

CHAPTER 11

Command-Line Master Class, Part 1

Some Linux users like to focus on the graphical environments that are available; they rush to tell new users that the command line isn't vital when using Linux. Although there are some amazing *graphical user interface (GUI)* desktops, and this statement is mostly true, avoiding the command line limits your options and makes some tasks more difficult. The command-line interface offers the greatest power and flexibility, and those who actively avoid learning how to use it are also actively limiting their abilities and options. You learned the basics in Chapter 10, "Command-Line Beginner's Class." In this chapter, we dig in deeper.

It is with some trepidation that this two-chapter set retains its classic title "Command-Line Master Class." Entire books have been published covering the depth and breadth of the command line. To believe that two short chapters make any reader a true master is foolish. Our greatest hope is to give enough information to enable any reader to perform all basic and vital tasks from the command line while inspiring readers to go on a quest to discover all the beauty and grandeur that we do not have space to cover here. Please keep this in mind as you continue.

In his book *The Art of Unix Programming*, Eric Raymond wrote a short story that perfectly illustrates the power of the command line versus the GUI. It's reprinted here with permission, for your reading pleasure:

One evening, Master Foo and Nubi attended a gathering of programmers who had met to learn from each other. One of the programmers asked Nubi to what school he and his master belonged. Upon being told they were followers of the Great Way of Unix, the programmer grew scornful.

IN THIS CHAPTER

- Why Use the Command Line?
- Using Basic Commands
- References

“The command-line tools of Unix are crude and backward,” he scoffed.

“Modern, properly designed operating systems do everything through a graphical user interface.”

Master Foo said nothing, but pointed at the moon. A nearby dog began to bark at the master’s hand.

“I don’t understand you!” said the programmer.

Master Foo remained silent, and pointed at an image of the Buddha. Then he pointed at a window. “What are you trying to tell me?” asked the programmer.

Master Foo pointed at the programmer’s head. Then he pointed at a rock.

“Why can’t you make yourself clear?” demanded the programmer.

Master Foo frowned thoughtfully, tapped the programmer twice on the nose, and dropped him in a nearby trash can.

As the programmer was attempting to extricate himself from the garbage, the dog wandered over and piddled on him.

At that moment, the programmer achieved enlightenment.

Whimsical as the story is, it does illustrate that there are some things that the GUI just does not do well. Enter the command line: It is a powerful and flexible operating environment and—if you practice—can actually be quite fun, too!

In this chapter, you learn more commands to help you master the command line so that you can perform common tasks through it.

Why Use the Command Line?

Moving from the GUI to the command line is a conscious choice for most people, although it is increasingly rare that it is an absolute choice accompanied by a complete abandoning of GUIs.

Reasons for using the command line include the following:

- ▶ You want to chain together two or more commands.
- ▶ You want to use a command or parameter available only on the shell.
- ▶ You are working on a text-only system.
- ▶ You have used it for a long time and feel comfortable there.
- ▶ You want to automate a task.

Chaining together two or more commands, or *piping*, is what gives the shell its real power. Hundreds of commands are available, and by combining them in different ways, you get tons of new options. Some of the shell commands are available through the GUI, but

these commands usually have only a small subset of their parameters available, which limits what you can do with them.

Working from a text-only system is useful both for working locally with a broken GUI and for connecting to a remote, text-only system. If your Linux server is experiencing problems, the last thing you want to do is load it down with a GUI connection; working in text mode is faster and more efficient.

Many people use the shell simply because it is familiar to them. Some people even use the shell to start GUI applications just because it saves them taking their hands off the keyboard for a moment. This is not a bad thing; it provides fluency and ease with the system and is a perfectly valid way of working. Working from the command line is faster. The mouse is slow, and taking your fingers away from the keyboard makes your work even slower. Anyone looking to achieve the Zen-like power user state hinted at by Eric Raymond will understand this after making the effort to learn.

Knowing how to work in the shell is also essential if you want to automate tasks on your system without use of the GUI. Whether you want to add a periodic task to `cron` or update a configuration management system, you need to know what text commands to give to run programs from the command line.

Using Basic Commands

It is impossible to know how many commands the average command-line citizen uses, but if we had to guess, we would place it at about 25. Some of these were introduced in Chapter 10, “Command-Line Beginner’s Class,” but are covered here in greater depth. Others may be new to you. Still others are mentioned in this list only to provide ideas for further study. Here are some commands that every command-line user will want to learn:

- ▶ **cat**—Prints the contents of a file
- ▶ **cd**—Changes directories
- ▶ **chmod**—Changes file access permissions
- ▶ **cp**—Copies files
- ▶ **du**—Prints disk usage
- ▶ **emacs**—Edits text files
- ▶ **find**—Finds files by searching
- ▶ **grep**—Searches for a string in input or files
- ▶ **head**—Prints the first lines of a file
- ▶ **less**—Displays files or input interactively
- ▶ **ln**—Creates links between files
- ▶ **locate**—Finds files from an index
- ▶ **ls**—Lists files in the current directory

- ▶ **make**—Compiles and installs programs
- ▶ **man**—Displays manual pages for reading
- ▶ **mkdir**—Makes directories
- ▶ **mv**—Moves files
- ▶ **nano**—Edits text files
- ▶ **rm**—Deletes files and directories
- ▶ **sort**—Takes a text file as input and outputs the contents of the file in the order you specify
- ▶ **ssh**—Connects to other machines using a secure shell connection
- ▶ **tail**—Prints the last lines of a file
- ▶ **vim**—Edits text files
- ▶ **which**—Prints the location of a command

Many other commands are also used fairly often—`cut`, `diff`, `gzip`, `history`, `ping`, `su`, `tar`, `uptime`, `who`, and so on—but if you can understand the ones listed here, you have sufficient skill to concoct your own command combinations.

Note that we say *understand* the commands—not know all their possible parameters and usages. This is because several of the commands, although commonly used, are used only in any complex manner by people with specific needs. `make` is a good example of this: Unless you plan to become a programmer, you need not worry about this command beyond just using `make` and `make install` now and then. If you want to learn more, see Chapter 38, “Using Programming Tools.”

Similarly, `emacs`, `nano`, and `vim` are text editors that have text-based interfaces all their own and are covered later in this chapter. `ssh` is covered in detail in Chapter 19, “Remote Access with SSH and VNC.”

The rest of this list is composed of commands that each have many parameters you can use to customize what the commands actually do. Again, many of the parameters are esoteric and rarely used, and the few times in your Linux life that you need them, you can just read the manual page.

We go over these commands one by one, explaining the most common ways to use them.

Printing the Contents of a File with `cat`

Many of Ubuntu’s shell commands manipulate text strings, so if you want to be able to feed them the contents of files, you need to be able to output those files as text. Enter the `cat` command, which prints the contents of any files you pass to it.

Its most basic use is like this:

```
matthew@seymour:~$ cat myfile.txt
```

This prints the contents of `myfile.txt`. For this usage, two extra parameters are often used: `-n` numbers the lines in the output, and `-s` (“squeeze”) prints a maximum of one blank line at a time. That is, if your file has 1 line of text, 10 blank lines, 1 line of text, 10 blank lines, and so on, `-s` shows the first line of text, a single blank line, the next line of text, a single blank line, and so forth. When you combine `-s` and `-n`, `cat` numbers only the lines that are printed—the 10 blank lines shown as 1 will count as 1 line for numbering.

Assuming that you are the user `matthew`, the following command prints information about your CPU, stripping out multiple blank lines and numbering the output:

```
matthew@seymour:~$ cat -sn /proc/cpuinfo
```

You can also use `cat` to print the contents of several files at once, like this:

```
matthew@seymour:~$ cat -s myfile.txt myotherfile.txt
```

In this command, `cat` merges `myfile.txt` and `myotherfile.txt` on the output and strips out multiple blank lines. The important thing is that `cat` does not distinguish between the files in the output; there are no filenames printed and no extra breaks between the two. This allows you to treat the 2 files as 1 or, by adding more files to the command line, to treat 20 files as 1.

Changing Directories with `cd`

Changing directories is surely something that has no options, right? Well, `cd` is actually more flexible than most people realize. Unlike most of the other commands here, `cd` is not a command in itself—it is built in to `bash` (or whichever shell interpreter you are using), but it is still used like a command.

The most basic usage of `cd` is this:

```
matthew@seymour:~$ cd somedir
```

This looks in the current directory for the `somedir` subdirectory and then moves you into it. You can also specify an exact location for a directory, like this:

```
matthew@seymour:~$ cd /home/matthew/stuff/somedir
```

The first part of `cd`’s magic lies in the characters (`-` and `~`, a dash and a tilde). The first means “switch to my previous directory,” and the second means “switch to my home directory.” The following conversation with `cd` shows this in action:

```
matthew@seymour:~$ cd /usr/local
matthew@seymour/usr/local$ cd bin
matthew@seymour/usr/local/bin$ cd -
/usr/local
matthew@seymour/usr/local$ cd ~
matthew@seymour:~$
```

In the first line, you change to `/usr/local` and get no output from the command. In the second line, you change to `bin`, which is a subdirectory of `/usr/local`. Next, `cd -` is used to change back to the previous directory. This time `bash` prints the name of the previous directory so you know where you are. Finally, `cd ~` is used to change back to your `/home/matthew` directory, although if you want to save an extra few keystrokes, you can just type `cd` by itself, which is equivalent to `cd ~`.

The second part of `cd`'s magic is its capability to look for directories in predefined locations. When you specify an absolute path to a directory (that is, one starting with a `/`), `cd` always switches to that exact location. However, if you specify a relative subdirectory—for example, `cd subdir`—you can tell `cd` where you would like that to be relative to. This is accomplished with the `CDPATH` environment variable. If this variable is not set, `cd` always uses the current directory as the base; however, you can set it to any number of other directories.

This next example shows a test of this. It starts in `/home/matthew/empty`, an empty directory, and the lines are numbered for later reference:

```
1 matthew@seymour:~/empty$ pwd
2 /home/matthew/empty
3 matthew@seymour:~/empty$ ls
4 matthew@seymour:~/empty$ mkdir local
5 matthew@seymour:~/empty$ ls
6 local
7 matthew@seymour:~/empty$ cd local
8 matthew@seymour:~/empty/local$ cd ..
9 matthew@seymour:~/empty$ export CDPATH=/usr
10 matthew@seymour:~/empty$ cd local
11 /usr/local
12 matthew@seymour:/usr/local$ cd -
13 /home/matthew/empty
14 matthew@seymour:~/empty$ export CDPATH=./usr
15 matthew@seymour:~/empty$ cd local
16 /home/matthew/empty/local
17 matthew@seymour:~/empty/local$
```

Lines 1–3 show that you are in `/home/matthew/empty` and that it is indeed empty; `ls` had no output. Lines 4–6 show the `local` subdirectory being made, so that `/home/matthew/empty/local` exists. Lines 7 and 8 show that you can `cd` into `/home/matthew/empty/local` and back out again.

In line 9, `CDPATH` is set to `/usr`. This was chosen because Ubuntu has the directory `/usr/local`, which means your current directory (`/home/matthew/empty`) and your `CDPATH` directory (`/usr`) both have local subdirectories. In line 10, while in the `/home/matthew/empty` directory, you use `cd local`. This time, `bash` switches you to `/usr/local` and even prints the new directory to ensure that you know what it has done.

Lines 12 and 13 move you back to the previous directory, `/home/matthew/empty`. In line 14, `CDPATH` is set to be `./usr`. The `.` is the directory separator, so this means `bash` should look first in the current directory, `.`, and then in the `/usr` directory. In line 15 `cd`

`local` is issued again, this time moving to `/home/matthew/empty/local`. Note that `bash` has still printed the new directory; it does that whenever it looks up a directory in `CDPATH`.

Changing File Access Permissions with `chmod`

Your use of `chmod` can be greatly extended through one simple parameter: `-c`. This instructs `chmod` to print a list of all the changes it made as part of its operation, which means you can capture the output and use it for other purposes. Consider this example:

```
matthew@seymour:~$ chmod -c 600 *
mode of '1.txt' changed to 0600 (rw-----)
mode of '2.txt' changed to 0600 (rw-----)
mode of '3.txt' changed to 0600 (rw-----)
matthew@seymour:~$ chmod -c 600 *
matthew@seymour:~$
```

Here the `chmod` command is issued with `-c`, and you can see it has output the result of the operation: Three files were changed to `rw-----` (read and write by user only). However, when the command is issued again, no output is returned. This is because `-c` prints only the changes it made. Files that already match the permissions you are setting are left unchanged and therefore are not printed.

Two other parameters of interest are `--reference` and `-R`. The first allows you to specify a file to use as a template for permissions rather than specifying permissions yourself. For example, if you want all files in the current directory to have the same permissions as the file `/home/matthew/myfile.txt`, you use this:

```
chmod --reference /home/matthew/myfile.txt *
```

You can use `-R` to enable recursive operation, which means you can use it to `chmod` a directory, and it will change the permissions of that directory as well as all files and subdirectories under that directory. You could use `chmod -R 600 /home` to change every file and directory under `/home` to become read/write to its owner(s).

Copying Files with `cp`

Like `mv`, which is covered later in this chapter, `cp` is a command that is easily used and mastered. However, two marvelous parameters rarely see much use, despite their power—which is a shame. These are `--parents` and `-u`. The first copies the full path of the file into the new directory; the second copies only if the source file is newer than the destination.

Using `--parents` requires a little explanation, so here is an example. Say that you have a file, `/home/matthew/desktop/documents/work/notes.txt`, and want to copy it to your `/home/matthew/backup` folder. You could do a normal `cp`, but that would give you `/home/matthew/backup/notes.txt`, so how would you know where that came from later? If you use `--parents`, the file is copied to `/home/matthew/backup/desktop/documents/work/notes.txt`.

The `-u` parameter is perfect for synchronizing two directories because it allows you to run a command like `cp -Ru myfiles myotherfiles` and have `cp` recopy only files that have changed. The `-R` parameter means *recursive* and enables you to copy directory contents.

Printing Disk Usage with `du`

The `du` command prints the size of each file and directory that is inside the current directory. Its most basic usage is as easy as it gets:

```
matthew@seymour:~$ du
```

This outputs a long list of directories and how much space their files take up. You can modify it with the `-a` parameter, which instructs `du` to print the sizes of individual files as well as directories. Another useful parameter is `-h`, which makes `du` use human-readable sizes like 18M (18MB) rather than 17532 (the same number in bytes, unrounded). The final useful basic option is `-c`, which prints the total sizes of files.

So, by using `du` you can get a printout of the size of each file in your `/home` directory, in human-readable format, and with a summary at the end, like this:

```
matthew@seymour:~$ du -ahc /home/matthew
```

Two advanced parameters deal with filenames you want excluded from your count. The first is `--exclude`, which enables you to specify a pattern that should be used to exclude files. This pattern is a standard shell file-matching pattern as opposed to a regular expression, which means you can use `?` to match a single character or `*` to match zero or many characters. You can specify multiple `--exclude` parameters to exclude several patterns. Here is an example:

```
matthew@seymour:~$ du --exclude="*.xml" --exclude="*.xsl"
```

However, typing numerous `--exclude` parameters repeatedly is a waste of time, and you can use `-x` to specify a file that has the list of patterns you want excluded. The file should look like this:

```
*.xml
*.xsl
```

That is, each pattern you want excluded should be on a line by itself. If that file were called `xml_exclude.txt`, you could use it in place of the previous example, like this:

```
matthew@seymour:~$ du -X xml_exclude.txt
```

You can make your exclusion file as long as you need, or you can just specify multiple `-x` parameters.

TIP

If you run `du` in a directory where several files are hard-linked to the same inode, you count the size of the file only once. If you want to count each hard link separately for some reason, use the `-l` parameter (lowercase *L*).

Using `echo`

You can do many things with `echo`, especially with redirection (see Chapter 12, “Command-Line Master Class, Part 2,” for more about redirecting output). In its simplest use, `echo` sends whatever you tell it to send to standard output. If you want to repeat text on

the screen (which is useful in a shell script, for example), just enter the text string in single quotation marks (`'`), and the output appears below it, like this:

```
matthew@seymour:~$ echo 'I have the power!'
I have the power!
```

If you want to know the value of a system variable, such as `TERM`, enter the variable name to output the value, like this:

```
matthew@seymour:~$ echo $TERM
xterm
```

You can redirect the output of `echo` into a text file, as is done here to add a new directory to `PATHS`:

```
matthew@seymour:~$ echo 'export PATH=$PATH:/usr/local/bin' >> ~/.bashrc
```

You can change or set a kernel setting (1 = on, 0 = off) in `/proc`, like this:

```
matthew@seymour:~$ sudo sh -c 'echo "1" > /proc/sys/location/of/setting'
```

Note that you can read the setting of a kernel value in `/proc` by using `cat`.

Finding Files by Searching with `find`

The `find` command is one of the darkest and least understood areas of Linux, but it is also one of the most powerful. Admittedly, the `find` command does not help itself by using X-style parameters. The UNIX standard is `-c`, `-s`, and so on, whereas the GNU standard is `--dosomething`, `--mooby`, and so forth. X-style parameters merge the two by having words preceded by only one dash.

However, the biggest problem with `find` is that it has more options than most people can remember; it truly is capable of doing most things you could want. The most basic usage is as follows:

```
matthew@seymour:~$ find -name "*.txt"
```

This option searches the current directory and all subdirectories for files that end in `.txt`. The previous search finds files ending in `.txt` but not `.TXT`, `.Txt`, or other case variations. To search without case-sensitivity, use `-iname` rather than `-name`. You can optionally specify where the search should start before the `-name` parameter, like this:

```
matthew@seymour:~$ find /home -name "*.txt"
```

Another useful test is `-size`, which you use to specify how big the files should be in order to match. You can specify the size in kilobytes and optionally use `+` or `-` to specify greater than or less than. Consider these examples:

```
matthew@seymour:~$ find /home -name "*.txt" -size 100k
matthew@seymour:~$ find /home -name "*.txt" -size +100k
matthew@seymour:~$ find /home -name "*.txt" -size -100k
```

The first example brings up files of exactly 100KB, the second only files larger than 100KB, and the last only files under 100KB.

Moving on, the `-user` option enables you to specify the user who owns the files you are looking for. So, to search for all files in `/home` that end with `.txt`, that are under 100KB, and that are owned by user `matthew`, you use this:

```
matthew@seymour:~$ find /home -name "*.txt" -size -100k -user matthew
```

You can flip any of the conditions by specifying `-not` before them. For example, you can add `-not` before `-user matthew` to find matching files owned by everyone except `matthew`:

```
matthew@seymour:~$ find /home -name "*.txt" -size -100k -not -user matthew
```

You can add as many `-not` parameters as you need, and you can even use `-not -not` to cancel out the first `-not`. (Yes, that is pointless.) Keep in mind, though, that `-not -size -100k` is essentially equivalent to `-size +100k`, with the exception that the former will match files of exactly 100KB, whereas the latter will not.

You can use `-perm` to specify which permissions a file should have in order to match it. This is tricky, so read carefully. The permissions are specified in the same way as with the `chmod` command: `u` for user, `g` for group, `o` for others, `r` for read, `w` for write, and `x` for execute. However, before you give the permissions, you need to specify a plus, a minus, or a blank space. If you specify neither a plus nor a minus, the files must exactly match the mode you give. If you specify `-`, the files must match all the modes you specify. If you specify `+`, the files must match any the modes you specify. Confused yet?

The confusion can be cleared up with some examples. This following command finds all files that have permission `o=r` (readable for other users). Notice that if you remove the `-name` parameter, it is equivalent to `*` because all filenames are matched:

```
matthew@seymour:~$ find /home -perm o=r
```

Any files that have `o=r` set are returned from this query. Those files also might have `u=rw` and other permissions, but as long as they have `o=r`, they will match. This next query matches all files that have `o=rw` set:

```
matthew@seymour:~$ find /home -perm o=rw
```

However, this query does not match files that are `o=r` or `o=w`. To be matched, a file must be readable and writable by other users. If you want to match readable or writable (or both), you need to use `+`, like this:

```
matthew@seymour:~$ find /home -perm +o=rw
```

Similarly, the next query matches only files that are readable by the user, the group, and others:

```
matthew@seymour:~$ find /home -perm -ugo=r
```

On the other hand, the following query matches files as long as they are readable by the user, or by the group, or by others, or by any combination of the three:

```
matthew@seymour:~$ find /home -perm +ugo=r
```

If you use neither `+` nor `-`, you are specifying the exact permissions to search for. For example, the following query searches for files that are readable by the user, the group, and others but not writable or executable by anyone:

```
matthew@seymour:~$ find /home -perm ugo=r
```

You can be as specific as you need to be with the permissions. For example, this query finds all files that are readable for the user, the group, and others and writable by the user:

```
matthew@seymour:~$ find /home -perm ugo=r,u=w
```

To find files that are not readable by others, use the `-not` condition, like this:

```
matthew@seymour:~$ find /home -not -perm +o=r
```

Now, on to the most advanced aspect of the `find` command: the `-exec` parameter. This parameter enables you to execute an external program each time a match is made, passing in the name of the matched file wherever you want it. This has very specific syntax: Your command and its parameters should follow immediately after `-exec`, terminated by `\;`. You can insert the filename match at any point by using `{}` (an opening and a closing brace side by side).

So, you can match all text files on the entire system (that is, searching recursively from `/` rather than from `/home`, as in the earlier examples) over 10KB, owned by `matthew`, that are not readable by other users, and then use `chmod` to enable reading, like this:

```
matthew@seymour:~$ find / -name "*.txt" -size +10k -user matthew -not -perm +o=r
-exec chmod o+r {} \;
```

When you type your own `-exec` parameters, be sure to include a space before `\;`. Otherwise, you might see an error such as `missing argument to '-exec'`.

Do you see now why some people think the `find` command is scary? Many people learn just enough about `find` to be able to use it in a very basic way, but hopefully you will see how much it can do if you give it chance.

Searches for a String in Input with `grep`

The `grep` command, like `find`, is an incredibly powerful search tool in the right hands. Unlike `find`, though, `grep` processes any text, whether in files or just in standard input.

The basic usage of `grep` is as follows:

```
matthew@seymour:~$ grep "some text" *
```

This searches all files in the current directory (but not subdirectories) for the string `some text` and prints matching lines along with the name of the file. To enable recursive searching in subdirectories, use the `-r` parameter, as follows:

```
matthew@seymour:~$ grep -r "some text" *
```

Each time a string is matched within a file, the filename and the match are printed. If a file contains multiple matches, each of the matches is printed. You can alter this behavior with the `-l` parameter (lowercase *L*), which forces `grep` to print the name of each file that contains at least one match without printing the matching text. If a file contains more than one match, it is still printed only once. Alternatively, the `-c` parameter prints each filename that was searched and includes the number of matches at the end, even if there were no matches.

You have a lot of control when specifying the pattern to search for. You can, as shown previously, specify a simple string like `some text`, or you can invert that search by specifying the `-v` parameter. For example, the following returns all the lines of the file `myfile.txt` that do not contain the word *hello*:

```
matthew@seymour:~$ grep -v "hello" myfile.txt
```

You can also use regular expressions for search terms. For example, you can search `myfile.txt` for all references to *cat*, *sat*, or *mat* with this command:

```
matthew@seymour:~$ grep "[cms]at" myfile.txt
```

Adding the `-i` parameter to this removes case-sensitivity, matching *Cat*, *CAT*, *MaT*, and so on:

```
matthew@seymour:~$ grep -i [cms]at myfile.txt
```

You can also control the output to some extent with the `-n` and `--color` parameters. The first tells `grep` to print the line number for each match, which is where it appears in the source file. The `--color` parameter tells `grep` to color the search terms in the output, which helps them stand out from among all the other text on the line. You choose which color you want by using the `GREP_COLOR` environment variable; for example, `export GREP_COLOR=36` gives you cyan, and `export GREP_COLOR=32` gives you lime green.

This next example shows how to use these two parameters to number and color all matches to the previous command:

```
matthew@seymour:~$ grep -in --color [cms]at myfile.txt
```

Later you learn how important `grep` is for piping with other commands.

Paging Through Output with `less`

The `less` command enables you to view large amounts of text in a more convenient way than by using the `cat` command. For example, your `/etc/passwd` file is probably more than a screen long, so if you run `cat /etc/passwd`, you are not able to see the lines at the

top. Using `less /etc/passwd` enables you to use the cursor keys to scroll up and down the output freely. Type `q` to quit and return to the shell.

On the surface, `less` sounds like an easy command; however, it has the infamy of being one of the few Linux commands to have a parameter for every letter of the alphabet. That is, `-a` does something, `-b` does something else, `-c`, `-d`, `-e`,...`-x`, `-y`, `-z`; they all do things, with some letters even differentiating between upper- and lowercase. Furthermore, these parameters are only used when invoking `less`. After you are viewing your text, even more commands are available. Make no mistake; `less` is a complex beast to master.

Input to `less` can be divided into two categories: what you type before running `less` and what you type while running it. The former category is easy, so we start there.

We have already discussed how many parameters `less` can take, but they can be distilled down to three that are very useful: `-M`, `-N`, and `+`. Adding `-M` (which is different from `-m`) enables verbose prompting in `less`. Instead of just printing a colon and a flashing cursor, `less` prints the filename, the line numbers being shown, the total number of lines, and the how far you are through the file, as a percentage. Adding `-N` (which is different from `-n`) enables line numbering.

The last option, `+`, enables you to pass a command to `less` for it to execute as it starts. To use it, you first need to know the commands available to you in `less`, which means it's time to move to the second category of `less` input: what you type while `less` is running.

The basic navigation keys are the up, down, left, and right arrows; Home and End (for navigating to the start and end of a file); and Page Up and Page Down. Beyond that, the most common command is `/`, which initiates a text search. You type what you want to search for and press Enter to have `less` find the first match and highlight all subsequent matches. Type `/` again and press Enter to have `less` jump to the next match. The inverse of `/` is `?`, which searches backward for text. Type `?`, enter a search string, and press Enter to go to the first previous match of that string, or just use `?` and press Enter to go to the next match preceding the current position. You can use `/` and `?` interchangeably by searching for something with `/` and then using `?` to go backward in the same search.

Searching with `/` and `?` is commonly used with the `+` command-line parameter from earlier, which passes `less` a command for execution after the file has loaded. For example, you can tell `less` to load a file and place the cursor at the first match for the search *hello*, like this:

```
matthew@seymour:~$ less +/hello myfile.txt
```

Or, you can use this to place the cursor at the last match for the search *hello*:

```
matthew@seymour:~$ less +?hello myfile.txt
```

Beyond the cursor keys, the controls primarily involve typing a number and then pressing a key. For example, to go to line 50, you type `50g`, or to go to the 75 percent point of the file, you type `75p`. You can also place invisible mark points through the file by pressing `m` and then typing a single letter. Later, while in the same `less` session, you can press

' (a single quote) and then type the letter, and it moves you back to that letter's position. You can set up to 52 marks, named a–z and A–Z.

One clever feature of `less` is that you can, at any time, press `v` to have your file open inside your text editor. This defaults to `vim`, but you can change it by setting the `EDITOR` environment variable to something else.

If you have made it this far, you can already use `less` better than most users. You can, at this point, justifiably skip to the next section and consider yourself proficient with `less`. However, if you want to be a `less` guru, there are two more things to learn: how to view multiple files simultaneously and how to run shell commands.

Like most other file-based commands in Linux, `less` can take several files as its parameters, as in this example:

```
matthew@seymour:~$ less -MN 1.txt 2.txt 3.txt
```

This loads all three files into `less`, starting at `1.txt`. When viewing several files, `less` usually tells you which file you are in and numbers the files: `1.txt (file 1 of 3)` should be at the bottom of the screen. However, certain things make that go away, so you should use `-M` anyway.

You can navigate between files by typing a colon and then pressing `n` to go to the next file or pressing `p` to go to the previous file; these are referred to from now on as `:n` and `:p`. You can open another file for viewing by typing `:e` and providing a filename. This can be any file you have permission to read, including files outside the local directory. Use `Tab` to complete filenames. Files you open in this way are inserted one place after your current position, so if you are viewing file 1 of 4 and open a new file, the new file is numbered 2 of 5 and is opened for viewing right away. To close a file and remove it from the list, use `:d`.

Viewing multiple files simultaneously has implications for searching. By default, `less` searches within only one file, but it is easy to search within all files. When you type `/` or `?` to search, follow it with a `*`. You should see `EOF-ignore` followed by a search prompt. You can now type a search, and it runs through the current file; if nothing is found, the search looks in subsequent files until it finds a match. You can repeat searches by pressing `Esc` and then either `n` or `N`. The lowercase option repeats the search forward across files, and the uppercase option repeats it backward.

The last thing you need to know is that you can get to a shell from `less` and execute commands. The simplest way to do this is just to type `!` and press `Enter`. This launches a shell and leaves you free to type all the commands you want, as per normal. Type `exit` to return to `less`. You can also type specific commands by entering them after the exclamation point (`!`), using the special character `%` for the current filename. For example, `du -h %` prints the size of the current file. Finally, you can use `!!` to repeat the previous command.

Creating Links Between Files with `ln`

Linux allows you to create links between files that look and work like normal files for the most part. Moreover, it allows you to make two types of links, known as hard links and

symbolic links (symlinks). The difference between the two is crucial, although it might not be obvious at first.

Each filename on your system points to what is known as an inode, which is the absolute location of a file. Linux allows you to point more than one filename to a given inode, and the result is a hard link—two filenames pointing to the same file. These two files share the same contents and attributes. So, if you edit one, the other changes because they are both the same file.

On the other hand, a symlink—sometimes called a soft link—is a redirect to the real file. When a program tries to read from a symlink, it automatically is redirected to what the symlink is pointing at. The fact that symlinks are really just dumb pointers has two advantages: You can link to something that does not exist (and create it later if you want), and you can link to directories.

Both types of links have uses. Instead of using redirection, a hard link instead creates a second reference to (name for) the same file, which means that it saves a little bit of time when accessing the file using the symlink. The big difference, however, is that you are able to delete the “original” filename used to create a hard link and the symlinks will still reference the original data file; if you delete the original filename for a soft link it actually does delete the original data and so all the soft symlinks will break. So why have soft links at all? The reason is that not all file systems permit the use of hard links, and sometimes that matters; for example, when hierarchical data needs to be backed up from a file system that does permit hard links to a file system that does not. In addition, only the root user or a user with `sudo` privileges can create a hard link, whereas soft links are permitted to be created by regular users. Symlinks are popular because they allow a file to appear to be in a different location; you could store your website in `/var/www/live` and an under-construction holding page in `/var/www/construction`. Then you could have Apache point to a symlink `/var/www/html` that is redirected to either the live directory or the construction directory, depending on what you need.

TIP

The `shred` command overwrites a file’s contents with random data, allowing for safe deletion. Because this directly affects a file’s contents, rather than just a filename, this means that all filenames hard linked to an inode are affected.

Both types of link are created using the `ln` command. By default, it creates hard links, but you can create symlinks by passing it the `-s` parameter. The syntax is `ln [-s] something somewhere`, as shown in this example:

```
matthew@seymour:~$ ln -s myfile.txt mylink
```

This creates the symlink `mylink` that points to `myfile.txt`. You don’t see it here, but the file created here is 341 bytes. This is important later. Remove the `-s` to create a hard link. You can verify that your link has been created by running `ls -l`. Your symlink should look something like this:

```
lrwxrwxrwx 1 matthew matthew 5 Feb 19 12:39 mylink -> myfile.txt
```

Note how the file properties start with `l` (lowercase *L*) for *link* and how `ls -l` also prints where a link is going. Symlinks are always very small in size; the previous link is 5 bytes. If you created a hard link, it should look like this:

```
-rw-rw-r 2 matthew matthew 341 Feb 19 12:39 mylink
```

This time the file has normal attributes, but the second number is 2 rather than 1. That number is how many hard links point to this file, which is why it is 2 now. The file size is also the same as that of the previous filename because it is the file, as opposed to just being a pointer.

Symlinks are used extensively in Linux. Programs that have been superseded, such as `sh`, now point to their replacements (in this case `bash`). Sometimes versioning is accomplished through symlinks. For example, applications that link against a fictional program called `snowflake` could load `/usr/bin/snowflake`. Internally, however, that is just a symlink that points to the actual `snowflake` release: `/usr/bin/snowflake.7.2.5`. This enables multiple versions of libraries to be installed without application developers needing to worry about the specific details.

Finding Files from an Index with `locate`

When you use the `find` command, it searches recursively through each directory each time you request a file. As you can imagine, this is slow. Fortunately, Ubuntu ships with a `cron` job that creates an index of all the files on your system every night. Searching this index is extremely fast, which means that if the file you are looking for has been around since the last index, this is the preferable way of searching.

To look for a file in your index, use the command `locate` followed by the names of the files you want to find, like this:

```
matthew@seymour:~$ locate myfile.txt
```

On a relatively modern computer (say, with at least one processor that runs at 1.5GHz or higher), `locate` should be able to return all the matching files in less than one second. The trade-off for this speed is lack of flexibility. You can search for matching filenames, but, unlike with `find`, you cannot search for sizes, owners, access permissions, or other attributes. The one thing you can change is case sensitivity; use the `-i` parameter to do a search that is not case sensitive.

Although Ubuntu rebuilds the filename index nightly, you can force a rebuild whenever you want by running the command `updatedb` with `sudo`. This usually takes a few minutes, but when it's done, the new database is immediately available.

Listing Files in the Current Directory with `ls`

The `ls` command, like `ln`, is a command that most people expect to be very straightforward. It lists files, but how many options can it possibly have? In true Linux style, the answer is many, although again you need only know a few to wield great power!

The basic usage is simply `ls`, which lists the files and directories in the current location. You can filter that by using normal wildcards, so all these are valid:

```
matthew@seymour:~$ ls *
matthew@seymour:~$ ls *.txt
matthew@seymour:~$ ls my*ls *.txt *.xml
```

Any directories that match these filters are recursed into one level. That is, if you run `ls my*` and you have the files `myfile1.txt` and `myfile2.txt` and a directory `mystuff`, the matching files are printed first. Then `ls` prints the contents of the `mystuff` directory.

The most popular parameters for customizing the output of `ls` are as follows:

- ▶ `-a`—Includes hidden files
- ▶ `-h`—Uses human-readable sizes
- ▶ `-l` (**lowercase** *L*)—Enables long listing
- ▶ `-r`—Reverse order
- ▶ `-R`—Recursively lists directories
- ▶ `-s`—Shows sizes
- ▶ `--sort`—Sorts the listing

All files that start with a period are hidden in Linux, so that includes the `.gnome` directory in your `/home` directory, as well as `.bash_history` and the `.` and `..` implicit directories that signify the current directory and the parent. By default, `ls` does not show these files, but if you run `ls -a`, they are shown. You can also use `ls -A` to show all the hidden files except `.` and `..`.

The `-h` parameter needs to be combined with the `-s` parameter, like this:

```
matthew@seymour:~$ ls -sh *.txt
```

This outputs the size of each matching file in a human-readable format, such as 108KB or 4.5MB.

Using the `-l` parameter enables you to get much more information about your files. Instead of just getting the names of the files, you get output like this:

```
drwxrwxr-x 24 matthew matthew 4096 Dec 24 21:33 arch
-rw-r--r-- 1 matthew matthew 18691 Dec 24 21:34 COPYING
-rw-r--r-- 1 matthew matthew 88167 Dec 24 21:35 CREDITS
drwxrwxr-x 2 matthew matthew 4096 Dec 24 21:35 crypto
```

This output shows four matches and prints a lot of information about each of them. The first row shows the `arch` directory; you can tell it is a directory because its file attributes starts with a `d`. The `rw-rwxr-x` following that shows the access permissions, and this has

special meanings because it is a directory. Read access for a directory enables users to see the directory contents, write access enables you to create files and subdirectories, and execute access enables you to `cd` into the directory. If a user has execute access but not read access, the user can `cd` into the directory but cannot list files.

Moving on, the next number on the line is 24, which also has a special meaning for directories: It is the number of subdirectories (including `.` and `..`). After that is `matthew matthew`, which is the name of the user owner and the group owner for the directory. Next are the size and modification time, and finally the directory name itself.

The next line shows the file `COPYING`, and most of the numbers have the same meaning, with the exception of the 1 immediately after the access permissions. For directories, this is the number of subdirectories, but for files it is the number of hard links to this file. A 1 in this column means this is the only filename pointing to this inode, so if you delete it, it is gone.

Ubuntu comes configured with a shortcut command for `ls -l: ll`.

The `--sort` parameter enables you to reorder the output from the default alphabetic sorting. You can sort by various things, although the most popular are extension (alphabetically), size (largest first), and time (newest first). To flip the sorting (making size sort by smallest first), use the `-r` parameter also. So, the following command lists all `.ogg` files, sorted smallest to largest:

```
matthew@seymour:~$ ls --sort size -r *.ogg
```

Finally, the `-R` parameter recurses through subdirectories. For example, `ls /etc` lists all the files and subdirectories in `/etc`, but `ls -R /etc` lists all the files and subdirectories in `/etc`, all the files and subdirectories in `/etc/acpi`, all the files and subdirectories in `/etc/acpi/actions`, and so on until every subdirectory has been listed.

Listing System Information with `lsblk`, `lshw`, `lsmod`, `lspci`, and `neofetch`

The commands `lsblk`, `lshw`, `lsmod`, `lspci`, and `neofetch` are not really related to `ls`, but they work in a similar way. Here, the focus is on listing information about your system rather than the contents of a directory.

To list the storage, or *block*, devices that are attached to your system, use the following:

```
matthew@seymour:~$ lsblk
NAME        MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda           8:0    0 465.8G  0 disk
├─sda1        8:1    0     1K  0 part
├─sda2        8:2    0 453.7G  0 part /
└─sda5        8:5    0  12.1G  0 part [SWAP]
sdb           8:16    0    1.4T  0 disk
└─sdb1        8:17    0    1.4T  0 part
sr0          11:0    1   1024M  0 rom
```

The next command must be run as root for a full listing. Note that the output may be quite long, so this command may be most useful if you pipe it into `grep` and search for a specific bit of text, as described in Chapter 12, “Command-Line Master Class, Part 2.” To list the hardware detected in your system, use the following:

```
matthew@seymour:~$ sudo lshw
```

To list the status of modules in the Linux kernel, use this, which takes the contents of `/proc/modules` and formats it nicely:

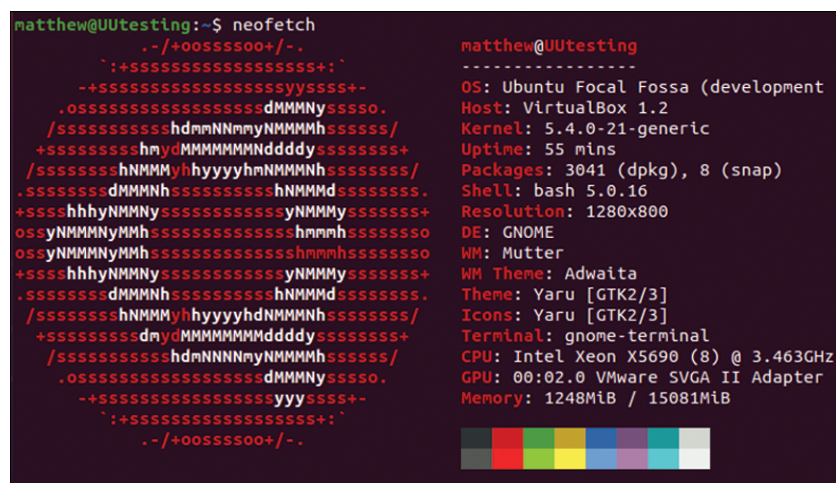
```
matthew@seymour:~$ lsmod
```

To list the PCI devices attached to your system, use the following:

```
matthew@seymour:~$ lspci
```

For an interesting, high-level listing of system information, use this, which will return output similar to Figure 11.1:

```
matthew@seymour:~$ neofetch
```



```
matthew@UUtesting:~$ neofetch
.-/+00ssssso+/-.
`:+ssssssssssssssss+:`
-+ssssssssssssssssyyss+-.
.o0sssssssssssssssdMMMNysssso.
/sss0sssssssssdmNnNNnyNMMNMhsssss/
+ssssssssshnydMMMMMMNdddyssssss+
/sss0ssssshNMMMyhhyyyhNMMMNhssssss/
.sssssssdMMMNhssssssshNMMMdssssss.
+ssshhnyNMMNysssssssssyNMMMyssssss+
ossyNMMMNyMMhssssssssshmmhssssssso
ossyNMMMNyMMhssssssssshmmhssssssso
+ssshhnyNMMNysssssssssyNMMMyssssss+
.sssssssdMMMNhssssssshNMMMdssssss.
/sss0ssshNMMMyhhyyyhNMMMNhssssss/
+sssssssdnydMMMMMMNdddyssssss+
/sss0ssssshdNnnNnyNMMNMhsssss/
.o0sssssssssssssssdMMMNysssso.
-+ssssssssssssssyyss+-.
`:+ssssssssssssss+:`
.-/+00ssssso+/-.

matthew@UUtesting
-----
OS: Ubuntu Focal Fossa (development)
Host: VirtualBox 1.2
Kernel: 5.4.0-21-generic
Uptime: 55 mins
Packages: 3041 (dpkg), 8 (snap)
Shell: bash 5.0.16
Resolution: 1280x800
DE: GNOME
WM: Mutter
WM Theme: Adwaita
Theme: Yaru [GTK2/3]
Icons: Yaru [GTK2/3]
Terminal: gnome-terminal
CPU: Intel Xeon X5690 (8) @ 3.463GHz
GPU: 00:02.0 VMware SVGA II Adapter
Memory: 1248MiB / 15081MiB
```

FIGURE 11.1 Neofetch screenshots are a popular way to share info about your system with online friends in places like Reddit.

Reading Manual Pages with `man`

It's time for a much-needed mental break, and the `man` command is easy to use. Most people use only the topic argument, like this: `man gcc`. However, two other commands that work closely with `man` are very useful: `whatis` and `apropos`.

The `whatis` command returns a one-line description of another command, which is the same text you get at the top of that command's man page. Here is an example:

```
matthew@seymour:~$ whatis ls
ls      (1) - list directory contents
```

The output explains what `ls` does and also provides the man page section number for the command so you can query it further.

The `apropos` command takes a search string as its parameter and returns all man pages that match the search. For example, `apropos mixer` returns this list:

```
alsamixer (1) - soundcard mixer for ALSA soundcard driver
mixer (1) - command-line mixer for ALSA soundcard driver
aumix (1) - adjust audio mixer
```

So, you can use `apropos` to help find commands and use `whatis` to find out what a command does.

One neat trick is that many of the tips and tricks you learned for `less` also work when viewing man pages (for example, using `/` and `?` to search). This is one of the beautiful things about UNIX systems: Gaining knowledge in one area often benefits you in other areas.

You can use a number of other commands to search the file system, including the following:

- ▶ **whereis command**—Returns the location of `command` (for example, `/bin`, `/sbin`, or `/usr/bin/command`) and its man page, which is an entry for the command included as a file already on your computer in a standardized manual format.
- ▶ **whatis command**—Returns a one-line synopsis from `command`'s man page.
- ▶ **type name**—Returns how a name would be interpreted if used as a command. This generally shows options or the location of the binary that will be used. For example, `type ls` returns `ls is aliased to 'ls -color=auto'`.

Making Directories with `mkdir`

Making directories is as easy as it sounds, although there is one parameter you should be aware of: `-p`. If you are in `/home/matthew` and you want to create the directory `/home/matthew/audio/sound`, you get an error like this:

```
mkdir: cannot create directory 'audio/sound': No such file or directory
```

At first glance, this seems wrong; `mkdir` cannot create the directory because it does not exist. What it actually means is that it cannot create the directory `sound` because the directory `audio` does not exist. This is where the `-p` parameter comes in: If you try to make a directory within another directory that does not exist, as in the preceding example, `-p` creates the parent directories, too, if you use it like this:

```
matthew@seymour:~$ mkdir -p audio/sound
```

This first creates the `audio` directory and then creates the `sound` directory inside it.

Moving Files with `mv`

The `mv` command is one of the easiest around. There are two helpful parameters to `mv`: `-f`, which overwrites files without asking; and `-u`, which moves the source file only if it is newer than the destination file. That's it. You can use absolute paths to indicate the destination directory (starting from `/`):

```
matthew@seymour:~$ mv filename /newdirectory/newfilename
```

Or you can use relative paths from the current directory (starting without a slash). This is generally entered in the source directory, but it doesn't have to be; you can use an absolute path to indicate the source directory, too.

Renaming Files with `rename`

We often use `mv` to rename a single file. This would be tedious, however, for renaming multiple files. For this, you use `rename`. The basic syntax is simple:

```
matthew@seymour:~$ rename 's/filename/newfilename/'
```

The part in the single quotes is a Perl expression, which means the command is far more powerful than this example suggests. Let's say you have a directory filled with files that end with the extension `.htm`, and you want to rename them all to `.html`:

```
matthew@seymour:~$ rename 's/\.htm/\.html/' *.htm
```

Notice here that the `.` characters must be preceded with a `\` to let the command know they are part of the text being searched and replaced rather than part of the command. This is called “escaping” the characters. Next, notice that the replace part of the command is followed by a wildcard character (`*`) and the remainder of the filename to search for in the directory. Anything matching the combination of `filename.htm` will be renamed to `filename.html`.

`rename` is incredibly powerful. See the man page for more.

Deleting Files and Directories with `rm`

The `rm` command has only one parameter of interest: `--preserve-root`. You should know that issuing `rm -rf /` with `sudo` will try to delete your entire Linux installation because `-r` means recursive and `-f` means force (that is, do not prompt for confirmation before deleting). It used to be possible for a clumsy person to issue this command by accident—not by typing the command on purpose but by putting a space in the wrong place; for example:

```
matthew@seymour:~$ rm -rf /home/matthew
```

This command deletes the `/home` directory of the user `matthew`. This is not an uncommon command; after you have removed a user and backed up the user's data, you will probably want to issue something similar. However, if you accidentally add a space between the `/` and the `h` in `home`, you get this:

```
matthew@seymour:~$ rm -rf / home/matthew
```

This time the command means “delete everything recursively from / and then delete home/matthew”—quite a different result! To stop this from happening, the `rm` command was changed to use `--preserve-root` by default, meaning that it is impossible to do this accidentally, and `root` now protects you from catastrophe with this message:

```
rm: it is dangerous to operate recursively on '/'
rm: use --no-preserve-root to override this failsafe.
```

Because `--preserve-root` is the default for the command, if you actually do want to remove the contents of your entire file system (this is dangerous), you use `--no-preserve-root`.

Sorting the Contents of a File with `sort`

Say that you have a text file, and you want to sort its contents alphabetically. That is easy. Assume that your text file is filled with one letter on each line, upper- or lowercase. Here is what you get when you run the `sort` command on this file:

```
matthew@seymour:~$ sort testfile.txt
a
A
b
B
```

This is useful. You can also sort in reverse order:

```
matthew@seymour:~$ sort -r testfile.txt
```

This command outputs the same contents but this time from Z to a.

You can use `sort` with files of numbers, but it doesn’t give you a numerical sort. It gives you output like this:

```
matthew@seymour:~$ sort numberfile.txt
1
14
5
58
6
```

You can see here that the `sort` command performs a rather useless alphabetic-style sort on the numbers. Fortunately, there is a switch you can use to fix this:

```
matthew@seymour:~$ sort -n numberfile.txt
1
5
6
14
58
```

There are tons of neat tricks you can do with `sort` and many more parameters explained in the `man` file. For files with a number of columns, such as a directory listing, you can specify the `-k` switch and the number of the column that you want to sort. Let's look at an example of a directory of old `conky` files and start with an `ls` command for comparison:

11

```
matthew@seymour:~/conky$ ls -la
total 60
drwxr-xr-x  2 matthew matthew 4096 Dec 25  2012 .
drwxr-xr-x 91 matthew matthew 4096 Jul 28 18:42 ..
-rwxr-xr-x  1 matthew matthew 5526 Dec 25  2012 conkyrc_main
-rwxr-xr-x  1 matthew matthew 5502 Dec 25  2012 conkyrc_main~
-rwxr-xr-x  1 matthew matthew 5387 Apr 16  2008 conkyrc_main (old)
-rwxr-xr-x  1 matthew matthew 1326 Mar 15  2008 conkyrc_weather
-rwxr-xr-x  1 matthew matthew 2549 Oct 23  2009 sample_conky.conf
-rwxr-xr-x  1 matthew matthew  128 Apr  8  2008 start_conky (copy).sh
-rwxr-xr-x  1 matthew matthew  139 Dec 25  2012 start_conky.sh
-rwxr-xr-x  1 matthew matthew  140 Dec 25  2012 start_conky.sh~
-rwxr-xr-x  1 matthew matthew 1503 Sep 30  2007 weather.sh
-rwxr-xr-x  1 matthew matthew 2379 Sep 30  2007 weather.xslt
```

Here is the same listing, this time sorted by file size:

```
matthew@seymour:~$ ls -la | sort -n -k5
total 60
-rwxr-xr-x  1 matthew matthew  128 Apr  8  2008 start_conky (copy).sh
-rwxr-xr-x  1 matthew matthew  139 Dec 25  2012 start_conky.sh
-rwxr-xr-x  1 matthew matthew  140 Dec 25  2012 start_conky.sh~
-rwxr-xr-x  1 matthew matthew 1326 Mar 15  2008 conkyrc_weather
-rwxr-xr-x  1 matthew matthew 1503 Sep 30  2007 weather.sh
-rwxr-xr-x  1 matthew matthew 2379 Sep 30  2007 weather.xslt
-rwxr-xr-x  1 matthew matthew 2549 Oct 23  2009 sample_conky.conf
drwxr-xr-x  2 matthew matthew 4096 Dec 25  2012 .
drwxr-xr-x 91 matthew matthew 4096 Jul 28 18:42 ..
-rwxr-xr-x  1 matthew matthew 5387 Apr 16  2008 conkyrc_main (old)
-rwxr-xr-x  1 matthew matthew 5502 Dec 25  2012 conkyrc_main~
-rwxr-xr-x  1 matthew matthew 5526 Dec 25  2012 conkyrc_main
```

Printing the Last Lines of a File with `tail`

If you want to watch a log file as it is written to, or if you want to monitor a user's actions as they are occurring, you need to be able to track log files as they change. In these situations, you need the `tail` command, which prints the last few lines of a file and updates as new lines are added. The following command tells `tail` to print the last few lines of `/var/log/apache2/access.log`, the Apache hit log:

```
matthew@seymour:~$ tail /var/log/apache2/access.log
```

To get `tail` to remain running and update as the file changes, add the `-f` (follow) parameter:

```
matthew@seymour:~$ tail -f /var/log/apache2/access.log
```

You can tie the life span of a `tail -f` to the existence of a process by specifying the `--pid` parameter. When you do this, `tail` continues to follow the file you asked for until it sees that the process identified by *process ID (PID)* is no longer running, at which point it stops tailing.

If you specify multiple files on the command line, `tail` follows them all, printing file headers whenever the input source changes. Press Ctrl+C to terminate `tail` when in follow mode.

Printing the Location of a Command with `which`

The purpose of `which` is to tell you exactly which command would be executed if you typed it. For example, `which mkdir` returns `/bin/mkdir`, telling you that running the command `mkdir` runs `/bin/mkdir`.

Downloading Files with `wget`

Let's say you see a website with useful content that you need to download to your server—for (for example, <https://releases.ubuntu.com>) because you want to make available to students who work in your computer lab a copy of the ISO of the current release of Ubuntu. This practice, called *mirroring*, is commonly accepted. Or, maybe you want to download the latest *Minecraft* server file to your server and don't want to download it from the website to your local machine first before you upload it to your server. You can do this with `wget`, which exists to download files using HTTP, HTTPS, and FTP, like this:

```
matthew@seymour:~$ wget http://releases.ubuntu.com/20.04/ubuntu-20.04-desktop-  
amd64.iso
```

This downloads the linked file directly to the directory in which you issue the command.

What if you want to copy all the content files from your existing web server onto a new server? You can use the `-m` or the `--mirror` flag to do this. This example downloads all the contents of the directory you specify, assuming that you have access, to your current directory:

```
matthew@seymour:~$ wget http://youroldserver.com/website/files
```

TIP

You can use `wget` with any standard URL syntax, including specifying ports and user-names and passwords, but you should realize that the username and password information will be transmitted in plain text; therefore, this is not considered a secure method of data transfer. For that, use `scp`, which is covered in Chapter 19, "Remote Access with SSH and VNC."

References

- ▶ **www.gnu.org**—The website of the GNU project, which contains manuals and downloads for lots of command-line software
- ▶ ***Linux in a Nutshell* by Stephen Figgins, Arnold Robbins, Ellen Siever, and Robert Love**—A dictionary of Linux commands that includes a wide selection of Linux commands and explanations of what they do
- ▶ **www.linuxcommand.org**—A “one-stop command-line shop” that contains a wealth of useful information about the console
- ▶ ***The Art of UNIX Programming* by Eric Raymond**—Focuses on the philosophy behind UNIX and manages to mix in much about the command line

