Chapter 1

Starting with Linux Shells

In this Chapter

- What is Linux?
- Parts of the Linux kernel
- Exploring the Linux desktop
- Visiting Linux distributions

Before you can dive into working with the Linux command line and shells, it's a good idea to first understand what Linux is, where it came from, and how it works. This chapter walks you through what Linux is, and explains where the shell and command line fit in the overall Linux picture.

What Is Linux?

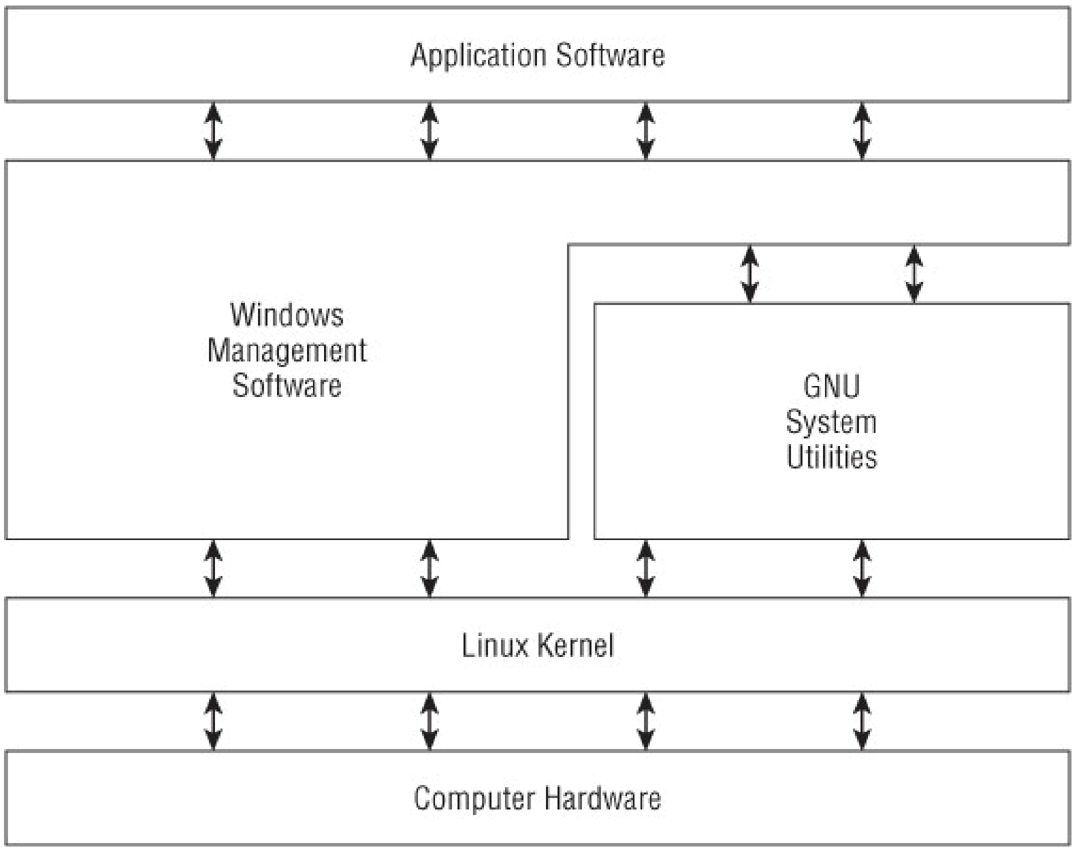
If you've never worked with Linux before, you may be confused as to why there are so many different versions of it available. I'm sure that you have heard various terms such as distribution, LiveCD, and GNU when looking at Linux packages and been confused. Wading through the world of Linux for the first time can be a tricky experience. This chapter takes some of the mystery out of the Linux system before you start working on commands and scripts.

For starters, there are four main parts that make up a Linux system:

- The Linux kernel
- The GNU utilities
- A graphical desktop environment
- Application software

Each of these four parts has a specific job in the Linux system. Each of the parts by itself isn't very useful.Figure 1.1 shows a basic diagram of how the parts fit together to create the overall Linux system.

Figure 1.1 The Linux system



This section describes these four main parts in detail, and gives you an overview of how they work together to create a complete Linux system.

Looking into the Linux Kernel

The core of the Linux system is the *kernel*. The kernel controls all of the hardware and software on the computer system, allocating hardware when necessary, and executing software when required.

If you've been following the Linux world at all, no doubt you've heard the name Linus Torvalds. Linus is the person responsible for creating the first Linux kernel software while he was a student at the University of Helsinki. He intended it to be a copy of the Unix system, at the time a popular operating system used at many universities.

After developing the Linux kernel, Linus released it to the Internet community and solicited suggestions for improving it. This simple process started a revolution in the world of computer operating systems. Soon Linus was receiving suggestions from students as well as professional programmers from around the world.

Allowing anyone to change programming code in the kernel would result in complete chaos. To simplify things, Linus acted as a central point for all improvement suggestions. It was ultimately Linus's decision whether or not to incorporate suggested code in the kernel. This same concept is still in place with the Linux kernel code, except that instead of just Linus controlling the kernel code, a team of developers has taken on the task.

The kernel is primarily responsible for four main functions:

- System memory management
- Software program management
- Hardware management
- Filesystem management

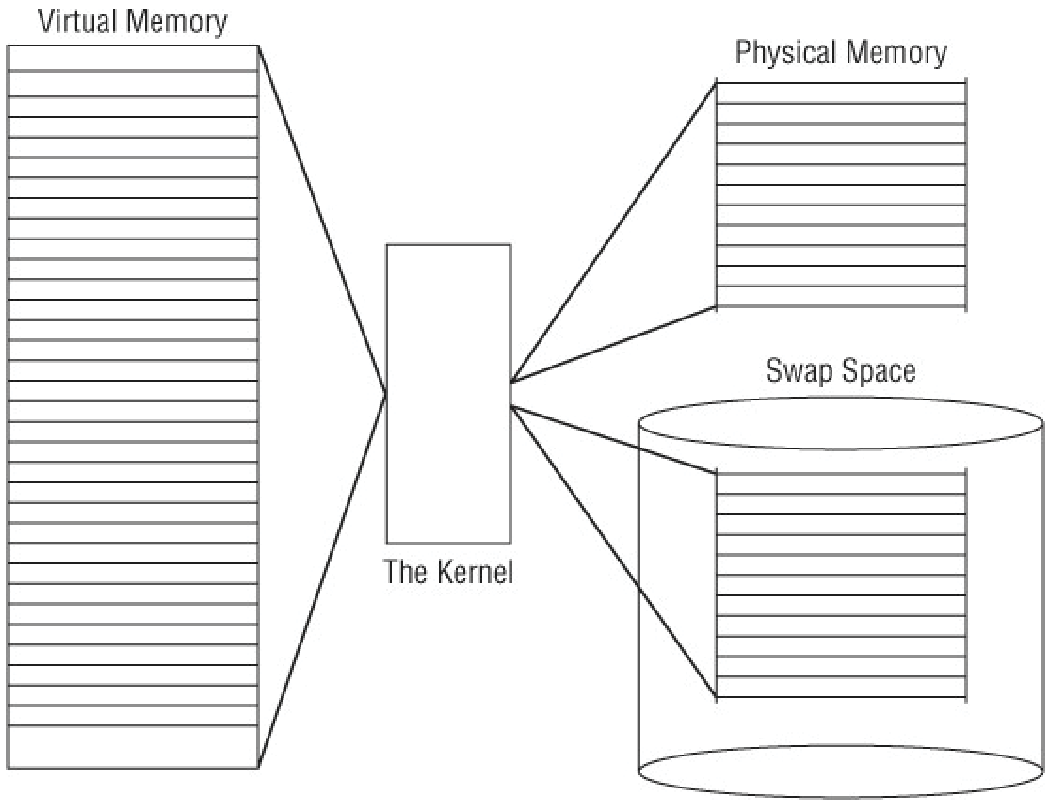
The following sections explore each of these functions in more detail.

System Memory Management

One of the primary functions of the operating system kernel is memory management. Not only does the kernel manage the physical memory available on the server, but it can also create and manage virtual memory, or memory that does not actually exist.

It does this by using space on the hard disk, called the *swap sp*ace. The kernel swaps the contents of virtual memory locations back and forth from the swap space to the actual physical memory. This allows the system to think there is more memory available than what physically exists (shown in Figure 1.2).

Figure 1.2 The Linux system memory map



The memory locations are grouped into blocks called *pages*. The kernel locates each page of memory either in the physical memory or the swap space. The kernel then maintains a table of the memory pages that indicates which pages are in physical memory and which pages are swapped out to disk.

The kernel keeps track of which memory pages are in use and automatically copies memory pages that have not been accessed for a period of time to the swap space area (called *swapping out*), even if there's other memory available. When a program wants to access a memory page that has been swapped out, the kernel must make room for it in physical memory by swapping out a different memory page, and swap in the required page from the swap space. Obviously, this process takes time, and can slow down a running process. The process of swapping out memory pages for running applications continues for as long as the Linux system is running.

You can see the current status of the virtual memory on your Linux system by viewing the special `/proc/meminfo`file. Here's an example of a sample `/proc/meminfo` entry:

```
rich@rich-desktop:~$ cat /proc/meminfo
MemTotal:        1026084 kB
MemFree:          666356 kB
Buffers:           49900 kB
Cached:           152272 kB
SwapCached:            0 kB
Active:           171468 kB
Inactive:         154196 kB
Active(anon):     131056 kB
Inactive(anon):       32 kB
Active(file):      40412 kB
Inactive(file):   154164 kB
Unevictable:          12 kB
Mlocked:              12 kB
HighTotal:        139208 kB
HighFree:            252 kB
LowTotal:         886876 kB
LowFree:          666104 kB
SwapTotal:       2781176 kB
SwapFree:        2781176 kB
Dirty:               588 kB
Writeback:             0 kB
AnonPages:        123500 kB
Mapped:            52232 kB
Shmem:              7600 kB
Slab:              17676 kB
SReclaimable:       9788 kB
SUnreclaim:         7888 kB
KernelStack:        2656 kB
PageTables:         5072 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     3294216 kB
Committed_AS:    1234480 kB
```

```
VmallocTotal:     122880 kB
VmallocUsed:        7520 kB
VmallocChunk:     110672 kB
HardwareCorrupted:     0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       4096 kB
DirectMap4k:       12280 kB
DirectMap4M:      897024 kB
rich@rich-desktop:~$
```

The `MemTotal:` line shows that this Linux server has 1GB of physical memory. It also shows that about 660MB is not currently being used (MemFree). The output also shows that there is about 2.5GB of swap space memory available on this system (SwapTotal).

By default, each process running on the Linux system has its own private memory pages. One process cannot access memory pages being used by another process. The kernel maintains its own memory areas. For security purposes, no processes can access memory used by the kernel processes.

To facilitate data sharing, you can create shared memory pages. Multiple processes can read and write to and from a common shared memory area. The kernel maintains and administers the shared memory areas and allows individual processes access to the shared area.

The special `ipcs` command allows you to view the current shared memory pages on the system. Here's the output from a sample `ipcs` command:

```
# ipcs -m


 ------ Shared Memory Segments --------
 key        shmid     owner     perms     bytes     nattch    status
 0x00000000 0         rich      600       52228     6         dest
 0x395ec51c 1         oracle    640       5787648   6


#
```

Each shared memory segment has an owner that created the segment. Each segment also has a standard Linux permissions setting that sets the availability of the segment for other users. The key value is used to allow other users to gain access to the shared memory segment.

Software Program Management

The Linux operating system calls a running program a *process*. A process can run in the foreground, displaying output on a display, or it can run in background, behind the scenes. The kernel controls how the Linux system manages all the processes running on the system.

The kernel creates the first process, called the *init process*, to start all other processes on the system. When the kernel starts, it loads the init process into virtual memory. As the kernel starts each additional process, it gives it a unique area in virtual memory to store the data and code that the process uses.

Some Linux implementations contain a table of processes to start automatically on bootup. On Linux systems, this table is usually located in the special file `/etc/inittabs`.

Other systems (such as the popular Ubuntu Linux distribution) utilize the `/etc/init.d` folder, which contains scripts for starting and stopping individual applications at boot time. The scripts are started via entries under the `/etc/rcX.d` folders, where *X* is a *run level*.

The Linux operating system uses an init system that utilizes run levels. A run level can be used to direct the init process to run only certain types of processes, as defined in the `/etc/inittabs` file or the `/etc/rcX.d`folders. There are five init run levels in the Linux operating system.

At run level 1, only the basic system processes are started, along with one console terminal process. This is called *single use*r mode. Single user mode is most often used for emergency filesystem maintenance when something is broken. Obviously, in this mode only one person (usually the administrator) can log in to the system to manipulate data.

The standard init run level is 3. At this run level, most application software such as network support software is started. Another popular run level in Linux is run level 5. This is the run level where the system starts the graphical X Window software, and allows you to log in using a graphical desktop window.

The Linux system can control the overall system functionality by controlling the init run level. By changing the run level from 3 to 5, the system can change from a console-based system to an advanced, graphical X Window system.

In Chapter 4, you'll see how to use the `ps` command to view the processes currently running on the Linux system. Here's an example of what you'll see using the `ps` command:

```
$ ps ax

 PID TTY      STAT   TIME COMMAND

  1 ?        S      0:03 init

  2 ?        SW     0:00 [kflushd]

  3 ?        SW     0:00 [kupdate]

  4 ?        SW     0:00 [kpiod]

  5 ?        SW     0:00 [kswapd]

 243 ?        SW     0:00 [portmap]

 295 ?        S      0:00 syslogd
```

```
305 ?        S      0:00 klogd
320 ?        S      0:00 /usr/sbin/atd
335 ?        S      0:00 crond
350 ?        S      0:00 inetd
365 ?        SW     0:00 [lpd]
403 ttyS0    S      0:00 gpm -t ms
418 ?        S      0:00 httpd
423 ?        S      0:00 httpd
424 ?        SW     0:00 [httpd]
425 ?        SW     0:00 [httpd]
426 ?        SW     0:00 [httpd]
427 ?        SW     0:00 [httpd]
428 ?        SW     0:00 [httpd]
429 ?        SW     0:00 [httpd]
430 ?        SW     0:00 [httpd]
436 ?        SW     0:00 [httpd]
437 ?        SW     0:00 [httpd]
438 ?        SW     0:00 [httpd]
470 ?        S      0:02 xfs -port -1
485 ?        SW     0:00 [smbd]
495 ?        S      0:00 nmbd -D
533 ?        SW     0:00 [postmaster]
538 tty1     SW     0:00 [mingetty]
539 tty2     SW     0:00 [mingetty]
540 tty3     SW     0:00 [mingetty]
541 tty4     SW     0:00 [mingetty]
542 tty5     SW     0:00 [mingetty]
543 tty6     SW     0:00 [mingetty]
544 ?        SW     0:00 [prefdm]
549 ?        SW     0:00 [prefdm]
559 ?        S      0:02 [kwm]
585 ?        S      0:06 kikbd
594 ?        S      0:00 kwmsound
```

```
  595 ?        S      0:03 kpanel
  596 ?        S      0:02 kfm
  597 ?        S      0:00 krootwm
  598 ?        S      0:01 kbgndwm
  611 ?        S      0:00 kcmlaptop -daemon
  666 ?        S      0:00 /usr/libexec/postfix/master
  668 ?        S      0:00 qmgr -l -t fifo -u
  787 ?        S      0:00 pickup -l -t fifo
  790 ?        S      0:00 telnetd: 192.168.1.2 [vt100]
  791 pts/0    S      0:00 login -- rich
  792 pts/0    S      0:00 -bash
  805 pts/0    R      0:00 ps ax
$
```

The first column in the output shows the *process ID* (or PID) of the process. Notice that the first process is our friend the init process, and assigned PID 1 by the Linux system. All other processes that start after the init process are assigned PIDs in numerical order. No two processes can have the same PID (although old PID numbers can be reused by the system after the original process terminates).

The third column shows the current status of the process (S for sleeping, SW for sleeping and waiting, and R for running). The process name is shown in the last column. Processes that are in brackets are processes that have been swapped out of memory to the disk swap space due to inactivity. You can see that some of the processes have been swapped out, but most of the running processes have not.

Hardware Management

Still another responsibility for the kernel is hardware management. Any device that the Linux system must communicate with needs driver code inserted inside the kernel code. The driver code allows the kernel to pass data back and forth to the device, acting as a middle man between applications and the hardware. There are two methods used for inserting device driver code in the Linux kernel:

- Drivers compiled in the kernel
- Driver modules added to the kernel

Previously, the only way to insert device driver code was to recompile the kernel. Each time you added a new device to the system, you had to recompile the kernel code. This process became even more inefficient as Linux kernels supported more hardware. Fortunately, Linux developers devised a better method to insert driver code into the running kernel.

Programmers developed the concept of kernel modules to allow you to insert driver code into a running kernel without having to recompile the kernel. Also, a kernel module could be removed from the kernel when the device was finished being used. This greatly simplified and expanded using hardware with Linux.

The Linux system identifies hardware devices as special files, called *device files*. There are three different classifications of device files:

- Character
- Block
- Network

Character device files are for devices that can only handle data one character at a time. Most types of modems and terminals are created as character files. Block files are for devices that can handle data in large blocks at a time, such as disk drives.

The network file types are used for devices that use packets to send and receive data. This includes network cards and a special loopback device that allows the Linux system to communicate with itself using common network programming protocols.

Linux creates special files, called nodes, for each device on the system. All communication with the device is performed through the device node. Each node has a unique number pair that identifies it to the Linux kernel. The number pair includes a major and a minor device number. Similar devices are grouped into the same major device number. The minor device number is used to identify a specific device within the major device group. The following is an example of a few device files on a Linux server:

```
rich@rich-desktop: ~$ cd /dev
rich@rich-desktop:/dev$ ls -al sda* ttyS*
brw-rw---- 1 root disk    8,  0 2010-09-18 17:25 sda
brw-rw---- 1 root disk    8,  1 2010-09-18 17:25 sda1
brw-rw---- 1 root disk    8,  2 2010-09-18 17:25 sda2
brw-rw---- 1 root disk    8,  5 2010-09-18 17:25 sda5
crw-rw---- 1 root dialout 4, 64 2010-09-18 17:25 ttyS0
crw-rw---- 1 root dialout 4, 65 2010-09-18 17:25 ttyS1
crw-rw---- 1 root dialout 4, 66 2010-09-18 17:25 ttyS2
crw-rw---- 1 root dialout 4, 67 2010-09-18 17:25 ttyS3
rich@rich-desktop:/dev$
```

Different Linux distributions handle devices using different device names. In this distribution, the `sda` device is the first ATA hard drive, and the `ttyS` devices are the standard IBM PC COM ports. The listing shows all of the`sda` devices that were created on the sample Linux system. Not all are actually used, but they are created in case the administrator needs them. Similarly, the listing shows all of the `ttyS` devices created.

The fifth column is the major device node number. Notice that all of the `sda` devices have the same major device node, 8, while all of the `ttyS` devices use 4. The sixth column is the minor device node number. Each device within a major number has its own unique minor device node number.

The first column indicates the permissions for the device file. The first character of the permissions indicates the type of file. Notice that the ATA hard drive files are all marked as block (b) device, while the COM port device files are marked as character (c) devices.

Filesystem Management

Unlike some other operating systems, the Linux kernel can support different types of filesystems to read and write data to and from hard drives. Besides having over a dozen filesystems of its own, Linux can read and write to and from filesystems used by other operating systems, such as Microsoft Windows. The kernel must be compiled with support for all types of filesystems that the system will use. Table 1.1 lists the standard filesystems that a Linux system can use to read and write data.

Table 1.1 Linux Filesystems

Filesystem	Description
ext	Linux Extended filesystem—the original Linux filesystem
ext2	Second extended filesystem, provided advanced features over ext
ext3	Third extended filesystem, supports journaling
ext4	Fourth extended filesystem, supports advanced journaling
hpfs	OS/2 high-performance filesystem
jfs	IBM's journaling file system
iso9660	ISO 9660 filesystem (CD-ROMs)
minix	MINIX filesystem
msdos	Microsoft FAT16
ncp	Netware filesystem
nfs	Network File System
ntfs	Support for Microsoft NT filesystem
proc	Access to system information
ReiserFS	Advanced Linux file system for better performance and disk recovery
smb	Samba SMB filesystem for network access
sysv	Older Unix filesystem
ufs	BSD filesystem
umsdos	Unix-like filesystem that resides on top of msdos
vfat	Windows 95 filesystem (FAT32)
XFS	High-performance 64-bit journaling filesystem

Any hard drive that a Linux server accesses must be formatted using one of the filesystem types listed in Table 1.1.

The Linux kernel interfaces with each filesystem using the Virtual File System (VFS). This provides a standard interface for the kernel to communicate with any type of filesystem. VFS caches information in memory as each filesystem is mounted and used.

The GNU Utilities

Besides having a kernel to control hardware devices, a computer operating system needs utilities to perform standard functions, such as controlling files and programs. While Linus created the Linux system kernel, he had no system utilities to run on it. Fortunately for him, at the same time he was working, a group of people were working together on the Internet trying to develop a standard set of computer system utilities that mimicked the popular Unix operating system.

The GNU organization (GNU stands for GNU's Not Unix) developed a complete set of Unix utilities, but had no kernel system to run them on. These utilities were developed under a software philosophy called open source software (OSS).

The concept of OSS allows programmers to develop software and then release it to the world with no licensing fees attached. Anyone can use the software, modify it, or incorporate it into his or her own system without having to pay a license fee. Uniting Linus's Linux kernel with the GNU operating system utilities created a complete, functional, free operating system.

While the bundling of the Linux kernel and GNU utilities is often just called Linux, you will see some Linux purists on the Internet refer to it as the GNU/Linux system to give credit to the GNU organization for its contributions to the cause.

The Core GNU Utilities

The GNU project was mainly designed for Unix system administrators to have a Unix-like environment available. This focus resulted in the project porting many common Unix system command line utilities. The core bundle of utilities supplied for Linux systems is called the *coreutils* package.

The GNU coreutils package consists of three parts:

- Utilities for handling files
- Utilities for manipulating text
- Utilities for managing processes

Each of these three main groups of utilities contains several utility programs that are invaluable to the Linux system administrator and programmer. This book covers each of the utilities contained in the GNU coreutils package in detail.

The Shell

The GNU/Linux shell is a special interactive utility. It provides a way for users to start programs, manage files on the filesystem, and manage processes running on the Linux system. The core of the shell is the command prompt. The command prompt is the interactive part of the shell. It allows you to enter text commands, and then it interprets the commands and then executes them in the kernel.

The shell contains a set of internal commands that you use to control things such as copying files, moving files, renaming files, displaying the programs currently running on the system, and stopping programs running on the system. Besides the internal commands, the shell also allows you to enter the name of a program at the command prompt. The shell passes the program name off to the kernel to start it.

You can also group shell commands into files to execute as a program. Those files are called *shell scripts*. Any command that you can execute from the command line can be placed in a shell script and run as a group of commands. This provides great flexibility in creating utilities for commonly run commands, or processes that require several commands grouped together.

There are quite a few Linux shells available to use on a Linux system. Different shells have different characteristics, some being more useful for creating scripts and some being more useful for managing processes. The default shell used in all Linux distributions is the bash shell. The bash shell was developed by the GNU project as a replacement for the standard Unix shell, called the Bourne shell (after its creator). The bash shell name is a play on this wording, referred to as the "Bourne again shell."

In addition to the bash shell, we will cover several other popular shells in this book. Table 1.2 lists the different shells we will examine.

Table 1.2 Linux Shells

Shell	Description
ash	A simple, lightweight shell that runs in low-memory environments but has full compatibility with the bash shell
korn	A programming shell compatible with the Bourne shell but supporting advanced programming features like associative arrays and floating-point arithmetic
tcsh	A shell that incorporates elements from the C programming language into shell scripts
zsh	An advanced shell that incorporates features from bash, tcsh, and korn, providing advanced programming features, shared history files, and themed prompts

Most Linux distributions include more than one shell, although usually they pick one of them to be the default. If your Linux distribution includes multiple shells, feel free to experiment with different shells and see which one fits your needs.

The Linux Desktop Environment

In the early days of Linux (the early 1990s) all that was available was a simple text interface to the Linux operating system. This text interface allowed administrators to start programs, control program operations, and move files around on the system.

With the popularity of Microsoft Windows, computer users expected more than the old text interface to work with. This spurred more development in the OSS community, and the Linux graphical desktops emerged.

Linux is famous for being able to do things in more than one way, and no place is this more relevant than in graphical desktops. There are a plethora of graphical desktops you can choose from in Linux. The following sections describe a few of the more popular ones.

The X Windows System

There are two basic elements that control your video environment—the video card in your PC and your monitor. To display fancy graphics on your computer, the Linux software needs to know how to talk to both of them. The X Windows software is the core element in presenting graphics.

The X Windows software is a low-level program that works directly with the video card and monitor in the PC, and controls how Linux applications can present fancy windows and graphics on your computer.

Linux isn't the only operating system that uses X Windows; there are versions written for many different operating systems. In the Linux world, there are only two software packages that can implement it.

The XFree86 software package is the older of the two, and for a long time was the only X Windows package available for Linux. As its name implies, it's a free open source version of the X Windows software.

The newer of the two packages, X.org, has made great inroads in the Linux world and is now the more popular of the two. It, too, provides an open source software implementation of the X Windows system, but has support for more of the newer video cards used today.

Both packages work the same way, controlling how Linux uses your video card to display content on your monitor. To do that, they have to be configured for your specific system. That is supposed to happen automatically when you install Linux.

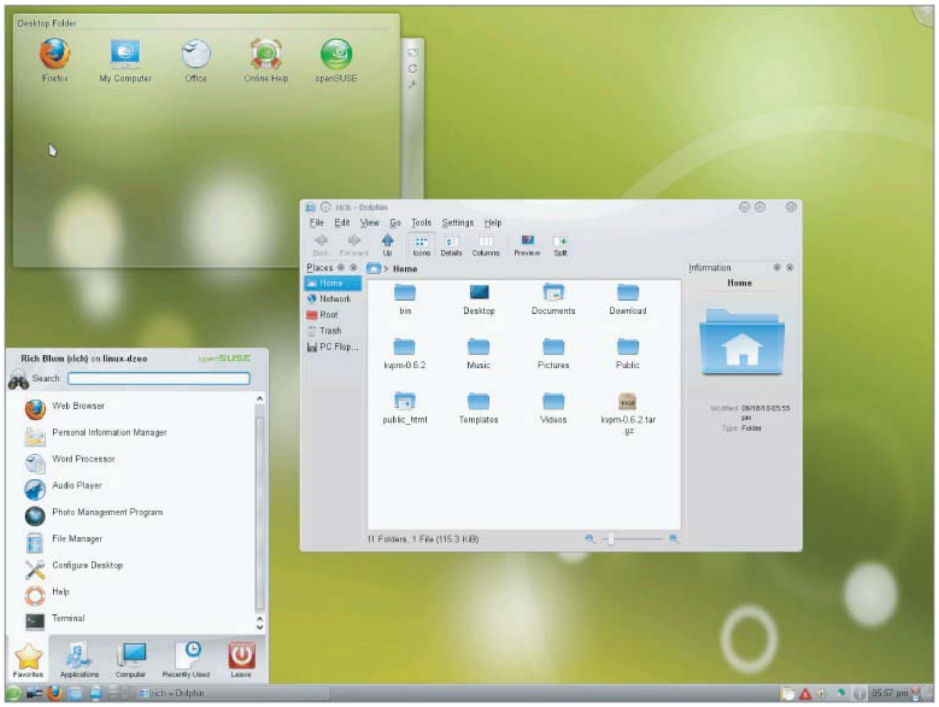
When you first install a Linux distribution, it attempts to detect your video card and monitor, and then creates an X Windows configuration file that contains the required information. During installation you may notice a time when the installation program scans your monitor for supported video modes. Sometimes this causes your monitor to go blank for a few seconds. Because there are lots of different types of video cards and monitors out there, this process can take a little while to complete.

The core X Windows software produces a graphical display environment, but nothing else. While this is fine for running individual applications, it is not too useful for day-to-day computer use. There is no desktop environment allowing users to manipulate files or launch programs. To do that, you need a desktop environment on top of the X Windows system software.

The KDE Desktop

The K Desktop Environment (KDE) was first released in 1996 as an open source project to produce a graphical desktop similar to the Microsoft Windows environment. The KDE desktop incorporates all of the features you are probably familiar with if you are a Windows user. Figure 1.3 shows a sample KDE 4 desktop running in the openSuSE Linux distribution.

Figure 1.3 The KDE 4 desktop on an openSuSE Linux system



The KDE desktop allows you to place both application and file icons in a special area on the desktop. If you single-click an application icon, the Linux system starts the application. If you single-click on a file icon, the KDE desktop attempts to determine what application to start to handle the file.

The bar at the bottom of the desktop is called the Panel. The Panel consists of four parts:

- **The K menu:** Much like the Windows Start menu, the K menu contains links to start installed applications.
- **Program shortcuts:** These are quick links to start applications directly from the Panel.
- **The taskbar:** The taskbar shows icons for applications currently running on the desktop.
- **Applets:** These are small applications that have an icon in the Panel that often can change depending on information from the application.

All of the Panel features are similar to what you would find in Windows. In addition to the desktop features, the KDE project has produced a wide assortment of applications that run in the KDE environment. These applications are shown in Table 1.3. (You may notice the trend of using a capital K in KDE application names.)

Table 1.3 KDE Applications

Application	Description
amaroK	Audio file player
digiKam	Digital camera software
dolphin	File manager

K3b	CD-burning software
Kaffeine	Video player
Kmail	E-mail client
Koffice	Office applications suite
Konqueror	File and Web browser
Kontact	Personal information manager
Kopete	Instant messaging client

This is only a partial list of applications produced by the KDE project. There are lots more applications that are included with the KDE desktop.

The GNOME Desktop

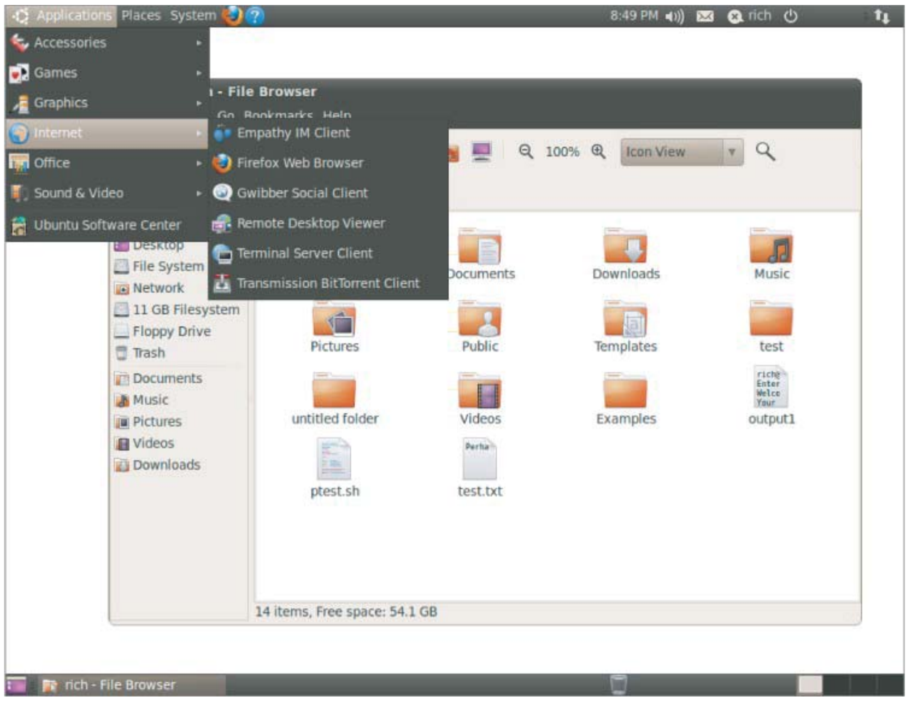
The GNU Network Object Model Environment (GNOME) is another popular Linux desktop environment. First released in 1999, GNOME has become the default desktop environment for many Linux distributions (the most popular being Red Hat Linux).

While GNOME chose to depart from the standard Microsoft Windows look-and-feel, it incorporates many features that most Windows users are comfortable with:

- A desktop area for icons
- Two panel areas
- Drag-and-drop capabilities

Figure 1.4 shows the standard GNOME desktop used in the Ubuntu Linux distribution.

Figure 1.4 A GNOME desktop on an Ubuntu Linux system



Not to be outdone by KDE, the GNOME developers have also produced a host of graphical applications that integrate with the GNOME desktop. These are shown in Table 1.4.

As you can see, there are also quite a few applications available for the GNOME desktop. Besides all of these applications, most Linux distributions that use the GNOME desktop also incorporate the KDE libraries, allowing you to run KDE applications on your GNOME desktop.

Table 1.4 GNOME Applications

Application	**Description**
epiphany	Web browser
evince	Document viewer
gcalc-tool	Calculator
gedit	GNOME text editor
gnome-panel	Desktop panel for launching applications
gnome-nettool	Network diagnostics tool
gnome-terminal	Terminal emulator
nautilus	Graphical file manager
nautilus-cd-burner	CD-burning tool
sound juicer	Audio CD–ripping tool
tomboy	Note-taking software
totem	Multimedia player

Other Desktops

The downside to a graphical desktop environment is that they require a fair amount of system resources to operate properly. In the early days of Linux, a hallmark and selling feature of Linux was its ability to operate on older, less powerful PCs that the newer Microsoft desktop products couldn't run on. However, with the popularity of KDE and GNOME desktops, this has changed, as it takes just as much memory to run a KDE or GNOME desktop as the latest Microsoft desktop environment.

If you have an older PC, don't be discouraged. The Linux developers have banded together to take Linux back to its roots. They've created several low-memory–oriented graphical desktop applications that provide basic features that run perfectly fine on older PCs.

While these graphical desktops don't have a plethora of applications designed around them, they still run many basic graphical applications that support features such as word processing, spreadsheets, databases, drawing, and, of course, multimedia support.

Table 1.5 shows some of the smaller Linux graphical desktop environments that can be used on lower-powered PCs and laptops.

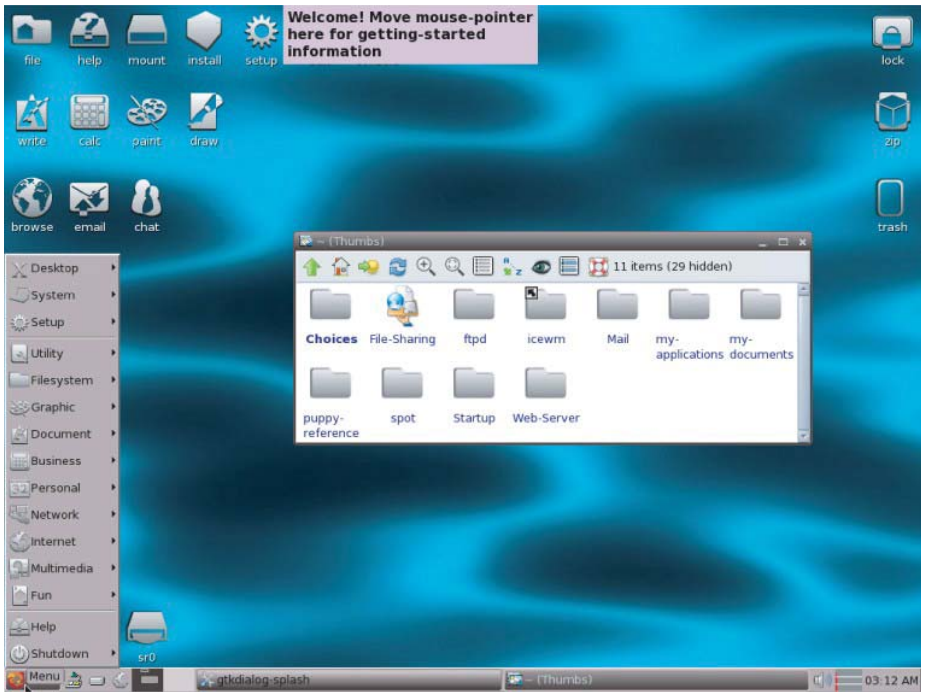
Table 1.5 Other Linux Graphical Desktops

Desktop	**Description**

fluxbox	A bare-bones desktop that doesn't include a Panel, only a pop-up menu to launch applications
xfce	A desktop that's similar to the KDE desktop, but with less graphics for low-memory environments
JWM	Joe's Window Manager, a very lightweight desktop ideal for low-memory and low-disk space environments
fvwm	Supports some advanced desktop features such as virtual desktops and Panels, but runs in low-memory environments
fvwm95	Derived from fvwm, but made to look like a Windows 95 desktop

These graphical desktop environments are not as fancy as the KDE and GNOME desktops, but they provide basic graphical functionality just fine. Figure 1.5 shows what the fluxbox desktop used in the Puppy Linux antiX distribution looks like.

Figure 1.5 The JWM desktop as seen in the Puppy Linux distribution



If you are using an older PC, try a Linux distribution that uses one of these desktops and see what happens. You may be pleasantly surprised.

Linux Distributions

Now that you have seen the four main components required for a complete Linux system, you may be wondering how you are going to get them all put together to make a Linux system. Fortunately, there are people who have already done that for you.

A complete Linux system package is called a *distribution*. There are lots of different Linux distributions available to meet just about any computing requirement you could have. Most distributions are customized for a specific user group, such as business users, multimedia enthusiasts, software developers, or average home users. Each customized distribution includes the software packages required to support specialized functions, such as audio- and video-editing software for multimedia enthusiasts, or compilers and integrated development environments (IDEs) for software developers.

The different Linux distributions are often divided into three categories:

- Full core Linux distributions
- Specialized distributions
- LiveCD test distributions

The following sections describe these different types of Linux distributions, and show some examples of Linux distributions in each category.

Core Linux Distributions

A core Linux distribution contains a kernel, one or more graphical desktop environments, and just about every Linux application that is available, precompiled for the kernel. It provides one-stop shopping for a complete Linux installation. Table 1.6 shows some of the more popular core Linux distributions.

Table 1.6 Core Linux Distributions

Distribution	Description
Slackware	One of the original Linux distribution sets, popular with Linux geeks
Red Hat	A commercial business distribution used mainly for Internet servers
Fedora	A spin-off from Red Hat but designed for home use
Gentoo	A distribution designed for advanced Linux users, containing only Linux source code
Mandriva	Designed mainly for home use (previously called Mandrake)
openSuSe	Different distributions for business and home use
Debian	Popular with Linux experts and commercial Linux products

In the early days of Linux, a distribution was released as a set of floppy disks. You had to download groups of files and then copy them onto disks. It would usually take 20 or more disks to make an entire distribution! Needless to say, this was a painful experience.

Nowadays, with home computers commonly having CD and DVD players built in, Linux distributions are released as either a CD set or a single DVD. This makes installing Linux much easier.

However, beginners still often run into problems when they install one of the core Linux distributions. To cover just about any situation in which someone might want to use Linux, a single distribution has to include lots of application software. They include everything from high-end Internet database servers to common games. Because of the quantity of applications available for Linux, a complete distribution often takes four or more CDs.

While having lots of options available in a distribution is great for Linux geeks, it can become a nightmare for beginning Linux users. Most distributions ask a series of questions during the installation process to determine which applications to load by default, what hardware is connected to the PC, and how to configure the hardware. Beginners often find these questions confusing. As a result, they often either load way too many programs on their computer or don't load enough and later discover that their computer won't do what they want it to.

Fortunately for beginners, there's a much simpler way to install Linux.

Specialized Linux Distributions

A new subgroup of Linux distributions has started to appear. These are typically based on one of the main distributions but contain only a subset of applications that would make sense for a specific area of use.

In addition to providing specialized software (such as only office products for business users), customized Linux distributions also attempt to help beginning Linux users by autodetecting and autoconfiguring common hardware devices. This makes installing Linux a much more enjoyable process.

Table 1.7 shows some of the specialized Linux distributions available and what they specialize in.

Table 1.7 Specialized Linux Distributions

Distribution	**Description**
Xandros	A commercial Linux package configured for beginners
SimplyMEPIS	A free distribution for home use
Ubuntu	A free distribution for school and home use
PCLinuxOS	A free distribution for home and office use
Mint	A free distribution for home entertainment use
dyne:bolic	A free distribution designed for audio and MIDI applications
Puppy Linux	A free small distribution that runs well on older PCs

That's just a small sampling of specialized Linux distributions. There are literally hundreds of specialized Linux distributions, and more are popping up all the time on the Internet. No matter what your specialty, you'll probably find a Linux distribution made for you.

Many of the specialized Linux distributions are based on the Debian Linux distribution. They use the same installation files as Debian but package only a small fraction of a full-blown Debian system.

The Linux LiveCD

A relatively new phenomenon in the Linux world is the bootable Linux CD distribution. This lets you see what a Linux system is like without actually installing it. Most modern PCs can boot from a CD instead of the standard hard drive. To take advantage of this, some Linux distributions create a bootable CD that contains a sample Linux system (called a *Linux LiveCD*). Because of the limitations of the single CD size, the sample can't contain a complete Linux system, but you'd be surprised at all the software they can cram in there. The result is that you can boot your PC from the CD and run a Linux distribution without having to install anything on your hard drive!

This is an excellent way to test various Linux distributions without having to mess with your PC. Just pop in a CD and boot! All of the Linux software will run directly off the CD. There are lots of Linux LiveCDs that you can download from the Internet and burn onto a CD to test drive.

Table 1.8 shows some popular Linux LiveCDs that are available.

Table 1.8 Linux LiveCD Distributions

Distribution	Description
Knoppix	A German Linux, the first Linux LiveCD developed
SimplyMEPIS	Designed for beginning home Linux users
PCLinuxOS	Full-blown Linux distribution on a LiveCD
Ubuntu	A worldwide Linux project, designed for many languages
Slax	A live Linux CD based on Slackware Linux
Puppy Linux	A full-featured Linux designed for older PCs

You may notice a familiarity in this table. Many specialized Linux distributions also have a Linux LiveCD version. Some Linux LiveCD distributions, such as Ubuntu, allow you to install the Linux distribution directly from the LiveCD. This enables you to boot with the CD, test drive the Linux distribution, and then if you like it, install it on your hard drive. This feature is extremely handy and user-friendly.

As with all good things, Linux LiveCDs have a few drawbacks. Because you access everything from the CD, applications run more slowly, especially if you're using older, slower computers and CD drives. Also, because you can't write to the CD, any changes you make to the Linux system will be gone the next time you reboot.

But there are advances being made in the Linux LiveCD world that help to solve some of these problems. These advances include the ability to:

- Copy Linux system files from the CD to memory
- Copy system files to a file on the hard drive
- Store system settings on a USB memory stick
- Store user settings on a USB memory stick

Some Linux LiveCDs, such as Puppy Linux, are designed with a minimum number of Linux system files. The LiveCD boot scripts copies them directly into memory when the CD boots. This allows you to remove the CD from the computer as soon as Linux boots. Not only does this make your applications run much faster (because applications run faster from memory), but it also gives you a free CD tray to use for ripping audio CDs or playing video DVDs from the software included in Puppy Linux.

Other Linux LiveCDs use an alternative method that allows you to remove the CD from the tray after booting. It involves copying the core Linux files onto the Windows hard drive as a single file. After the CD boots, it looks for that file and reads the system files from it. The dyne:bolic Linux LiveCD uses this technique, which is called docking. Of course, you must copy the system file to your hard drive before you can boot from the CD.

A very popular technique for storing data from a live Linux CD session is to use a common USB memory stick (also called a flash drive or a thumb drive). Just about every

Linux LiveCD can recognize a plugged-in USB memory stick (even if the stick is formatted for Windows) and read and write files to and from it. This allows you to boot a Linux LiveCD, use the Linux applications to create files, store those files on your memory stick, and then access them from your Windows applications later (or from a different computer). How cool is that?

Summary

This chapter discussed the Linux system, and the basics of how it works. The Linux kernel is the core of the system, controlling how memory, programs, and hardware all interact with one another. The GNU utilities are also an important piece in the Linux system. The Linux shell, which is the main focus of this book, is part of the GNU core utilities. The chapter also discussed the final piece of a Linux system, the Linux desktop environment. Things have changed over the years, and Linux now supports several graphical desktop environments.

The chapter also discussed the various Linux distributions. A Linux distribution bundles the various parts of a Linux system into a simple package that you can easily install on your PC. The Linux distribution world consists of full-blown Linux distributions that include just about every application imaginable, as well as specialized Linux distributions that only include applications focused on a special function. The Linux LiveCD craze has created another group of Linux distributions that allow you to easily test drive Linux without even having to install it on your hard drive.

In the next chapter, you look at what you need to start your command line and shell scripting experience. You'll see what you need to do to get to the Linux shell utility from your fancy graphical desktop environment. These days that's not always an easy thing.

Chapter 2

Getting to the Shell

In This Chapter

- Terminal emulation
- The terminfo database
- The Linux console

Chapter 3

Basic bash Shell Commands

In This Chapter

- Starting the shell
- The shell prompt
- The bash manual
- Filesystem navigation
- File and directory listing
- File handling
- Directory handling
- Viewing file contents

The default shell used in all Linux distributions is the GNU bash shell. This chapter describes the basic features available in the bash shell, and walks you through how to work with Linux files and directories using the basic commands provided by the bash shell. If you're already comfortable working with files and directories in the Linux environment, feel free to skip this chapter and continue with Chapter 4 to see more advanced commands.

Starting the Shell

The GNU bash shell is a program that provides interactive access to the Linux system. It runs as a regular program, normally started whenever a user logs in to a terminal. The shell that the system starts depends on your user ID configuration.

The `/etc/passwd` file contains a list of all the system user accounts, along with some basic configuration information about each user. Here's a sample entry from a `/etc/passwd` file:

```
rich:x:501:501:Rich Blum:/home/rich:/bin/bash
```

Each entry has seven data fields, with each field separated by a colon. The system uses the data in these fields to assign specific features for the user. These fields are:

- The username
- The user's password (or a placeholder if the password is stored in another file)
- The user's system user ID number
- The user's system group ID number
- The user's full name
- The user's default home directory
- The user's default shell program

Most of these entries will be discussed in more detail in Chapter 6. For now, just pay attention to the shell program specified.

Most Linux systems use the default bash shell program when starting a command line interface (CLI) environment for the user. The bash program also uses command line parameters to modify the type of shell you can start. Table 3.1 lists the command line parameters available in bash that define what type of shell to use.

Table 3.1 The bash Command Line Parameters

Parameter	**Description**
`-c string`	Read commands from string and process them.
`-r`	Start a restricted shell, limiting the user to the default directory.
`-i`	Start an interactive shell, allowing input from the user.
`-s`	Read commands from the standard input.

By default, when the bash shell starts, it automatically processes commands in the `.bashrc` file in the user's home directory. Many Linux distributions use this file to also load a common file that contains commands and settings for everyone on the system. This common file is normally located in the file `/etc/bashrc`. This file often sets environment variables (see Chapter 5) used in various applications.

The Shell Prompt

Once you start a terminal emulation package or log in from the Linux console, you get access to the shell CLI*prompt*. The prompt is your gateway to the shell. This is the place where you enter shell commands.

The default prompt symbol for the bash shell is the dollar sign ($). This symbol indicates that the shell is waiting for you to enter text. However, you can change the format of the prompt used by your shell. The different Linux distributions use different formats for the prompt. On this Ubuntu Linux system, the bash shell prompt looks like this:

```
rich@user-desktop:~$
```

On this Fedora Linux system, it looks like this:

```
[rich@testbox~]$
```

You can configure the prompt to provide basic information about your environment. The first example shows three pieces of information in the prompt:

- The username that started the shell
- The current virtual console number
- The current directory (the tilde sign is shorthand for the home directory)

The second example provides similar information, except that it uses the hostname instead of the virtual console number. There are two environment variables that control the format of the command line prompt:

- PS1: Controls the format of the default command line prompt
- PS2: Controls the format of the second-tier command line prompt

The shell uses the default PS1 prompt for initial data entry into the shell. If you enter a command that requires additional information, the shell displays the second-tier prompt specified by the PS2 environment variable.

To display the current settings for your prompts, use the echo command:

```
rich@ user-desktop:~$ echo $PS1
${debian_chroot:+($debian_chroot)}\u@\h:\w\$
rich@ user-desktop:~$ echo $PS2
>
rich@ user-desktop:~$
```

The format of the prompt environment variables can look pretty odd. The shell uses special characters to signify elements within the command line prompt. Table 3.2 shows the special characters that you can use in the prompt string.

Table 3.2 Bash Shell Prompt Characters

Character	Description
\a	Bell character
\d	Date in the format "Day Month Date"
\e	ASCII escape character
\h	Local hostname
\H	Fully qualified domain hostname
\j	Number of jobs currently managed by the shell
\l	Basename of the shell's terminal device name
\n	ASCII newline character
\r	ASCII carriage return

`\s`	Name of the shell
`\t`	Current time in 24-hour HH:MM:SS format
`\T`	Current time in 12-hour HH:MM:SS format
`\@`	Current time in 12-hour am/pm format
`\u`	Username of the current user
`\v`	Version of the bash shell
`\V`	Release level of the bash shell
`\w`	Current working directory
`\W`	Basename of the current working directory
`\!`	Bash shell history number of this command
`\#`	Command number of this command
`\$`	A dollar sign if a normal user, or a pound sign if the root user
`\nnn`	Character corresponding to the octal value `nnn`
`\\`	Backslash
`\[`	Begins a control code sequence
`\]`	Ends a control code sequence

Notice that all of the special prompt characters begin with a backslash (\). This is what delineates a prompt character from normal text in the prompt. In the earlier example, the prompt contained both prompt characters and a normal character (the "at" sign, and the square brackets). You can create any combination of prompt characters in your prompt. To create a new prompt, just assign a new string to the PS1 variable:

```
[rich@testbox~]$ PS1="[\t][\u]\$ "
[14:40:32][rich]$
```

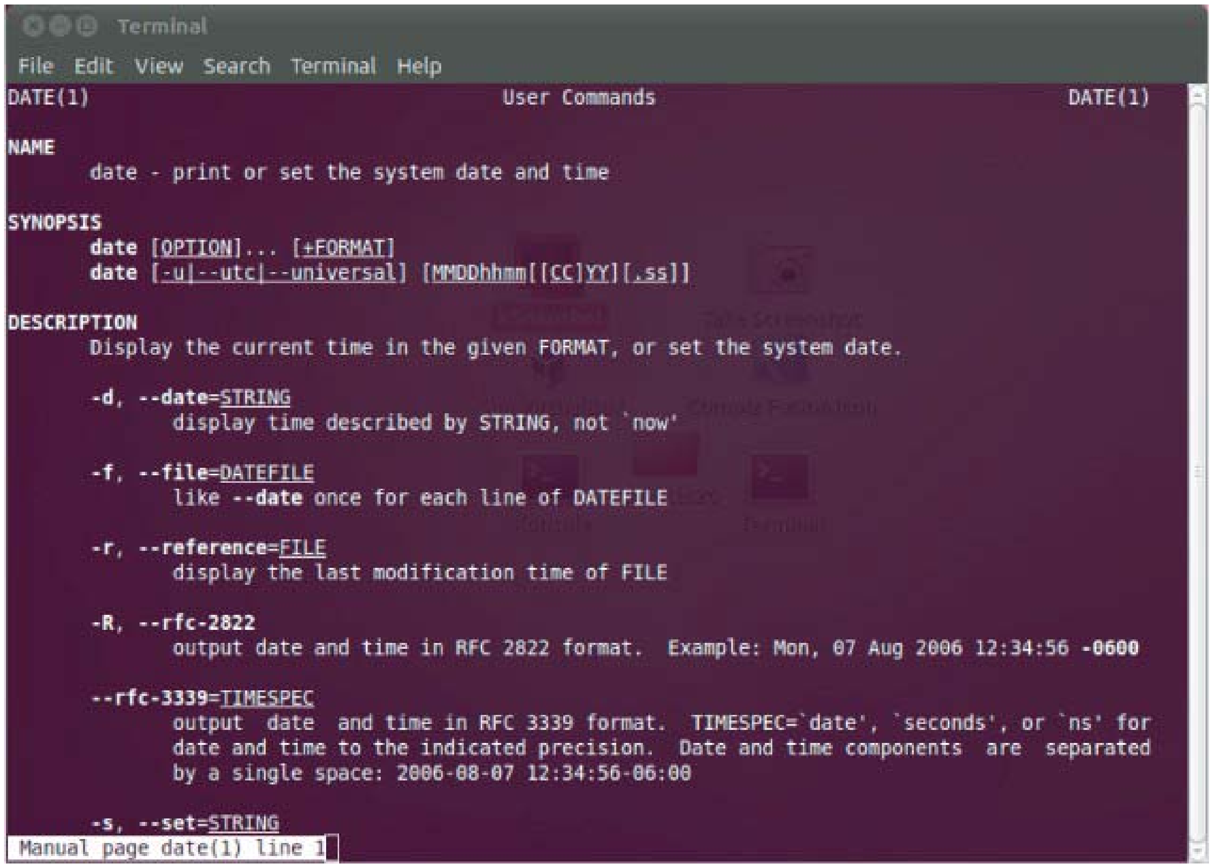
This new shell prompt now shows the current time, along with the username. The new PS1 definition only lasts for the duration of the shell session. When you start a new shell, the default shell prompt definition is reloaded. In Chapter 5 you'll see how you can change the default shell prompt for all shell sessions.

The bash Manual

Most Linux distributions include an online manual for looking up information on shell commands, as well as lots of other GNU utilities included in the distribution. It is a good idea to become familiar with the manual, as it's invaluable for working with utilities, especially when you're trying to figure out various command line parameters.

The `man` command provides access to the manual pages stored on the Linux system. Entering the `man` command followed by a specific utility name provides the manual entry for that utility. Figure 3.1 shows an example of looking up the manual pages for the date command.

Figure 3.1 Displaying the manual pages for the Linux date command



```
DATE(1)                          User Commands                          DATE(1)

NAME
       date - print or set the system date and time

SYNOPSIS
       date [OPTION]... [+FORMAT]
       date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]]

DESCRIPTION
       Display the current time in the given FORMAT, or set the system date.

       -d, --date=STRING
              display time described by STRING, not `now'

       -f, --file=DATEFILE
              like --date once for each line of DATEFILE

       -r, --reference=FILE
              display the last modification time of FILE

       -R, --rfc-2822
              output date and time in RFC 2822 format.  Example: Mon, 07 Aug 2006 12:34:56 -0600

       --rfc-3339=TIMESPEC
              output  date  and time in RFC 3339 format.  TIMESPEC=`date', `seconds', or `ns' for
              date and time to the indicated precision.  Date and time components  are  separated
              by a single space: 2006-08-07 12:34:56-06:00

       -s, --set=STRING
 Manual page date(1) line 1
```

The manual page divides information about the command into separate sections, shown in Table 3.3.

Table 3.3 The Linux man Page Format

Section	**Description**
Name	Displays the command name and a short description
Synopsis	Shows the format of the command
Description	Describes each command option
Author	Provides information on the person who developed the command
Reporting bugs	Provides information on where to report any bugs found
Copyright	Provides information on the copyright status of the command code
See Also	Refers you to any similar commands available

You can step through the man pages by pressing the spacebar or using the arrow keys to scroll forward and backward through the man page text (assuming that your terminal emulation package supports the arrow key functions). When you are done with the man pages, press the q key to quit.

To see information about the bash shell, look at the man pages for it using the following command:

```
$ man bash
```

This allows you to step through all of the man pages for the bash shell. This is extremely handy when building scripts, as you don't have to refer back to books or Internet sites to look up specific formats for commands. The manual is always there for you in your session.

Filesystem Navigation

As you can see from the shell prompt, when you start a shell session, you are usually placed in your home directory. Most often, you will want to break out of your home directory and explore other areas in the Linux system. This section describes how to do that using shell commands. Before we do that, however, let's take a tour of just what the Linux filesystem looks like so we know where we're going.

The Linux Filesystem

If you're new to the Linux system, you may be confused by how it references files and directories, especially if you're used to the way that the Microsoft Windows operating system does that. Before exploring the Linux system, it helps to have an understanding of how it's laid out.

The first difference you'll notice is that Linux does not use drive letters in pathnames. In the Windows world, the physical drives installed on the PC determine the pathname of the file. Windows assigns a letter to each physical disk drive, and each drive contains its own directory structure for accessing files stored on it.

For example, in Windows you may be used to seeing the filepaths such as:

```
c:\Users\Rich\Documents\test.doc.
```

This indicates that the file test.doc is located in the directory Documents, which itself is located in the directory Rich. The Rich directory is contained under the directory Users, which is located on the hard disk partition assigned the letter C (usually the first hard drive on the PC).

The Windows filepath tells you exactly which physical disk partition contains the file named test.doc. If you wanted to save a file on a flash drive, it could be, for example, designated by the J drive. You would click the icon for the J drive, which would automatically use the filepath J:\test.doc. This path indicates that the file is located at the root of the drive assigned the letter J.

This is not the method used by Linux. Linux stores files within a single directory structure, called a *virtual directory*. The virtual directory contains filepaths from all the storage devices installed on the PC, merged into a single directory structure.

The Linux virtual directory structure contains a single base directory, called the root. Directories and files beneath the root directory are listed based on the directory path used to get to them, similar to the way Windows does it.

Tip

You'll notice that Linux uses a forward slash (/) instead of a backward slash (\) to denote directories in filepaths. The backslash character in Linux denotes an escape character and causes all sorts of problems when you use it in a filepath. This may take some getting used to if you're coming from a Windows environment.

For example, the Linux filepath `/home/rich/Documents/test.doc` indicates only that the file test.doc is in the directory Documents, under the directory rich, which is contained in the directory home. It doesn't provide any information as to which physical disk on the PC the file is stored on.

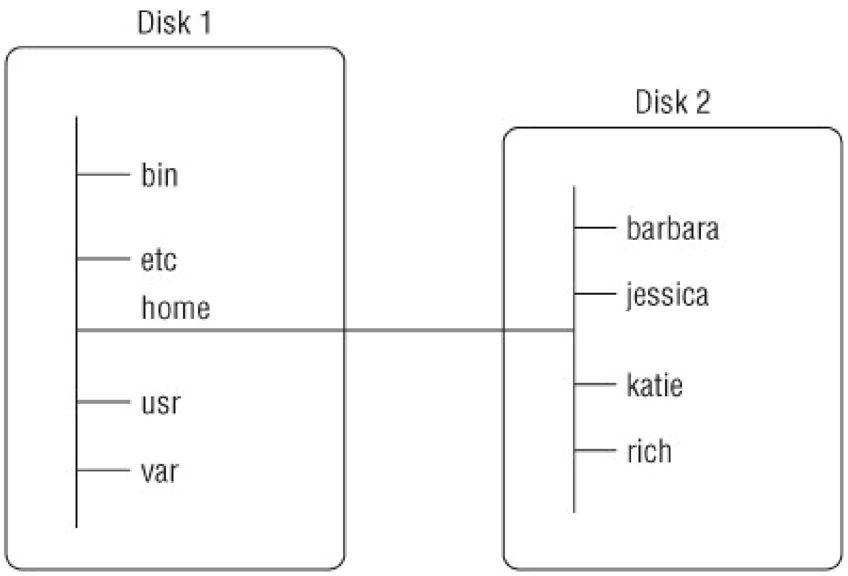
The tricky part about the Linux virtual directory is how it incorporates each storage device. The first hard drive installed in a Linux PC is called the *root drive*. The root drive contains the core of the virtual directory. Everything else builds from there.

On the root drive, Linux creates special directories called *mount points*. Mount points are directories in the virtual directory where you assign additional storage devices.

The virtual directory causes files and directories to appear within these mount point directories, even though they are physically stored on a different drive.

Often the system files are physically stored on the root drive, while user files are stored on a different drive, as shown in Figure 3.2.

Figure 3.2 The Linux file structure



In Figure 3.2, there are two hard drives on the PC. One hard drive is associated with the root of the virtual directory (indicated by a single forward slash). Other hard drives can be mounted anywhere in the virtual directory structure. In this example, the second hard drive is mounted at the location `/home`, which is where the user directories are located.

The Linux filesystem structure has evolved from the Unix file structure. Unfortunately, the Unix file structure has been somewhat convoluted over the years by different flavors of Unix. Nowadays it seems that no two Unix or Linux systems follow the same filesystem structure. However, there are a few common directory names that are used for common functions. Table 3.4 lists some of the more common Linux virtual directory names.

Table 3.4 Common Linux Directory Names

Directory	Usage
/	root of the virtual directory. Normally, no files are placed here.
/bin	binary directory, where many GNU user-level utilities are stored.
/boot	boot directory, where boot files are stored.
/dev	device directory, where Linux creates device nodes.
/etc	system configuration files directory.
/home	home directory, where Linux creates user directories.
/lib	library directory, where system and application library files are stored.
/media	media directory, a common place for mount points used for removable media.
/mnt	mount directory, another common place for mount points used for removable media.
/opt	optional directory, often used to store optional software packages.
/root	root home directory.
/sbin	system binary directory, where many GNU admin-level utilities are stored.
/tmp	temporary directory, where temporary work files can be created and destroyed.
/usr	user-installed software directory.
/var	variable directory, for files that change frequently, such as log files.

When you start a new shell prompt, your session starts in your home directory, which is a unique directory assigned to your user account. When you create a user account, the system normally assigns a unique directory for the account (see Chapter 6).

In the Windows world, you're probably used to moving around the directory structure using a graphical interface. To move around the virtual directory from a CLI prompt, you'll need to learn to use the `cd` command.

Traversing Directories

You use the change directory command (`cd`) to move your shell session to another directory in the Linux filesystem. The format of the `cd` command is pretty simplistic:

```
cd destination
```

The `cd` command may take a single parameter, `destination`, which specifies the directory name you want to go to. If you don't specify a destination on the `cd` command, it will take you to your home directory.

The destination parameter, however, can be expressed using two different methods:

- An absolute filepath
- A relative filepath

The following sections describe the differences between these two methods.

Absolute Filepaths

You can reference a directory name within the virtual directory using an *absolute filepath*. The absolute filepath defines exactly where the directory is in the virtual directory structure, starting at the root of the virtual directory, sort of like a full name for a directory.

Thus, to reference the apache directory, which is contained within the lib directory, which in turn is contained within the usr directory, you would use the absolute filepath:

```
/usr/lib/NetworkManager
```

With the absolute filepath, there's no doubt as to exactly where you want to go. To move to a specific location in the filesystem using the absolute filepath, you just specify the full pathname in the cd command:

```
rich@testbox[~]$cd /etc
rich@testbox[etc]$
```

The prompt shows that the new directory for the shell after the cd command is now `/etc`. You can move to any level within the entire Linux virtual directory structure using the absolute filepath:

```
rich@testbox[~]$ cd /usr/lib/NetworkManager
rich@testbox[NetworkManager]$
```

However, if you're just working within your own home directory structure, often using absolute filepaths can get tedious. For example, if you're already in the directory `/home/rich`, it seems somewhat cumbersome to have to type the command

```
cd /home/rich/Documents
```

just to get to your Documents directory. Fortunately, there's a simpler solution.

Relative Filepaths

Relative filepaths allow you to specify a destination filepath relative to your current location, without having to start at the root. A relative filepath doesn't start with a forward slash, indicating the root directory.

Instead, a relative filepath starts with either a directory name (if you're traversing to a directory under your current directory), or a special character indicating a relative location to your current directory location. The two special characters used for this are:

- The dot (`.`) to represent the current directory
- The double dot (`..`) to represent the parent directory

The double dot character is extremely handy when trying to traverse a directory hierarchy. For example, if you are in the Documents directory under your home directory and need to go to your Desktop directory, also under your home directory, you can do this:

```
rich@testbox[Documents]$ cd ../Desktop
rich@testbox[Desktop]$
```

The double dot character takes you back up one level to your home directory; then the `/Desktop` portion then takes you back down into the Desktop directory. You can use as many double dot characters as necessary to move around. For example, if you are in your home directory (`/home/rich`) and want to go to the `/etc` directory, you could type the following:

```
rich@testbox[~]$ cd ../../etc
rich@testbox[etc]$
```

Of course, in a case like this, you actually have to do more typing to use the relative filepath rather than just typing the absolute filepath, `/etc`!

File and Directory Listing

The most basic feature of the shell is the ability to see what files are available on the system. The list command (`ls`) is the tool that helps do that. This section describes the `ls` command and all of the options available to format the information it can provide.

Basic Listing

The `ls` command at its most basic form displays the files and directories located in your current directory:

```
$ ls
4rich  Desktop   Download Music   Pictures store     store.zip test
backup Documents Drivers  myprog Public   store.sql Templates Videos
```

Notice that the `ls` command produces the listing in alphabetical order (in columns rather than rows). If you're using a terminal emulator that supports color, the `ls` command may also show different types of entries in different colors. The `LS_COLORS` environment variable controls this feature. Different Linux distributions set this environment variable depending on the capabilities of the terminal emulator.

If you don't have a color terminal emulator, you can use the `-F` parameter with the `ls` command to easily distinguish files from directories. Using the `-F` parameter produces the following output:

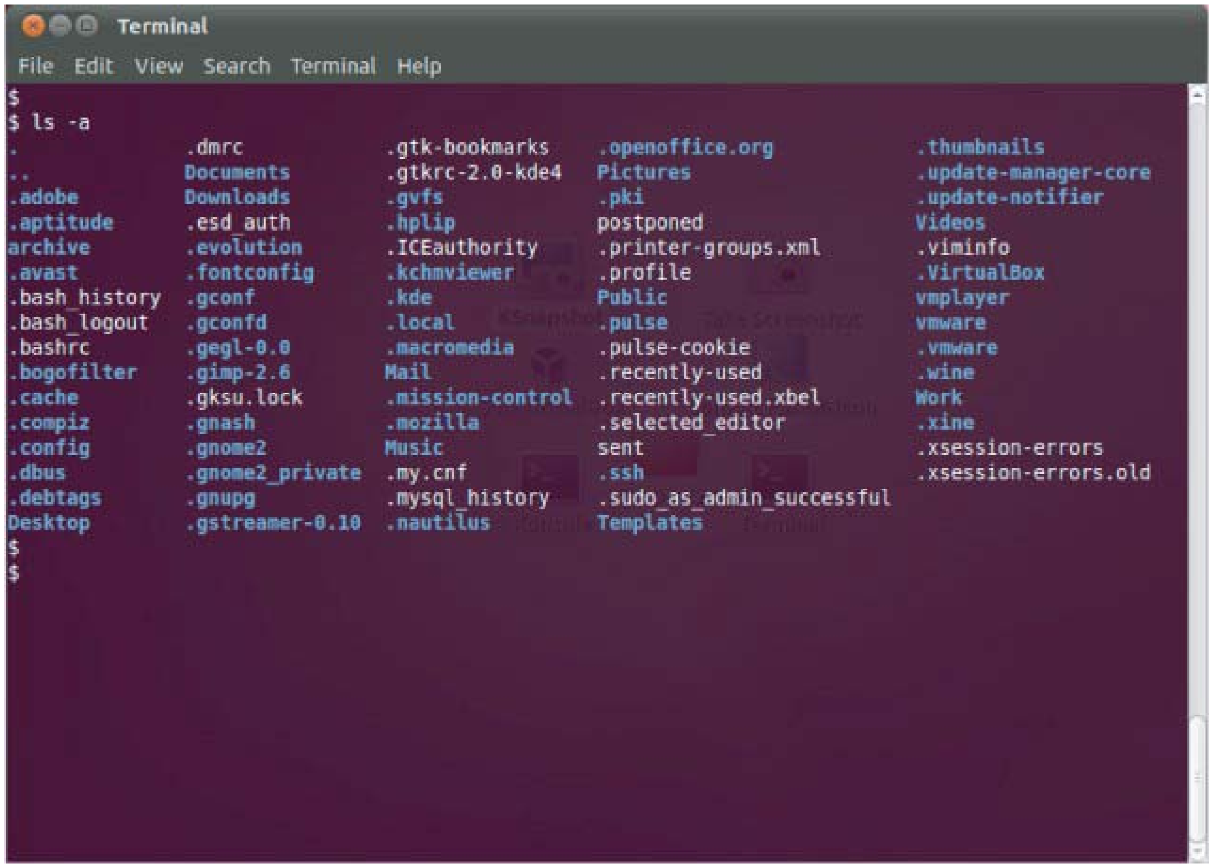
```
$ ls -F
4rich/      Documents/  Music/    Public/    store.zip   Videos/
backup.zip  Download/   myprog*   store/     Templates/
Desktop/    Drivers/    Pictures/ store.sql  test
$
```

The `-F` parameter now flags the directories with a forward slash, to help identify them in the listing. Similarly, it flags executable files (like the myprog file above) with an asterisk, to help you more easily find the files that can be run on the system.

The basic `ls` command can be somewhat misleading. It shows the files and directories contained in the current directory, but not necessarily all of them. Linux often uses *hidden files* to store configuration information. In Linux, hidden files are files with file names that start with a period. These files don't appear in the default `ls`listing (thus, they are called hidden).

To display hidden files along with normal files and directories, use the `-a` parameter. Figure 3.3 shows an example of using the `-a` parameter with the `ls` command.

Figure 3.3 Using the -a parameter with the `ls` command



```
$
$ ls -a
.               .dmrc             .gtk-bookmarks     .openoffice.org             .thumbnails
..              Documents         .gtkrc-2.0-kde4    Pictures                    .update-manager-core
.adobe          Downloads         .gvfs              .pki                        .update-notifier
.aptitude       .esd_auth         .hplip             postponed                   Videos
archive         .evolution        .ICEauthority      .printer-groups.xml         .viminfo
.avast          .fontconfig       .kchmviewer        .profile                    .VirtualBox
.bash_history   .gconf            .kde               Public                      vmplayer
.bash_logout    .gconfd           .local             .pulse                      vmware
.bashrc         .gegl-0.0         .macromedia        .pulse-cookie               .vmware
.bogofilter     .gimp-2.6         Mail               .recently-used              .wine
.cache          .gksu.lock        .mission-control   .recently-used.xbel         Work
.compiz         .gnash            .mozilla           .selected_editor            .xine
.config         .gnome2           Music              sent                        .xsession-errors
.dbus           .gnome2_private   .my.cnf            .ssh                        .xsession-errors.old
.debtags        .gnupg            .mysql_history     .sudo_as_admin_successful
Desktop         .gstreamer-0.10   .nautilus          Templates
$
$
```

Wow, that's quite a difference. In a home directory for a user who has logged in to the system from a graphical desktop, you'll see lots of hidden configuration files. This particular example is from a user logged in to a GNOME desktop session. Also notice that there are three files that begin with .bash. These files are hidden files that are used by the bash shell environment. These features are covered in detail in Chapter 5.

The -R parameter is another option the `ls` command can use. It shows files that are contained within directories in the current directory. If you have lots of directories, this can be quite a long listing. Here's a simple example of what the -R parameter produces:

```
$ ls -F -R
.:
file1 test1/  test2/


./test1:
myprog1*  myprog2*
./test2:
$
```

Notice that first, the -R parameter shows the contents of the current directory, which is a file (`file1`) and two directories (`test1` and `test2`). Following that, -R traverses each of the two directories, showing if any files are contained within each directory. The `test1` directory shows two files (`myprog1` and `myprog2`), while the `test2`directory doesn't contain any files. If there had been further subdirectories within the `test1` or `test2`directories, the -R parameter would have continued to traverse those as well. As you can see, for large directory structures this can become quite a large output listing.

Modifying the Information Presented

As you can see in the basic listings, the `ls` command doesn't produce a whole lot of information about each file. For listing additional information, another popular parameter is `-l`. The `-l` parameter produces a long listing format, providing more information about each file in the directory:

```
$ ls -l
total 2064
drwxrwxr-x  2 rich rich    4096 2010-08-24 22:04 4rich
-rw-r--r--  1 rich rich 1766205 2010-08-24 15:34 backup.zip
drwxr-xr-x  3 rich rich    4096 2010-08-31 22:24 Desktop
drwxr-xr-x  2 rich rich    4096 2009-11-01 04:06 Documents
drwxr-xr-x  2 rich rich    4096 2009-11-01 04:06 Download
drwxrwxr-x  2 rich rich    4096 2010-07-26 18:25 Drivers
drwxr-xr-x  2 rich rich    4096 2009-11-01 04:06 Music
-rwxr--r--  1 rich rich      30 2010-08-23 21:42 myprog
drwxr-xr-x  2 rich rich    4096 2009-11-01 04:06 Pictures
drwxr-xr-x  2 rich rich    4096 2009-11-01 04:06 Public
drwxrwxr-x  5 rich rich    4096 2010-08-24 22:04 store
-rw-rw-r--  1 rich rich   98772 2010-08-24 15:30 store.sql
-rw-r--r--  1 rich rich  107507 2010-08-13 15:45 store.zip
drwxr-xr-x  2 rich rich    4096 2009-11-01 04:06 Templates
drwxr-xr-x  2 rich rich    4096 2009-11-01 04:06 Videos
[rich@testbox~]$
```

The long listing format lists each file and directory contained in the directory on a single line. In addition to the file name, the listing shows additional useful information. The first line in the output shows the total number of blocks contained within the directory. Following that, each line contains the following information about each file (or directory):

- The file type—such as directory (d), file (-), character device (c), or block device (b)
- The permissions for the file (see Chapter 6)
- The number of hard links to the file (see the section “Linking Files” in this chapter)
- The username of the owner of the file
- The group name of the group the file belongs to
- The size of the file in bytes
- The time the file was modified last
- The file or directory name

The `-l` parameter is a powerful tool to have. Armed with this information, you can see just about any information you need to for any file or directory on the system.

The Complete Parameter List

There are lots of parameters for the `ls` command that can come in handy as you do file management. If you use theman command for `ls`, you'll see several pages of available parameters for you to use to modify the output of the `ls`command.

The `ls` command uses two types of command line parameters:

- Single-letter parameters
- Full-word (long) parameters

The single-letter parameters are always preceded by a single dash. Full-word parameters are more descriptive and are preceded by a double dash. Many parameters have both a single-letter and full-word version, while some have only one type. Table 3.5 lists some of the more popular parameters that will help you out with using the bash `ls`command.

Table 3.5 Some Popular ls Command Parameters

Single Letter	**Full Word**	**Description**
`-a`	`--all`	Don't ignore entries starting with a period.
`-A`	`--almost-all`	Don't list the . and .. files.
	`--author`	Print the author of each file.
`-b`	`--escape`	Print octal values for nonprintable characters.
	`--block-size=size`	Calculate the block sizes using size-byte blocks.
`-B`	`--ignore-backups`	Don't list entries with the tilde (~) symbol (used to denote backup copies).
`-c`		Sort by time of last modification.
`-C`		List entries by columns.
	`--color=when`	When to use colors (always, never, or auto).
`-d`	`--directory`	List directory entries instead of contents, and don't dereference symbolic links.
`-F`	`--classify`	Append file-type indicator to entries.
	`--file-type`	Only append file-type indicators to some filetypes (not executable files).
	`--format=word`	Format output as either across, commas, horizontal, long, single-column, verbose, or vertical.
`-g`		List full file information except for the file's owner.
	`--group-directories-first`	List all directories before files.
`-G`	`--no-group`	In long listing don't display group names.
`-h`	`--human-readable`	Print sizes using K for kilobytes, M for megabytes, and G for gigabytes.
	`--si`	Same as `-h`, but use powers of 1000 instead of 1024.

`-i`	`--inode`	Display the index number (inode) of each file.
`-l`		Display the long listing format.
`-L`	`--dereference`	Show information for the original file for a linked file.
`-n`	`--numeric-uid-gid`	Show numeric userid and groupid instead of names.
`-o`		In long listing don't display owner names.
`-r`	`--reverse`	Reverse the sorting order when displaying files and directories.
`-R`	`--recursive`	List subdirectory contents recursively.
`-s`	`--size`	Print the block size of each file.
`-S`	`--sort=size`	Sort the output by file size.
`-t`	`--sort=time`	Sort the output by file modification time.
`-u`		Display file last access time instead of last modification time.
`-U`	`--sort=none`	Don't sort the output listing.
`-v`	`--sort=version`	Sort the output by file version.
`-x`		List entries by line instead of columns.
`-X`	`--sort=extension`	Sort the output by file extension.

You can use more than one parameter at a time if you want to. The double dash parameters must be listed separately, but the single dash parameters can be combined together into a string behind the dash. A common combination to use is the `-a` parameter to list all files, the `-i` parameter to list the *inode* for each file, the `-l`parameter to produce a long listing, and the `-s` parameter to list the block size of the files. The inode of a file or directory is a unique identification number the kernel assigns to each object in the filesystem. Combining all of these parameters creates the easy-to-remember `-sail` parameter:

```
$ ls -sail
total 2360
301860    8 drwx------ 36 rich rich    4096 2010-09-03 15:12 .
 65473    8 drwxr-xr-x  6 root root    4096 2010-07-29 14:20 ..
360621    8 drwxrwxr-x  2 rich rich    4096 2010-08-24 22:04 4rich
301862    8 -rw-r--r--  1 rich rich     124 2010-02-12 10:18 .bashrc
361443    8 drwxrwxr-x  4 rich rich    4096 2010-07-26 20:31 .ccache
301879    8 drwxr-xr-x  3 rich rich    4096 2010-07-26 18:25 .config
301871    8 drwxr-xr-x  3 rich rich    4096 2010-08-31 22:24 Desktop
301870    8 -rw-------  1 rich rich      26 2009-11-01 04:06 .dmrc
301872    8 drwxr-xr-x  2 rich rich    4096 2009-11-01 04:06 Download
360207    8 drwxrwxr-x  2 rich rich    4096 2010-07-26 18:25 Drivers
301882    8 drwx------  5 rich rich    4096 2010-09-02 23:40 .gconf
301883    8 drwx------  2 rich rich    4096 2010-09-02 23:43 .gconfd
360338    8 drwx------  3 rich rich    4096 2010-08-06 23:06 .gftp
```

In addition to the normal `-l` parameter output information, you'll see two additional numbers added to each line. The first number in the listing is the file or directory inode number. The second number is the block size of the file. The third entry is a diagram of the type of file, along with the file's permissions. We dive into that in more detail in Chapter 6.

Following that, the next number is the number of hard links to the file (discussed later in the "Linking Files" section), the owner of the file, the group the file belongs to, the size of the file (in bytes), a timestamp showing the last modification time by default, and finally, the actual file name.

Filtering Listing Output

As you've seen in the examples, by default the `ls` command lists all of the files in a directory. Sometimes this can be overkill, especially when you're just looking for information on a single file.

Fortunately, the `ls` command also provides a way for you to define a filter on the command line. It uses the filter to determine which files or directories it should display in the output.

The filter works as a simple text-matching string. Include the filter after any command line parameters you want to use:

```
$ ls -l myprog
-rwxr--r-- 1 rich rich 30 2007-08-23 21:42 myprog
$
```

When you specify the name of specific file as the filter, the `ls` command only shows the information for that one file. Sometimes you might not know the exact name of the file you're looking for. The `ls` command also recognizes standard wildcard characters and uses them to match patterns within the filter:

- A question mark to represent one character
- An asterisk to represent zero or more characters

The question mark can be used to replace exactly one character anywhere in the filter string. For example:

```
$ ls -l mypro?
-rw-rw-r-- 1 rich rich  0 2010-09-03 16:38 myprob
-rwxr--r-- 1 rich rich 30 2010-08-23 21:42 myprog
$
```

The filter `mypro?` matched two files in the directory. Similarly, the asterisk can be used to match zero or more characters:

```
$ ls -l myprob*
-rw-rw-r-- 1 rich rich 0 2010-09-03 16:38 myprob
-rw-rw-r-- 1 rich rich 0 2010-09-03 16:40 myproblem
$
```

The asterisk matches zero characters in the `myprob` file, but it matches three characters in the myproblem file.

This is a powerful feature to use when searching for files when you're not quite sure of the file names.

File Handling

The bash shell provides lots of commands for manipulating files on the Linux filesystem. This section walks you through the basic commands you will need to work with files from the CLI for all your file-handling needs.

Creating Files

Every once in a while you will run into a situation where you need to create an empty file. Sometimes applications expect a log file to be present before they can write to it. In these situations, you can use the `touch` command to easily create an empty file:

```
$ touch test1
$ ls -il test1
1954793 -rw-r--r--  1 rich     rich         0 Sep  1 09:35 test1
$
```

The `touch` command creates the new file you specify and assigns your username as the file owner. Because the `-il`parameter was used for the `ls` command, the first entry in the listing shows the inode number assigned to the file. Every file on a Linux filesystem has a unique inode number.

Notice that the file size is zero because the `touch` command just created an empty file. The touch command can also be used to change the access and modification times on an existing file without changing the file contents:

```
$ touch test1
$ ls -l test1
-rw-r--r--    1 rich     rich       0 Sep  1 09:37 test1
$
```

The modification time of `test1` is now updated from the original time. If you want to change only the access time, use the `-a` parameter. To change only the modification time, use the `-m` parameter. By default, `touch` uses the current time. You can specify the time by using the `-t` parameter with a specific timestamp:

```
$ touch -t 201112251200 test1
$ ls -l test1
-rw-r--r--    1 rich     rich       0 Dec 25  2011 test1
$
```

Now the modification time for the file is set to a date significantly in the future from the current time.

Copying Files

Copying files and directories from one location in the filesystem to another is a common practice for system administrators. The `cp` command provides this feature.

In its most basic form, the `cp` command uses two parameters, the source object and the destination object:

```
cp source destination
```

When both the `source` and `destination` parameters are file names, the `cp` command copies the source file to a new file with the file name specified as the destination. The new file acts like a brand new file, with an updated file creation and last modified times:

```
$ cp test1 test2
$ ls -il
total 0
1954793 -rw-r--r--    1 rich     rich       0 Dec 25  2011 test1
1954794 -rw-r--r--    1 rich     rich       0 Sep  1 09:39 test2
$
```

The new file `test2` shows a different inode number, indicating that it's a completely new file. You'll also notice that the modification time for the `test2` file shows the time that it was created. If the destination file already exists, the `cp` command will prompt you to answer whether or not you want to overwrite it:

```
$ cp test1 test2
cp: overwrite ‘test2’? y
$
```

If you don't answer `y`, the file copy will not proceed. You can also copy a file to an existing directory:

```
$ cp test1 dir1
$ ls -il dir1
total 0
1954887 -rw-r--r--    1 rich     rich       0 Sep  6 09:42 test1
$
```

The new file is now under the `dir1` directory, using the same file name as the original. These examples all used relative pathnames, but you can just as easily use the absolute pathname for both the source and destination objects.

To copy a file to the current directory you're in, you can use the dot symbol:

```
$ cp /home/rich/dir1/test1 .
cp: overwrite ‘./test1’?
```

As with most commands, the `cp` command has a few command line parameters to help you out. These are shown in Table 3.6.

Table 3.6 The cp Command Parameters

Parameter	Description
-a	Archive files by preserving their attributes.
-b	Create a backup of each existing destination file instead of overwriting it.
-d	Preserve.
-f	Force the overwriting of existing destination files without prompting.
-i	Prompt before overwriting destination files.
-l	Create a file link instead of copying the files.
-p	Preserve file attributes if possible.
-r	Copy files recursively.

-R	Copy directories recursively.
-s	Create a symbolic link instead of copying the file.
-S	Override the backup feature.
-u	Copy the source file only if it has a newer date and time than the destination (update).
-v	Verbose mode, explaining what's happening.
-x	Restrict the copy to the current filesystem.

Use the `-p` parameter to preserve the file access or modification times of the original file for the copied file.

```
$ cp -p test1 test3
$ ls -il
total 4
1954886 drwxr-xr-x    2 rich     rich         4096 Sep  1 09:42 dir1/
1954793 -rw-r--r--    1 rich     rich            0 Dec 25  2011 test1
1954794 -rw-r--r--    1 rich     rich            0 Sep  1 09:39 test2
1954888 -rw-r--r--    1 rich     rich            0 Dec 25  2011 test3
$
```

Now, even though the `test3` file is a completely new file, it has the same timestamps as the original `test1` file.

The `-R` parameter is extremely powerful. It allows you to recursively copy the contents of an entire directory in one command:

```
$ cp -R dir1 dir2
$ ls -l
total 8
drwxr-xr-x    2 rich     rich         4096 Sep  6 09:42 dir1/
drwxr-xr-x    2 rich     rich         4096 Sep  6 09:45 dir2/
-rw-r--r--    1 rich     rich            0 Dec 25  2011 test1
-rw-r--r--    1 rich     rich            0 Sep  6 09:39 test2
-rw-r--r--    1 rich     rich            0 Dec 25  2011 test3
$
```

Now `dir2` is a complete copy of `dir1`. You can also use wildcard characters in your `cp` commands:

```
$ cp -f test* dir2
$ ls -al dir2
total 12
drwxr-xr-x    2 rich     rich         4096 Sep  6 10:55 ./
drwxr-xr-x    4 rich     rich         4096 Sep  6 10:46 ../
-rw-r--r--    1 rich     rich            0 Dec 25  2011 test1
-rw-r--r--    1 rich     rich            0 Sep  6 10:55 test2
-rw-r--r--    1 rich     rich            0 Dec 25  2011 test3
$
```

This command copied all of the files that started with `test` to `dir2`. The `-f` parameter was included to force the overwrite of the `test1` file that was already in the directory without asking.

Linking Files

You may have noticed a couple of the parameters for the cp command referred to linking files. This is a pretty cool option available in the Linux filesystems. If you need to maintain two (or more) copies of the same file on the system, instead of having separate physical copies, you can use one physical copy and multiple virtual copies, called *links*. A link is a placeholder in a directory that points to the real location of the file. There are two different types of file links in Linux:

- A symbolic, or soft link
- A hard link

The hard link creates a separate file that contains information about the original file and where to locate it. When you reference the hard link file, it's just as if you're referencing the original file:

```
$ cp -l test1 test4
$ ls -il
total 16
1954886 drwxr-xr-x    2 rich     rich    4096 Sep  1 09:42 dir1/
1954889 drwxr-xr-x    2 rich     rich    4096 Sep  1 09:45 dir2/
1954793 -rw-r--r--    2 rich     rich       0 Sep  1 09:51 test1
1954794 -rw-r--r--    1 rich     rich       0 Sep  1 09:39 test2
1954888 -rw-r--r--    1 rich     rich       0 Dec 25  2011 test3
1954793 -rw-r--r--    2 rich     rich       0 Sep  1 09:51 test4
$
```

The `-l` parameter created a hard link for the `test1` file called `test4`. In the file listing, you can see that the inode number of both the `test1` and `test4` files is the same, indicating that, in reality, they are both the same file. Also notice that the link count (the third item in the listing) now shows that both files have two links.

Note

You can only create a hard link between files on the same physical medium. You can't create a hard link between files under separate mount points. In that case, you'll have to use a soft link.

On the other hand, the `-s` parameter creates a symbolic, or soft link:

```
$ cp -s test1 test5
$ ls -il test*
total 16
1954793 -rw-r--r--    2 rich     rich   6 Sep  1 09:51 test1
1954794 -rw-r--r--    1 rich     rich   0 Sep  1 09:39 test2
1954888 -rw-r--r--    1 rich     rich   0 Dec 25  2011 test3
1954793 -rw-r--r--    2 rich     rich   6 Sep  1 09:51 test4
1954891 lrwxrwxrwx    1 rich     rich   5 Sep  1 09:56 test5 -> test1
$
```

There are a couple of things to notice in the file listing, First, you'll notice that the new `test5` file has a different inode number than the `test1` file, indicating that the Linux system treats it as a separate file. Second, the file size is smaller. A linked file needs to store only information about the source file, not the actual data in the file. The file name area of the listing shows the relationship between the two files.

Tip

Instead of using the cp command, if you want to link files you can also use the ln command. By default, the ln command creates hard links. If you want to create a soft link, you'll still need to use the -s parameter.

Be careful when copying linked files. If you use the cp command to copy a file that's linked to another source file, all you're doing is making another copy of the source file. This can quickly get confusing. Instead of copying the linked file, you can create another link to the original file. You can have many links to the same file with no problems. However, you also don't want to create soft links to other soft-linked files. This creates a chain of links that can not only be confusing but also be easily broken, causing all sorts of problems.

Renaming Files

In the Linux world, renaming files is called *moving*. The mv command is available to move both files and directories to another location:

```
$ mv test2 test6
$ ls -il test*
1954793 -rw-r--r--    2 rich     rich   6 Sep  1 09:51 test1
1954888 -rw-r--r--    1 rich     rich   0 Dec 25  2011 test3
1954793 -rw-r--r--    2 rich     rich   6 Sep  1 09:51 test4
1954891 lrwxrwxrwx    1 rich     rich   5 Sep  1 09:56 test5 -> test1
1954794 -rw-r--r--    1 rich     rich   0 Sep  1 09:39 test6
$
```

Notice that moving the file changed the file name but kept the same inode number and the timestamp value. Moving a file with soft links is a problem:

```
$ mv test1 test8
$ ls -il test*
total 16
1954888 -rw-r--r--    1 rich     rich  0 Dec 25  2011 test3
1954793 -rw-r--r--    2 rich     rich  6 Sep  1 09:51 test4
1954891 lrwxrwxrwx    1 rich     rich  5 Sep  1 09:56 test5 -> test1
1954794 -rw-r--r--    1 rich     rich  0 Sep  1 09:39 test6
1954793 -rw-r--r--    2 rich     rich  6 Sep  1 09:51 test8
[rich@test2 clsc]$ mv test8 test1
```

The test4 file that uses a hard link still uses the same inode number, which is perfectly fine. However, thetest5 file now points to an invalid file, and it is no longer a valid link.

You can also use the mv command to move directories:

```
$ mv dir2 dir4
```

The entire contents of the directory are unchanged. The only thing that changes is the name of the directory. Thus, the mv command operates much faster than the cp command.

Deleting Files

Most likely at some point in your Linux career, you'll want to be able to delete existing files. Whether it's to clean up a filesystem or to remove a software package, there are always opportunities to delete files.

In the Linux world, deleting is called *removing*. The command to remove files in the bash shell is rm. The basic form of the rm command is pretty simple:

```
$ rm -i test2
rm: remove ‘test2’? y
$ ls -l
total 16
drwxr-xr-x    2 rich     rich    4096 Sep  1 09:42 dir1/
drwxr-xr-x    2 rich     rich    4096 Sep  1 09:45 dir2/
-rw-r--r--    2 rich     rich       6 Sep  1 09:51 test1
-rw-r--r--    1 rich     rich       0 Dec 25  2011 test3
-rw-r--r--    2 rich     rich       6 Sep  1 09:51 test4
lrwxrwxrwx    1 rich     rich       5 Sep  1 09:56 test5 -> test1
$
```

Notice that the command prompts you to make sure that you're serious about removing the file. There's no recycle bin or trashcan in the bash shell. Once you remove a file, it's gone forever.

Now, here's an interesting tidbit about deleting a file that has links to it:

```
$ rm test1
$ ls -l
total 12
drwxr-xr-x    2 rich     rich    4096 Sep  1 09:42 dir1/
drwxr-xr-x    2 rich     rich    4096 Sep  1 09:45 dir2/
-rw-r--r--    1 rich     rich       0 Dec 25  2011 test3
-rw-r--r--    1 rich     rich       6 Sep  1 09:51 test4
lrwxrwxrwx    1 rich     rich       5 Sep  1 09:56 test5 -> test1
$ cat test4
hello
$ cat test5
cat: test5: No such file or directory
$
```

The test1 file was removed, which had both a hard link with the test4 file and a soft link with the test5 file. Notice what happened. Both of the linked files still appear, even though the test1 file is now gone (although on my color terminal the test5 file name now appears in red). When you look at the contents of the test4 file that was a hard link, it still shows the contents of the file. When you look at the contents of the test5 file that was a soft link, bash indicates that it doesn't exist anymore.

Remember that the hard link file uses the same inode number as the original file. The hard link file maintains that inode number until you remove the last file hard-linked to it, preserving the data! All the soft link file knows is that the underlying file is now gone, so it has nothing to point to. This is an important feature to remember when working with linked files.

One other feature of the `rm` command, if you're removing lots of files and don't want to be bothered with the prompt, is to use the `-f` parameter to force the removal. Just be careful!

Tip

As with copying files, you can use wildcard characters with the `rm` command. Again, use caution when doing this, as anything your remove, even by accident, is gone forever!

Directory Handling

In Linux there are a few commands that work for both files and directories (such as the `cp` command), and some that only work for directories. To create a new directory, you'll need to use a specific command, which is covered in this section. Removing directories can get interesting, so that is covered in this section as well.

Creating Directories

There's not much to creating a new directory in Linux—just use the `mkdir` command:

```
$ mkdir dir3
$ ls -il
total 16
1954886 drwxr-xr-x    2 rich     rich    4096 Sep  1 09:42 dir1/
1954889 drwxr-xr-x    2 rich     rich    4096 Sep  1 10:55 dir2/
1954893 drwxr-xr-x    2 rich     rich    4096 Sep  1 11:01 dir3/
1954888 -rw-r--r--    1 rich     rich       0 Dec 25  2011 test3
1954793 -rw-r--r--    1 rich     rich       6 Sep  1 09:51 test4
$
```

The system creates a new directory and assigns it a new inode number.

Deleting Directories

Removing directories can be tricky, but there's a reason for that. There are lots of opportunities for bad things to happen when you start deleting directories. The bash shell tries to protect us from accidental catastrophes as much as possible. The basic command for removing a directory is `rmdir`:

```
$ rmdir dir3
$ rmdir dir1
rmdir: dir1: Directory not empty
$
```

By default, the `rmdir` command only works for removing empty directories. Because there is a file in the `dir1`directory, the `rmdir` command refuses to remove it. You can remove nonempty directories using the `--ignore-fail-on-non-empty` parameter.

Our friend the `rm` command can also help us out some when handling directories.

If you try using it with no parameters, as with files, you'll be somewhat disappointed:

```
$ rm dir1
rm: dir1: is a directory
$
```

However, if you really want to remove a directory, you can use the `-r` parameter to recursively remove the files in the directory, then the directory itself:

```
$ rm -r dir2
rm: descend into directory ‘dir2’? y
rm: remove ‘dir2/test1’? y
rm: remove ‘dir2/test3’? y
rm: remove ‘dir2/test4’? y
rm: remove directory ‘dir2’? y
$
```

While this works, it's somewhat awkward. Notice that you still must verify every file that gets removed. For a directory with lots of files and subdirectories, this can become tedious.

The ultimate solution for throwing caution to the wind and removing an entire directory, contents and all, is the`rm` command with both the `-r` and `-f` parameters:

```
$ rm -rf dir2
$
```

That's it. No warnings, no fanfare, just another shell prompt. This, of course, is an extremely dangerous tool to have, especially if you're logged in as the root user account. Use it sparingly, and only after triple checking to make sure that you're doing exactly what you want to do.

Note

You may have noticed in the last example that the two command line parameters were combined using one dash. This is a feature in the bash shell that allows you to combine command line parameters to help cut down on typing.

Viewing File Contents

So far we've covered everything there is to know about files, except for how to peek inside of them. There are several commands available for taking a look inside files without having to pull out an editor (see Chapter 11). This section demonstrates a few of the commands you have available to help you examine files.

Viewing File Statistics

You've already seen that the `ls` command can be used to provide lots of useful information about files. However, there's still more information that you can't see in the `ls` command (or at least not all at once).

The `stat` command provides a complete rundown of the status of a file on the filesystem:

```
$ stat test10
  File: “test10”
  Size: 6              Blocks: 8          Regular File
Device: 306h/774d       Inode: 1954891     Links: 2
Access: (0644/-rw-r--r--)  Uid: (  501/ rich) Gid: ( 501/ rich)
Access: Sat Sep  1 12:10:25 2010
Modify: Sat Sep  1 12:11:17 2010
Change: Sat Sep  1 12:16:42 2010
$
```

The results from the `stat` command show just about everything you'd want to know about the file being examined, even down to the major and minor device numbers of the device where the file is being stored.

Viewing the File Type

Despite all of the information the `stat` command produces, there's still one piece of information missing—the file type. Before you go charging off trying to list out a 1000-byte file, it's usually a good idea to get a handle on what type of file it is. If you try listing a binary file, you'll get lots of gibberish on your monitor and possibly even lock up your terminal emulator.

The `file` command is a handy little utility to have around. It has the ability to peek inside of a file and determine just what kind of file it is:

```
$ file test1
test1: ASCII text
$ file myscript
myscript: Bourne shell script text executable
$ file myprog
myprog: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), not stripped
$
```

The `file` command classifies files into three categories:

- **Text files:** Files that contain printable characters
- **Executable files:** Files that you can run on the system
- **Data files:** Files that contain nonprintable binary characters, but that you can't run on the system

The first example shows a text file. The `file` command determined not only that the file contains text but also the character code format of the text. The second example shows a text script file. While the file is text, because it's a script file, you can execute (run) it on the system. The final example is a binary executable program. The `file` command determines the platform that the program was compiled for and what types of libraries it requires. This is an especially handy feature if you have a binary executable program from an unknown source.

Viewing the Whole File

If you have a large text file on your hands, you may want to be able to see what's inside of it. There are three different commands in Linux that can help you out here.

The cat Command

The `cat` command is a handy tool for displaying all of the data inside a text file:

```
$ cat test1
hello


This is a test file.


That we'll use to       test the cat command.
$
```

Nothing too exciting, just the contents of the text file. There are a few parameters you can use with the `cat`command, however, that can help you out.

The `-n` parameter numbers all of the lines for you:

```
$ cat -n test1
     1  hello

     2

     3  This is a test file.

     4

     5

     6  That we'll use to       test the cat command.
$
```

That feature will come in handy when you're examining scripts. If you just want to number the lines that have text in them, the `-b` parameter is for you:

```
$ cat -b test1
     1  hello


     2  This is a test file.


     3  That we'll use to       test the cat command.
$
```

If you need to compress multiple blank lines into a single blank line, use the -s parameter:

```
$ cat -s test1
hello


This is a test file.


That we'll use to       test the cat command.
$
```

Finally, if you don't want tab characters to appear, use the `-T` parameter:

```
$ cat -T test1
hello


This is a test file.


That we'll use to^Itest the cat command.
$
```

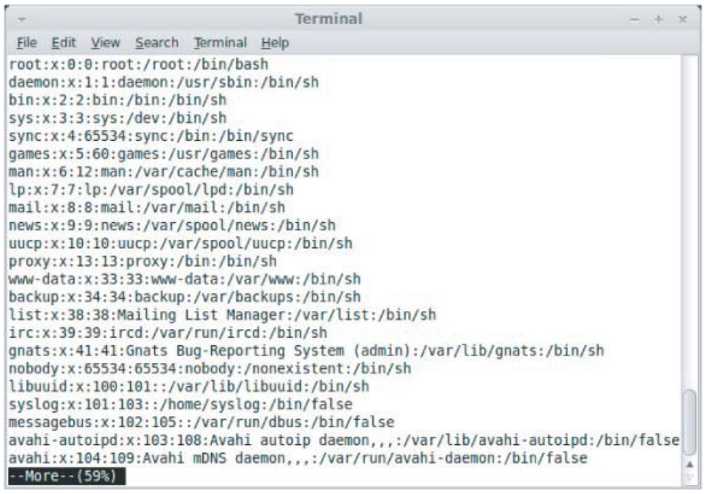
The `-T` parameter replaces any tabs in the text with the ^I character combination.

For large files, the `cat` command can be somewhat annoying. The text in the file will just quickly scroll off of the monitor without stopping. Fortunately, there's a simple way to solve this problem.

The more Command

The main drawback of the `cat` command is that you can't control what's happening once you start it. To solve that problem, developers created the `more` command. The `more` command displays a text file, but stops after it displays each page of data. A sample `more` screen is shown in Figure 3.4.

Figure 3.4 Using the `more` command to display a text file



```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuuid:x:100:101::/var/lib/libuuid:/bin/sh
syslog:x:101:103::/home/syslog:/bin/false
messagebus:x:102:105::/var/run/dbus:/bin/false
avahi-autoipd:x:103:108:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/bin/false
avahi:x:104:109:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false
--More--(59%)
```

Notice that at the bottom of the screen in Figure 3.4, the `more` command displays a tag showing that you're still in the `more` application and how far along in the text file you are. This is the prompt for the `more` command. At this point, you can enter one of several options, shown in Table 3.7.

Table 3.7 The more Command Options

Option	**Description**
H	Display a help menu.
spacebar	Display the next screen of text from the file.
z	Display the next screen of text from the file.
ENTER	Display one more line of text from the file.
d	Display a half-screen (11 lines) of text from the file.
q	Exit the program.
s	Skip forward one line of text.
f	Skip forward one screen of text.
b	Skip backward one screen of text.
/expression	Search for the text expression in the file.
n	Search for the next occurrence of the last specified expression.
’	Go to the first occurrence of the specified expression.
!cmd	Execute a shell command.
v	Start up the vi editor at the current line.
CTRL-L	Redraw the screen at the current location in the file.
=	Display the current line number in the file.
.	Repeat the previous command.

The `more` command allows some rudimentary movement through the text file. For more advanced features, try the `less`command.

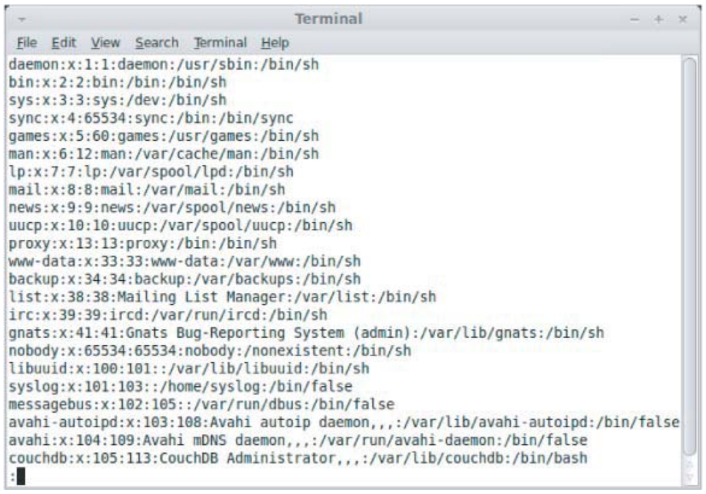
The less Command

Although from its name it sounds like it shouldn't be as advanced as the `more` command, the `less` command is actually a play on words and is an advanced version of the `more` command (the `less` command name comes from the phrase “less is more”). It provides several very handy features for scrolling both forward and backward through a text file, as well as some pretty advanced searching capabilities.

The `less` command can also display the contents of a file before it finishes reading the entire file. This is a serious drawback for both the `cat` and `more` commands when viewing extremely large files.

The `less` command operates much the same as the `more` command, displaying one screen of text from a file at a time.Figure 3.5 shows the `less` command in action.

Figure 3.5 Viewing a file using the `less` command



```
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuuid:x:100:101::/var/lib/libuuid:/bin/sh
syslog:x:101:103::/home/syslog:/bin/false
messagebus:x:102:105::/var/run/dbus:/bin/false
avahi-autoipd:x:103:108:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/bin/false
avahi:x:104:109:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false
couchdb:x:105:113:CouchDB Administrator,,,:/var/lib/couchdb:/bin/bash
:
```

Notice that the `less` command provides additional information in its prompt, showing the total number of lines in the file and the range of lines currently displayed. The `less` command supports the same command set as the `more`command plus lots more options. To see all of the options available, look at the man pages for the `less` command. One set of features is that the `less` command recognizes the up and down arrow keys as well as the page up and page down keys (assuming that you're using a properly defined terminal). This gives you full control when viewing a file.

Viewing Parts of a File

Often the data you want to view is located either right at the top or buried at the bottom of a text file. If the information is at the top of a large file, you still need to wait for the `cat` or `more` commands to load the entire file before you can view it. If the information is located at the bottom of a file (such as a log file), you need to wade through thousands of lines of text just to get to the last few entries. Fortunately, Linux has specialized commands to solve both of these problems.

The tail Command

The tail command displays the last group of lines in a file. By default, it will show the last 10 lines in the file, but you can change that with command line parameters, shown in Table 3.8.

Table 3.8 The tail Command Line Parameters

Parameter	**Description**
`-c bytes`	Display the last byte's number of bytes in the file.
`-n lines`	Display the last line's number of lines in the file.
`-f`	Keeps the tail program active and continues to display new lines as they're added to the file.
`--pid=PID`	Along with `-f`, follows a file until the process with ID PID terminates.
`-s sec`	Along with `-f`, sleeps for sec seconds between iterations.
`-v`	Always displays output headers giving the file name.

`-q`	Never displays output headers giving the file name.

The `-f` parameter is a pretty cool feature of the `tail` command. It allows you to peek inside a file as it's being used by other processes. The tail command stays active and continues to display new lines as they appear in the text file. This is a great way to monitor the system log file in real-time mode.

The head Command

While not as exotic as the `tail` command, the `head` command does what you'd expect; it displays the first group of lines at the start of a file. By default, it will display the first 10 lines of text. Similar to the tail command, it supports the `-c` and `-n` parameters so that you can alter what's displayed.

Usually the beginning of a file doesn't change, so the `head` command doesn't support the `-f` parameter feature. The`head` command is a handy way to just peek at the beginning of a file if you're not sure what's inside, without having to go through the hassle of displaying the entire file.

Summary

This chapter covered the basics of working with the Linux filesystem from a shell prompt. We began with a discussion of the bash shell and showed you how to interact with the shell. The command line interface (CLI) uses a prompt string to indicate when it's ready for you to enter commands. You can customize the prompt string to display useful information about your system, your logon ID, and even dates and times.

The bash shell provides a wealth of utilities you can use to create and manipulate files. Before you start playing with files, it's a good idea to understand how Linux stores them. This chapter discussed the basics of the Linux virtual directory and showed you how Linux references store media devices. After describing the Linux filesystem, the chapter walked you through using the `cd` command to move around the virtual directory.

After showing you how to get to a directory, the chapter demonstrated how to use the `ls` command to list the files and subdirectories. There are lots of parameters that customize the output of the `ls` command. You can obtain information on files and directories just by using the `ls` command.

The `touch` command is useful for creating empty files and for changing the access or modification times on an existing file. The chapter also discussed using the `cp` command to copy existing files from one location to another. It walked you through the process of linking files instead of copying them, providing an easy way to have the same file in two locations without making a separate copy. The `cp` command does this, as does the `ln` command.

Next, you learned how to rename files (called *moving*) in Linux using the `mv` command, and saw how to delete files (called *removing*) using the `rm` command. It also showed you how to perform the same tasks with directories, using the `mkdir` and `rmdir` commands.

Finally, the chapter closed with a discussion on viewing the contents of files. The `cat`, `more`, and `less` commands provide easy methods for viewing the entire contents of a file, while the `tail` and head commands are great for peeking inside a file to just see a small portion of it.

The next chapter continues the discussion on bash shell commands. We'll take a look at more advanced administrator commands that will come in handy as you administer your Linux system.

Chapter 4

More bash Shell Commands

In This Chapter

This step changes the permissions of the directory to give you complete access and everyone else no access at all. (The new permissions should read `rwx------`.)

5. Make the test directory your current directory as follows:

```
$ cd test
$ pwd
/home/joe/test
```

If you followed along, at this point a subdirectory of your home directory called `test` is your current working directory. You can create files and directories in the `test` directory along with the descriptions in the rest of this chapter.

Using Metacharacters and Operators

Whether you are listing, moving, copying, removing, or otherwise acting on files in your Linux system, certain special characters, referred to as metacharacters and operators, help you to work with files more efficiently. Metacharacters can help you match one or more files without completely typing each file name. Operators enable you to direct information from one command or file to another command or file.

Using file-matching metacharacters

To save you some keystrokes and to enable you to refer easily to a group of files, the bash shell lets you use metacharacters. Any time you need to refer to a file or directory, such as to list it, open it, or remove it, you can use metacharacters to match the files you want. Here are some useful metacharacters for matching filenames:

- `*` — Matches any number of characters.
- `?` — Matches any one character.
- `[...]` — Matches any one of the characters between the brackets, which can include a hyphen-separated range of letters or numbers.

Try out some of these file-matching metacharacters by first going to an empty directory (such as the `test` directory described in the previous section) and creating some empty files:

```
$ touch apple banana grape grapefruit watermelon
```

The `touch` command creates empty files. The commands that follow show you how to use shell metacharacters with the `ls` command to match filenames. Try the following commands to see whether you get the same responses:

```
$ ls a*
apple
$ ls g*
grape grapefruit
$ ls g*t
```

```
grapefruit
$ ls *e*
apple grape grapefruit watermelon
$ ls *n*
banana watermelon
```

The first example matches any file that begins with an `a` (`apple`). The next example matches any files that begin with `g` (`grape`, `grapefruit`). Next, files beginning with `g` and ending in `t` are matched (`grapefruit`). Next, any file that contains an `e` in the name is matched (`apple`, `grape`, `grapefruit`, `watermelon`). Finally, any file that contains an `n` is matched (`banana`, `watermelon`).

Here are a few examples of pattern matching with the question mark (`?`):

```
$ ls ????e
apple grape
$ ls g???e*
grape grapefruit
```

The first example matches any five-character file that ends in `e` (`apple`, `grape`). The second matches any file that begins with `g` and has `e` as its fifth character (`grape`, `grapefruit`).

The following examples use braces to do pattern matching:

```
$ ls [abw]*
apple banana watermelon
$ ls [agw]*[ne]
apple grape watermelon
```

In the first example, any file beginning with `a`, `b`, or `w` is matched. In the second, any file that begins with `a`, `g`, or `w` and also ends with either `n` or `e` is matched. You can also include ranges within brackets. For example:

```
$ ls [a-g]*
apple banana grape grapefruit
```

Here, any filenames beginning with a letter from `a` through `g` are matched.

Using file-redirection metacharacters

Commands receive data from standard input and send it to standard output. Using pipes (described earlier), you can direct standard output from one command to the standard input of another. With files, you can use less than (<) and greater than (>) signs to direct data to and from files. Here are the file-redirection characters:

- < — Directs the contents of a file to the command. In most cases, this is the default action expected by the command and the use of the character is optional; using `less bigfile` is the same as `less < bigfile`.

- `>` — Directs the standard output of a command to a file. If the file exists, the content of that file is overwritten.
- `2>` — Directs standard error (error messages) to the file.
- `&>` — Directs both standard output and standard error to the file.
- `>>` — Directs the output of a command to a file, adding the output to the end of the existing file.

The following are some examples of command lines where information is directed to and from files:

```
$ mail root  < ~/.bashrc
$ man chmod | col -b  > /tmp/chmod
$ echo "I finished the project on $(date)" >> ~/projects
```

In the first example, the content of the `.bashrc` file in the home directory is sent in a mail message to the computer's root user. The second command line formats the `chmod` man page (using the `man` command), removes extra back spaces (`col -b`), and sends the output to the file `/tmp/chmod` (erasing the previous `/tmp/chmod` file, if it exists). The final command results in the following text being added to the user's project file:

```
I finished the project on Sat Jan 22 13:46:49 PST 2011
```

Another type of redirection, referred to as *here text* (also called *here document*), enables you to type text that can be used as standard input for a command. Here documents involve entering two less-than characters (`<<`) after a command, followed by a word. All typing following that word is taken as user input until the word is repeated on a line by itself. Here is an example:

```
$ mail root cnegus rjones bdecker <<thetext
> I want to tell everyone that there will be a 10 a.m.
> meeting in conference room B. Everyone should attend.
>
> -- James
> thetext
$
```

This example sends a mail message to root, cnegus, rjones, and bdecker usernames. The text entered between `<<thetext` and `thetext` becomes the content of the message.
A common use of here text is to use it with a text editor to create or add to a file from within a script:

```
/bin/ed /etc/resolv.conf <<resendit
a
nameserver 100.100.100.100
.
w
q
resendit
```

With these lines added to a script run by the root user, the `ed` text editor adds the IP address of a DNS server to the `/etc/resolv.conf` file.

Using brace expansion characters

By using curly braces ({}), you can expand out a set of characters across filenames, directory names, or other arguments you give commands. For example, if you want to create a set of files such as memo1 through memo5, you can do that as follows:

```
$ touch memo{1,2,3,4,5}
$ ls
memo1    memo2    memo3    memo4    memo5
```

The items that are expanded don't have to be number or even single digits. For example, you could use ranges of numbers or digits. You could also use any string of characters, as long as you separate them with commas. Here are some examples:

```
$ touch {John,Bill,Sally}-{Breakfast,Lunch,Dinner}
$ ls
Bill-Breakfast  Bill-Lunch      John-Dinner  Sally-Breakfast  Sally-Lunch
Bill-Dinner     John-Breakfast  John-Lunch   Sally-Dinner
$ rm -f {John,Bill,Sally}-{Breakfast,Lunch,Dinner}
$ touch {a..f}{1..5}
$ ls
a1  a3  a5  b2  b4  c1  c3  c5  d2  d4  e1  e3  e5  f2  f4
a2  a4  b1  b3  b5  c2  c4  d1  d3  d5  e2  e4  f1  f3  f5
```

In the first example, the use of two sets of braces means John, Bill, and Sally each have filenames associated with Breakfast, Lunch, and Dinner. If I had made a mistake, I could easily recall the command and change `touch` to `rm -f` to delete all the files. In the next example, the use of two dots between letters a and f and numbers 1 and 5 specifies the ranges to be used. Notice the files that were created from those few characters.

Listing Files and Directories

The `ls` command is the most common command used to list information about files and directories. Many options available with the `ls` command allow you to gather different sets of files and directories, as well as to view different kinds of information about them.

By default, when you type the `ls` command, the output shows you all non-hidden files and directories contained in the current directory. When you type `ls`, however, many Linux systems (including Fedora and RHEL) assign an alias `ls` to add options. To see if `ls` is aliased, type the following:

```
$ alias ls
alias ls='ls --color=auto'
```

Chapter 15. redirection and pipes

Table of Contents

One of the powers of the Unix command line is the use of **redirection** and **pipes**.

This chapter first explains **redirection** of input, output and error streams. It then introduces **pipes** that consist of several **commands**.

15.1. stdin, stdout, and stderr

The shell (and almost every other Linux command) takes input from **stdin** (stream **0**) and sends output to **stdout** (stream **1**) and error messages to **stderr** (stream **2**) .

The keyboard often server as **stdin**, **stdout** and **stderr** both go to the disply. The shell allows you to redirect these streams.

15.2. output redirection

> stdout

stdout can be redirected with a **greater than** sign. While scanning the line, the shell will see the **>** sign and will clear the file.

```
[paul@RHELv4u3 ~]$ echo It is cold today!
It is cold today!
[paul@RHELv4u3 ~]$ echo It is cold today! > winter.txt
[paul@RHELv4u3 ~]$ cat winter.txt
It is cold today!
[paul@RHELv4u3 ~]$
```

Note that the **>** notation is in fact the abbreviation of **1>** (**stdout** being referred to as stream **1**.

output file is erased

To repeat: While scanning the line, the shell will see the > sign and **will clear the file**! This means that even when the command fails, the file will be cleared!

```
[paul@RHELv4u3 ~]$ cat winter.txt
It is cold today!
[paul@RHELv4u3 ~]$ zcho It is cold today! > winter.txt
-bash: zcho: command not found
[paul@RHELv4u3 ~]$ cat winter.txt
[paul@RHELv4u3 ~]$
```

noclobber

Erasing a file while using > can be prevented by setting the **noclobber** option.

```
[paul@RHELv4u3 ~]$ cat winter.txt
It is cold today!
[paul@RHELv4u3 ~]$ set -o noclobber
```

```
[paul@RHELv4u3 ~]$ echo It is cold today! > winter.txt
-bash: winter.txt: cannot overwrite existing file
[paul@RHELv4u3 ~]$ set +o noclobber
[paul@RHELv4u3 ~]$
```

overruling noclobber

The **noclobber** can be overruled with **>|**.

```
[paul@RHELv4u3 ~]$ set -o noclobber
[paul@RHELv4u3 ~]$ echo It is cold today! > winter.txt
-bash: winter.txt: cannot overwrite existing file
[paul@RHELv4u3 ~]$ echo It is very cold today! >| winter.txt
[paul@RHELv4u3 ~]$ cat winter.txt
It is very cold today!
[paul@RHELv4u3 ~]$
```

>> append

Use >> to **append** output to a file.

```
[paul@RHELv4u3 ~]$ echo It is cold today! > winter.txt
[paul@RHELv4u3 ~]$ cat winter.txt
It is cold today!
[paul@RHELv4u3 ~]$ echo Where is the summer ? >> winter.txt
[paul@RHELv4u3 ~]$ cat winter.txt
It is cold today!
Where is the summer ?
[paul@RHELv4u3 ~]$
```

15.3. error redirection

2> stderr

Redirecting **stderr** is done with **2>**. This can be very useful to prevent error messages from cluttering your screen. The screenshot below shows redirection of **stdout** to a file, and **stderr** to **/dev/null**. Writing **1>** is the same as >.

```
[paul@RHELv4u3 ~]$ find / > allfiles.txt 2> /dev/null
[paul@RHELv4u3 ~]$
```

2>&1

To redirect both **stdout** and **stderr** to the same file, use **2>&1**.

```
[paul@RHELv4u3 ~]$ find / > allfiles_and_errors.txt 2>&1
[paul@RHELv4u3 ~]$
```

Note that the order of redirections is significant. For example, the command

```
ls > dirlist 2>&1
```

directs both standard output (file descriptor 1) and standard error (file descriptor 2) to the file dirlist, while the command

```
ls 2>&1 > dirlist
```

directs only the standard output to file dirlist, because the standard error was made a copy of the standard output before the standard output was redirected to dirlist.

15.4. input redirection

< stdin

Redirecting **stdin** is done with < (short for 0<).

```
[paul@RHEL4b ~]$ cat < text.txt
one
two
[paul@RHEL4b ~]$ tr 'onetw' 'ONEZZ' < text.txt
ONE
ZZO
[paul@RHEL4b ~]$
```

<< here document

The **here document** (sometimes called here-is-document) is a way to append input until a certain sequence (usually EOF) is encountered. The **EOF** marker can be typed literally or can be called with Ctrl-D.

```
[paul@RHEL4b ~]$ cat <<EOF > text.txt
> one
> two
> EOF
[paul@RHEL4b ~]$ cat text.txt
one
two
[paul@RHEL4b ~]$ cat <<brol > text.txt
> brel
> brol
[paul@RHEL4b ~]$ cat text.txt
brel
[paul@RHEL4b ~]$
```

15.5. confusing redirection

The shell will scan the whole line before applying redirection. The following command line is very readable and is correct.

```
cat winter.txt > snow.txt 2> errors.txt
```

But this one is also correct, but less readable.

```
2> errors.txt cat winter.txt > snow.txt
```

Even this will be understood perfectly by the shell.

```
< winter.txt > snow.txt 2> errors.txt cat
```

15.6. quick file clear

So what is the quickest way to clear a file ?

```
>foo
```

And what is the quickest way to clear a file when the **noclobber** option is set ?

```
>|bar
```

15.7. swapping stdout and stderr

When filtering an output stream, e.g. through a regular pipe (|) you only can filter **stdout**. Say you want to filter out some unimportant error, out of the **stderr** stream. This cannot be done directly, and you need to 'swap' **stdout** and **stderr**. This can be done by using a 4th stream referred to with number 3:

```
3>&1 1>&2 2>&3
```

This Tower Of Hanoi like construction uses a temporary stream 3, to be able to swap **stdout** (1) and **stderr** (2). The following is an example of how to filter out all lines in the **stderr** stream, containing $uninterestingerror.

```
$command 3>&1 1>&2 2>&3 | grep -v $error 3>&1 1>&2 2>&3
```

But in this example, it can be done in a much shorter way, by using a pipe on STDERR:

```
/usr/bin/$somecommand |& grep -v $uninterestingerror
```

15.8. pipes

One of the most powerful advantages of **Linux** is the use of **pipes**.

A pipe takes **stdout** from the previous command and sends it as **stdin** to the next command. All commands in a **pipe** run simultaneously.

| vertical bar

Consider the following example.

```
paul@debian5:~/test$ ls /etc > etcfiles.txt
paul@debian5:~/test$ tail -4 etcfiles.txt
X11
xdg
xml
xpdf
paul@debian5:~/test$
```

This can be written in one command line using a **pipe**.

```
paul@debian5:~/test$ ls /etc | tail -4
X11
xdg
xml
xpdf
paul@debian5:~/test$
```

The **pipe** is represented by a vertical bar **|** between two commands.

multiple pipes

One command line can use multiple **pipes**. All commands in the **pipe** can run at the same time.

```
paul@deb503:~/test$ ls /etc | tail -4 | tac
xpdf
xml
xdg
X11
```

Chapter 16. filters

Table of Contents

Commands that are created to be used with a **pipe** are often called **filters**. These **filters** are very small programs that do one specific thing very efficiently. They can be used as **building blocks**.

This chapter will introduce you to the most common **filters**. The combination of simple commands and filters in a long **pipe** allows you to design elegant solutions.

16.1. cat

When between two **pipes**, the **cat** command does nothing (except putting **stdin** on **stdout**.

```
[paul@RHEL4b pipes]$ tac count.txt | cat | cat | cat | cat | cat
five
four
three
two
one
[paul@RHEL4b pipes]$
```

16.2. tee

Writing long **pipes** in Unix is fun, but sometimes you might want intermediate results. This is were **tee** comes in handy. The **tee** filter puts **stdin** on **stdout** and also into a file. So **tee** is almost the same as **cat**, except that it has two identical outputs.

```
[paul@RHEL4b pipes]$ tac count.txt | tee temp.txt | tac
one
two
three
four
five
[paul@RHEL4b pipes]$ cat temp.txt
five
four
three
two
one
[paul@RHEL4b pipes]$
```

16.3. grep

The **grep** filter is famous among Unix users. The most common use of **grep** is to filter lines of text containing (or not containing) a certain string.

```
[paul@RHEL4b pipes]$ cat tennis.txt
Amelie Mauresmo, Fra
Kim Clijsters, BEL
Justine Henin, Bel
Serena Williams, usa
Venus Williams, USA
[paul@RHEL4b pipes]$ cat tennis.txt | grep Williams
Serena Williams, usa
Venus Williams, USA
```

You can write this without the cat.

```
[paul@RHEL4b pipes]$ grep Williams tennis.txt
Serena Williams, usa
Venus Williams, USA
```

One of the most useful options of grep is **grep -i** which filters in a case insensitive way.

```
[paul@RHEL4b pipes]$ grep Bel tennis.txt
Justine Henin, Bel
[paul@RHEL4b pipes]$ grep -i Bel tennis.txt
Kim Clijsters, BEL
Justine Henin, Bel
[paul@RHEL4b pipes]$
```

Another very useful option is **grep -v** which outputs lines not matching the string.

```
[paul@RHEL4b pipes]$ grep -v Fra tennis.txt
Kim Clijsters, BEL
Justine Henin, Bel
Serena Williams, usa
Venus Williams, USA
[paul@RHEL4b pipes]$
```

And of course, both options can be combined to filter all lines not containing a case insensitive string.

```
[paul@RHEL4b pipes]$ grep -vi usa tennis.txt
Amelie Mauresmo, Fra
Kim Clijsters, BEL
Justine Henin, Bel
[paul@RHEL4b pipes]$
```

With **grep -A1** one line **after** the result is also displayed.

```
paul@debian5:~/pipes$ grep -A1 Henin tennis.txt
Justine Henin, Bel
Serena Williams, usa
```

With **grep -B1** one line **before** the result is also displayed.

```
paul@debian5:~/pipes$ grep -B1 Henin tennis.txt
Kim Clijsters, BEL
Justine Henin, Bel
```

With **grep -C1** (context) one line **before** and one **after** are also displayed. All three options (A,B, and C) can display any number of lines (using e.g. A2, B4 or C20).

```
paul@debian5:~/pipes$ grep -C1 Henin tennis.txt
Kim Clijsters, BEL
Justine Henin, Bel
Serena Williams, usa
```

16.4. cut

The **cut** filter can select columns from files, depending on a delimiter or a count of bytes. The screenshot below uses **cut** to filter for the username and userid in the **/etc/passwd** file. It uses the colon as a delimiter, and selects fields 1 and 3.

```
[[paul@RHEL4b pipes]$ cut -d: -f1,3 /etc/passwd | tail -4
Figo:510
Pfaff:511
Harry:516
Hermione:517
[paul@RHEL4b pipes]$
```

When using a space as the delimiter for **cut**, you have to quote the space.

```
[paul@RHEL4b pipes]$ cut -d" " -f1 tennis.txt
Amelie
Kim
Justine
Serena
Venus
[paul@RHEL4b pipes]$
```

This example uses **cut** to display the second to the seventh character of **/etc/passwd**.

```
[paul@RHEL4b pipes]$ cut -c2-7 /etc/passwd | tail -4
igo:x:
faff:x
arry:x
ermion
[paul@RHEL4b pipes]$
```

16.5. tr

You can translate characters with **tr**. The screenshot shows the translation of all occurrences of e to E.

```
[paul@RHEL4b pipes]$ cat tennis.txt | tr 'e' 'E'
AmEliE MaurEsmo, Fra
Kim ClijstErs, BEL
JustinE HEnin, BEl
SErEna Williams, usa
VEnus Williams, USA
```

Here we set all letters to uppercase by defining two ranges.

```
[paul@RHEL4b pipes]$ cat tennis.txt | tr 'a-z' 'A-Z'
AMELIE MAURESMO, FRA
KIM CLIJSTERS, BEL
JUSTINE HENIN, BEL
SERENA WILLIAMS, USA
VENUS WILLIAMS, USA
[paul@RHEL4b pipes]$
```

Here we translate all newlines to spaces.

```
[paul@RHEL4b pipes]$ cat count.txt
one
```

```
two
three
four
five
[paul@RHEL4b pipes]$ cat count.txt | tr '\n' ' '
one two three four five [paul@RHEL4b pipes]$
```

The **tr -s** filter can also be used to squeeze multiple occurrences of a character to one.

```
[paul@RHEL4b pipes]$ cat spaces.txt
one    two        three
     four   five  six
[paul@RHEL4b pipes]$ cat spaces.txt | tr -s ' '
one two three
 four five six
[paul@RHEL4b pipes]$
```

You can also use **tr** to 'encrypt' texts with **rot13**.

```
[paul@RHEL4b pipes]$ cat count.txt | tr 'a-z' 'nopqrstuvwxyzabcdefghijklm'
bar
gjb
guerr
sbhe
svir
[paul@RHEL4b pipes]$ cat count.txt | tr 'a-z' 'n-za-m'
bar
gjb
guerr
sbhe
svir
[paul@RHEL4b pipes]$
```

This last example uses **tr -d** to delete characters.

```
paul@debian5:~/pipes$ cat tennis.txt | tr -d e
Amli Maursmo, Fra
Kim Clijstrs, BEL
Justin Hnin, Bl
Srna Williams, usa
Vnus Williams, USA
```

16.6. wc

Counting words, lines and characters is easy with **wc**.

```
[paul@RHEL4b pipes]$ wc tennis.txt
  5  15 100 tennis.txt
[paul@RHEL4b pipes]$ wc -l tennis.txt
5 tennis.txt
[paul@RHEL4b pipes]$ wc -w tennis.txt
15 tennis.txt
[paul@RHEL4b pipes]$ wc -c tennis.txt
100 tennis.txt
```

```
[paul@RHEL4b pipes]$
```

16.7. sort

The **sort** filter will default to an alphabetical sort.

```
paul@debian5:~/pipes$ cat music.txt
Queen
Brel
Led Zeppelin
Abba
paul@debian5:~/pipes$ sort music.txt
Abba
Brel
Led Zeppelin
Queen
```

But the **sort** filter has many options to tweak its usage. This example shows sorting different columns (column 1 or column 2).

```
[paul@RHEL4b pipes]$ sort -k1 country.txt
Belgium, Brussels, 10
France, Paris, 60
Germany, Berlin, 100
Iran, Teheran, 70
Italy, Rome, 50
[paul@RHEL4b pipes]$ sort -k2 country.txt
Germany, Berlin, 100
Belgium, Brussels, 10
France, Paris, 60
Italy, Rome, 50
Iran, Teheran, 70
```

The screenshot below shows the difference between an alphabetical sort and a numerical sort (both on the third column).

```
[paul@RHEL4b pipes]$ sort -k3 country.txt
Belgium, Brussels, 10
Germany, Berlin, 100
Italy, Rome, 50
France, Paris, 60
Iran, Teheran, 70
[paul@RHEL4b pipes]$ sort -n -k3 country.txt
Belgium, Brussels, 10
Italy, Rome, 50
France, Paris, 60
Iran, Teheran, 70
Germany, Berlin, 100
```

16.8. uniq

With **uniq** you can remove duplicates from a **sorted list**.

```
paul@debian5:~/pipes$ cat music.txt
Queen
Brel
Queen
Abba
paul@debian5:~/pipes$ sort music.txt
Abba
Brel
Queen
Queen
paul@debian5:~/pipes$ sort music.txt |uniq
Abba
Brel
Queen
```

uniq can also count occurrences with the **-c** option.

```
paul@debian5:~/pipes$ sort music.txt |uniq -c
      1 Abba
      1 Brel
      2 Queen
```

16.9. comm

Comparing streams (or files) can be done with the **comm**. By default **comm** will output three columns. In this example, Abba, Cure and Queen are in both lists, Bowie and Sweet are only in the first file, Turner is only in the second.

```
paul@debian5:~/pipes$ cat > list1.txt
Abba
Bowie
Cure
Queen
Sweet
paul@debian5:~/pipes$ cat > list2.txt
Abba
Cure
Queen
Turner
paul@debian5:~/pipes$ comm list1.txt list2.txt
                Abba
Bowie
                Cure
                Queen
Sweet
        Turner
```

The output of **comm** can be easier to read when outputting only a single column. The digits point out which output columns should not be displayed.

```
paul@debian5:~/pipes$ comm -12 list1.txt list2.txt
Abba
Cure
Queen
paul@debian5:~/pipes$ comm -13 list1.txt list2.txt
Turner
paul@debian5:~/pipes$ comm -23 list1.txt list2.txt
```

17.1. find

The **find** command can be very useful at the start of a pipe to search for files. Here are some examples. You might want to add **2>/dev/null** to the command lines to avoid cluttering your screen with error messages.

Find all files in **/etc** and put the list in etcfiles.txt

```
find /etc > etcfiles.txt
```

Find all files of the entire system and put the list in allfiles.txt

```
find / > allfiles.txt
```

Find files that end in .conf in the current directory (and all subdirs).

```
find . -name "*.conf"
```

Find files of type file (not directory, pipe or etc.) that end in .conf.

```
find . -type f -name "*.conf"
```

Find files of type directory that end in .bak .

```
find /data -type d -name "*.bak"
```

Find files that are newer than file42.txt

```
find . -newer file42.txt
```

Find can also execute another command on every file found. This example will look for *.odf files and copy them to /backup/.

```
find /data -name "*.odf" -exec cp {} /backup/ \;
```

Find can also execute, after your confirmation, another command on every file found. This example will remove *.odf files if you approve of it for every file found.

```
find /data -name "*.odf" -ok rm {} \;
```

17.2. locate

The **locate** tool is very different from **find** in that it uses an index to locate files. This is a lot faster than traversing all the directories, but it also means that it is always outdated. If the index does not exist yet, then you have to create it (as root on Red Hat Enterprise Linux) with the **updatedb** command.

```
[paul@RHEL4b ~]$ locate Samba
warning: locate: could not open database: /var/lib/slocate/slocate.db:...
warning: You need to run the 'updatedb' command (as root) to create th...
Please have a look at /etc/updatedb.conf to enable the daily cron job.
[paul@RHEL4b ~]$ updatedb
fatal error: updatedb: You are not authorized to create a default sloc...
[paul@RHEL4b ~]$ su -
Password:
```

```
[root@RHEL4b ~]# updatedb
[root@RHEL4b ~]#
```

Most Linux distributions will schedule the **updatedb** to run once every day.

17.3. date

The **date** command can display the date, time, timezone and more.

```
paul@rhel55 ~$ date
Sat Apr 17 12:44:30 CEST 2010
```

A date string can be customized to display the format of your choice. Check the man page for more options.

```
paul@rhel55 ~$ date +'%A %d-%m-%Y'
Saturday 17-04-2010
```

Time on any Unix is calculated in number of seconds since 1969 (the first second being the first second of the first of January 1970). Use **date +%s** to display Unix time in seconds.

```
paul@rhel55 ~$ date +%s
1271501080
```

When will this seconds counter reach two thousand million ?

```
paul@rhel55 ~$ date -d '1970-01-01 + 2000000000 seconds'
Wed May 18 04:33:20 CEST 2033
```

17.4. cal

The **cal** command displays the current month, with the current day highlighted.

```
paul@rhel55 ~$ cal
     April 2010
Su Mo Tu We Th Fr Sa
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
```

You can select any month in the past or the future.

```
paul@rhel55 ~$ cal 2 1970
   February 1970
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
```

17.5. sleep

The **sleep** command is sometimes used in scripts to wait a number of seconds. This example shows a five second **sleep**.

```
paul@rhel55 ~$ sleep 5
paul@rhel55 ~$
```

17.6. time

The **time** command can display how long it takes to execute a command. The **date** command takes only a little time.

```
paul@rhel55 ~$ time date
Sat Apr 17 13:08:27 CEST 2010

real    0m0.014s
user    0m0.008s
sys     0m0.006s
```

The **sleep 5** command takes five **real** seconds to execute, but consumes little **cpu time**.

```
paul@rhel55 ~$ time sleep 5

real    0m5.018s
user    0m0.005s
sys     0m0.011s
```

This **bzip2** command compresses a file and uses a lot of **cpu time**.

```
paul@rhel55 ~$ time bzip2 text.txt

real    0m2.368s
user    0m0.847s
sys     0m0.539s
```

17.7. gzip - gunzip

Users never have enough disk space, so compression comes in handy. The **gzip** command can make files take up less space.

Compressing Data

If you've done any work in the Microsoft Windows world, no doubt you've used zip files. It became such a popular feature that Microsoft eventually incorporated it into the Windows XP operating system. The zip utility allows you to easily compress large files (both text and executable) into smaller files that take up less space.

Linux contains several file compression utilities. While this may sound great, it often leads to confusion and chaos when trying to download files. Table 4.8 lists the file compression utilities available for Linux.

Table 4.8 Linux File Compression Utilities

Utility	File Extension	Description
`bzip2`	`.bz2`	Uses the Burrows-Wheeler block sorting text compression algorithm and Huffman coding
`compress`	`.Z`	Original Unix file compression utility; starting to fade away into obscurity
`gzip`	`.gz`	The GNU Project's compression utility; uses Lempel-Ziv coding
`zip`	`.zip`	The Unix version of the PKZIP program for Windows

The `compress` file compression utility is not often found on Linux systems. If you download a file with a `.Z`extension, you can usually install the compress package (called `ncompress` in many Linux distributions) using the software installation methods discussed in Chapter 8, and then uncompress the file with the `uncompress` command.

The bzip2 Utility

The `bzip2` utility is a relatively new compression package that is gaining popularity, especially when compressing large binary files. The utilities in the `bzip2` package are:

- `bzip2` for compressing files
- `bzcat` for displaying the contents of compressed text files
- `bunzip2` for uncompressing compressed .bz2 files
- `bzip2recover` for attempting to recover damaged compressed files

By default, the `bzip2` command attempts to compress the original file and replaces it with the compressed file, using the same file name with a `.bz2` extension:

```
$ ls -l myprog
-rwxrwxr-x 1 rich rich 4882 2007-09-13 11:29 myprog
$ bzip2 myprog
$ ls -l my*
-rwxrwxr-x 1 rich rich 2378 2007-09-13 11:29 myprog.bz2
$
```

The original size of the `myprog` program was 4882 bytes, and after the `bzip2` compression it is now 2378 bytes. Also, notice that the `bzip2` command automatically renamed the original file with the `.bz2` extension, indicating what compression technique we used to compress it.

To uncompress the file, just use the `bunzip2` command:

```
$ bunzip2 myprog.bz2
$ ls -l myprog
-rwxrwxr-x 1 rich rich 4882 2007-09-13 11:29 myprog
$
```

As you can see, the uncompressed file is back to the original file size. Once you compress a text file, you can't use the standard `cat`, `more`, or `less` commands to view the data. Instead, you need to use the `bzcat` command:

```
$ bzcat test.bz2
This is a test text file.
The quick brown fox jumps over the lazy dog.
This is the end of the test text file.
$
```

The `bzcat` command displays the text inside the compressed file without uncompressing the actual file.

The gzip Utility

By far the most popular file compression utility in Linux is the `gzip` utility. The `gzip` package is a creation of the GNU Project, in their attempt to create a free version of the original Unix compress utility. This package includes the files:

- `gzip` for compressing files
- `gzcat` for displaying the contents of compressed text files
- `gunzip` for uncompressing files

These utilities work the same way as the bzip2 utilities:

```
$ gzip myprog
$ ls -l my*
-rwxrwxr-x 1 rich rich 2197 2007-09-13 11:29 myprog.gz
$
```

The `gzip` command compresses the file you specify on the command line. You can also specify more than one file name or even use wildcard characters to compress multiple files at once:

```
$ gzip my*
$ ls -l my*
-rwxr--r--    1 rich     rich          103 Sep  6 13:43 myprog.c.gz
-rwxr-xr-x    1 rich     rich         5178 Sep  6 13:43 myprog.gz
-rwxr--r--    1 rich     rich           59 Sep  6 13:46 myscript.gz
-rwxr--r--    1 rich     rich           60 Sep  6 13:44 myscript□.gz
```

```
$
```

The gzip command compresses every file in the directory that matches the wildcard pattern.

The zip Utility

The zip utility is compatible with the popular PKZIP package created by Phil Katz for MS-DOS and Windows. There are four utilities in the Linux zip package:

- zip creates a compressed file containing listed files and directories.
- zipcloak creates an encrypted compress file containing listed files and directories.
- zipnote extracts the comments from a zip file.
- zipsplit splits a zip file into smaller files of a set size (used for copying large zip files to floppy disks).
- unzip extracts files and directories from a compressed zip file.

To see all of the options available for the zip utility, just enter it by itself on the command line:

```
$ zip
Copyright (C) 1990-2005 Info-ZIP
Type ‘zip “-L”’ for software license.
Zip 2.31 (March 8th 2005). Usage:
zip [-options] [-b path] [-t mmddyyyy] [-n suffixes] [zipfile list]
[-xi list]
  The default action is to add or replace zipfile entries from list,
which can include the special name - to compress standard input.
 If zipfile and list are omitted, zip compresses stdin to stdout.
-f freshen: only changed files  -u update: only changed or new files
-d delete entries in zipfile    -m move into zipfile (delete files)
-r recurse into directories     -j junk directory names
-0 store only                   -l convert LF to CR LF
-1 compress faster              -9 compress better
-q quiet operation              -v verbose operation
-c add one-line comments        -z add zipfile comment
-@ read names from stdin        -o make file as old as latest entry
-x exclude the following names  -i include only the following names
-F fix zipfile (-FF try harder) -D do not add directory entries
```

```
-A adjust self-extracting exe   -J junk zipfile prefix (unzipsfx)
-T test zipfile integrity       -X eXclude eXtra file attributes
-y store symbolic links as the link instead of the referenced file
-R PKZIP recursion (see manual)
-e encrypt                      -n don't compress these suffixes
$
```

The power of the `zip` utility is its ability to compress entire directories of files into a single compressed file. This makes it ideal for archiving entire directory structures:

```
$ zip -r testzip test
 adding: test/ (stored 0%)
 adding: test/test1/ (stored 0%)
 adding: test/test1/myprog2 (stored 0%)
 adding: test/test1/myprog1 (stored 0%)
 adding: test/myprog.c (deflated 39%)
 adding: test/file3 (deflated 2%)
 adding: test/file4 (stored 0%)
 adding: test/test2/ (stored 0%)
 adding: test/file1.gz (stored 0%)
 adding: test/file2 (deflated 4%)
 adding: test/myprog.gz (stored 0%)
$
```

This example creates the zip file named `testzip.zip` and recurses through the directory `test`, adding each file and directory found to the zip file. Notice from the output that not all of the files stored in the zip file could be compressed. The `zip` utility automatically determines the best compression type to use for each individual file.

Caution

When you use the recursion feature in the zip command, files are stored in the same directory structure in the zip file. Files contained in subdirectories are stored in the zip file within the same subdirectories. You must be careful when extracting the files; the unzip command will rebuild the entire directory structure in the new location. Sometimes this gets confusing when you have lots of subdirectories and files.

Archiving Data

While the `zip` command works great for compressing and archiving data into a single file, it's not the standard utility used in the Unix and Linux worlds. By far the most popular archiving tool used in Unix and Linux is the `tar`command.

The `tar` command was originally used to write files to a tape device for archiving. However, it can also write the output to a file, which has become a popular way to archive data in Linux.

The following is the format of the `tar` command:

```
tar function [options] object1 object2 …
```

The function parameter defines what the `tar` command should do, as shown in Table 4.9.

Table 4.9 The tar Command Functions

Function	Long Name	Description
`-A`	`--concatenate`	Append an existing tar archive file to another existing tar archive file.
`-c`	`--create`	Create a new tar archive file.
`-d`	`--diff`	Check the differences between a tar archive file and the filesystem.
	`--delete`	Delete from an existing tar archive file.
`-r`	`--append`	Append files to the end of an existing tar archive file.
`-t`	`--list`	List the contents of an existing tar archive file.
`-u`	`--update`	Append files to an existing tar archive file that are newer than a file with the same name in the existing archive.
`-x`	`--extract`	Extract files from an existing archive file.

Each function uses *options* to define a specific behavior for the tar archive file. Table 4.10 lists the common options that you can use with the `tar` command.

Table 4.10 The tar Command Options

Option	Description
`-C` *`dir`*	Change to the specified directory.
`-f` *`file`*	Output results to file (or device) *`file`*.
`-j`	Redirect output to the `bzip2` command for compression.
`-p`	Preserve all file permissions.
`-v`	List files as they are processed.
`-z`	Redirect the output to the gzip command for compression.

These options are usually combined to create the following scenarios. First, you'll want to create an archive file using this command:

```
tar -cvf test.tar test/ test2/
```

The above command creates an archive file called `test.tar` containing the contents of both the `test` directory and the `test2` directory. Next, this command:

```
tar -tf test.tar
```

lists (but doesn't extract) the contents of the tar file `test.tar`. Finally, this command:

```
tar -xvf test.tar
```

extracts the contents of the tar file `test.tar`. If the tar file was created from a directory structure, the entire directory structure is re-created starting at the current directory.

As you can see, using the `tar` command is a simple way to create archive files of entire directory structures. This is a common method for distributing source code files for open source applications in the Linux world.

Tip

If you download open source software, often you'll see filenames that end in `.tgz`. These are gzipped tar files, and can be extracted using the command `tar -zxvf` *`filename`*`.tgz`.

Summary

This chapter discussed some of the more advanced bash commands used by Linux system administrators and programmers. The `ps` and `top` commands are vital in determining the status of the system, allowing you to see what applications are running and how many resources they are consuming.

In this day of removable media, another popular topic for system administrators is mounting storage devices. The`mount` command allows you to mount a physical storage device into the Linux virtual directory structure. To remove the device, use the `umount` command.

Finally, the chapter discussed various utilities used for handling data. The `sort` utility easily sorts large data files to help you organize data, and the `grep` utility allows you to quickly scan through large data files looking for specific information. There are a few different file compression utilities available in Linux, including `bzip2`,`gzip`, and `zip`. Each one allows you to compress large files to help save space on your filesystem. The Linux `tar`utility is a popular way to archive directory structures into a single file that can easily be ported to another system.

The next chapter discusses Linux environment variables. Environment variables allow you to access information about the system from your scripts, as well as provide a convenient way to store data within your scripts.