



Understanding Linux Files and Users

Most of us are used to dealing with files—the things that live on our hard disks, floppies, and DVD-ROMs, and contain data and program code. It should come as no surprise that Linux has its own file structure, which is different from Windows, in terms of where data is stored and also the underlying technology.

Taking a page from Unix, Ubuntu takes the concept of the file system to an extreme. To Ubuntu, almost everything is treated as a file: your PC's hardware, network computers connected to your PC, information about the current state of your computer—almost everything finds a home within the Linux file system.

Linux places an equal emphasis on the users of the system. They own the various files and can decide who can and cannot access various files that they create or that are transferred to their ownership.

In this chapter, we'll delve into users, files, and permissions. You'll be introduced to how Ubuntu handles files and how files are tied into the system of user accounts.

Real Files and Virtual Files

Linux sees virtually everything as a series of files. This might sound absurd and certainly requires further explanation.

Let's start with the example of plugging in a piece of hardware. Whenever you attach something to a USB socket, the Linux kernel finds it, sees if it can make the hardware work, and if everything checks out okay, it will usually make the hardware available as a file under the `/dev` directory on your hard disk (`dev` is short for devices). Figure 14-1 shows an example of a `/dev` directory.

The file created in the `/dev` directory is not a real file, of course. It's a file system shortcut plumbed through to the input and output components of the hardware you've just attached.

Note As a user, you're not expected to delve into the `/dev` directory and deal with this hardware directly. Most of the time, you'll use various software packages that will access the hardware for you or use special BASH commands or GUI programs to make the hardware available in a more accessible way for day-to-day use.

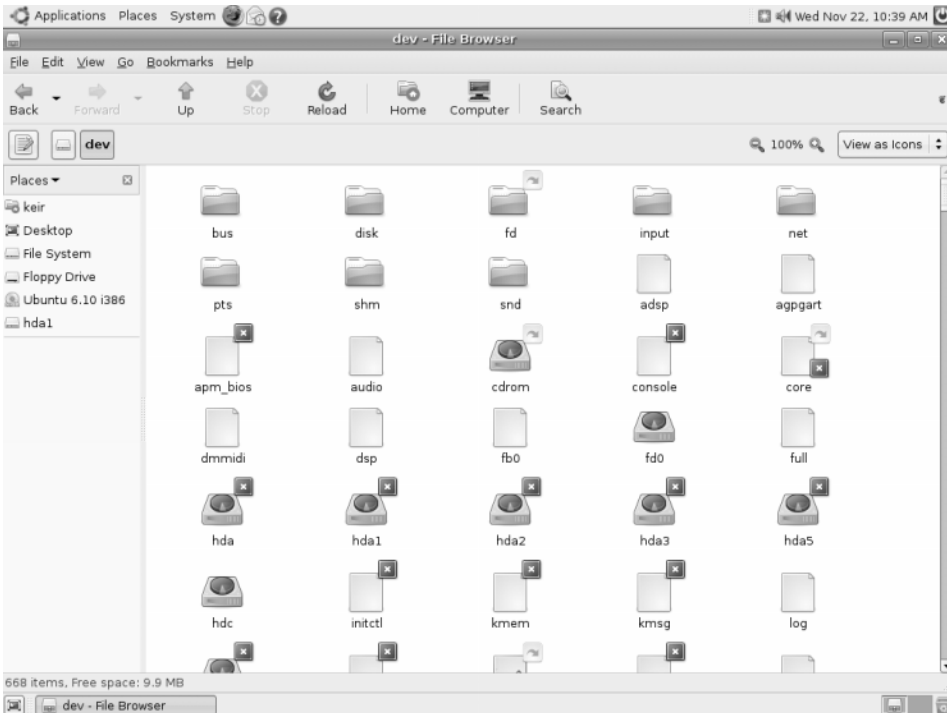


Figure 14-1. Hardware devices under Linux are accessed as if they were files and can be found in the `/dev` folder.

Here's another example. Say you're working in an office, and you want to connect to a central file server. To do this under Linux, you must *mount* the files that the server offers, making it a part of the Ubuntu file system. Doing this involves creating an empty directory (or using one that already exists) and using the `mount` command at the BASH shell to make the server's contents magically appear whenever that directory is accessed. We'll discuss how this is done later in this chapter, in the "Mounting" section.

Note Bear in mind that, in most cases, Ubuntu takes care of mounting automatically, as discussed in Chapter 12. For example, when you try to connect to a shared folder by clicking Places ► Network Servers, Ubuntu will automatically handle the mounting of the shared folder.

Once the network server is mounted, it is treated exactly like a directory on your hard disk. You can copy files to and from it, just as you would normally, using the same tools as you use for dealing with any other files. In fact, less knowledgeable users won't even be aware that they're accessing something that isn't located on their PC's hard disk (or, technically speaking, within their Ubuntu partition and file system).

By treating everything as a file, Linux makes system administration easier. To probe and test your hardware, for example, you can use the same tools you use to manipulate files.

So how do you know which files are real and which are virtual? One method is to use the following command, which was introduced in the previous chapter:

```
ls -l
```

The `-l` option tells the `ls` command to list nearly all the details about the files. If you do this in GNOME Terminal, you'll see that the listing is color-coded. There are many different combinations of colors, but Table 14-1 shows some typical examples that you're likely to come across.

The `ls -l` command returns a lot of additional information, including who owns which file and what you and others can do with it. This requires an understanding of users and file permissions, which we'll discuss next.

Tip The command `ls -la` will give you even more information—perhaps too much for general use. In most instances, `ls -l` should show enough information.

Table 14-1. *Color-Coding Within GNOME Terminal*

Color	Type of File
Black text	Standard file
Light-blue text	Directory
Black outline with yellow text	Virtual device ¹
Green text	Program or script ²
Cyan text	Symbolic link to another file ³
Pink text	Image file
Red text	Archive ⁴

¹ This is found only in the `/dev` directory.

² Technically speaking, green text indicates a program or script that has merely been marked as being executable.

³ This is similar to a Windows desktop shortcut.

⁴ Installation files are also marked red, because they're usually contained in archives.

Users and File Permissions

The concept of users and permissions is as important to Ubuntu as the idea of a central and all-encompassing file system. In fact, the two are implicitly linked.

When initially installing Linux, you should have created at least one user account. By now, this will have formed the day-to-day login that you use to access Linux and run programs.

Although you might not realize it, as a user, you also belong to a group. In fact, every user on the system belongs to a group. Under Ubuntu, ordinary users belong to a group based on their usernames (under other versions of Linux, you might find that you belong to a group called users).

Note Groups are yet another reminder of Ubuntu's Unix origins. Unix is often used on huge computer systems with hundreds or thousands of users. Putting each user into a group makes the system administrator's job a lot easier. When controlling system resources, the administrator can control groups of users rather than hundreds of individual users. On most home user PCs, the concept of groups is a little redundant, because there's normally a single user, or at most, two or three. However, the concept of groups is central to the way that Linux handles files.

A standard user account under Ubuntu is normally limited in what it can do. As a standard user, you can save files to your own private area of the disk, located in the `/home` directory, but usually nowhere else. You can move around the file system, but some directories are strictly out of bounds. In a similar way, some files can be opened as read-only, so you cannot save changes to them. All of this is enforced using file permissions.

Every file and directory is owned by a user. In addition, files and directories have three separate settings that indicate who within the Linux system can read them, who can write to them, and, if the file in question is "runnable" (usually a program or a script), who can run it (execute it). In the case of directories, it's also possible to set who can browse them, as well as who can write files to them. If you try to access a file or directory for which you don't have permission, you'll be turned away with an "access denied" error message.

ROOT VS. SUDO

Most versions of Linux have two types of user accounts: standard and root. Standard users are those who can run programs on the system but are limited in what they can do. Root users have complete run of the system, and as such, are often referred to as "superusers." They can access and/or delete whatever files they want. They can configure hardware, change settings, and so on.

Most versions of Linux create a user account called `root` and let users log in as `root` to perform system maintenance. However, for practical as well as security reasons, most of the time the user is logged in as a standard user.

Ubuntu is different in that it does away with the `root` account. Instead, it allows certain users, including the one created during installation, to temporarily adopt root-like powers. You will already have encountered this when configuring hardware. As you've seen, all you need to do is type your password when prompted in order to administer the system.

This way of working is referred to as *sudo*, which is short for “superuser do.” In fact, the command `sudo` will let you adopt root powers at the shell prompt—simply preface any command with `sudo` in order to run it with root privileges. (A different command is normally used if you want to run graphical applications from the shell prompt—`gksu`. However, the effect is the same.) Ubuntu remembers when you last used `sudo` too, so that it won't annoy you by asking you again for your password within 15 minutes of its first use. You can avoid this grace period by typing `sudo -k`.

In some ways, the `sudo` system is slightly less secure than using a standard root account. But it's also a lot simpler. It reduces the chance of serious errors, too. Any command or tweak that can cause damage will invariably require administrative powers, and therefore require you to type your password or preface the command with `sudo`. This serves as a warning and prevents mistakes.

If you're an experienced Linux user and want to invoke the root account, simply type the following at the command prompt:

```
sudo passwd root
```

Then, type a password. If you subsequently want to deactivate the root account, type this:

```
sudo passwd -l root
```

If you ever want to slip into the root account for a short period, even if you haven't followed the previous instructions to activate the root account login, you can do so by typing the following:

```
sudo su
```

You'll be prompted to type your password; do so. When you've finished, type `exit` to return to your standard user account.

Viewing Permissions

When you issue the `ls -l` command, each file is listed on an individual line. Here's an example of one line of a file listing from our test PC:

```
-rw-r--r--  2 keir keir 673985982 2006-11-31 17:19 myfile
```

The `r`, `w`, and `-` symbols on the very left of the listing indicate the file permissions. The permission list usually consists of the characters `r` (for read), `w` (for write), `x` (for execute), or `-` (meaning none are applicable).

They're followed by a number indicating the link count, which indicates how many hard/soft links have been made to the file, but you can ignore this for the moment (for more information about file links, see the "Creating File Links" sidebar).

After this is listed the owner of the file (`keir` in the example) and then the group that also has permission to access the file (in this case, the group is also called `keir`). This is followed by the file size (in bytes), the date and time the file was last accessed, and finally, the filename itself appears.

The file permissions part of the listing might look confusing, but it's actually quite simple. To understand what's going on, you need to split it into groups of four, as illustrated in Figure 14-2.

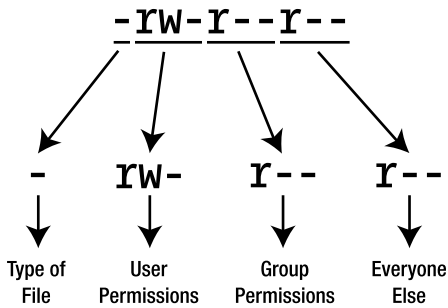


Figure 14-2. The file permissions part of a file listing can be broken down into four separate parts.

The four groups are as follows:

Type of file: This character represents the file type. A standard data file is indicated with a dash (`-`). Most files on your system fall into this category. A `d` shows that the entry is not a file but a directory. Table 14-2 lists the file type codes.

User permissions: Next come the permissions of the person who owns the file. The three characters indicate what the person who owns the file can do with it. The owner of a file is usually the user who created it, although it's also possible to change the owner later on. In this example, you see `rw-`. This means that the owner of the file can read (`r`) and write (`w`) the file. In other words, he can look at it and also save changes to it. However, there's a dash afterward, and this indicates that the user cannot execute the file. If this were possible, there would be an `x` in this spot instead.

Group permissions: After the owner's permissions are the permissions given to the specified group. This is indicated by another three characters in the same style as those for user permissions. In the example, the group's permission is `r--`, which means that the members of the specified group can read the file but don't have permission to write to it, since there's a dash where the `w` would normally appear. In other words, as far as they're concerned, the file is read-only.

Everyone else's permissions: The last set of permissions indicates the permissions of everyone else on the system (other users in other groups). In the example, they can only read the file (`r`); the two dashes afterward indicate that they cannot write to the file nor execute it.

Table 14-2. *File Type Codes*

Code	File Type
-	Standard file
d	Standard directory
l	Symbolic link (a shortcut to another file)
p	Named pipe (a file that acts as a conduit for data between two programs)
s	Socket (a file designed to send and receive data over a network)
c	Character device (a hardware device driver, usually found in <code>/dev</code>)
b	Block device (a hardware device driver, usually found in <code>/dev</code>)

As you might remember from Windows, programs are stored as files on your hard disk, just like standard data files. On Linux, program files need to be explicitly marked as being executable. This is indicated in the permission listing by an `x`. Therefore, if there's no `x` in a file's permissions, it's a good bet that the file in question isn't a program or script (although this isn't always true for various technical reasons).

To make matters a little more confusing, if the entry in the list of files is a directory (indicated by a `d`), then the rules are different. In this case, an `x` indicates that the user can access that directory. If there's no `x`, then the user's attempts to browse to that directory will be met with an "access denied" message.

File permissions can be difficult to understand, so let's look at a few real-world examples. These examples assume that you're logged in to Linux as the user `keir`.

LESS COMMON FILE PERMISSIONS

Instead of the `x` or dash in the list of permissions for a directory, you might sometimes see a `t`. This is referred to as the “sticky bit” and means that the only people who can delete or alter a file in that directory are the users who created the file in the first place. This is a useful option to have in some circumstances. It’s used with the `/tmp` (temporary) folder, for example, to ensure that one user can’t delete another user’s temporary files but is able to delete his own temporary files. To set the sticky bit for a directory, type `chmod +t directoryname`.

You might sometimes see a set of permissions like `rxs`. The `s` stands for “set user id” and is often referred to as the “suid bit”. Like `x`, it indicates that the file is executable, except in this case, it means that the file will be run with the permissions of the person who owns it, rather than the user who is executing it. In other words, if user `frank` tries to run a program owned by `keir` that has the execute permission set as `s`, that program will be run as if `keir` were running it. This is very useful, because it can make programs that require root powers usable by ordinary users, although this brings with it obvious security risks.

To set the suid bit, type `chmod +s filename`. However, it’s very unlikely you’ll ever need to use this command.

Typical Data File Permissions

Here’s the first example:

```
-rw-rw---- 2 keir keir 1450 2006-07-07 09:19 myfile2
```

You see immediately that this file is owned by user `keir`, because that username appears directly after the permissions. You also see that the group `keir` has access to the file, although precisely how much depends on the permissions.

Reading the file permissions from left to right, you see that the initial character is a dash. That indicates that this is an ordinary file and has no special characteristics. It’s also not a directory.

After that is the first part of the permissions, `rw-`. These are the permissions for the owner of the file, `keir`. You’re logged in as that user, so this file belongs to you, and these permissions apply to you. You can read and write the file but not execute it. Because you cannot execute the file, you can infer that this is a data file rather than a program (there are certain exceptions to this rule, but we’ll ignore them for the sake of simplicity).

Following this is the next part of the file permissions, `rw-`. This tells you what members of the group called `keir` can do with the file. It’s fairly useless information if you’re the only user of your PC, but for the record, you’re told that anyone else belonging to the group called `keir` can also read and write the file but not execute it. If you’re not the only user of a computer, group permissions can be important. The “Altering Permissions” section,

coming up shortly, describes how to change file permissions to control who can access files.

Finally, the last three characters tell you the permissions of everyone else on the system. The three dashes (---) mean that they have no permissions at all regarding the file. There's a dash where the *r* normally appears, so they cannot even read it. The dashes afterward tell you they cannot write to the file or execute it. If they try to do anything with the file, they'll get a "permission denied" error.

Permissions on a User's Directory

Here's example number two:

```
drwxr-xr-x  7 keir  keir  824  2006-07-07  10:01  mydirectory
```

The list of permissions starts with *d*, which tells you that this isn't a file but a directory. After this is the list of permissions for the owner of the directory (*keir*), who can read files in the directory and also create new ones there. The *x* indicates that you can access this directory, as opposed to being turned away with an "access denied" message. You might think being able to access the directory is taken for granted if the user can read and write to it, but that's not the case.

Next are the permissions for the group members. They can read files in the directory but not write any new ones there (although they can modify files already in there, provided the permissions of the individual files allow this). Once again, there's an *x* at the end of their particular permission listing, which indicates that the group members can access the directory.

Following the group's permissions are those of everyone else. They can read the directory and browse it, but not write new files to it, as with the group users' permissions.

Permissions on a Directory Owned by Root

Here's the last example:

```
drwx----- 25 root  root 1000 2004-08-06 15:44 root
```

You can see that the file is owned by *root*. Remember that in this example, you're logged in as *keir* and your group is *keir*.

The list of permissions starts with a `d`, so you can tell that this is actually a directory. After this, you see that the owner of the directory, `root`, has permission to read, write, and access the directory.

Next are the permissions for the group: three dashes. In other words, members of the group called `root` have no permission to access this directory in any way. They cannot browse it, create new files in it, or even access it.

Following this are the permissions for the rest of the users. This includes you, because you're not the user `root` and don't belong to its group. The three dashes means you don't have permission to read, write, or access this directory. In other words, it's out of bounds to you, probably because it contains files that only the `root` user should access!

SWITCHING USERS

If you have more than one user set up on your system, it's possible to switch users on the fly while you're working at the shell. On our test PC, we have an additional user account called `frank`. While logged in as any user, we can temporarily switch to this user by typing the following command, which stands for substitute user:

```
su frank
```

We'll then be asked for user `frank`'s password. Once this is typed, we will effectively have logged in as user `frank`. Any files we create will be saved with `frank`'s ownership.

If you created a root account (by using the command `sudo passwd root`), you can temporarily switch into it by typing just `su`, without any username afterward.

To return to your own account from any other account, type `exit`.

Altering Permissions

You can easily change permissions of files and directories by using the `chmod` command. For example, if you want to change a file so that everyone on the system can read and write to it, type the following:

```
chmod a+rw myfile
```

In other words, you're adding read and write (`rw`) permissions for all users (`a`), including the owner, the group, and everybody else. Here's another example:

```
chmod a-w myfile
```

This tells Linux that you want to take away (-) the ability of all users (`a`) to write (`w`) to the file. However, you want to leave the other permissions as they are.

Tip If you leave out the `a`, `chmod` assumes you mean “all”. In other words, commands like `chmod a+r myfile` and `chmod +r myfile` do the same thing.

If you specify `u`, you can change permissions just for the owner (`u` is for “user”, which is the same as “owner”):

```
chmod u+rw
```

This will add (+) read/write (`rw`) permissions for the owner.

As you might already have guessed, you can substitute a `g` to change group permissions:

```
chmod g-rw
```

This will configure the file so that members of the group that owns the file can’t read or write to it. Using an `o`, which is for “others”, will configure the file permissions for those who aren’t the owner of the file or who are not in the group that owns the file—the last three digits of the permission list.

A typical day-to-day use of `chmod` is in making a program file that you’ve downloaded executable. Because of the way the Internet works, if you download a program to install on your computer, it can lose its executable status while in transit. In this case, issue the following command:

```
chmod u+x myprogram
```

This will configure the file so that the owner (`u`) can execute (`x`) it.

Changing the Ownership of a File

To change the owner of a file, use the `chown` command. For security reasons, this must be prefaced with the `sudo` command, which is to say that `chown` and `chgrp` (to change the group ownership) require superuser powers.

For example, to set the owner of `myfile` as `frank`, type this command:

```
sudo chown frank myfile
```

You can also change the owner *and* the group of a file using `chown`. Simply type each separated by a period:

```
sudo chown frank.mygroup myfile
```

This will change `myfile` so that its owner is `frank` and its group is `mygroup`.

To change just the group of a file, you can use the `chgrp` command in exactly the same way as `chown`:

```
sudo chgrp mygroup myfile
```

CREATING FILE SHORTCUTS

We touched upon the idea of file system shortcuts in Chapter 12, when we discussed creating launchers on the GNOME desktop. The problem with launchers is that they are only recognized within GNOME. In other words, they mean nothing when you're using the command prompt (or virtually every other program that loads/saves files, with the exception of some programs created specially for the GNOME desktop environment).

The Ubuntu file system offers two types of genuine shortcuts, which it refers to as *file links*. They are *symbolic links* and *hard links*. Both are created using the `ln` command.

Symbolic links are the most commonly used. A symbolic link is the most similar to a Windows shortcut in that a small file is created that “points toward” another file. Unlike a Windows shortcut, the symbolic link file exists at the file system level, so it can't be viewed in a text editor, for example.

You can spot a symbolic link in a file listing, because it will be followed by an arrow and then the name and path (if necessary) of the file it links to. For example, in your `/home` directory, the directory `Examples` is symbolically linked to `/usr/share/example-content`. When you type `ls -l`, it appears as follows:

```
Examples -> /usr/share/example-content
```

A hard link is more complex and needs some understanding of how files work. In simple terms, all files consist of a pointer and actual data. As you might expect, the pointer tells the file system where on the disk to find the data. Creating a hard link effectively creates an additional pointer to the data that has exactly the same attributes as the original pointer, except with a different name. Performing any operation on the linked file will perform that operation on the original file. Additionally, there will be no obvious sign the hard link isn't actually a genuine file, apart from the fact that the *link count*—the number after the file permissions—will be more than 1. This indicates that more than one file *links* to the data. Maybe now you can see why people prefer to use the more obviously detectable symbolic links!

To create a symbolic link, the `-s` command option is used with the `ln` command. First, specify the original file and then the new link's name. Here's an example, followed immediately by the output of the `ls -l` command, which shows the results:

```
ln -s original_file link
ls -l
lrwxrwxrwx 1 keir keir 13 2006-11-22 12:05 link -> original_file
-rw-r--r-- 1 keir keir 0 2006-11-21 15:30 original_file
```

The new link has odd file permissions. It claims to have read/write/execute permissions for everybody (`lrwxrwxrwx`) but actually, because it's a link, it mirrors the permissions of the file it links to. So if you attempt to access a shortcut that links to a file you don't have permission to access, you'll see the appropriate error message.

To create a hard link, simply use `ln` on its own:

```
ln original_file link
```

As mentioned, apart from the link count, there will be no obvious sign the new link is, in fact, a link:

```
-rw-r--r--  2 keir keir  0 2006-11-21 15:30 original_file
-rw-r--r--  2 keir keir  0 2006-11-21 15:30 link
```

The hard link adopts all the properties of the file, including its permissions and date/time of creation. It even has the same link count!

The File System Explained

Now that you understand the principles of files and users, we can take a bird's-eye view of the Linux file system and start to make sense of it.

You might already have ventured beyond the /home directory and wandered through the file system. You no doubt found it thoroughly confusing, largely because it's not like anything you're used to. The good news is that it's not actually very hard to understand. If nothing else, you should be aware that nearly everything can be ignored during everyday use.

Note The Ubuntu file system is referred to as a *hierarchical* file system. This means that it consists of a lot of directories that contain files. Windows also uses a hierarchical file system. Ubuntu refers to the very bottom level of the file system as the root. This has no connection with the root user.

You can switch to the root of the file system by typing the following shell command:

```
cd /
```

When used on its own, the forward slash is interpreted as a shortcut for root.

If we do this on our PC and then ask for a long file listing (`ls -l`), we see the following:

```
total 108
drwxr-xr-x  2 root root  4096 2006-11-14 04:29 bin
drwxr-xr-x  3 root root  4096 2006-11-14 04:27 boot
lrwxrwxrwx  1 root root    11 2006-11-14 04:20 cdrom -> media/cdrom
drwxr-xr-x 12 root root 13500 2006-11-22 05:44 dev
drwxr-xr-x 102 root root  4096 2006-11-22 11:35 etc
drwxr-xr-x  3 root root  4096 2006-11-14 04:26 home
drwxr-xr-x  2 root root  4096 2006-10-25 09:26 initrd
lrwxrwxrwx  1 root root    33 2006-11-14 04:27 initrd.img -> boot/ ➡
```

```

initrd.img-2.6.17-10-generic
drwxr-xr-x 17 root root 4096 2006-11-14 04:29 lib
drwxr-xr-x 2 root root 49152 2006-11-14 04:20 lost+found
drwxr-xr-x 5 root root 4096 2006-10-25 09:26 media
drwxr-xr-x 2 root root 4096 2006-10-19 18:49 mnt
drwxr-xr-x 2 root root 4096 2006-10-25 09:26 opt
dr-xr-xr-x 94 root root 0 2006-11-22 00:43 proc
drwxr-xr-x 7 root root 4096 2006-11-22 11:14 root
drwxr-xr-x 2 root root 4096 2006-11-14 04:28/sbin
drwxr-xr-x 2 root root 4096 2006-10-25 09:26 srv
drwxr-xr-x 11 root root 0 2006-11-22 00:43 sys
drwxrwxrwt 12 root root 4096 2006-11-22 11:00 tmp
drwxr-xr-x 11 root root 4096 2006-10-25 09:26 usr
drwxr-xr-x 15 root root 4096 2006-10-25 09:39 var
lrwxrwxrwx 1 root root 30 2006-11-14 04:27 vmlinuz -> boot/ ➔
vmlinuz-2.6.17-10-generic

```

The first thing you'll notice from this is that the root of the file system contains largely directories and that all files and directories are owned by root.

Only users with administrative powers can write files to the root of the file system. That means if you wanted to write to the root of the file system or otherwise access those files, you would need to use the `sudo` command. This is to prevent damage, since most of the directories in the root of the file system are vital to the correct running of Linux and contain essential programs or data.

Caution It's incredibly easy to slip up when using the command-line shell and thereby cause a lot of damage. For example, simply mistyping a forward slash in a command can mean the difference between deleting the files in a directory and deleting the directory itself. This is just another reason why you should always be careful when working at the command line, especially if you use the `sudo` command.

As you can see from the file permissions of each directory in the root of the file system, most directories allow all users to browse them and access the files within (the last three characters of the permissions read `r-x`). You just won't be able to write new files there or delete the directories themselves. You might be able to modify or execute programs contained within the directory, but this will depend on the permissions of each individual file.

Table 14-3 provides a brief description of what each directory and file in the Ubuntu root file system contains. This is for reference only; there's no need for you to learn this

information. The Ubuntu file system broadly follows the principles in the Filesystem Hierarchy Standard, as do most versions of Linux, but it does have its own subtleties.

Table 14-3. *Directories and Files in the Ubuntu Root File System*

Directory	Contents
bin	Vital tools necessary to get the system running or for use when repairing the system and diagnosing problems
boot	Boot loader programs and configuration files (the boot loader is the menu that appears when you first boot Linux)
cdrom -> media/cdrom	Symbolic link (shortcut) to the entry for the CD- or DVD-ROM drive in the /dev folder (accessing this file will let you access the CD- or DVD-ROM drive)
dev	Virtual files representing hardware installed on your system
etc	Central repository of configuration files for your system
home	Where each user's personal directory is stored
initrd	Used during booting to mount the initial ramdisk
initrd.img -> boot/ initrd.img-2.6.17-10-generic	Symbolic link to the initial ramdisk, which is used to boot Linux
lib	Shared system files used by Linux as well as the software that runs on it
lost+found	Folder where salvaged scraps of files are saved in the event of a problematic shutdown and subsequent file system check
media	Where the directories representing various mounted storage systems are made available (including Windows partitions on the disk)
mnt	Directory in which external file systems can be temporarily mounted
opt	Software that is theoretically optional and not vital to the running of the system (many software packages you use daily can be found here)
proc	Virtual directory containing data about your system and its current status
root	The root user's personal directory
sbin	Programs essential to administration of the system
srv	Configuration files for any network servers you might have running on your system
sys	Mount point of the sysfs file system, which is used by the kernel to administer your system's hardware
tmp	Temporary files stored by the system

Table 14-3. *Directories and Files in the Ubuntu Root File System (Continued)*

Directory	Contents
usr	Programs and data that might be shared with other systems (such as in a large networking setup with many users) ¹
var	Used by the system to store data that is constantly updated, such as printer spooling output
vmlinux -> boot/ vmlinux-2.6.17-10-generic	Symbolic link to the kernel file used during bootup

¹ The *usr* directory contains its own set of directories that are full of programs and data. Many system programs, such as the X11 GUI software, are located within the */usr* directory. Note that the */usr* directory is used even if your system will never act as a server to other systems.

TYPES OF FILE SYSTEMS

Linux is all about choice, and this extends to the technology that makes the file system work. Unlike with Windows, where the only choice is NTFS, Linux offers many different types of file system technology. Each is designed for varying tasks. Most are scalable, however, which means that they will work just as happily on a desktop PC as on a massive cluster of computers.

Ubuntu uses the ext3 file system. This is a popular choice among distros, and nearly all home- or office-oriented distros use it. That said, people are constantly arguing about which file system is best. The principal measuring stick is performance. Your computer spends a lot of time writing and reading files, so the faster a file system is, the faster your PC will be overall (although, in reality, the hardware is of equal importance).

Note that what we're talking about here is the underlying and invisible technology of the file system. In day-to-day use, the end user won't be aware of any difference between ext3, reiserfs, or another file system technology (although when things go wrong, different tools are used to attempt repairs; their selection is automated within Ubuntu).

Here are the various types along with notes about what they offer:

- **ext2:** Fast, stable, and well established, ext2 was once the most popular type of file system technology used on Linux. It has now been eclipsed by ext3.
- **ext3:** An extension of ext2, ext3 allows journaling, a way of recording what has been written to disk so that a recovery can be attempted when things go wrong.
- **reiserfs:** This is another journaling file system, which claims to be faster than others and also offers better security features.
- **jfs:** This is a journaling file system created by IBM. It's used on industrial implementations of Unix.
- **xfs:** This is a 64-bit journaling file system created by Silicon Graphics, Inc. (SGI) and used on its own version of Unix, as well as Linux.

Mounting

Described in technical terms, mounting is the practice of making a file system available under Linux. Whereas Windows uses drive letters to make other file systems available within Windows Explorer, Linux integrates the new file system within the root file system, usually by making the contents appear whenever a particular directory is accessed. The mounted file system can be a partition on your hard disk, a CD-ROM, a network server, or many other things.

Mounting drives might seem a strange concept, but it actually makes everything much simpler than it might be otherwise. For example, once a drive is mounted, you don't need to use any special commands or software to access its contents. You can use the same programs and tools that you use to access all of your other files. Mounting creates a level playing field on which everything is equal and can therefore be accessed quickly and efficiently.

Using the mount Command

At the command line, mounting is done via the `mount` command. Under Ubuntu, you must have administrator powers to use the `mount` command, which means prefacing it with `sudo` and providing your password when prompted.

With most modern versions of Linux, `mount` can be used in two ways: by specifying all the settings immediately after the command or by making reference to an entry within the `fstab` file. `fstab` stands for File System Table, and that gives an indication of what it's used for—it's a look-up file stored in the `/etc` directory that contains details of all file systems on the PC that are regularly mounted. Figure 14-3 shows an example of a typical `fstab` file.

Note The root file system is itself mounted automatically during bootup, shortly after the kernel has started and has all your hardware up and running. If you look within `/etc/fstab`, you'll see that it too has its own entry, as does the swap partition. Every file system that Linux uses must be mounted at some point.

Let's say that you insert a CD or DVD into your computer's DVD-ROM drive. To mount the CD or DVD and make it available to Linux (something that is actually done automatically as soon as you put a disk in the drive, so this example is for demonstration purposes only), you would type the following:

```
sudo mount /media/cdrom
```

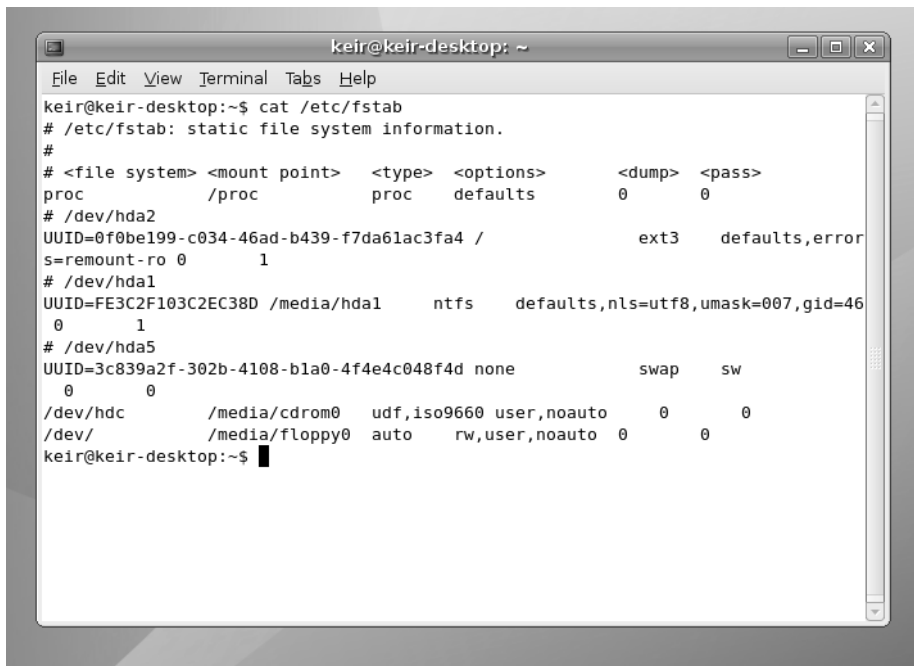
The `mount` command first looks in your `fstab` file to find what you're referring to. It then matches up the directory you've specified as the mount point against the hardware details,

in the form of a UUID number, which is then translated by Ubuntu into a file within the `/dev` directory. The two are then magically connected together.

Note that the contents of the mounted file system are made available in a virtual way. The files are not literally copied into the directory. The directory is merely a conduit that allows you to read the CD's contents.

There aren't any special commands used to work with drives that have been mounted. The shell commands discussed in Chapter 13 should do everything you need, and Nautilus will have no trouble browsing its contents.

The `mount` command doesn't see widespread usage by most users nowadays, because most removable storage devices like CDs, and even memory card readers, are mounted automatically under Ubuntu, and an icon for them appears on the desktop. However, there may be occasions when you need to mount a drive manually.



```

keir@keir-desktop: ~
File Edit View Terminal Tabs Help
keir@keir-desktop:~$ cat /etc/fstab
# /etc/fstab: static file system information.
#
# <file system> <mount point> <type> <options> <dump> <pass>
proc /proc proc defaults 0 0
# /dev/hda2
UUID=0f0be199-c034-46ad-b439-f7da61ac3fa4 / ext3 defaults,error
s=remount-ro 0 1
# /dev/hda1
UUID=FE3C2F103C2EC38D /media/hda1 ntfs defaults,nls=utf8,umask=007,gid=46
0 1
# /dev/hda5
UUID=3c839a2f-302b-4108-b1a0-4f4e4c048f4d none swap sw
0 0
/dev/hdc /media/cdrom0 udf,iso9660 user,noauto 0 0
/dev/ /media/floppy0 auto rw,user,noauto 0 0
keir@keir-desktop:~$

```

Figure 14-3. Details of all frequently mounted file systems are held in the `/etc/fstab` file.

Mounting a Drive Manually

Let's look at an example of when you might need to mount a drive manually. Suppose that you've just added a second hard disk to your PC that has previously been used on a Windows system, and you want to salvage some data before formatting the disk. Let's also assume the new disk has been added as the slave on the primary IDE channel, which is the usual method of adding a second disk to an IDE-based computer.

■ **Tip** To learn more about installing hard disks, see www.computerhope.com/issues/ch000413.htm.

Here are the steps you would typically follow:

1. The first thing to do is create a *mount point*, which is a directory that will act as a location where you can tell `mount` to make the disk accessible.

■ **Note** The mount point doesn't necessarily have to be empty or new! You can use any directory as a mount point, and as long as the file system is mounted, the original contents of the directory will be invisible. However, to avoid confusion, it's best to create a new independent mount point.

You can create the new directory anywhere, but under Ubuntu, the convention is to create it in the `/media` directory. Therefore, the following command should do the trick (note that you need to use the `sudo` command, because writing to any directory other than your `/home` directory requires administrator privileges):

```
sudo mkdir /media/newdisk
```

2. You now need to know what kind of partition type is used on the disk, because you need to specify this when mounting. To find this out, use the `fdisk` command. Type the following exactly as it appears:

```
sudo fdisk -l /dev/hdb
```

3. This will list the partitions on the second disk drive (assuming an average PC system). With most hard disks used under Windows, you should find a single partition that will be either NTFS or FAT32. The examples here assume that this is `hdb1`.

■ **Caution** Be aware that `fdisk` is a dangerous system command that can damage your system. The program is designed to partition disks and can wipe out your data if you're not careful!

4. With this information in hand, you're now ready to mount the disk. For a FAT32 disk, type the following:

```
sudo mount -t vfat -o umask=000 /dev/hdb1 /media/newdisk
```

For an NTFS disk, type the following:

```
sudo mount -t ntfs -o umask=0222 /dev/hdb1 /media/newdisk
```

The `-t` command option is used to specify the file system type. The `-o` flag indicates that you're going to specify some more command options, and you do so in the form of `umask`, which tells `mount` to ensure that the directory is readable (and writable in the case of the FAT32 drive). After this, you specify the relevant file in the `/dev` directory (this file is only virtual, of course, and merely represents the hardware), and then specify the directory that is acting as your mount point.

Note Although the `fstab` file refers to UUID numbers, for a temporary mount, it's fine to refer specifically to the hardware within the `/dev` directory.

Now when you browse to the `/mnt/newdisk` directory by typing `cd /mnt/newdisk`, you should find the contents of the hard disk accessible. You should also have found that a new icon appeared on the desktop for the file system which you can double-click to access the new disk via Nautilus.

For more information about the `mount` command, read its man page (type `man mount`).

Removing a Mounted System

To unmount a system, you use the special command `umount` (notice there's no `n` after the first `u`). Here's an example of using the command to unmount the hard disk we mounted previously:

```
sudo umount /media/newdisk
```

All you need to do is tell the `umount` command the mount point. Alternatively, you can specify the file in the `/dev` directory that refers to mounted resource, but this is a little complicated, so in most cases, it's better to simply specify the file system location of the mount.

If you're currently browsing the mounted directory, you'll need to leave it before you can unmount it. The same is true of all kinds of access to the mounted directory. If you're browsing the mounted drive with Nautilus or if a piece of software is accessing it, you won't be able to unmount it until you've quit the program and closed the Nautilus window (or browsed to a different part of the file system).

USEFUL BASIC SHELL COMMANDS

Here are some additional shell commands that you might find useful on a day-to-day basis. Don't forget you can view the man pages of these commands to learn more. Note that commands for manipulating text files are dealt with in the next chapter.

- `clear`: Clear the terminal window, and put the cursor at the top of the window.
- `date`: Display current date and time.
- `dmesg`: Show the output of the kernel, including error messages (useful for problem solving).
- `eject`: Eject a CD/DVD.
- `exit`: Log out of current user account being accessed at the command line (if issued in a terminal window, the window will close).
- `file`: Display useful information about the specified file; the filename should be stated immediately afterward (that is, `file myfile.txt`).
- `free`: Display information about memory usage; add `-m` command option to see output in megabytes.
- `halt`: Shut down the computer (needs to be run as root, so preface with `sudo`).
- `help`: Show a list of commonly used BASH commands.
- `last`: Show recent system logins.
- `pwd`: Print Working Directory; this will simply tell you the full path of where you're currently browsing.
- `reboot`: Reboot the computer (needs to be run as root, so preface with `sudo`).
- `shred`: Destroy the specified file beyond recovery by overwriting with junk data; the filename should be specified immediately afterward.
- `touch`: Give the specified file's current date and time; if the specified file doesn't exist, then create an empty file. The filename should be specified immediately afterward.
- `uptime`: Display how long the computer has been booted, plus various CPU usage statistics.
- `whatis`: Display a one-line summary of the specified command; the command name should follow immediately afterward.

File Searches

Files frequently get lost. Well, technically speaking, they don't actually get lost. We just forget where we've put them. But because of this, the shell includes some handy commands to search for files.

Using the find Command

The `find` command manually searches through all the files on the hard disk. It's not a particularly fast way of finding a file, but it is reliable.

Here's an example:

```
find /home/keir -name "myfile"
```

This will search for `myfile` using `/home/keir` as a starting point (which is to say that it will search all directories within `/home/keir`, any directories within those directories, and so on, because it's recursive). To search the entire file system, type `/` as the path. Remember that `/` is interpreted by BASH as the root of the file system.

If the file is found, you'll see it appear in the output of the command. The full path will be shown next to the filename.

If you give `find` a try, you'll see that it's not a particularly good way of searching. Apart from being slow, it will also return a lot of error messages about directories it cannot search. This is because, when you run the `find` command, it takes on your user permissions. Whenever `find` comes across a directory it cannot access, it will report it to you, as shown in the example in Figure 14-4. There are frequently so many of these warnings that the output can hide the instances where `find` actually locates the file in question!

You can avoid these error messages in various ways, but perhaps the quickest solution is to preface the `find` command with `sudo` to invoke superuser powers. In this way, you'll have access to every file on the hard disk, so the `find` command will be unrestricted in where it can search and won't run into any directories it doesn't have permission to enter.

Caution Using the `sudo` command with `find` may represent an invasion of privacy if you have more than one user on your system. The `find` command will search other users' `/home` directories and report any instances of files found there, too.

However, an even better solution for finding files is to use the `locate` command.

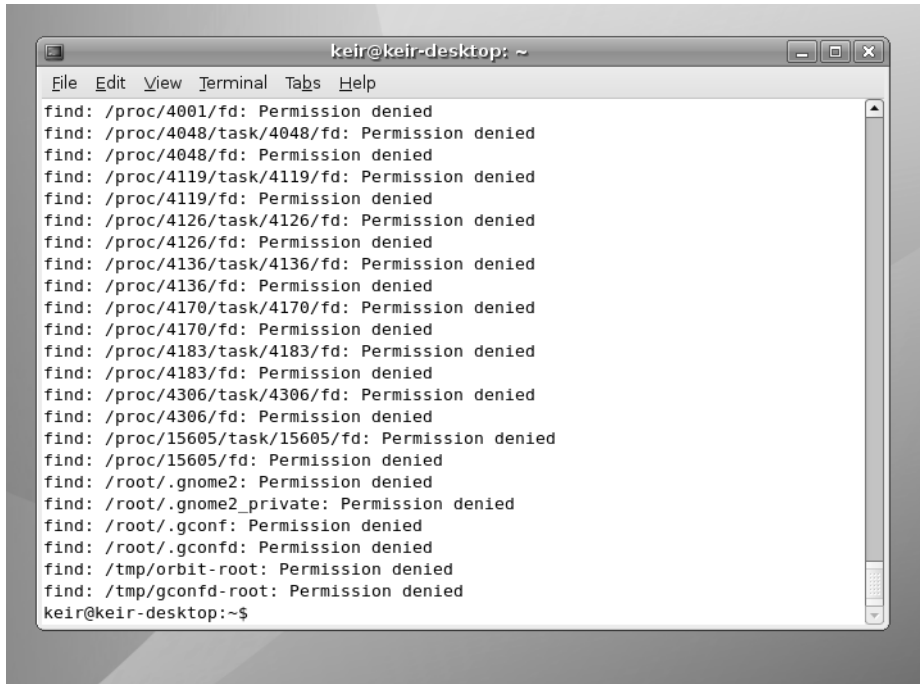


Figure 14-4. The *find* command is useful for finding files but isn't problem-free.

Using the locate Command

The alternative to using *find* is to use the *locate* command. This is far quicker than *find*, because it relies on a central database of files, which is periodically updated. In other words, it doesn't literally search the file system each time.

The problem is that if you've saved a file recently and are hoping to find it, there's a chance that it won't yet appear in *locate*'s database, so it won't turn up in the list of results.

Using *locate* is easy. You can use the following command to search for a file (you don't need to precede the command with *sudo*):

```
locate myfile
```

It's possible to update the *locate* database manually, although this might take a few minutes to work through. Simply issue the command:

```
sudo updatedb
```

After this, all files in the system should be indexed, making your search results more accurate.

Using the `whereis` Command

One other command worth mentioning in the context of searching is `whereis`. This locates where programs are stored and is an excellent way of exploring your system. Using it is simply a matter of typing something like this:

```
whereis cp
```

This will tell you where the `cp` program is located on your hard disk. It will also tell you where its source code and man page are located (if applicable). However, the first path returned by the search will be the location of the program itself.

File Size and Free Space

Often, it's necessary to know how large files are and to know how much space they're taking up on the hard disk. In addition, it's often handy to know how much free space is left on a disk.

Viewing File Sizes

Using the `ls -l` command option will tell you how large each file is in terms of bytes. Adding the `-h` option converts these file sizes to kilobytes, megabytes, and even gigabytes, depending on how large they are.

In order to get an idea of which are the largest files and which are the smallest, you can add the `-S` command option. This will order the files in the list in terms of the largest and smallest files.

The following will return a list of all the files in the current directory, in order of size (largest first), detailing the sizes in kilobytes, megabytes, or gigabytes:

```
ls -Slh
```

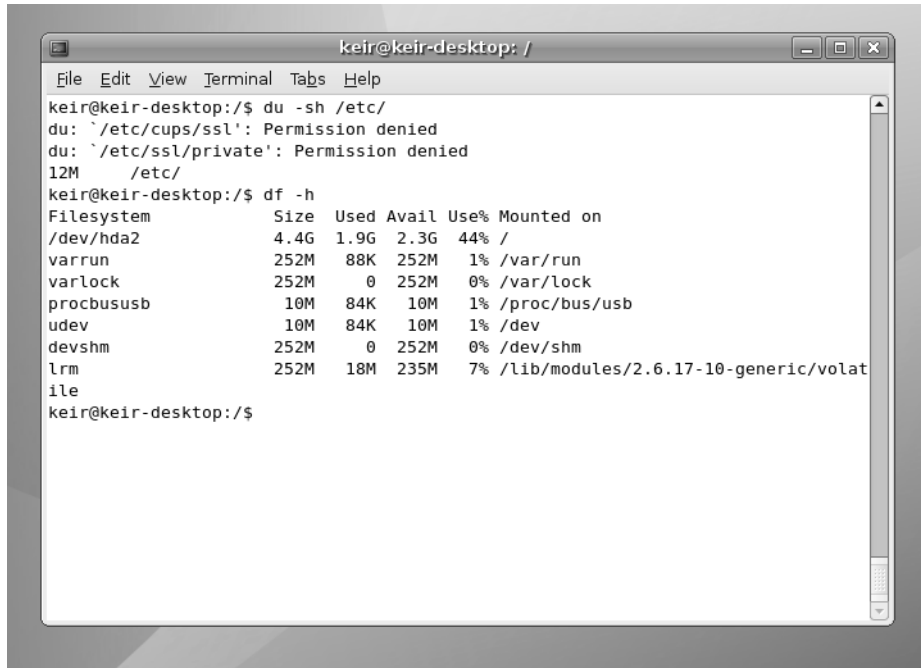
There's another, more powerful way of presenting this information: using the `du` command, which stands for "disk usage." When used on its own without command switches, `du` simply presents the size of directories alongside their names (starting in the current directory). It will show any hidden directories (directories whose names start with a period) and will also present a total at the end of the list. This will probably be quite a long list. Once again, you can add the `-h` command option to force the `du` command to produce human-readable measurements of kilobytes and megabytes.

If you specify a file or directory when using the `du` command, along with the `-s` command option, you can find out its total file size:

```
du -sh mydirectory
```


This will show the size taken up on the disk by `mydirectory`, adding to the total any files and/or subdirectories it contains.

However, `du` is limited by the same file permission problems as the `find` tool, as shown in Figure 14-5. If you run `du` as an ordinary user, it won't be able to calculate the total for any directories you don't have permission to access. Therefore, you might consider prefacing the command with `sudo`.



```

keir@keir-desktop: /
File Edit View Terminal Tabs Help
keir@keir-desktop:/$ du -sh /etc/
du: '/etc/cups/ssl': Permission denied
du: '/etc/ssl/private': Permission denied
12M    /etc/
keir@keir-desktop:/$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/hda2        4.4G   1.9G   2.3G  44% /
varrun          252M   88K   252M   1% /var/run
varlock         252M    0   252M   0% /var/lock
procbususb       10M   84K    10M   1% /proc/bus/usb
udev            10M   84K    10M   1% /dev
devshm          252M    0   252M   0% /dev/shm
lrmm             252M   18M   235M   7% /lib/modules/2.6.17-10-generic/volat
ile
keir@keir-desktop:/$

```

Figure 14-5. The `du` command shows the size of a file, and the `df` command can be used to gauge the amount of free space on the disk.

Finding Out the Amount of Free Space

What if you want to find out how much free space is left on the disk? In this case, you can use the `df` command. This command is also demonstrated in Figure 14-5.

The `df` command reports the free space in *all* mounted file systems, as well as just the root file system. This can lead to confusing results, because you'll also see results for the `/var` directory, for example, or `/dev`. You'll see results for your Windows file system too!

To make sense of it all, look under the Mounted On heading for `/`, which always indicates the root file system. If you have a memory card inserted and want to find out its free space, look under this list for its mount point (which will probably be in `/media`).

Once again, you can add the `-h` option to the `df` command to have the file sizes returned in megabytes and gigabytes (and even terabytes if your hard disk is big enough!).

Note There is as much space free in any directory as there is space on the disk, which is why `df` displays data about the entire partition. If you're using a system managed by a system administrator within a business environment, you might find that quotas have been used to limit how much disk space you can take up. However, if you're using a desktop PC and are the only user, this won't be activated.

Summary

In this chapter, we examined how the Ubuntu file system lies at the heart of an understanding of how the operating system works. We also discussed how the file system and user accounts go hand-in-hand and are inextricably linked. This involved discussing the concept of file ownership and usage permissions, plus how these can be manipulated using command-line shell tools.

We also discussed the overall structure of the Ubuntu file system and how external file systems can be mounted and made available within Ubuntu. Finally, we looked at how to find files and how to gauge how much free space there is within the file system.

In the next chapter, we'll look at how the BASH shell can be used to view and otherwise manipulate text files, which are also important to the way Ubuntu works.