rw-rw-r-- 1 me   me   18 2012-02-20 13:03 two words.txt
[me@linuxbox ~]$ mv "two words.txt" two_words.txt
```

There! Now we don't have to keep typing those pesky double quotes.

Remember: Parameter expansion, arithmetic expansion, and command substitution still take place within double quotes:

```
[me@linuxbox ~]$ echo "$USER $((2+2)) $(cal)"
me 4   February 2012
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29
```

We should take a moment to look at the effect of double quotes on command substitution. First let's look a little deeper at how word splitting works. In our earlier example, we saw how word splitting appears to remove extra spaces in our text:

```
[me@linuxbox ~]$ echo this is a     test
this is a test
```

By default, word splitting looks for the presence of spaces, tabs, and newlines (linefeed characters) and treats them as *delimiters* between words. This means that unquoted spaces, tabs, and newlines are not considered to be part of the text. They serve only as separators. Since they separate the words into different arguments, our example command line contains a command followed by four distinct arguments. If we add double quotes, however, word splitting is suppressed and the embedded spaces are not treated as delimiters; rather, they become part of the argument:

```
[me@linuxbox ~]$ echo "this is a     test"
this is a     test
```

Once the double quotes are added, our command line contains a command followed by a single argument.

The fact that newlines are considered delimiters by the word splitting mechanism causes an interesting, albeit subtle, effect on command substitution. Consider the following:

```
[me@linuxbox ~]$ echo $(cal)
February 2012 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29
[me@linuxbox ~]$ echo "$(cal)"
```

```
   February 2012
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29
```

In the first instance, the unquoted command substitution resulted in a command line containing 38 arguments; in the second, the result was a command line with 1 argument that includes the embedded spaces and newlines.

Single Quotes

If we need to suppress *all* expansions, we use *single quotes*. Here is a comparison of unquoted, double quotes, and single quotes:

```
[me@linuxbox ~]$ echo text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
text /home/me/ls-output.txt a b foo 4 me
[me@linuxbox ~]$ echo "text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER"
text ~/*.txt {a,b} foo 4 me
[me@linuxbox ~]$ echo 'text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER'
text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
```

As we can see, with each succeeding level of quoting, more and more expansions are suppressed.

Escaping Characters

Sometimes we want to quote only a single character. To do this, we can precede a character with a backslash, which in this context is called the *escape character*. Often this is done inside double quotes to selectively prevent an expansion.

```
[me@linuxbox ~]$ echo "The balance for user $USER is: \$5.00"
The balance for user me is: $5.00
```

It is also common to use escaping to eliminate the special meaning of a character in a filename. For example, it is possible to use characters in filenames that normally have special meaning to the shell. These would include `$`, `!`, `&`, (a space), and others. To include a special character in a filename, you can do this:

```
[me@linuxbox ~]$ mv bad\&filename good_filename
```

To allow a backslash character to appear, escape it by typing `\\`. Note that within single quotes, the backslash loses its special meaning and is treated as an ordinary character.

BACKSLASH ESCAPE SEQUENCES

In addition to its role as the escape character, the backslash is also used as part of a notation to represent certain special characters called *control codes*. The first 32 characters in the ASCII coding scheme are used to transmit commands to teletype-like devices. Some of these codes are familiar (tab, backspace, linefeed, and carriage return), while others are not (null, end-of-transmission, and acknowledge), as shown in Table 7-2.

Table 7-2: Backslash Escape Sequences

Escape Sequence	Meaning
`\a`	Bell ("alert"—causes the computer to beep)
`\b`	Backspace
`\n`	Newline (on Unix-like systems, this produces a linefeed)
`\r`	Carriage return
`\t`	Tab

This table lists some of the common backslash escape sequences. The idea behind using the backslash originated in the C programming language and has been adopted by many others, including the shell.

Adding the `-e` option to `echo` will enable interpretation of escape sequences. You may also place them inside `$' '`. Here, using the `sleep` command, a simple program that just waits for the specified number of seconds and then exits, we can create a primitive countdown timer.

```
sleep 10; echo -e  "Time's up\a"
```

We could also do this:

```
sleep 10; echo "Time's up" $'\a'
```

Final Note

As we move forward with using the shell, we will find that expansions and quoting will be used with increasing frequency, so it makes sense to get a good understanding of the way they work. In fact, it could be argued that they are the most important subjects to learn about the shell. Without a proper understanding of expansion, the shell will always be a source of mystery and confusion, and much of its potential power will be wasted.

8

ADVANCED KEYBOARD TRICKS

I often kiddingly describe Unix as "the operating system for people who like to type." Of course, the fact that it even has a command line is a testament to that. But command line users don't like to type *that* much. Why else would so many commands have such short names, like `cp`, `ls`, `mv`, and `rm`?

In fact, one of the most cherished goals of the command line is laziness—doing the most work with the fewest keystrokes. Another goal is never having to lift your fingers from the keyboard—never reaching for the mouse. In this chapter, we will look at `bash` features that make keyboard use faster and more efficient.

The following commands will make an appearance:

- `clear`—Clear the screen.
- `history`—Display the contents of the history list.

Command Line Editing

bash uses a library (a shared collection of routines that different programs can use) called *Readline* to implement command line editing. We have already seen some of this. We know, for example, that the arrow keys move the cursor, but there are many more features. Think of these as additional tools that we can employ in our work. It's not important to learn all of them, but many of them are very useful. Pick and choose as desired.

Note: *Some of the key sequences below (particularly those that use the ALT key) may be intercepted by the GUI for other functions. All of the key sequences should work properly when using a virtual console.*

Cursor Movement

Table 8-1 lists the keys used to move the cursor.

Table 8-1: Cursor Movement Commands

Key	Action
CTRL-A	Move cursor to the beginning of the line.
CTRL-E	Move cursor to the end of the line.
CTRL-F	Move cursor forward one character; same as the right arrow key.
CTRL-B	Move cursor backward one character; same as the left arrow key.
ALT-F	Move cursor forward one word.
ALT-B	Move cursor backward one word.
CTRL-L	Clear the screen and move the cursor to the top left corner. The `clear` command does the same thing.

Modifying Text

Table 8-2 lists keyboard commands that are used to edit characters on the command line.

Cutting and Pasting (Killing and Yanking) Text

The Readline documentation uses the terms *killing* and *yanking* to refer to what we would commonly call cutting and pasting. Table 8-3 lists the commands for cutting and pasting. Items that are cut are stored in a buffer called the *kill-ring*.

Table 8-2: Text Editing Commands

Key	Action
CTRL-D	Delete the character at the cursor location.
CTRL-T	Transpose (exchange) the character at the cursor location with the one preceding it.
ALT-T	Transpose the word at the cursor location with the one pre ceding it.
ALT-L	Convert the characters from the cursor location to the end of the word to lowercase.
ALT-U	Convert the characters from the cursor location to the end of the word to uppercase.

Table 8-3: Cut and Paste Commands

Key	Action
CTRL-K	Kill text from the cursor location to the end of line.
CTRL-U	Kill text from the cursor location to the beginning of the line.
ALT-D	Kill text from the cursor location to the end of the current word.
ALT-BACKSPACE	Kill text from the cursor location to the beginning of the cur rent word. If the cursor is at the beginning of a word, kill the previous word.
CTRL-Y	Yank text from the kill-ring and insert it at the cursor location.

THE META KEY

If you venture into the Readline documentation, which can be found in the "READLINE" section of the bash man page, you will encounter the term *meta key*. On modern keyboards this maps to the ALT key, but it wasn't always so.

Back in the dim times (before PCs but after Unix) not everybody had their own computer. What they might have had was a device called a *terminal*. A terminal was a communication device that featured a text-display screen and a keyboard and had just enough electronics inside to display text characters and move the cursor around. It was attached (usually by serial cable) to a larger computer or the communication network of a larger computer. There were many different brands of terminals, and they all had different keyboards and display feature sets. Since they all tended to at least understand ASCII, software

developers wanting portable applications wrote to the lowest common denominator. Unix systems have a very elaborate way of dealing with terminals and their different display features. Since the developers of Readline could not be sure of the presence of a dedicated extra control key, they invented one and called it *meta*. While the ALT key serves as the meta key on modern keyboards, you can also press and release the ESC key to get the same effect as holding down the ALT key if you're still using a terminal (which you can still do in Linux!).

Completion

Another way that the shell can help you is through a mechanism called *completion*. Completion occurs when you press the TAB key while typing a command. Let's see how this works. Say your home directory looks like this:

```
[me@linuxbox ~]$ ls
Desktop    ls-output.txt  Pictures  Templates      Videos
Documents  Music          Public
```

Try typing the following but *don't press the ENTER key*:

```
[me@linuxbox ~]$ ls l
```

Now press the TAB key:

```
[me@linuxbox ~]$ ls ls-output.txt
```

See how the shell completed the line for you? Let's try another one. Again, don't press ENTER:

```
[me@linuxbox ~]$ ls D
```

Press TAB:

```
[me@linuxbox ~]$ ls D
```

No completion—just a beep. This happened because `D` matches more than one entry in the directory. For completion to be successful, the "clue" you give it has to be unambiguous. We can go further:

```
[me@linuxbox ~]$ ls Do
```

Then press TAB:

```
[me@linuxbox ~]$ ls Documents
```

The completion is successful.

While this example shows completion of pathnames, which is completion's most common use, completion will also work on variables (if the beginning of the word is a `$`), usernames (if the word begins with `~`), commands (if the word is the first word on the line), and hostnames (if the beginning of the word is `@`). Hostname completion works only for hostnames listed in */etc/hosts*.

A number of control and meta key sequences are associated with completion (see Table 8-4).

Table 8-4: Completion Commands

Key	Action
ALT-?	Display list of possible completions. On most systems you can also do this by pressing the TAB key a second time, which is much easier.
ALT-*	Insert all possible completions. This is useful when you want to use more than one possible match.

There quite a few more that I find rather obscure. You can see a list in the `bash` man page under the "READLINE" section.

PROGRAMMABLE COMPLETION

Recent versions of `bash` have a facility called *programmable completion*. Programmable completion allows you (or, more likely, your distribution provider) to add additional completion rules. Usually this is done to add support for specific applications. For example, it is possible to add completions for the option list of a command or match particular file types that an application supports. Ubuntu has a fairly large set defined by default. Programmable completion is implemented by shell functions, a kind of mini shell script that we will cover in later chapters. If you are curious, try

```
set | less
```

and see if you can find them. Not all distributions include them by default.

Using History

As we discovered in Chapter 1, `bash` maintains a history of commands that have been entered. This list of commands is kept in your home directory in a file called *.bash_history*. The history facility is a useful resource for reducing the amount of typing you have to do, especially when combined with command-line editing.

Searching History

At any time, we can view the contents of the history list:

```
[me@linuxbox ~]$ history | less
```

By default, bash stores the last 500 commands you have entered. We will see how to adjust this value in Chapter 11. Let's say we want to find the commands we used to list */usr/bin*. Here is one way we could do this:

```
[me@linuxbox ~]$ history | grep /usr/bin
```

And let's say that among our results we got a line containing an interesting command like this:

```
   88  ls -l /usr/bin > ls-output.txt
```

The number 88 is the line number of the command in the history list. We could use this immediately with another type of expansion called *history expansion*. To use our discovered line, we could do this:

```
[me@linuxbox ~]$ !88
```

bash will expand !88 into the contents of the 88th line in the history list. We will cover other forms of history expansion a little later.

bash also provides the ability to search the history list incrementally. This means that we can tell bash to search the history list as we enter characters, with each additional character further refining our search. To start an incremental search, enter CTRL-R followed by the text you are looking for. When you find it, you can either press ENTER to execute the command or press CTRL-J to copy the line from the history list to the current command line. To find the next occurrence of the text (moving "up" the history list), press CTRL-R again. To quit searching, press either CTRL-G or CTRL-C. Here we see it in action:

```
[me@linuxbox ~]$
```

First press CTRL-R:

```
(reverse-i-search)`':
```

The prompt changes to indicate that we are performing a reverse incremental search. It is "reverse" because we are searching from "now" to some time in the past. Next, we start typing our search text, which in this example is /usr/bin:

```
(reverse-i-search)`/usr/bin': ls -l /usr/bin > ls-output.txt
```

Immediately, the search returns its result. Now we can execute the command by pressing ENTER, or we can copy the command to our current command line for further editing by pressing CTRL-J. Let's copy it. Press CTRL-J:

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Our shell prompt returns, and our command line is loaded and ready for action!

Table 8-5 lists some of the keystrokes used to manipulate the history list.

Table 8-5: History Commands

Key	Action
CTRL-P	Move to the previous history entry. Same action as the up arrow.
CTRL-N	Move to the next history entry. Same action as the down arrow.
ALT-<	Move to the beginning (top) of the history list.
ALT->	Move to the end (bottom) of the history list; i.e., the current command line.
CTRL-R	Reverse incremental search. Searches incrementally from the current command line up the history list.
ALT-P	Reverse search, non-incremental. With this key, type the search string and press ENTER before the search is performed.
ALT-N	Forward search, non-incremental.
CTRL-O	Execute the current item in the history list and advance to the next one. This is handy if you are trying to re-execute a sequence of commands in the history list.

History Expansion

The shell offers a specialized type of expansion for items in the history list by using the ! character. We have already seen how the exclamation point can be followed by a number to insert an entry from the history list. There are a number of other expansion features (see Table 8-6).

I would caution against using the `!string` and `!?string` forms unless you are absolutely sure of the contents of the history list items.

Many more elements are available in the history expansion mechanism, but this subject is already too arcane and our heads may explode if we continue. The "HISTORY EXPANSION" section of the bash man page goes into all the gory details. Feel free to explore!

Table 8-6: History Expansion Commands

Sequence	Action
`!!`	Repeat the last command. It is probably easier to press the up arrow and ENTER.
`!`*`number`*	Repeat history list item *`number`*.
`!`*`string`*	Repeat last history list item starting with *`string`*.
`!?`*`string`*	Repeat last history list item containing *`string`*.

SCRIPT

In addition to the command history feature in `bash`, most Linux distributions include a program called `script`, which can be used to record an entire shell session and store it in a file. The basic syntax of the command is

```
script [file]
```

where *`file`* is the name of the file used for storing the recording. If no file is specified, the file *typescript* is used. See the `script` man page for a complete list of the program's options and features.

Final Note

In this chapter we have covered *some* of the keyboard tricks that the shell provides to help hardcore typists reduce their workloads. I suspect that as time goes by and you become more involved with the command line, you will refer to this chapter to pick up more of these tricks. For now, consider them optional and potentially helpful.

9

PERMISSIONS

Operating systems in the Unix tradition differ from those in the MS-DOS tradition in that they are not only *multitasking* systems but also *multiuser* systems.

What exactly does this mean? It means that more than one person can use the computer at the same time. While a typical computer will likely have only one keyboard and monitor, it can still be used by more than one user. For example, if a computer is attached to a network or the Internet, remote users can log in via `ssh` (secure shell) and operate the computer. In fact, remote users can execute graphical applications and have the graphical output appear on a remote display. The X Window System supports this as part of its basic design.

The multiuser capability of Linux is not a recent "innovation" but rather a feature that is deeply embedded into the design of the operating system. Considering the environment in which Unix was created, this makes perfect sense. Years ago, before computers were "personal," they were large, expensive, and centralized. A typical university computer system, for example, consisted of a large central computer located in one building and terminals located throughout the campus, each connected to the large central computer. The computer would support many users at the same time.

In order to make this practical, a method had to be devised to protect the users from each other. After all, the actions of one user could not be allowed to crash the computer, nor could one user interfere with the files belonging to another user.

In this chapter we are going to look at this essential part of system security and introduce the following commands:

- `id`—Display user identity.
- `chmod`—Change a file's mode.
- `umask`—Set the default file permissions.
- `su`—Run a shell as another user.
- `sudo`—Execute a command as another user.
- `chown`—Change a file's owner.
- `chgrp`—Change a file's group ownership.
- `passwd`—Change a user's password.

Owners, Group Members, and Everybody Else

When we were exploring the system back in Chapter 4, we may have encountered the following problem when trying to examine a file such as */etc/shadow*.

```
[me@linuxbox ~]$ file /etc/shadow
/etc/shadow: regular file, no read permission
[me@linuxbox ~]$ less /etc/shadow
/etc/shadow: Permission denied
```

The reason for this error message is that, as regular users, we do not have permission to read this file.

In the Unix security model, a user may *own* files and directories. When a user owns a file or directory, the user has control over its access. Users can, in turn, belong to a *group* consisting of one or more users who are given access to files and directories by their owners. In addition to granting access to a group, an owner may also grant some set of access rights to everybody, which in Unix terms is referred to as the *world*. To find out information about your identity, use the `id` command:

```
[me@linuxbox ~]$ id
uid=500(me) gid=500(me) groups=500(me)
```

Let's look at the output. When user accounts are created, users are assigned a number called a *user ID*, or *uid*. This is then, for the sake of the humans, mapped to a username. The user is assigned a *primary group ID,* or *gid*, and may belong to additional groups. The previous example is from a Fedora system. On other systems, such as Ubuntu, the output may look a little different.

```
[me@linuxbox ~]$ id
uid=1000(me) gid=1000(me)
groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(
plugdev),108(lpadmin),114(admin),1000(me)
```

As we can see, the uid and gid numbers are different. This is simply because Fedora starts its numbering of regular user accounts at 500, while Ubuntu starts at 1000. We can also see that the Ubuntu user belongs to a lot more groups. This has to do with the way Ubuntu manages privileges for system devices and services.

So where does this information come from? Like so many things in Linux, it comes from a couple of text files. User accounts are defined in the */etc/passwd* file, and groups are defined in the */etc/group* file. When user accounts and groups are created, these files are modified along with */etc/shadow*, which holds information about the user's password. For each user account, the */etc/passwd* file defines the user (login) name, the uid, the gid, the account's real name, the home directory, and the login shell. If you examine the contents of */etc/passwd* and */etc/group*, you will notice that besides the regular user accounts there are accounts for the superuser (uid 0) and various other system users.

In Chapter 10, when we cover processes, you will see that some of these other "users" are, in fact, quite busy.

While many Unix-like systems assign regular users to a common group such as *users*, modern Linux practice is to create a unique, single-member group with the same name as the user. This makes certain types of permission assignment easier.

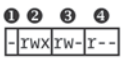
Reading, Writing, and Executing

Access rights to files and directories are defined in terms of read access, write access, and execution access. If we look at the output of the `ls` command, we can get some clue as to how this is implemented:

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me   0 2012-03-06 14:52 foo.txt
```

The first 10 characters of the listing are the *file attributes* (see Figure 9-1). The first of these characters is the *file type*. Table 9-1 lists the file types you are most likely to see (there are other, less common types too).

The remaining nine characters of the file attributes, called the *file mode,* represent the read, write, and execute permissions for the file's owner, the file's group owner, and everybody else.



❶ File type (see Table 9-1)
❷ Owner permissions (see Table 9-2)
❸ Group permissions (see Table 9-2)
❹ World permissions (see Table 9-2)

Figure 9-1: Breakdown of file attributes

When set, the `r`, `w`, and `x` mode attributes have certain effects on files and directories, as shown in Table 9-2.

Table 9-1: File Types

Attribute	File Type
`-`	A regular file.
`d`	A directory.
`l`	A symbolic link. Notice that with symbolic links, the remaining file attributes are always `rwxrwxrwx` and are dummy values. The real file attributes are those of the file the symbolic link points to.
`c`	A *character special file*. This file type refers to a device that handles data as a stream of bytes, such as a terminal or modem.
`b`	A *block special file*. This file type refers to a device that handles data in blocks, such as a hard drive or CD-ROM drive.

Table 9-2: Permission Attributes

Attribute	Files	Directories
`r`	Allows a file to be opened and read.	Allows a directory's contents to be listed if the execute attribute is also set.
`w`	Allows a file to be written to or truncated; however, this attribute does not allow files to be renamed or deleted. The ability to delete or rename files is determined by directory attributes.	Allows files within a directory to be created, deleted, and renamed if the execute attribute is also set.
`x`	Allows a file to be treated as a program and executed. Program files written in scripting languages must also be set as readable to be executed.	Allows a directory to be entered; e.g., `cd` *`directory`*.

Table 9-3 shows some examples of file attribute settings.

Table 9-3: Permission Attribute Examples

File Attributes	Meaning
`-rwx------`	A regular file that is readable, writable, and executable by the file's owner. No one else has any access.
`-rw-------`	A regular file that is readable and writable by the file's owner. No one else has any access.

Table 9-3 (*continued*)

File Attributes	Meaning
`-rw-r--r--`	A regular file that is readable and writable by the file's owner. Members of the file's owner group may read the file. The file is world readable.
`-rwxr-xr-x`	A regular file that is readable, writable, and executable by the file's owner. The file may be read and executed by everybody else.
`-rw-rw----`	A regular file that is readable and writable by the file's owner and members of the file's owner group only.
`Lrwxrwxrwx`	A symbolic link. All symbolic links have "dummy" permissions. The real permissions are kept with the actual file pointed to by the symbolic link.
`drwxrwx---`	A directory. The owner and the members of the owner group may enter the directory and create, rename, and remove files within the directory.
`drwxr-x---`	A directory. The owner may enter the directory and create, rename, and delete files within the directory. Members of the owner group may enter the directory but cannot create, delete, or rename files.

chmod—Change File Mode

To change the mode (permissions) of a file or directory, the `chmod` command is used. Be aware that only the file's owner or the superuser can change the mode of a file or directory. `chmod` supports two distinct ways of specifying mode changes: octal number representation and symbolic representation. We will cover octal number representation first.

Octal Representation

With octal notation we use octal numbers to set the pattern of desired permissions. Since each digit in an octal number represents three binary digits, this maps nicely to the scheme used to store the file mode. Table 9-4 shows what we mean.

Table 9-4: File Modes in Binary and Octal

Octal	Binary	File Mode
0	000	`---`
1	001	`--x`
2	010	`-w-`

(*continued*)

Table 9-4 (*continued*)

Octal	Binary	File Mode
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

WHAT THE HECK IS OCTAL?

Octal (base 8) and its cousin *hexadecimal* (base 16) are number systems often used to express numbers on computers. We humans, owing to the fact that we (or at least most of us) were born with 10 fingers, count using a base 10 number system. Computers, on the other hand, were born with only one finger and thus do all all their counting in *binary* (base 2). Their number system has only two numerals, zero and one. So in binary, counting looks like this: 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011 . . .

In octal, counting is done with the numerals zero through seven, like so: 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21 . . .

Hexadecimal counting uses the numerals zero through nine plus the letters *A* through *F*: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13 . . .

While we can see the sense in binary (since computers have only one finger), what are octal and hexadecimal good for? The answer has to do with human convenience. Many times, small portions of data are represented on computers as *bit patterns*. Take for example an RGB color. On most computer displays, each pixel is composed of three color components: 8 bits of red, 8 bits of green, and 8 bits of blue. A lovely medium blue would be a 24-digit number: 010000110110111111001101.

How would you like to read and write those kinds of numbers all day? I didn't think so. Here's where another number system would help. Each digit in a hexadecimal number represents four digits in binary. In octal, each digit represents three binary digits. So our 24-digit medium blue could be condensed to a 6-digit hexadecimal number: 436FCD. Since the digits in the hexadecimal number "line up" with the bits in the binary number, we can see that the red component of our color is 43, the green 6F, and the blue CD.

These days, hexadecimal notation (often called *hex*) is more common than octal, but as we shall soon see, octal's ability to express three bits of binary is very useful.

By using three octal digits, we can set the file mode for the owner, group owner, and world.

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me   0 2012-03-06 14:52 foo.txt
[me@linuxbox ~]$ chmod 600 foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw------- 1 me    me   0 2012-03-06 14:52 foo.txt
```

By passing the argument `600`, we were able to set the permissions of the owner to read and write while removing all permissions from the group owner and world. Though remembering the octal-to-binary mapping may seem inconvenient, you will usually have to use only a few common ones: 7 (`rwx`), 6 (`rw-`), 5 (`r-x`), 4 (`r--`), and 0 (`---`).

Symbolic Representation

`chmod` also supports a symbolic notation for specifying file modes. Symbolic notation is divided into three parts: whom the change will affect, which operation will be performed, and which permission will be set. To specify who is affected, a combination of the characters *u*, *g*, *o*, and *a* is used, as shown in Table 9-5.

Table 9-5: chmod Symbolic Notation

Symbol	Meaning
u	Short for *user* but means the file or directory owner.
g	Group owner.
o	Short for *others* but means world.
a	Short for *all*; the combination of *u*, *g*, and *o*.

If no character is specified, *all* will be assumed. The operation may be a `+` indicating that a permission is to be added, a `-` indicating that a permission is to be taken away, or a `=` indicating that only the specified permissions are to be applied and that all others are to be removed.

Permissions are specified with the `r`, `w`, and `x` characters. Table 9-6 lists some examples of symbolic notation.

Table 9-6: chmod Symbolic Notation Examples

Notation	Meaning
u+x	Add execute permission for the owner.
u-x	Remove execute permission from the owner.
+x	Add execute permission for the owner, group, and world. Equivalent to a+x.

(*continued*)

Table 9-6 (*continued*)

Notation	Meaning
`o-rw`	Remove the read and write permissions from anyone besides the owner and group owner.
`go=rw`	Set the group owner and anyone besides the owner to have read and write permission. If either the group owner or world previously had execute permissions, remove them.
`u+x,go=rx`	Add execute permission for the owner and set the permissions for the group and others to read and execute. Multiple specifications may be separated by commas.

Some people prefer to use octal notation; some folks really like the symbolic. Symbolic notation does offer the advantage of allowing you to set a single attribute without disturbing any of the others.

Take a look at the `chmod` man page for more details and a list of options. A word of caution regarding the `--recursive` option: It acts on both files and directories, so it's not as useful as one would hope because we rarely want files and directories to have the same permissions.

Setting File Mode with the GUI

Now that we have seen how the permissions on files and directories are set, we can better understand the permission dialogs in the GUI. In both Nautilus (GNOME) and Konqueror (KDE), right-clicking a file or directory icon will expose a properties dialog. Figure 9-2 is an example from KDE 3.5.

Here we can see the settings for the owner, group, and world. In KDE, clicking the Advanced Permissions button brings up another dialog that allows you to set each of the mode attributes individually. Another victory for understanding brought to us by the command line!

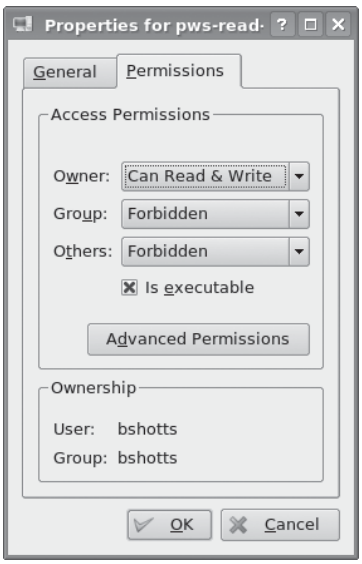


Figure 9-2: KDE 3.5 File Properties dialog

umask—Set Default Permissions

The `umask` command controls the default permissions given to a file when it is created. It uses octal notation to express a *mask* of bits to be removed from a file's mode attributes.

Let's take a look:

```
[me@linuxbox ~]$ rm -f foo.txt
[me@linuxbox ~]$ umask
0002
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me   0 2012-03-06 14:53 foo.txt
```

We first removed any existing copy of *foo.txt* to make sure we were starting fresh. Next, we ran the `umask` command without an argument to see the current value. It responded with the value `0002` (the value `0022` is another common default value), which is the octal representation of our mask. We then created a new instance of the file *foo.txt* and observed its permissions.

We can see that both the owner and group get read and write permissions, while everyone else gets only read permission. World does not have write permission because of the value of the mask. Let's repeat our example, this time setting the mask ourselves:

```
[me@linuxbox ~]$ rm foo.txt
[me@linuxbox ~]$ umask 0000
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-rw- 1 me    me   0 2012-03-06 14:58 foo.txt
```

When we set the mask to `0000` (effectively turning it off), we see that the file is now world writable. To understand how this works, we have to look at octal numbers again. If we expand the mask into binary and then compare it to the attributes, we can see what happens:

Original file mode	--- rw- rw- rw-
Mask	000 000 000 010
Result	--- rw- rw- r--

Ignore for the moment the leading 0s (we'll get to those in a minute) and observe that where the 1 appears in our mask, an attribute was removed—in this case, the world write permission. That's what the mask does. Everywhere a 1 appears in the binary value of the mask, an attribute is unset. If we look at a mask value of `0022`, we can see what it does:

Original file mode	--- rw- rw- rw-
Mask	000 000 000 010
Result	--- rw- rw- r--

Again, where a 1 appears in the binary value, the corresponding attribute is unset. Play with some values (try some 7s) to get used to how this works. When you're done, remember to clean up:

```
[me@linuxbox ~]$ rm foo.txt; umask 0002
```

Most of the time you won't have to change the mask; the default provided by your distribution will be fine. In some high-security situations, however, you will want to control it.

SOME SPECIAL PERMISSIONS

Though we usually see an octal permission mask expressed as a three-digit number, it is more technically correct to express it in four digits. Why? Because, in addition to read, write, and execute permissions, there are some other, less-used permission settings.

The first of these is the *setuid bit* (octal 4000). When applied to an executable file, it sets the *effective user ID* from that of the real user (the user actually running the program) to that of the program's owner. Most often this is given to a few programs owned by the superuser. When an ordinary user runs a program that is s*etuid root*, the program runs with the effective privileges of the superuser. This allows the program to access files and directories that an ordinary user would normally be prohibited from accessing. Clearly, because this raises security concerns, the number of setuid programs must be held to an absolute minimum.

The second less-used setting is the *setgid bit* (octal 2000). This, like the setuid bit, changes the *effective group ID* from that of the *real group ID* of the user to that of the file owner. If the setgid bit is set on a directory, newly created files in the directory will be given the group ownership of the directory rather the group ownership of the file's creator. This is useful in a shared directory when members of a common group need access to all the files in the directory, regardless of the file owner's primary group.

The third is called the *sticky bit* (octal 1000). This is a holdover from ancient Unix, where it was possible to mark an executable file as "not swappable." On files, Linux ignores the sticky bit, but if applied to a directory, it prevents users from deleting or renaming files unless the user is either the owner of the directory, the owner of the file, or the superuser. This is often used to control access to a shared directory, such as */tmp*.

Here are some examples of using `chmod` with symbolic notation to set these special permissions. First, assign setuid to a program:

```
chmod u+s program
```

Next, assign setgid to a directory:

```
chmod g+s dir
```

Finally, assign the sticky bit to a directory:

```
chmod +t dir
```

By viewing the output from `ls`, you can determine the special permissions. Here are some examples. First, a program that is setuid:

```
-rwsr-xr-x
```

Now, a directory that has the setgid attribute:

```
drwxrwsr-x
```

Finally, a directory with the sticky bit set:

```
drwxrwxrwt
```

Changing Identities

At various times, we may find it necessary to take on the identity of another user. Often we want to gain superuser privileges to carry out some administrative task, but it is also possible to "become" another regular user to perform such tasks as testing an account. There are three ways to take on an alternate identity:

- Log out and log back in as the alternate user.
- Use the `su` command.
- Use the `sudo` command.

We will skip the first technique because we know how to do it and it lacks the convenience of the other two. From within your own shell session, the `su` command allows you to assume the identity of another user and either start a new shell session with that user's ID or issue a single command as that user. The `sudo` command allows an administrator to set up a configuration file called */etc/sudoers* and define specific commands that particular users are permitted to execute under an assumed identity. The choice of which command to use is largely determined by which Linux distribution you use. Your distribution probably includes both commands, but its configuration will favor either one or the other. We'll start with `su`.

su—Run a Shell with Substitute User and Group IDs

The `su` command is used to start a shell as another user. The command syntax looks like this:

```
su [-[l]] [user]
```

If the -l option is included, the resulting shell session is a *login shell* for the specified user. This means that the user's environment is loaded and the working directory is changed to the user's home directory. This is usually what we want. If the user is not specified, the superuser is assumed. Notice that (strangely) the -l may be abbreviated as -, which is how it is most often used. To start a shell for the superuser, we would do this:

```
[me@linuxbox ~]$ su -
Password:
[root@linuxbox ~]#
```

After entering the command, we are prompted for the superuser's password. If it is successfully entered, a new shell prompt appears indicating that this shell has superuser privileges (the trailing # rather than a $) and that the current working directory is now the home directory for the superuser (normally */root*). Once in the new shell, we can carry out commands as the superuser. When finished, enter exit to return to the previous shell:

```
[root@linuxbox ~]# exit
[me@linuxbox ~]$
```

It is also possible to execute a single command rather than starting a new interactive command by using su this way:

```
su -c 'command'
```

Using this form, a single command line is passed to the new shell for execution. It is important to enclose the command in quotes, as we do not want expansion to occur in our shell but rather in the new shell:

```
[me@linuxbox ~]$ su -c 'ls -l /root/*'
Password:
-rw------- 1 root root     754 2011-08-11 03:19 /root/anaconda-ks.cfg

/root/Mail:
total 0
[me@linuxbox ~]$
```

sudo—Execute a Command as Another User

The sudo command is like su in many ways but has some important additional capabilities. The administrator can configure sudo to allow an ordinary user to execute commands as a different user (usually the superuser) in a very controlled way. In particular, a user may be restricted to one or more specific commands and no others. Another important difference is that the use of sudo does not require access to the superuser's password. To authenticate using sudo, the user enters his own password. Let's say, for example, that sudo has been configured to allow us to run a fictitious backup program called backup_script, which requires superuser privileges.

With sudo it would be done like this:

```
[me@linuxbox ~]$ sudo backup_script
Password:
System Backup Starting...
```

After entering the command, we are prompted for our password (not the superuser's), and once the authentication is complete, the specified command is carried out. One important difference between su and sudo is that sudo does not start a new shell, nor does it load another user's environment. This means that commands do not need to be quoted any differently than they would be without using sudo. Note that this behavior can be overridden by specifying various options. See the sudo man page for details.

To see what privileges are granted by sudo, use the -l option to list them:

```
[me@linuxbox ~]$ sudo -l
User me may run the following commands on this host:
    (ALL) ALL
```

UBUNTU AND SUDO

One of the recurrent problems for regular users is how to perform certain tasks that require superuser privileges. These tasks include installing and updating software, editing system configuration files, and accessing devices. In the Windows world, this is often done by giving users administrative privileges. This allows users to perform these tasks. However, it also enables programs executed by the user to have the same abilities. This is desirable in most cases, but it also permits *malware* (malicious software) such as viruses to have free run of the computer.

In the Unix world, there has always been a larger division between regular users and administrators, owing to the multiuser heritage of Unix. The approach taken in Unix is to grant superuser privileges only when needed. To do this, the su and sudo commands are commonly used.

Up until a few of years ago, most Linux distributions relied on su for this purpose. su didn't require the configuration that sudo required, and having a root account is traditional in Unix. This introduced a problem. Users were tempted to operate as root unnecessarily. In fact, some users operated their systems as the root user exclusively, because it does away with all those annoying "permission denied" messages. This is how you reduce the security of a Linux system to that of a Windows system. Not a good idea.

When Ubuntu was introduced, its creators took a different tack. By default, Ubuntu disables logins to the root account (by failing to set a password for the account) and instead uses sudo to grant superuser privileges. The initial user account is granted full access to superuser privileges via sudo and may grant similar powers to subsequent user accounts.

chown—Change File Owner and Group

The `chown` command is used to change the owner and group owner of a file or directory. Superuser privileges are required to use this command. The syntax of `chown` looks like this:

```
chown [owner][:[group]] file...
```

`chown` can change the file owner and/or the file group owner depending on the first argument of the command. Table 9-7 lists some examples.

Table 9-7: chown Argument Examples

Argument	Results
`bob`	Changes the ownership of the file from its current owner to user *bob*.
`bob:users`	Changes the ownership of the file from its current owner to user *bob* and changes the file group owner to group *users*.
`:admins`	Changes the group owner to the group *admins*. The file owner is unchanged.
`bob:`	Change the file owner from the current owner to user *bob* and changes the group owner to the login group of user *bob*.

Let's say that we have two users: *janet*, who has access to superuser privileges, and *tony*, who does not. User *janet* wants to copy a file from her home directory to the home directory of user *tony*. Since user *janet* wants *tony* to be able to edit the file, *janet* changes the ownership of the copied file from *janet* to *tony*:

```
[janet@linuxbox ~]$ sudo cp myfile.txt ~tony
Password:
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
 -rw-r--r-- 1 root  root  8031 2012-03-20 14:30 /home/tony/myfile.txt
[janet@linuxbox ~]$ sudo chown tony: ~tony/myfile.txt
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
 -rw-r--r-- 1 tony  tony  8031 2012-03-20 14:30 /home/tony/myfile.txt
```

Here we see user *janet* copy the file from her directory to the home directory of user *tony*. Next, *janet* changes the ownership of the file from *root* (a result of using `sudo`) to *tony*. Using the trailing colon in the first argument, *janet* also changed the group ownership of the file to the login group of *tony*, which happens to be group *tony*.

Notice that after the first use of `sudo`, *janet* was not prompted for her password? This is because `sudo`, in most configurations, "trusts" you for several minutes (until its timer runs out).

chgrp—Change Group Ownership

In older versions of Unix, the `chown` command changed only file ownership, not group ownership. For that purpose a separate command, `chgrp`, was used. It works much the same way as `chown`, except for being more limited.

Exercising Your Privileges

Now that we have learned how this permissions thing works, it's time to show it off. We are going to demonstrate the solution to a common problem—setting up a shared directory. Let's imagine that we have two users named *bill* and *karen*. They both have music CD collections and wish to set up a shared directory, where they will each store their music files as Ogg Vorbis or MP3. User *bill* has access to superuser privileges via `sudo`.

The first thing that needs to happen is the creation of a group that will have both *bill* and *karen* as members. Using GNOME's graphical user management tool, *bill* creates a group called *music* and adds users *bill* and *karen* to it, as shown in Figure 9-3.

Figure 9-3: Creating a new group with GNOME

Next, *bill* creates the directory for the music files:

```
[bill@linuxbox ~]$ sudo mkdir /usr/local/share/Music
Password:
```

Since *bill* is manipulating files outside his home directory, superuser privileges are required. After the directory is created, it has the following ownerships and permissions:

```
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxr-xr-x 2 root root 4096 2012-03-21 18:05 /usr/local/share/Music
```

As we can see, the directory is owned by *root* and has 755 permissions. To make this directory shareable, *bill* needs to change the group ownership and the group permissions to allow writing:

```
[bill@linuxbox ~]$ sudo chown :music /usr/local/share/Music
[bill@linuxbox ~]$ sudo chmod 775 /usr/local/share/Music
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxrwxr-x 2 root music 4096 2012-03-21 18:05 /usr/local/share/Music
```

So what does this all mean? It means that we now have a directory */usr/local/share/Music* that is owned by *root* and allows read and write access to group *music*. Group *music* has members *bill* and *karen*; thus *bill* and *karen* can create files in directory */usr/local/share/Music*. Other users can list the contents of the directory but cannot create files there.

But we still have a problem. With the current permissions, files and directories created within the *Music* directory will have the normal permissions of the users *bill* and *karen*:

```
[bill@linuxbox ~]$ > /usr/local/share/Music/test_file
[bill@linuxbox ~]$ ls -l /usr/local/share/Music
-rw-r--r-- 1 bill   bill   0 2012-03-24 20:03 test_file
```

Actually there are two problems. First, the default `umask` on this system is 0022, which prevents group members from writing files belonging to other members of the group. This would not be a problem if the shared directory contained only files, but since this directory will store music and music is usually organized in a hierarchy of artists and albums, members of the group will need the ability to create files and directories inside directories created by other members. We need to change the `umask` used by *bill* and *karen* to 0002 instead.

Second, each file and directory created by one member will be set to the primary group of the user, rather than the group *music*. This can be fixed by setting the setgid bit on the directory:

```
[bill@linuxbox ~]$ sudo chmod g+s /usr/local/share/Music
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxrwsr-x 2 root music 4096 2012-03-24 20:03 /usr/local/share/Music
```

Now we test to see if the new permissions fix the problem. *bill* sets his `umask` to 0002, removes the previous test file, and creates a new test file and directory:

```
[bill@linuxbox ~]$ umask 0002
[bill@linuxbox ~]$ rm /usr/local/share/Music/test_file
[bill@linuxbox ~]$ > /usr/local/share/Music/test_file
[bill@linuxbox ~]$ mkdir /usr/local/share/Music/test_dir
[bill@linuxbox ~]$ ls -l /usr/local/share/Music
drwxrwsr-x 2 bill   music 4096 2012-03-24 20:24 test_dir
-rw-rw-r-- 1 bill   music 0 2012-03-24 20:22 test_file
[bill@linuxbox ~]$
```

Both files and directories are now created with the correct permissions to allow all members of the group *music* to create files and directories inside the *Music* directory.

The one remaining issue is `umask`. The necessary setting lasts only until the end of the session and then must be reset. In Chapter 11, we'll look at making the change to `umask` permanent.

Changing Your Password

The last topic we'll cover in this chapter is setting passwords for yourself (and for other users if you have access to superuser privileges). To set or change a password, the `passwd` command is used. The command syntax looks like this:

```
passwd [user]
```

To change your password, just enter the `passwd` command. You will be prompted for your old password and your new password:

```
[me@linuxbox ~]$ passwd
(current) UNIX password:
New UNIX password:
```

The `passwd` command will try to enforce use of "strong" passwords. This means it will refuse to accept passwords that are too short, are too similar to previous passwords, are dictionary words, or are too easily guessed:

```
[me@linuxbox ~]$ passwd
(current) UNIX password:
New UNIX password:
BAD PASSWORD: is too similar to the old one
New UNIX password:
BAD PASSWORD: it is WAY too short
New UNIX password:
BAD PASSWORD: it is based on a dictionary word
```

If you have superuser privileges, you can specify a username as an argument to the `passwd` command to set the password for another user. Other options are available to the superuser to allow account locking, password expiration, and so on. See the `passwd` man page for details.

10

PROCESSES

Modern operating systems are usually *multitasking*, meaning that they create the illusion of doing more than one thing at once by rapidly switching from one executing program to another. The Linux kernel manages this through the use of *processes*. Processes are how Linux organizes the different programs waiting for their turn at the CPU.

Sometimes a computer will become sluggish, or an application will stop responding. In this chapter, we will look at some of the tools available at the command line that let us examine what programs are doing and how to terminate processes that are misbehaving.

This chapter will introduce the following commands:

- `ps`—Report a snapshot of current processes.
- `top`—Display tasks.
- `jobs`—List active jobs.

- `bg`—Place a job in the background.
- `fg`—Place a job in the foreground.
- `kill`—Send a signal to a process.
- `killall`—Kill processes by name.
- `shutdown`—Shut down or reboot the system.

How a Process Works

When a system starts up, the kernel initiates a few of its own activities as processes and launches a program called `init`. `init`, in turn, runs a series of shell scripts (located in */etc*) called *init scripts*, which start all the system services. Many of these services are implemented as *daemon programs*, programs that just sit in the background and do their thing without having any user interface. So even if we are not logged in, the system is at least a little busy performing routine stuff.

The fact that a program can launch other programs is expressed in the process scheme as a *parent process* producing a *child process*.

The kernel maintains information about each process to help keep things organized. For example, each process is assigned a number called a *process ID (PID)*. PIDs are assigned in ascending order, with `init` always getting PID 1. The kernel also keeps track of the memory assigned to each process, as well as the processes' readiness to resume execution. Like files, processes also have owners and user IDs, effective user IDs, and so on.

Viewing Processes with ps

The most commonly used command to view processes (there are several) is `ps`. The `ps` program has a lot of options, but in it simplest form it is used like this:

```
[me@linuxbox ~]$ ps
  PID TTY          TIME CMD
 5198 pts/1    00:00:00 bash
10129 pts/1    00:00:00 ps
```

The result in this example lists two processes: process 5198 and process 10129, which are `bash` and `ps` respectively. As we can see, by default `ps` doesn't show us very much, just the processes associated with the current terminal session. To see more, we need to add some options, but before we do that, let's look at the other fields produced by `ps`. `TTY` is short for *teletype* and refers to the *controlling terminal* for the process. Unix is showing its age here. The `TIME` field is the amount of CPU time consumed by the process. As we can see, neither process makes the computer work very hard.

If we add an option, we can get a bigger picture of what the system is doing:

```
[me@linuxbox ~]$ ps x
  PID TTY      STAT   TIME COMMAND
 2799 ?        Ssl    0:00 /usr/libexec/bonobo-activation-server -ac
 2820 ?        Sl     0:01 /usr/libexec/evolution-data-server-1.10 --
15647 ?        Ss     0:00 /bin/sh /usr/bin/startkde
15751 ?        Ss     0:00 /usr/bin/ssh-agent /usr/bin/dbus-launch --
15754 ?        S      0:00 /usr/bin/dbus-launch --exit-with-session
15755 ?        Ss     0:01 /bin/dbus-daemon --fork --print-pid 4 -pr
15774 ?        Ss     0:02 /usr/bin/gpg-agent -s -daemon
15793 ?        S      0:00 start_kdeinit --new-startup +kcminit_start
15794 ?        Ss     0:00 kdeinit Running...
15797 ?        S      0:00 dcopserver -nosid

and many more...
```

Adding the x option (note that there is no leading dash) tells ps to show all of our processes regardless of what terminal (if any) they are controlled by. The presence of a ? in the TTY column indicates no controlling terminal. Using this option, we see a list of every process that we own.

Since the system is running a lot of processes, ps produces a long list. It is often helpful to pipe the output from ps into less for easier viewing. Some option combinations also produce long lines of output, so maximizing the terminal emulator window may be a good idea, too.

A new column titled STAT has been added to the output. STAT is short for *state* and reveals the current status of the process, as shown in Table 10-1.

Table 10-1: Process States

State	Meaning
R	Running. The process is running or ready to run.
S	Sleeping. The process is not running; rather, it is waiting for an event, such as a keystroke or network packet.
D	Uninterruptible sleep. Process is waiting for I/O such as a disk drive.
T	Stopped. Process has been instructed to stop (more on this later).
Z	A defunct or "zombie" process. This is a child process that has terminated but has not been cleaned up by its parent.
<	A high-priority process. It's possible to grant more importance to a process, giving it more time on the CPU. This property of a process is called *niceness*. A process with high priority is said to be less nice because it's taking more of the CPU's time, which leaves less for everybody else.
N	A low-priority process. A process with low priority (a nice process) will get processor time only after other processes with higher priority have been serviced.

The process state may be followed by other characters. These indicate various exotic process characteristics. See the ps man page for more detail.

Another popular set of options is aux (without a leading dash). This gives us even more information:

```
[me@linuxbox ~]$ ps aux
USER     PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root       1  0.0  0.0   2136   644 ?        Ss   Mar05   0:31 init
root       2  0.0  0.0      0     0 ?        S<   Mar05   0:00 [kt]
root       3  0.0  0.0      0     0 ?        S<   Mar05   0:00 [mi]
root       4  0.0  0.0      0     0 ?        S<   Mar05   0:00 [ks]
root       5  0.0  0.0      0     0 ?        S<   Mar05   0:06 [wa]
root       6  0.0  0.0      0     0 ?        S<   Mar05   0:36 [ev]
root       7  0.0  0.0      0     0 ?        S<   Mar05   0:00 [kh]

and many more...
```

This set of options displays the processes belonging to every user. Using the options without the leading dash invokes the command with "BSD-style" behavior. The Linux version of ps can emulate the behavior of the ps program found in several Unix implementations. With these options, we get the additional columns shown in Table 10-2.

Table 10-2: BSD-Style ps Column Headers

Header	Meaning
USER	User ID. This is the owner of the process.
%CPU	CPU usage as a percent.
%MEM	Memory usage as a percent.
VSZ	Virtual memory size.
RSS	Resident Set Size. The amount of physical memory (RAM) the process is using in kilobytes.
START	Time when the process started. For values over 24 hours, a date is used.

Viewing Processes Dynamically with top

While the ps command can reveal a lot about what the machine is doing, it provides only a snapshot of the machine's state at the moment the ps command is executed. To see a more dynamic view of the machine's activity, we use the top command:

```
[me@linuxbox ~]$ top
```

The top program displays a continuously updating (by default, every 3 seconds) display of the system processes listed in order of process activity.

Its name comes from the fact that the `top` program is used to see the "top" processes on the system. The `top` display consists of two parts: a system summary at the top of the display, followed by a table of processes sorted by CPU activity:

```
top - 14:59:20 up  6:30,  2 users,  load average: 0.07, 0.02, 0.00
Tasks: 109 total,   1 running, 106 sleeping,   0 stopped,   2 zombie
Cpu(s):  0.7%us,  1.0%sy,  0.0%ni, 98.3%id,  0.0%wa,  0.0%hi,  0.0%si
Mem:    319496k total,   314860k used,     4636k free,    19392k buff
Swap:   875500k total,   149128k used,   726372k free,   114676k cach

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 6244 me        39  19 31752 3124 2188 S  6.3  1.0  16:24.42 trackerd
11071 me        20   0  2304 1092  840 R  1.3  0.3   0:00.14 top
 6180 me        20   0  2700 1100  772 S  0.7  0.3   0:03.66 dbus-dae
 6321 me        20   0 20944 7248 6560 S  0.7  2.3   2:51.38 multiloa
 4955 root      20   0  104m 9668 5776 S  0.3  3.0   2:19.39 Xorg
    1 root      20   0  2976  528  476 S  0.0  0.2   0:03.14 init
    2 root      15  -5     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 migratio
    4 root      15  -5     0    0    0 S  0.0  0.0   0:00.72 ksoftirq
    5 root      RT  -5     0    0    0 S  0.0  0.0   0:00.04 watchdog
    6 root      15  -5     0    0    0 S  0.0  0.0   0:00.42 events/0
    7 root      15  -5     0    0    0 S  0.0  0.0   0:00.06 khelper
   41 root      15  -5     0    0    0 S  0.0  0.0   0:01.08 kblockd/
   67 root      15  -5     0    0    0 S  0.0  0.0   0:00.00 kseriod
  114 root      20   0     0    0    0 S  0.0  0.0   0:01.62 pdflush
  116 root      15  -5     0    0    0 S  0.0  0.0   0:02.44 kswapd0
```

The system summary contains a lot of good stuff; see Table 10-3 for a rundown.

Table 10-3: top Information Fields

Row	Field	Meaning
1	`top`	Name of the program.
	`14:59:20`	Current time of day.
	`up 6:30`	This is called *uptime*. It is the amount of time since the machine was last booted. In this example, the system has been up for 6½ hours.
	`2 users`	Two users are logged in.
	`load average:`	*Load average* refers to the number of processes that are waiting to run; that is, the number of processes that are in a runnable state and are sharing the CPU. Three values are shown, each for a different period of time. The first is the average for the last 60 seconds, the next the previous 5 minutes, and finally the previous 15 minutes. Values under 1.0 indicate that the machine is not busy.

(*continued*)

Table 10-3 (*continued*)

Row	Field	Meaning
2	`Tasks:`	This summarizes the number of processes and their various process states.
	`0.7%us`	0.7% of the CPU is being used for *user processes*. This means processes outside of the kernel itself.
	`1.0%sy`	1.0% of the CPU is being used for *system* (kernel) processes.
	`0.0%ni`	0.0% of the CPU is being used by nice (low-priority) processes.
	`98.3%id`	98.3% of the CPU is idle.
	`0.0%wa`	0.0% of the CPU is waiting for I/O.
4	`Mem:`	Shows how physical RAM is being used.
5	`Swap:`	Shows how swap space (virtual memory) is being used.

The `top` program accepts a number of keyboard commands. The two most interesting are `h`, which displays the program's help screen, and `q`, which quits `top`.

Both major desktop environments provide graphical applications that display information similar to `top` (in much the same way that Task Manager in Windows does), but I find that `top` is better than the graphical versions because it is faster and consumes far fewer system resources. After all, our system monitor program shouldn't add to the system slowdown that we are trying to track.

Controlling Processes

Now that we can see and monitor processes, let's gain some control over them. For our experiments, we're going to use a little program called `xlogo` as our guinea pig. The `xlogo` program is a sample program supplied with the X Window System (the underlying engine that makes the graphics on our display go), which simply displays a resizable window containing the X logo. First, we'll get to know our test subject:

```
[me@linuxbox ~]$ xlogo
```

After we enter the command, a small window containing the logo should appear somewhere on the screen. On some systems, `xlogo` may print a warning message, but it may be safely ignored.

Note: *If your system does not include the* `xlogo` *program, try using* `gedit` *or* `kwrite` *instead.*

We can verify that `xlogo` is running by resizing its window. If the logo is redrawn in the new size, the program is running.

Notice how our shell prompt has not returned? This is because the shell is waiting for the program to finish, just like all the other programs we have used so far. If we close the `xlogo` window, the prompt returns.

Interrupting a Process

Let's observe what happens when we run `xlogo` again. First, enter the `xlogo` command and verify that the program is running. Next, return to the terminal window and press CTRL-C.

```
[me@linuxbox ~]$ xlogo
[me@linuxbox ~]$
```

In a terminal, pressing CTRL-C *interrupts* a program. This means that we politely asked the program to terminate. After we pressed CTRL-C, the `xlogo` window closed and the shell prompt returned.

Many (but not all) command-line programs can be interrupted by using this technique.

Putting a Process in the Background

Let's say we wanted to get the shell prompt back without terminating the `xlogo` program. We'll do this by placing the program in the *background*. Think of the terminal as having a *foreground* (with stuff visible on the surface, like the shell prompt) and a background (with hidden stuff below the surface). To launch a program so that it is immediately placed in the background, we follow the command with an ampersand character (`&`):

```
[me@linuxbox ~]$ xlogo &
[1] 28236
[me@linuxbox ~]$
```

After the command was entered, the `xlogo` window appeared and the shell prompt returned, but some funny numbers were printed too. This message is part of a shell feature called *job control*. With this message, the shell is telling us that we have started job number 1 (`[1]`) and that it has PID 28236. If we run `ps`, we can see our process:

```
[me@linuxbox ~]$ ps
  PID TTY          TIME CMD
10603 pts/1    00:00:00 bash
28236 pts/1    00:00:00 xlogo
28239 pts/1    00:00:00 ps
```

The shell's job control facility also gives us a way to list the jobs that have been launched from our terminal. Using the `jobs` command, we can see the following list:

```
[me@linuxbox ~]$ jobs
[1]+  Running                 xlogo &
```

The results show that we have one job, numbered 1, that it is running, and that the command was `xlogo &`.

Returning a Process to the Foreground

A process in the background is immune from keyboard input, including any attempt to interrupt it with a CTRL-C. To return a process to the foreground, use the `fg` command, as in this example:

```
[me@linuxbox ~]$ jobs
[1]+  Running                 xlogo &
[me@linuxbox ~]$ fg %1
xlogo
```

The command `fg` followed by a percent sign and the job number (called a *jobspec*) does the trick. If we have only one background job, the jobspec is optional. To terminate `xlogo`, type CTRL-C.

Stopping (Pausing) a Process

Sometimes we'll want to stop a process without terminating it. This is often done to allow a foreground process to be moved to the background. To stop a foreground process, type CTRL-Z. Let's try it. At the command prompt, type `xlogo`, press the ENTER key, and then type CTRL-Z:

```
[me@linuxbox ~]$ xlogo
[1]+  Stopped                 xlogo
[me@linuxbox ~]$
```

After stopping `xlogo`, we can verify that the program has stopped by attempting to resize the `xlogo` window. We will see that it appears quite dead. We can either restore the program to the foreground, using the `fg` command, or move the program to the background with the `bg` command:

```
[me@linuxbox ~]$ bg %1
[1]+ xlogo &
[me@linuxbox ~]$
```

As with the `fg` command, the jobspec is optional if there is only one job.

Moving a process from the foreground to the background is handy if we launch a graphical program from the command but forget to place it in the background by appending the trailing `&`.

Why would you want to launch a graphical program from the command line? There are two reasons. First, the program you wish to run might not be listed on the window manager's menus (such as `xlogo`).

Second, by launching a program from the command line, you might be able to see error messages that would be invisible if the program were launched graphically. Sometimes, a program will fail to start up when launched from the graphical menu. By launching it from the command line instead, we may see an error message that will reveal the problem. Also, some graphical programs have many interesting and useful command-line options.

Signals

The `kill` command is used to "kill" (terminate) processes. This allows us to end the execution of a program that is behaving badly or otherwise refuses to terminate on its own. Here's an example:

```
[me@linuxbox ~]$ xlogo &
[1] 28401
[me@linuxbox ~]$ kill 28401
[1]+  Terminated              xlogo
```

We first launch `xlogo` in the background. The shell prints the jobspec and the PID of the background process. Next, we use the `kill` command and specify the PID of the process we want to terminate. We could also have specified the process using a jobspec (for example, `%1`) instead of a PID.

While this is all very straightforward, there is more to it. The `kill` command doesn't exactly "kill" processes; rather it sends them *signals*. Signals are one of several ways that the operating system communicates with programs. We have already seen signals in action with the use of CTRL-C and CTRL-Z. When the terminal receives one of these keystrokes, it sends a signal to the program in the foreground. In the case of CTRL-C, a signal called `INT` (Interrupt) is sent; with CTRL-Z, a signal called `TSTP` (Terminal Stop) is sent. Programs, in turn, "listen" for signals and may act upon them as they are received. The fact that a program can listen and act upon signals allows it to do things like save work in progress when it is sent a termination signal.

Sending Signals to Processes with kill

The most common syntax for the `kill` command looks like this:

```
kill [-signal] PID...
```

If no signal is specified on the command line, then the `TERM` (Terminate) signal is sent by default. The `kill` command is most often used to send the signals shown in Table 10-4.

Table 10-4: Common Signals

Number	Name	Meaning
1	HUP	Hang up. This is a vestige of the good old days when terminals were attached to remote computers with phone lines and modems. The signal is used to indicate to programs that the controlling terminal has "hung up." The effect of this signal can be demonstrated by closing a terminal session. The foreground program running on the terminal will be sent the signal and will terminate. This signal is also used by many daemon programs to cause a reinitialization. This means that when a daemon is sent this signal, it will restart and reread its configuration file. The Apache web server is an example of a daemon that uses the HUP signal in this way.
2	INT	Interrupt. Performs the same function as the CTRL-C key sent from the terminal. It will usually terminate a program.
9	KILL	Kill. This signal is special. Whereas programs may choose to handle signals sent to them in different ways, including by ignoring them altogether, the KILL signal is never actually sent to the target program. Rather, the kernel immediately terminates the process. When a process is terminated in this manner, it is given no opportunity to "clean up" after itself or save its work. For this reason, the KILL signal should be used only as a last resort when other termination signals fail.
15	TERM	Terminate. This is the default signal sent by the `kill` command. If a program is still "alive" enough to receive signals, it will terminate.
18	CONT	Continue. This will restore a process after a STOP signal.
19	STOP	Stop. This signal causes a process to pause without terminating. Like the KILL signal, it is not sent to the target process, and thus it cannot be ignored.

Let's try out the kill command:

```
[me@linuxbox ~]$ xlogo &
[1] 13546
[me@linuxbox ~]$ kill -1 13546
[1]+  Hangup                  xlogo
```

In this example, we start the xlogo program in the background and then send it a HUP signal with kill. The xlogo program terminates, and the shell indicates that the background process has received a hangup signal. You may need to press the ENTER key a couple of times before you see the message. Note that signals may be specified either by number or by name, including the name prefixed with the letters *SIG*:

```
[me@linuxbox ~]$ xlogo &
[1] 13601
[me@linuxbox ~]$ kill -INT 13601
[1]+  Interrupt               xlogo
[me@linuxbox ~]$ xlogo &
[1] 13608
[me@linuxbox ~]$ kill -SIGINT 13608
[1]+  Interrupt               xlogo
```

Repeat the example above and try out the other signals. Remember, you can also use jobspecs in place of PIDs.

Processes, like files, have owners, and you must be the owner of a process (or the superuser) in order to send it signals with kill.

In addition to the signals listed in Table 10-4, which are most often used with kill, other signals are frequently used by the system. Table 10-5 lists the other common signals.

Table 10-5: Other Common Signals

Number	Name	Meaning
3	QUIT	Quit.
11	SEGV	Segmentation violation. This signal is sent if a program makes illegal use of memory; that is, it tried to write somewhere it was not allowed to.
20	TSTP	Terminal stop. This is the signal sent by the terminal when CTRL-Z is pressed. Unlike the STOP signal, the TSTP signal is received by the program but the program may choose to ignore it.
28	WINCH	Window change. This is a signal sent by the system when a window changes size. Some programs, like top and less, will respond to this signal by redrawing themselves to fit the new window dimensions.

For the curious, a complete list of signals can be seen with the following command:

```
[me@linuxbox ~]$ kill -l
```

Sending Signals to Multiple Processes with killall

It's also possible to send signals to multiple processes matching a specified program or username by using the `killall` command. Here is the syntax:

```
killall [-u user] [-signal] name...
```

To demonstrate, we will start a couple of instances of the `xlogo` program and then terminate them:

```
[me@linuxbox ~]$ xlogo &
[1] 18801
[me@linuxbox ~]$ xlogo &
[2] 18802
[me@linuxbox ~]$ killall xlogo
[1]-  Terminated              xlogo
[2]+  Terminated              xlogo
```

Remember, as with `kill`, you must have superuser privileges to send signals to processes that do not belong to you.

More Process-Related Commands

Since monitoring processes is an important system administration task, there are a lot of commands for it. Table 10-6 lists some to play with.

Table 10-6: Other Process-Related Commands

Command	Description
`pstree`	Outputs a process list arranged in a tree-like pattern showing the parent/child relationships between processes.
`vmstat`	Outputs a snapshot of system resource usage including memory, swap, and disk I/O. To see a continuous display, follow the command with a time delay (in seconds) for updates (e.g., `vmstat 5`). Terminate the output with CTRL-C.
`xload`	A graphical program that draws a graph showing system load over time.
`tload`	Similar to the `xload` program, but draws the graph in the terminal. Terminate the output with CTRL-C.