



Introducing the BASH Shell

As you learned in Chapter 1, strictly speaking, the word *Linux* refers to just the kernel, which is the fundamental, invisible program that runs your PC and lets everything happen. However, on its own, the kernel is completely useless. It needs programs to let users interact with the PC and do cool stuff, and it needs a lot of system files (also referred to as *libraries*) to provide vital functions.

The GNU Project provides many of these low-level pieces of code and programs. This is why many people refer to the Linux operating system as GNU/Linux, giving credit to the fact that, without the GNU components, Linux wouldn't have gotten off the starting blocks.

The GNU Project provides various shell programs, too. A *shell* is what the user interacts with on a day-to-day basis, whether by mouse or keyboard. The word originates from the fact that the shell is the outer layer of the operating system, which encompasses the kernel (and in some instances protects it by filtering out bad user commands!). Some shells offer graphical functionality but, in general, the word *shell* is understood to mean text-only interfaces. These text shell programs are also known as *terminal programs*, and they're often colloquially referred to as *command-line prompts*, in reference to the most important component they provide. This kind of shell lets you take control of your system in a quick and efficient way.

By learning how to use the shell, you'll become the true master of your own system. In this part of the book, you'll learn all you need to know about using the shell. This chapter introduces the BASH shell, which is the default one in Ubuntu.

What Is the BASH Shell?

The best way of explaining the BASH shell to a Windows user is to compare it to the DOS command prompt. It lets you issue commands directly to the operating system via the keyboard without needing to mess around with the mouse and windows (although it is sometimes possible to use the mouse within a BASH shell to copy and paste text, and sometimes to control simple text-based menus). The big difference is that the BASH shell has commands for just about everything you might do on your system, whereas the DOS

command prompt is restricted to tools capable of manipulating and viewing files and directories, and on Windows 2000/XP/Vista machines, configuring certain system settings.

In the old days, the DOS command prompt was also the visible layer of an entire operating system in which DOS programs were designed to be run. However, the shell is merely one of the many ways of accessing the Linux kernel and subsystems. It's true that many programs are designed to run via the BASH shell, but technically speaking, most actually run on the Linux operating system, and simply take input and show their output via the BASH shell.

Note Linux purists will point out another reason why the shell isn't exactly the same as a DOS command prompt within Windows: it doesn't run in virtual machine mode, a CPU trick by which part of the memory is subdivided to let programs run as if they had the PC all to themselves.

Linux finds itself with the BASH shell largely because Linux is a clone of Unix. In the early days of Unix, the text-based shell was all that was offered as a way of letting users control the computer. Typing in commands directly is one of the most fundamental ways of controlling any type of computer and, in the evolutionary scale, comes straight after needing to set switches and watch blinking lights in order to run programs.

That the BASH shell can trace its history back to the early days of Unix might sound like a tacit indication that the BASH is somehow primitive—far from it. It's one of the most efficient and immediate ways of working with your computer. Many people consider the command-line shell to be a way of using a computer that has yet to be superseded by a better method.

Note When you run a shell on a Linux system, the system refers to it as a `tty` device. This stands for tele-typewriter, a direct reference to the old system of inputting data on what were effectively electronic typewriters connected to mainframe computers. These, in turn, took their names from the devices used to automate the sending and receiving of telegrams in the early part of the twentieth century.

Most Linux distributions come with a choice of different kinds of shell programs. However, the default shell is BASH, as is the case in Ubuntu. BASH stands for Bourne Again SHell. This is based on the Bourne shell, a tried-and-tested program that originated in the early days of Unix.

The other shells available include PDKSH (Public Domain Korn SHell, based on Korn Shell, another early Unix shell), and ZSH (Z SHell), a more recent addition. These are usually used by people who want to program Linux in various ways, or by those who simply aren't happy with BASH.

The BASH shell is considered by many to be the best of all worlds in that it's easy enough for beginners to learn, yet is able to grow with them and offer additional capabilities as necessary. BASH is capable of scripting, for example, which means you can even create your own simple programs.

Why Bother with the Shell?

You might have followed the instructions in Part 2 of this book and consider yourself an expert in Linux. But the real measure of a Linux user comes from your abilities at the shell.

In our modern age, the GUI is mistakenly considered “progress.” For instance, users of the Microsoft and Apple-based operating systems are quite accustomed to using a mouse to navigate and perform various tasks. While it's handy in certain situations—it would be difficult to imagine image editing without a mouse, for example—in many other situations, such as when manipulating files, directly typing commands is far more efficient.

Most modern Linux distributions prefer you to use the GUI to do nearly everything. This is because they acknowledge the dominance of Windows and realize they need to cater to mouse users who might not even know the shell exists (and, of course, programs like web browsers would be unusable without a GUI!). To this end, they provide GUI tools for just about every task you might wish to undertake. Ubuntu is strong in this regard, and you can configure a lot of things from the desktop.

However, it's well worth developing at least some command-line shell skills, for a number of reasons:

It's simple and fast. The shell is the simplest and fastest way of working with Ubuntu. As just one example, consider the task of changing the IP address of your network card. You could click the Systems menu, then the Administration option, then the Networking option, and then double-click the entry in this list relating to your network card. That will take at least a minute or two if you know what you're doing, and perhaps longer if it's new to you. Alternatively, you could simply open a shell and type this:

```
ifconfig eth0 192.168.0.15 up
```

It's versatile. Everything can be done via the shell—from deleting files, to configuring hardware, to creating MP3s. A lot of GUI programs actually make use of programs you can access via the shell.

It's consistent among distributions. All Linux systems have shells and understand the same commands (broadly speaking). However, not all Linux systems will have Ubuntu's graphical configuration programs. SUSE Linux uses its own GUI configuration tool, as does Mandriva Linux. Therefore, if you ever need to use another system, or decide to switch distributions, a reliance on GUI tools will mean learning everything from scratch. Knowing a few shell commands will help you get started instantly.

It's crucial for troubleshooting. The shell offers a vital way of fixing your system should it go wrong. Your Linux installation might be damaged to the extent that it cannot boot to the GUI, but you'll almost certainly be able to boot into a shell. A shell doesn't require much of the system other than the ability to display characters on the screen and take input from the keyboard, which most PCs can do, even when they're in a sorry state. This is why most rescue floppies offer shells to let you fix your system.

It's useful for remote access. One handy thing about the shell is that you don't need to be in front of your PC to use it. Programs like `ssh` let you log in to your PC across the Internet and use the shell to control your PC (as described in Chapter 34). For example, you can access data on a remote machine, or even fix it when you're unable to attend the machine's location. This is why Linux is preferred on many server systems when the system administrator isn't always present on the site.

It's respected in the community. Using a shell earns you enormous brownie points when speaking to other Linux users. It separates the wheat from the chaff and the men from the boys (or women from the girls). If you intend to use Linux professionally, you will most certainly need to be a master at the shell.

Seen in this light, learning at least a handful of shell commands is vital to truly mastering your PC.

The drawback when using a command-line shell is that it's not entirely intuitive. Take for instance the command to change the network card's IP address:

```
ifconfig eth0 192.168.0.15 up
```

If you've never used the shell before, it might as well be Sanskrit. What on earth does `ifconfig` mean? And why is there the word `up` at the end?

Note If you're curious, the command tells the network card, referred to by Linux as `eth0`, to adopt the specified IP address. The word `up` at the end merely tells it to activate—to start working now. If the word `down` were there instead, it would deactivate! Don't worry about understanding all this right now; later in this chapter, we'll explain how you can learn about every Linux command.

Learning to use the shell involves learning terms like these. Hundreds of commands are available, but you really need to learn only around 10 or 20 for everyday use. The comparison with a new language is apt because, although you might think it daunting to learn new terminology, with a bit of practice, it will all become second nature. Once you've used a command a few times, you'll know how to use it in the future.

The main thing to realize is that the shell is your friend. It's there to help you get stuff done as quickly as possible. When you become familiar with it, you'll see that it is a beautiful concept. The shell is simple, elegant, and powerful.

When Should You Use the Shell?

The amount of use the Linux shell sees is highly dependent on the user. Some Linux buffs couldn't manage without it. They use it to read and compose e-mail, and even to browse the Web (usually using the Mutt and Lynx programs, respectively).

However, most people simply use it to manage files, view text files (like program documentation), and run programs. All kinds of programs—including GUI and command-line—can be started from the shell. As you'll learn in Chapter 29, unlike with Windows, installing a program on Ubuntu doesn't necessarily mean the program will automatically appear on the Applications menu. In fact, unless the installation routine is specifically made for the version of Linux you're running, this is unlikely. Therefore, using the shell is a necessity for most people.

Note Unlike with DOS programs, Ubuntu programs that describe themselves as “command-line” are rarely designed to run solely via the command-line shell. All programs are like machines that take input at one end and output objects at the other. Where the input comes from and where the output goes to is by no means limited to the command line. Usually, with a command-line program, the input and output are provided via the shell, and the programmer makes special dispensation for this, but this way of working is why GUI programs often make use of what might be considered shell programs. You'll often find that a GUI program designed to, for example, burn CDs, will also require the installation of a command-line program that will actually do the hard work for it.

There's another reason why the shell is used to run programs: you can specify how a particular program runs before starting it. For example, to launch the Totem Movie Player in full-screen mode playing the `myvideofile.mpg` file, you could type this:

```
totem --fullscreen myvideofile.mpg
```

This saves the bother of starting the program, loading a clip, and then selecting the full-screen option. After you've typed the command once or twice, you'll be able to remember it for the next time. No matter how much you love the mouse, you'll have to admit that this method of running programs is more efficient.

When you get used to using the shell, it's likely you'll have it open most of the time behind your other program windows.

Getting Started with the Shell

You can start the shell in a number of ways. The most common is to use a terminal emulator program. As its name suggests, this runs a shell inside a program window on your desktop.

You can start GNOME Terminal, the built-in GNOME shell emulator, by clicking **Applications ► Accessories ► Terminal**, as shown in Figure 13-1.

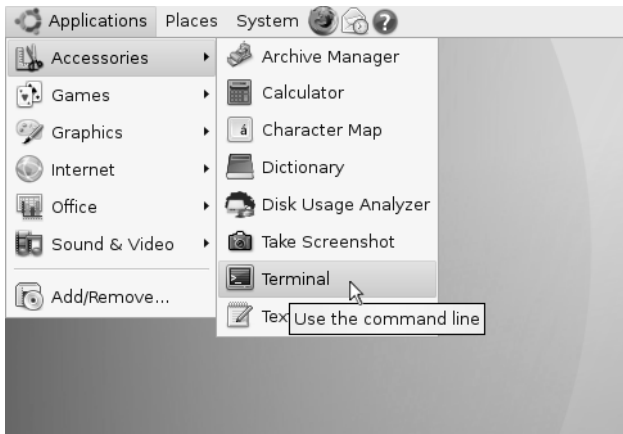


Figure 13-1. Start the *The GNOME Terminal* program from the *Accessories* submenu.

You'll see the terminal window—a blank, white window that's similar to a simple text editor window. When you run the terminal for the first time, at the top of the window will be a handful of lines telling you about the `sudo` command. We explain the importance of this in Chapter 14, but right now there's no need to worry about it.

Below this will be the most important component of the terminal window—the *command prompt*: a few words followed by the dollar symbol: `$`. On our test system, this is what we see:

```
keir@keir-desktop:~$
```

The first part is the username—the user account we created during installation and use to log in to the PC. After the `@` sign is the hostname of the PC, which we also chose when installing Ubuntu. The hostname of the PC isn't important on most desktop PCs; it's a legacy from the days of Unix.

Note What's with the `@` sign? Again, it's a legacy from the days of Unix when the hostname referred to the site the computer was located at (such as the university or military facility). Reading the command prompt in this context, the line reads that the user `keir` is logged into the computer located at the location specified in the hostname! Like we said, this is a legacy of Unix's origins and doesn't mean much nowadays.

After the colon is the current directory you're browsing. In this example, the `~` symbol appears instead of an actual path or directory name. This is merely Linux shorthand for

the user's home directory. In other words, wherever we see a ~ on our test PC, we read it as /home/keir/. After this is the dollar symbol (\$), which indicates that we're currently logged in as an ordinary user, as opposed to the root user. However, unlike most other Linux distributions, Ubuntu doesn't use the root account during day-to-day operations, so this is a moot point. Finally, there is a cursor, and this is where you can start typing commands!

Note If you were to log in as root, a hash (#) would appear instead of the dollar symbol prompt. This is important to remember, because often in magazines and some computer manuals, the use of the hash symbol before a command indicates that it should be run as root. In addition, if you use the rescue function of the install CD, you'll be running as root, and a hash will appear at the prompt. See Chapter 14 for more information about the root user.

Running Programs

When we refer to *commands* at the shell, we're actually talking about small programs. When you type a command to list a directory, for example, you're actually starting a small program that will do that job. Seen in this light, the shell's main function is to simply let you run programs—either those that are built into the shell, such as ones that let you manipulate files, or other, more complicated programs that you've installed yourself.

The shell is clever enough to know where your programs are likely to be stored. This information was given to it when you first installed Ubuntu and is stored in a system variable.

Note A *variable* is the method Linux uses to remember things like names, directory paths, or other data. There are many system variables that are vital for the running of Ubuntu. These variables can be seen by typing `set` at the command prompt.

The information about where your programs are stored, and therefore where Ubuntu should look for commands you type in, as well as any programs you might want to run, is stored in the `PATH` variable. You can take a look at what's currently stored there by typing the following:

```
echo $PATH
```

Don't forget that the difference between uppercase and lowercase letters matters to Ubuntu, unlike with Windows and DOS.

The `echo` command merely tells the shell to print something on screen. In this case, you're telling it to "echo" the `PATH` variable onto your screen. On our test PC, this returned the following information:

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/bin/X11: ➡
/usr/games
```

Several directories are in this list, each separated by a colon.

Don't worry too much about the details right now. The important thing to know is that whenever you type a program name, the shell looks in each of the listed directories in sequence. In other words, when you type `ls`, the shell will look in each of the directories stored in the `PATH` variable, starting with the first in the list, to see if the `ls` program can be found. The first instance it finds is the one it will run. (The `ls` command gives you a directory listing, as described in the "Listing Files" section later in this chapter.)

But what if you want to run a program that is not contained in a directory listed in your `PATH`? In this case, you must tell the shell exactly where the program is. Here's an example:

```
/home/keir/myprogram
```

This will run a program called `myprogram` in the `/home/keir` directory. It will do this regardless of the directory you're currently browsing, and regardless of whether there is anything else on your system called `myprogram`.

If you're already in the directory where the program in question is located, you can type the following:

```
./myprogram
```

So, just enter a dot and a forward slash, followed by the program name. The dot tells BASH that what you're referring to is "right here." Like the tilde symbol (`~`) mentioned earlier, this dot is BASH shorthand.

Getting Help

Each command usually has help built in, which you can query (a little like typing `/?` after a command when using DOS). This will explain what the command does and how it should be used. In most cases, you'll see an example of the command in use, along with the range of command options that can be used with it. For example, you can get some instant help on the `ifconfig` command by typing this:

```
ifconfig --help
```

You'll see the help screen shown in Figure 13-2.

The `--help` option is fairly universal, and most programs will respond to it, although sometimes you might need to use a single dash. Just type the command along with `--help` to see what happens. You'll be told if you're doing anything wrong.

In addition, most commands have manuals that you can read to gain a fairly complete understanding of how they work. Virtually every Ubuntu setup has a set of these man pages, which can be accessed by typing this:

```
man <command>
```


However, man pages are often technical and designed for experienced Ubuntu users who understand the terminology.

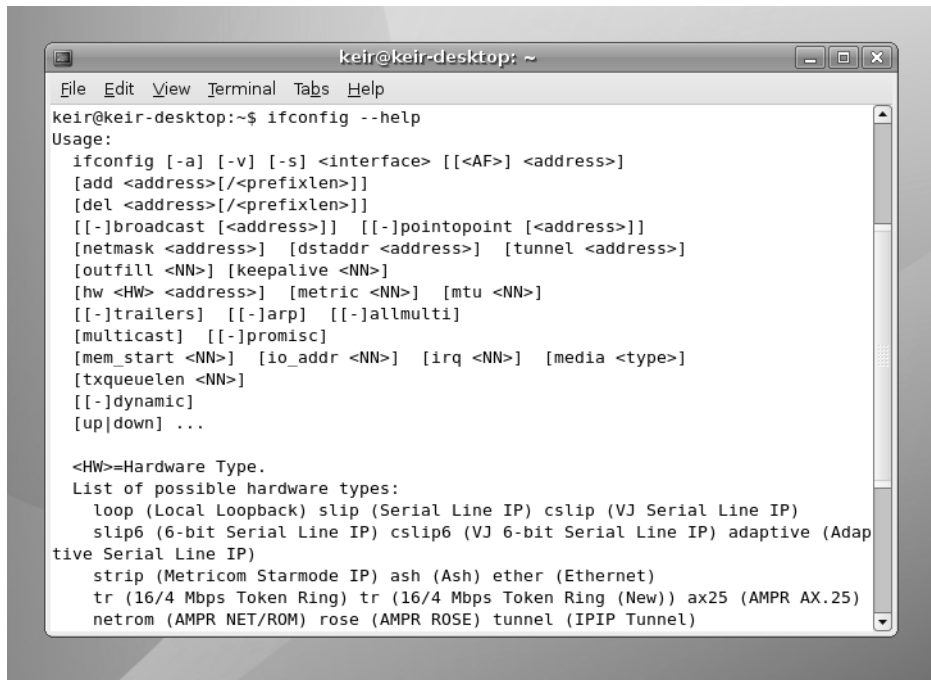


Figure 13-2. Most commands contain built-in help to give you a clue as to how they're used.

Some commands also have info pages, which offer slightly more down-to-earth guides. You can read these by typing this:

```
info <command>
```

If a command isn't covered by the info system, you'll be shown the default screen explaining basic facts about how the info command works.

Note that both man and info have their own man and info pages, explaining how they work. Just type `man man` or `info info`. We explain how to read man and info pages in Appendix C.

Running the Shell via a Virtual Console

As noted earlier, you can start the shell in a number of ways. The most common way among Linux diehards is via a virtual console. To access a virtual console, press Ctrl+Alt, and then press one of the function keys from F1 through F6 (the keys at the top of your keyboard).

Using a virtual console is a little like switching desks to a completely different PC. Pressing Ctrl+Alt+F1 will cause your GUI to disappear, and the screen to be taken over by a

command-line prompt (don't worry; your GUI is still there and running in the background). You'll be asked to enter your username and your password.

Any programs you run in a virtual console won't affect the rest of the system, unless they're system commands. (As discussed in Chapter 16, one way to rescue a crashed GUI program is to switch to a virtual console and attempt to terminate the program from there.)

You can switch back to the GUI by pressing Ctrl+Alt+F7. Don't forget to quit your virtual console when you're finished with it, by typing `exit`.

BOOTING INTO THE SHELL

If you're really in love with the shell, you can choose to boot into it, avoiding the GUI completely (although you can later start the GUI by typing `startx` at the command line).

Booting into the shell is done by defining a custom run level. A *run level* is how the operating mode that Ubuntu is currently running in is described. For example, one particular run level might start a GUI, while another might start only a command prompt.

There are usually seven run levels under Linux, numbered from 0 to 6. Not all of them do something interesting. On Ubuntu, run levels 2 through 5 are all the same. Each runs the GUI. Run level 1 runs a command prompt, so it might seem ideal for booting into the shell, but it also shuts down a few essential services. This means it isn't suitable for day-to-day use.

The trick is to take one of the existing run levels and alter it slightly so that it doesn't run a GUI by default. On many distributions, run level 3 is reserved for this purpose, so it makes sense to alter it under Ubuntu. (For what it's worth, the default Ubuntu run level is 2.)

Stopping Ubuntu from running a GUI upon booting is simply a matter of stopping the program that appears when Ubuntu boots—GDM. This provides the login window that appears and starts the whole graphical subsystem. Type the following command at the shell to remove the shortcut to GDM within the run level 3 configuration:

```
sudo rm /etc/rc3.d/S13GDM
```

After this, you'll need to tell Ubuntu to boot straight to run level 3, rather than the default of 2. You do this by creating the `/etc/inittab` file, which then becomes one of the first configuration files Ubuntu reads when booting. Issue the following command at the shell to create and then open the file in the Gedit text editor:

```
gksu gedit /etc/inittab
```

Then add the following line at the top of the file:

```
id:3:initdefault:
```

Then save the file. From now on, you'll always boot straight to a BASH prompt. To restore things to the way they were, simply delete the `/etc/inittab` file by typing the following at the prompt:

```
sudo rm /etc/inittab
```

Working with Files

So let's start actually using the shell. If you've ever used DOS, then you have a head start over most shell beginners, although you'll still need to learn some new commands. Table 13-1 shows various DOS commands alongside their Ubuntu equivalents. This table also serves as a handy guide to some BASH commands, even if you've never used DOS. In Appendix B, you'll find a comprehensive list of useful shell commands, together with explanations of what they do and examples of typical usage.

Table 13-1. *DOS Commands and Their Shell Equivalents*

Command	DOS Command	Linux Shell Command	Usage
Copy files	COPY	cp	cp <filename> <new location>
Move files	MOVE	mv	mv <filename> <new location>
Rename files	RENAME	mv	mv <old filename> <new filename> ¹
Delete files	DEL	rm	rm <filename> ²
Create directories	MKDIR	mkdir	mkdir <directory name>
Delete directories	DELTREE/RMDIR	rm	rm -rf <directory name>
Change directory	CD	cd	cd <directory name>
Edit text files	EDIT	vi	vi <filename>
View text files	TYPE	less	less <filename> ³
Print text files	PRINT	lpr	lpr <filename>
Compare files	FC	diff	diff <file1> <file2>
Find files	FIND	find	find -name <filename>
Check disk integrity	SCANDISK	fsck	fsck ⁴
View network settings	IPCONFIG	ifconfig	ifconfig
Check a network connection	PING	ping	ping <address>
View a network route	TRACERT	tracert	tracert <address>
Clear screen	CLS	clear	clear
Get help	HELP	man	man <command> ⁵
Quit	EXIT	exit	exit

¹ The BASH shell offers a rename command, but this is chiefly used to rename many files at once.

² To avoid being asked to confirm each file deletion, you can add the `-f` option. Be aware that the `rm` command deletes data instantly, without the safety net of the Recycle Bin, as with the GNOME desktop.

³ Use the cursor keys to move up and down in the document. Type `Q` to quit.

⁴ This is a system command and can be run only on a disk that isn't currently in use. To scan the main partition, you'll need to boot from the installation CD and select the rescue option. Then issue the `fsck` command.

⁵ The `info` command can also be used.

CREATING ALIASES

If you've ever used DOS, you might find yourself inadvertently typing DOS commands at the shell prompt. Some of these will actually work, because most distribution companies create command aliases to ease the transition of newcomers to Linux.

Aliases mean that whenever you type certain words, they will be interpreted as meaning something else. However, an alias won't work with any of the command-line switches used in DOS. In the long run, you should try to learn the BASH equivalents.

You can create your own command aliases quickly and simply. Just start a BASH shell and type the following:

```
alias <DOS command>='<Linux shell command>'
```

For example, to create an alias that lets you type `del` instead of `rm`, type this:

```
alias del='rm'
```

Note that the Ubuntu command must appear in single quotation marks.

To make aliases permanent, you need to add them to your `.bashrc` file.

Open the file in the Gedit text editor by typing the following:

```
gedit .bashrc
```

At the bottom of the file, add new lines for all the aliases you want to make permanent. Simply type the command shown previously. Save the file when you've finished.

Note that the aliases won't go into effect until you open a new terminal window or reboot the computer.

Listing Files

Possibly the most fundamentally useful BASH command is `ls`. This will list the files in the current directory, as shown in Figure 13-3. If you have a lot of files, they might scroll off the screen. If you're running GNOME Terminal, you can use the scroll bar on the right side of the window to view the list.

Having the files scroll off the screen can be annoying, so you can cram as many as possible onto each line by typing the following:

```
ls -m
```

The dash after the command indicates that you're using a command option. These are also referred to as command-line *flags* or *switches*. Nearly all shell commands have options like this. In fact, some commands won't do anything unless you specify various options. In the case of the `ls` command, only one dash is necessary, but some commands need two dashes to indicate an option.

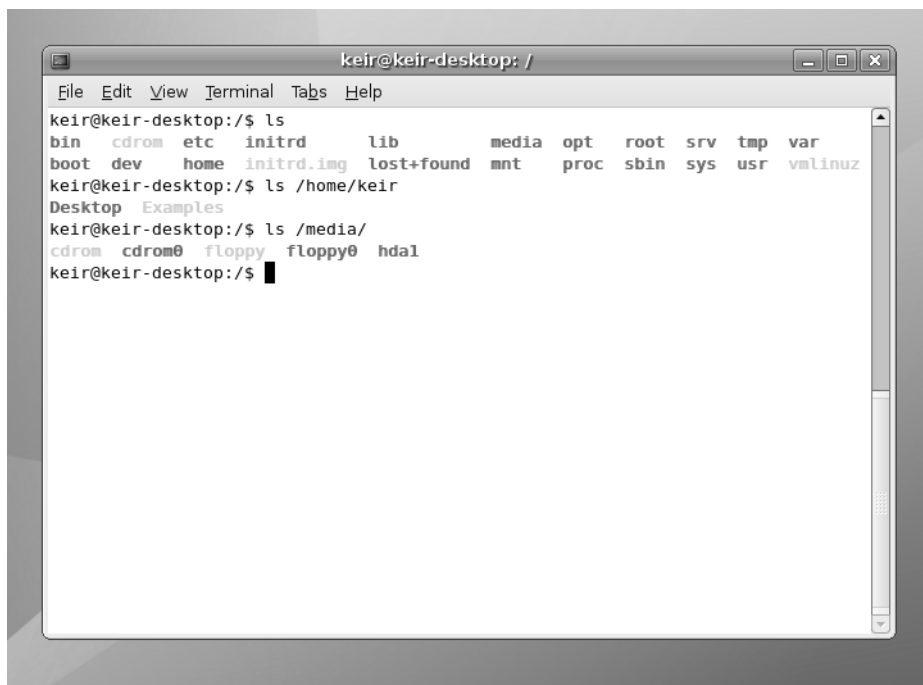


Figure 13-3. The `ls` command lists the files in the current directory.

You can see a list of all the command options for `ls` by typing the following (ironically, itself a command option):

```
ls --help
```

Once again, the output will scroll off the screen, and you can use the window’s scroll bars to examine it. (In Chapter 17, you’ll learn a trick you can use to be able to read this output without needing to fiddle around with the scroll bars, even if there’s screen after screen of it.)

With most commands, you can use many command options at once, as long as they don’t contradict each other. For example, you could type the following:

```
ls -lh
```

This tells the `ls` command to produce “long” output and also to produce “human-readable” output. The long option (`-l`) lists file sizes and ownership permissions, among other details (permissions are covered in the next chapter). The human-readable option (`-h`) means that rather than listing files in terms of bytes (such as 1029725 bytes), it will list them in kilobytes and megabytes. Notice that you can simply list the options after the dash; you don’t need to give each option its own dash.

Caution Don't forget that case-sensitivity is vitally important in Ubuntu! Typing `ls -L` is not the same as typing `ls -l`. It will produce different results.

Copying Files and Directories

Another useful command for dealing with files is `cp`, which copies files. You can use the `cp` command in the following way:

```
cp myfile /home/keir/
```

This will copy the file to the location specified.

One important command-line option for `cp` is `-r`. This stands for recursive and tells BASH that you want to copy a directory and its contents (as well as any directories within this directory). Most commands that deal with files have a recursive option.

Note Only a handful of BASH commands default to recursive copying. Even though it's extremely common to copy folders, you still need to specify the `-r` command option most of the time.

One curious trick is that you can copy a file from one place to another but, by specifying a filename in the destination part of the command, change its name. Here's an example:

```
cp myfile /home/keir/myfile2
```

This will copy `myfile` to `/home/keir`, but rename it as `myfile2`. Be careful not to add a final slash to the command when you do this. In the example here, doing so would cause BASH to think that `myfile2` is a directory.

This way of copying files is a handy way of duplicating files. By not specifying a new location in the destination part of the command, but still specifying a different filename, you effectively duplicate the file within the same directory:

```
cp myfile myfile2
```

This will result in two identical files: one called `myfile` and one called `myfile2`.

Moving Files and Directories

The `mv` command is similar to `cp`, except that rather than copying the file, the old one is removed. You can move files from one directory to another, for example, like this:

```
mv myfile /home/keir/
```

You can also use the `mv` command to quickly rename files:

```
mv myfile myfile2
```

Figure 13-4 shows the results of using `mv` to rename a file. The `mv` command can be used to move a directory in the same way as with files. However, there's no need to use a command option to specify recursivity, as with other commands.

For instance, to move the directory `daffodil` into the directory `flowers`, you could type the following (assuming both directories are in the one you're currently browsing):

```
mv daffodil/ flowers/
```

Note the use of the slash after each directory.

To rename directories, simply leave off the slashes. To rename the directory `daffodil` to `hyacinth`, for example, you could type the following:

```
mv daffodil hyacinth
```

Note Getting technical for a moment, moving a file in Linux isn't the same as in Windows, where a file is copied and then the original deleted. Under Ubuntu, the file's absolute path is rewritten, causing it to simply appear in a different place in the file structure. However, the end result is the same.

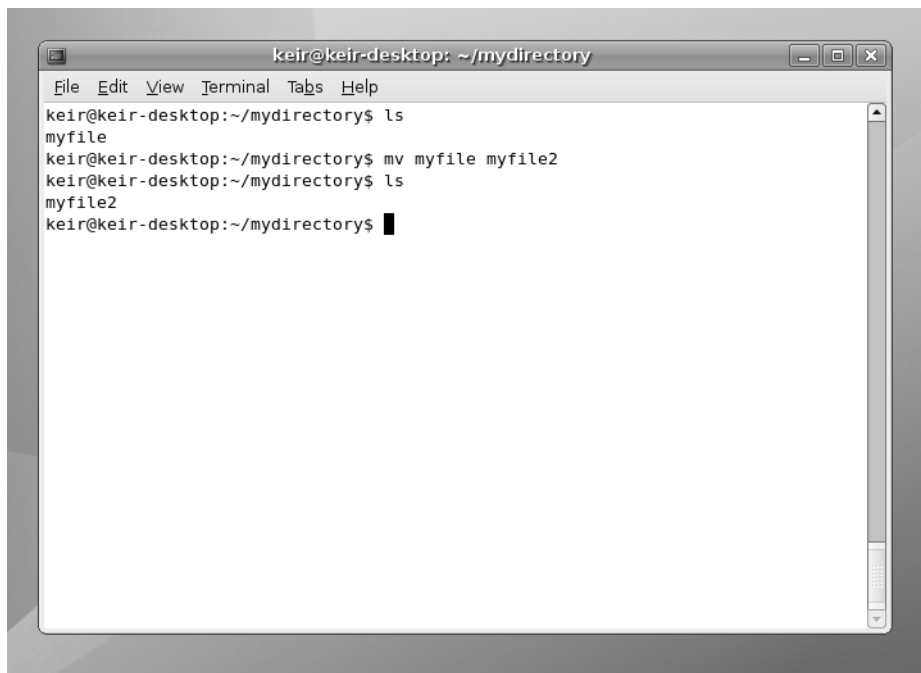


Figure 13-4. You can also use the `mv` command to rename files.

Deleting Files and Directories

But how do you get rid of files? Again, this is relatively easy, but first a word of caution: the shell doesn't operate any kind of Recycle Bin. Once a file is deleted, it's gone forever. (There are utilities you can use to recover files, but these are specialized tools and aren't to be relied on for day-to-day use.)

Removing a file is achieved by typing something like this:

```
rm myfile
```

It's as simple as that.

In some instances, you'll be asked to confirm the deletion after you issue the command. If you want to delete a file without being asked to confirm it, type the following:

```
rm -f myfile
```

The `f` stands for force (that is, force the deletion).

If you try to use the `rm` command to remove a directory, you'll see an error message. This is because the command needs an additional option:

```
rm -rf mydirectory
```

As noted earlier, the `r` stands for recursive and indicates that any folder specified afterward should be deleted, in addition to any files it contains.

Tip You might have used wildcards within Windows and DOS. They can be used within Ubuntu, too. For example, the asterisk (*) can be used to mean any file. So, you can type `rm -f *` to delete all files within a directory, or type `rm -f myfile*` to delete all files that start with the word `myfile`. But remember to be careful with the `rm` command. Keep in mind that you cannot salvage files easily if you accidentally delete them!

WORKING WITH FILES WITH SPACES IN THEM

If, at the command prompt, you try to copy, move or otherwise manipulate files that have spaces in their names, you'll run into problems. For example, suppose you want to move the file `picture from germany.jpg` to the directory `mydirectory`. In theory the following command should do the trick:

```
mv picture from germany.jpg mydirectory/
```

But when we tried it on our test Ubuntu setup, we got the following errors:

```
mv: cannot stat 'picture': No such file or directory
mv: cannot stat 'from': No such file or directory
mv: cannot stat 'germany.jpg': No such file or directory
```


In other words, BASH had interpreted each word as a separate file and tried to move each of them! The error messages tell us that BASH cannot find the file `picture`, `from`, or `germany.jpg`.

There are two solutions. The easiest is to enclose the filename in quotation marks ("), so the previous command would read as follows:

```
mv "picture from germany.jpg" mydirectory/
```

The other solution is to precede each space with a backslash. This tells BASH you're including a *literal character* in the filename. In other words, you're telling BASH not to interpret the space in the way it normally does, which is as a separator between filenames or commands. Here's how the command looks if you use backslashes:

```
mv picture\ from\ germany.jpg mydirectory/
```

The backslash can also be used to stop BASH from interpreting other symbols in the way it normally does. For example, the less than and greater than symbols (<>) have a specific meaning in BASH, which we discuss in Chapter 17, but they're allowed in filenames. So to copy the file `<bach>.mp3` to the directory `mydirectory`, you could type:

```
cp \<bach\>.mp3 mydirectory/
```

Generally speaking, however, simply enclosing filenames in quotation marks is the easiest approach.

Changing and Creating Directories

Another handy command is `cd`, for change directory. This lets you move around the file system, from directory to directory. Say you're in a directory that has another directory in it, named `mydirectory2`. Switching to it is easy:

```
cd mydirectory2
```

But how do you get out of this directory once you're in it? Try the following command:

```
cd ..
```

The `..` refers to the "parent" directory, which is the one containing the directory you're currently browsing. Using two dots to indicate this may seem odd, but it's just the way that Ubuntu (and Unix before it) does things. It's one of the many conventions that Unix relies on and that you'll pick up as you go along.

You can create directories with the `mkdir` command:

```
mkdir mydirectory
```

What if you want to create a new directory and, at the same time, create a new directory to contain it? Simply use the `-p` command option. The following command will create a new folder called `flowers` and, at the same time, create a directory within `flowers` called `daffodil`:

```
mkdir -p flowers/daffodil
```

Summary

This chapter introduced the command-line shell, considered by many to be the heart of Linux. We've discussed its similarities to MS-DOS, and shown that these are only cursory; knowledge of DOS doesn't equate to skill within BASH. In the long run, you should work to polish your BASH skills.

This chapter also introduced some elementary commands used within BASH, such as those used to provide directory listings and to copy files. We looked at how you can use command-line options to control BASH tools. In many cases, these are mandatory, so you learned how the BASH shell itself can be used to investigate a command and find out vital information about how it works.

At this point, your newfound knowledge will have no doubt caused you to venture into the Ubuntu file system itself, which can be a confusing, if not terrifying, place for the inexperienced. But don't worry. The next chapter explains everything you need to know about the file system and what you'll find in it.