5 Controlling Processes

A process is the abstraction used by UNIX and Linux to represent a running program. It's the object through which a program's use of memory, processor time, and I/O resources can be managed and monitored.

It is part of the UNIX philosophy that as much work as possible be done within the context of processes, rather than handled specially by the kernel. System and user processes all follow the same rules, so you can use a single set of tools to control them both.

5.1 Components of a process

A process consists of an address space and a set of data structures within the kernel. The address space is a set of memory pages[1] that the kernel has marked for the process's use. It contains the code and libraries that the process is executing, the process's variables, its stacks, and various extra information needed by the kernel while the process is running. Because UNIX and Linux are virtual memory systems, there is no correlation between a page's location within a process's address space and its location inside the machine's physical memory or swap space.

1. Pages are the units in which memory is managed, usually between 1KiB and 8KiB in size.

The kernel's internal data structures record various pieces of information about each process. Here are some of the more important of these:

- The process's address space map
- The current status of the process (sleeping, stopped, runnable, etc.)
- The execution priority of the process
- Information about the resources the process has used
- Information about the files and network ports the process has opened
- The process's signal mask (a record of which signals are blocked)
- The owner of the process

An execution thread, usually known simply as a thread, is the result of a fork in execution within a process. A thread inherits many of the attributes of the process that contains it (such as the process's address space), and multiple threads can execute concurrently within a single process under a model called multithreading.

Concurrent execution is simulated by the kernel on old-style uniprocessor systems, but on multicore and multi-CPU architectures the threads can run simultaneously on different cores. Multithreaded applications such as BIND and Apache benefit the most from multicore systems since the applications can work on more than one request simultaneously. All our example operating systems support multithreading.

Many of the parameters associated with a process directly affect its execution: the amount of processor time it gets, the files it can access, and so on. In the following sections, we discuss the meaning and significance of the parameters that are most interesting from a system administrator's point of view. These attributes are common to all versions of UNIX and Linux.

PID: process ID number

The kernel assigns a unique ID number to every process.[2] Most commands and system calls that manipulate processes require you to specify a PID to identify the target of the operation. PIDs are assigned in order as processes are created.

PPID: parent PID

Neither UNIX nor Linux has a system call that initiates a new process running a particular program. Instead, an existing process must clone itself to create a new process. The clone can then exchange the program it's running for a different one.

When a process is cloned, the original process is referred to as the parent, and the copy is called the child. The PPID attribute of a process is the PID of the parent from which it was cloned.[3]

2. As pointed out by our reviewer Jon Corbet, Linux kernel 2.6.24 introduced process ID namespaces, which allow multiple processes with the same PID to exist concurrently. This feature was implemented to support container-based virtualization.
3. At least initially. If the original parent dies, **init** (process 1) becomes the new parent. See page 124.

The parent PID is a useful piece of information when you're confronted with an unrecognized (and possibly misbehaving) process. Tracing the process back to its origin (whether a shell or another program) may give you a better idea of its purpose and significance.

UID and EUID: real and effective user ID

See page 180 for more information about UIDs.

A process's UID is the user identification number of the person who created it, or more accurately, it is a copy of the UID value of the parent process. Usually, only the creator (aka the "owner") and the superuser can manipulate a process.

The EUID is the "effective" user ID, an extra UID used to determine what resources and files a process has permission to access at any given moment. For most processes, the UID and EUID are the same, the usual exception being programs that are setuid.

Why have both a UID and an EUID? Simply because it's useful to maintain a distinction between identity and permission, and because a setuid program may not wish to operate with expanded permissions all the time. On most systems, the effective UID can be set and reset to enable or restrict the additional permissions it grants.

Most systems also keep track of a "saved UID," which is a copy of the process's EUID at the point at which the process first begins to execute. Unless the process takes steps to obliterate this saved UID, it remains available for use as the real or effective UID. A conservatively written setuid program can therefore renounce its special privileges for the majority of its execution, accessing them only at the specific points at which extra privileges are needed.

Linux also defines a nonstandard FSUID process parameter that controls the determination of filesystem permissions. It is infrequently used outside the kernel and is not portable to other UNIX systems.

GID and EGID: real and effective group ID

See page 181 for more information about groups.

The GID is the group identification number of a process. The EGID is related to the GID in the same way that the EUID is related to the UID in that it can be "upgraded" by the execution of a setgid program. A saved GID is maintained. It is similar in intent to the saved UID.

The GID attribute of a process is largely vestigial. For purposes of access determination, a process can be a member of many groups at once. The complete group list is stored separately from the distinguished GID and EGID. Determinations of access permissions normally take into account the EGID and the supplemental group list, but not the GID.

The only time at which the GID really gets to come out and play is when a process creates new files. Depending on how the filesystem permissions have been set, new files may adopt the GID of the creating process. See page 154 for details.

Niceness

A process's scheduling priority determines how much CPU time it receives. The kernel uses a dynamic algorithm to compute priorities, allowing for the amount of CPU time that a process has recently consumed and the length of time it has been waiting to run. The kernel also pays attention to an administratively set value that's usually called the "nice value" or "niceness," so called because it tells how nice you are planning to be to other users of the system. We discuss niceness in detail on page 129.

In an effort to provide better support for low-latency applications, Linux has added "scheduling classes" to the traditional UNIX scheduling model. There are currently three classes, and each process is assigned to one class. Unfortunately, the real-time classes are neither widely used nor well supported from the command line. System processes use the traditional (niceness) scheduler, which is the only one we discuss in this book. See realtimelinuxfoundation.org for more discussion of issues related to real-time scheduling.

Control terminal

Most nondaemon processes have an associated control terminal. The control terminal determines default linkages for the standard input, standard output, and standard error channels. When you start a command from the shell, your terminal window normally becomes the process's control terminal. The concept of a control terminal also affects the distribution of signals, which are discussed starting on page 124.

5.2 The life cycle of a process

To create a new process, a process copies itself with the **fork** system call. **fork** creates a copy of the original process; that copy is largely identical to the parent. The new process has a distinct PID and has its own accounting information.

fork has the unique property of returning two different values. From the child's point of view, it returns zero. The parent receives the PID of the newly created child. Since the two processes are otherwise identical, they must both examine the return value to figure out which role they are supposed to play.

After a **fork**, the child process will often use one of the **exec** family of system calls to begin the execution of a new program.[4] These calls change the program that the process is executing and reset the memory segments to a predefined initial state. The various forms of **exec** differ only in the ways in which they specify the command-line arguments and environment to be given to the new program.

When the system boots, the kernel autonomously creates and installs several processes. The most notable of these is **init**, which is always process number 1. **init** is responsible for executing the system's startup scripts, although the exact manner

4. Actually, all but one are library routines rather than system calls.

in which this is done differs slightly between UNIX and Linux. All processes other than the ones the kernel creates are descendants of **init**. See Chapter 3 for more information about booting and the **init** daemon.

init also plays another important role in process management. When a process completes, it calls a routine named **_exit** to notify the kernel that it is ready to die. It supplies an exit code (an integer) that tells why it's exiting. By convention, 0 is used to indicate a normal or "successful" termination.

Before a process can be allowed to disappear completely, the kernel requires that its death be acknowledged by the process's parent, which the parent does with a call to **wait**. The parent receives a copy of the child's exit code (or an indication of why the child was killed if the child did not exit voluntarily) and can also obtain a summary of the child's use of resources if it wishes.

This scheme works fine if parents outlive their children and are conscientious about calling **wait** so that dead processes can be disposed of. If the parent dies first, however, the kernel recognizes that no **wait** will be forthcoming and adjusts the process to make the orphan a child of **init**. **init** politely accepts these orphaned processes and performs the **wait** needed to get rid of them when they die.

5.3 Signals

Signals are process-level interrupt requests. About thirty different kinds are defined, and they're used in a variety of ways:

- They can be sent among processes as a means of communication.
- They can be sent by the terminal driver to kill, interrupt, or suspend processes when keys such as <Control-C> and <Control-Z> are typed.[5]
- They can be sent by an administrator (with **kill**) to achieve various ends.
- They can be sent by the kernel when a process commits an infraction such as division by zero.
- They can be sent by the kernel to notify a process of an "interesting" condition such as the death of a child process or the availability of data on an I/O channel.

A core dump is a process's memory image. It can be used for debugging.

When a signal is received, one of two things can happen. If the receiving process has designated a handler routine for that particular signal, the handler is called with information about the context in which the signal was delivered. Otherwise, the kernel takes some default action on behalf of the process. The default action varies from signal to signal. Many signals terminate the process; some also generate a core dump.

5. The functions of <Control-Z> and <Control-C> can be reassigned to other keys with the **stty** command, but this is rare in practice. In this chapter we refer to them by their conventional bindings.

Specifying a handler routine for a signal within a program is referred to as catching the signal. When the handler completes, execution restarts from the point at which the signal was received.

To prevent signals from arriving, programs can request that they be either ignored or blocked. A signal that is ignored is simply discarded and has no effect on the process. A blocked signal is queued for delivery, but the kernel doesn't require the process to act on it until the signal has been explicitly unblocked. The handler for a newly unblocked signal is called only once, even if the signal was received several times while reception was blocked.

Table 5.1 lists some signals with which all administrators should be familiar. The uppercase convention for the names derives from C language tradition. You might also see signal names written with a SIG prefix (e.g., SIGHUP) for similar reasons.

Table 5.1 Signals every administrator should know[a]

#	Name	Description	Default	Can catch?	Can block?	Dump core?
1	HUP	Hangup	Terminate	Yes	Yes	No
2	INT	Interrupt	Terminate	Yes	Yes	No
3	QUIT	Quit	Terminate	Yes	Yes	Yes
9	KILL	Kill	Terminate	No	No	No
–[b]	BUS	Bus error	Terminate	Yes	Yes	Yes
11	SEGV	Segmentation fault	Terminate	Yes	Yes	Yes
15	TERM	Software termination	Terminate	Yes	Yes	No
–[b]	STOP	Stop	Stop	No	No	No
–[b]	TSTP	Keyboard stop	Stop	Yes	Yes	No
–[b]	CONT	Continue after stop	Ignore	Yes	No	No
–[b]	WINCH	Window changed	Ignore	Yes	Yes	No
–[b]	USR1	User-defined #1	Terminate	Yes	Yes	No
–[b]	USR2	User-defined #2	Terminate	Yes	Yes	No

a. A list of signal names and numbers is also available from the **bash** built-in command **kill -l**.
b. Varies among systems. See **/usr/include/signal.h** or **man signal** for more specific information.

Other signals, not shown in Table 5.1, mostly report obscure errors such as "illegal instruction." The default handling for signals like that is to terminate with a core dump. Catching and blocking are generally allowed because some programs may be smart enough to try to clean up whatever problem caused the error before continuing.

The BUS and SEGV signals are also error signals. We've included them in the table because they're so common: when a program crashes, it's usually one of these two signals that finally brings it down. By themselves, the signals are of no specific diagnostic value. Both of them indicate an attempt to use or access memory improperly.[6]

The signals named KILL and STOP cannot be caught, blocked, or ignored. The KILL signal destroys the receiving process, and STOP suspends its execution until a CONT signal is received. CONT may be caught or ignored, but not blocked.

TSTP is a "soft" version of STOP that might be best described as a request to stop. It's the signal generated by the terminal driver when <Control-Z> is typed on the keyboard. Programs that catch this signal usually clean up their state, then send themselves a STOP signal to complete the stop operation. Alternatively, programs can ignore TSTP to prevent themselves from being stopped from the keyboard.

Terminal emulators send a WINCH signal when their configuration parameters (such as the number of lines in the virtual terminal) change. This convention allows emulator-savvy programs such as text editors to reconfigure themselves automatically in response to changes. If you can't get windows to resize properly, make sure that WINCH is being generated and propagated correctly.[7]

The signals KILL, INT, TERM, HUP, and QUIT all sound as if they mean approximately the same thing, but their uses are actually quite different. It's unfortunate that such vague terminology was selected for them. Here's a decoding guide:

- KILL is unblockable and terminates a process at the kernel level. A process can never actually receive this signal.
- INT is sent by the terminal driver when you type <Control-C>. It's a request to terminate the current operation. Simple programs should quit (if they catch the signal) or simply allow themselves to be killed, which is the default if the signal is not caught. Programs that have an interactive command line (such as a shell) should stop what they're doing, clean up, and wait for user input again.
- TERM is a request to terminate execution completely. It's expected that the receiving process will clean up its state and exit.
- HUP has two common interpretations. First, it's understood as a reset request by many daemons. If a daemon is capable of rereading its configuration file and adjusting to changes without restarting, a HUP can generally be used to trigger this behavior.

6. More specifically, bus errors result from violations of alignment requirements or the use of nonsensical addresses. Segmentation violations represent protection violations such as attempts to write to read-only portions of the address space.
7. Which may be easier said than done. The terminal emulator (e.g., **xterm**), terminal driver, and user-level commands may all have a role in propagating SIGWINCH. Common problems include sending the signal to a terminal's foreground process only (rather than to all processes associated with the terminal) and failing to propagate notification of a size change across the network to a remote computer. Protocols such as Telnet and SSH explicitly recognize local terminal size changes and communicate this information to the remote host. Simpler protocols (e.g., direct serial lines) cannot do this.

Second, HUP signals are sometimes generated by the terminal driver in an attempt to "clean up" (i.e., kill) the processes attached to a particular terminal. This behavior is largely a holdover from the days of wired terminals and modem connections, hence the name "hangup."

Shells in the C shell family (**tcsh** et al.) usually make background processes immune to HUP signals so that they can continue to run after the user logs out. Users of Bourne-ish shells (**ksh**, **bash**, etc.) can emulate this behavior with the **nohup** command.

- QUIT is similar to TERM, except that it defaults to producing a core dump if not caught. A few programs cannibalize this signal and interpret it to mean something else.

The signals USR1 and USR2 have no set meaning. They're available for programs to use in whatever way they'd like. For example, the Apache web server interprets the USR1 signal as a request to gracefully restart.

5.4 KILL: SEND SIGNALS

As its name implies, the **kill** command is most often used to terminate a process. **kill** can send any signal, but by default it sends a TERM. **kill** can be used by normal users on their own processes or by root on any process. The syntax is

```
kill [-signal] pid
```

where *signal* is the number or symbolic name of the signal to be sent (as shown in Table 5.1) and *pid* is the process identification number of the target process.

A **kill** without a signal number does not guarantee that the process will die, because the TERM signal can be caught, blocked, or ignored. The command

```
kill -9 pid
```

"guarantees" that the process will die because signal 9, KILL, cannot be caught. Use **kill -9** only if a polite request fails. We put quotes around "guarantees" because processes can occasionally become so wedged that even KILL does not affect them (usually because of some degenerate I/O vapor lock such as waiting for a disk that has stopped spinning). Rebooting is usually the only way to get rid of these processes.

The **killall** command performs wildly different functions on UNIX and Linux. Under Linux, **killall** kills processes by name. For example, the following command kills all Apache web server processes:

```
ubuntu$ sudo killall httpd
```

The standard UNIX **killall** command that ships with Solaris, HP-UX, and AIX takes no arguments and simply kills all the current user's processes. Running it as root kills **init** and shuts down the machine. Oops.

The **pgrep** and **pkill** commands for Solaris, HP-UX, and Linux (but not AIX) search for processes by name (or other attributes, such as EUID) and display or signal them, respectively. For example, the following command sends a TERM signal to all processes running as the user ben:

```
$ sudo pkill -u ben
```

5.5 Process states

A process is not automatically eligible to receive CPU time just because it exists. You need to be aware of the four execution states listed in Table 5.2.

Table 5.2 Process states

State	Meaning
Runnable	The process can be executed.
Sleeping	The process is waiting for some resource.
Zombie	The process is trying to die.
Stopped	The process is suspended (not allowed to execute).

A runnable process is ready to execute whenever CPU time is available. It has acquired all the resources it needs and is just waiting for CPU time to process its data. As soon as the process makes a system call that cannot be immediately completed (such as a request to read part of a file), the kernel puts it to sleep.

Sleeping processes are waiting for a specific event to occur. Interactive shells and system daemons spend most of their time sleeping, waiting for terminal input or network connections. Since a sleeping process is effectively blocked until its request has been satisfied, it will get no CPU time unless it receives a signal or a response to one of its I/O requests.

Some operations cause processes to enter an uninterruptible sleep state. This state is usually transient and not observed in **ps** output (indicated by a D in the STAT column; see Table 5.4 on page 132). However, a few degenerate situations can cause it to persist. The most common cause involves server problems on an NFS filesystem mounted with the "hard" option. Since processes in the uninterruptible sleep state cannot be roused even to service a signal, they cannot be killed. To get rid of them, you must fix the underlying problem or reboot.

Zombies are processes that have finished execution but have not yet had their status collected. If you see zombies hanging around, check their PPIDs with **ps** to find out where they're coming from.

Stopped processes are administratively forbidden to run. Processes are stopped on receipt of a STOP or TSTP signal and are restarted with CONT. Being stopped is similar to sleeping, but there's no way for a process to get out of the stopped state other than having some other process wake it up (or kill it).

5.6 NICE AND RENICE: INFLUENCE SCHEDULING PRIORITY

The “niceness” of a process is a numeric hint to the kernel about how the process should be treated in relation to other processes contending for the CPU. The strange name is derived from the fact that it determines how nice you are going to be to other users of the system. A high nice value means a low priority for your process: you are going to be nice. A low or negative value means high priority: you are not very nice.

The range of allowable niceness values varies among systems. The most common range is -20 to +19. Some systems use a range of a similar size beginning at 0 instead of a negative number (typically 0 to 39). The ranges used on our example systems are shown in Table 5.3 on the next page.

Despite their numeric differences, all systems handle nice values in much the same way. Unless the user takes special action, a newly created process inherits the nice value of its parent process. The owner of the process can increase its nice value but cannot lower it, even to return the process to the default niceness. This restriction prevents processes with low priority from bearing high-priority children. The superuser may set nice values arbitrarily.

It’s rare to have occasion to set priorities by hand these days. On the puny systems of the 1970s and 80s, performance was significantly affected by which process was on the CPU. Today, with more than adequate CPU power on every desktop, the scheduler does a good job of servicing all processes. The addition of scheduling classes gives developers additional control when fast response is essential.

I/O performance has not kept up with increasingly fast CPUs, and the major bottleneck on most systems has become the disk drives. Unfortunately, a process’s nice value has no effect on the kernel’s management of its memory or I/O; high-nice processes can still monopolize a disproportionate share of these resources.

A process’s nice value can be set at the time of creation with the **nice** command and adjusted later with the **renice** command. **nice** takes a command line as an argument, and **renice** takes a PID or (sometimes) a username.

Some examples:

```
$ nice -n 5 ~/bin/longtask          // Lowers priority (raise nice) by 5
$ sudo renice -5 8829               // Sets nice value to -5
$ sudo renice 5 -u boggs            // Sets nice value of boggs’s procs to 5
```

Unfortunately, there is little agreement among systems about how the desired priorities should be specified; in fact, even **nice** and **renice** from the same system usually don’t agree. Some commands want a nice value increment, whereas others want an absolute nice value. Some want their nice values preceded by a dash. Others want a flag (-**n**), and some just want a value.

To complicate things, a version of **nice** is built into the C shell and some other common shells (but not **bash**). If you don’t type the full path to **nice**, you’ll get the shell's version rather than the operating system's. This duplication can be confusing because shell-**nice** and command-**nice** use different syntax: the shell wants its priority increment expressed as *+incr* or *-incr,* but the stand-alone command wants an **-n** flag followed by the priority increment.[8]

Table 5.3 summarizes all these variations. A *prio* is an absolute nice value, while an *incr* is relative to the niceness of the shell from which **nice** or **renice** is run. Wherever an *-incr* or a *-prio* is called for, you can use a double dash to enter negative values (e.g., --**10**). Only the shell **nice** understands plus signs (in fact, it requires them); leave them out in all other circumstances.

Table 5.3 How to express priorities for various versions of nice and renice

System	Range	OS nice	csh nice	renice
Linux	-20 to 19	*-incr* or **-n** *incr*	*+incr* or *-incr*	*prio*
Solaris	0 to 39	*-incr* or **-n** *incr*	*+incr* or *-incr*	*incr* or **-n** *incr*
HP-UX	0 to 39	*-prio* or **-n** *prio*	*+incr* or *-incr*	**-n** *prio*[a]
AIX	-20 to 19	*-incr* or **-n** *incr*	*+incr* or *-incr*	**-n** *incr*

a. Uses absolute priority, but adds 20 to the value you specify.

The most commonly **nice**d process in the modern world is **ntpd**, the clock synchronization daemon. Since promptness is critical to its mission, it usually runs at a nice value about 12 below the default (that is, at a higher priority than normal).

If a problem drives the system's load average to 65, you may need to use **nice** to start a high-priority shell before you can run commands to investigate the problem. Otherwise, you may have difficulty running even simple commands.

5.7 PS: MONITOR PROCESSES

ps is the system administrator's main tool for monitoring processes. While versions of **ps** differ in their arguments and display, they all deliver essentially the same information. Part of the enormous variation among versions of **ps** can be traced back to differences in the development history of UNIX. However, **ps** is also a command that vendors tend to customize for other reasons. It's closely tied to the kernel's handling of processes, so it tends to reflect all of a vendors' underlying kernel changes.

ps can show the PID, UID, priority, and control terminal of processes. It also gives information about how much memory a process is using, how much CPU time it has consumed, and its current status (running, stopped, sleeping, etc.). Zombies show up in a **ps** listing as <exiting> or <defunct>.

8. Actually, it's worse than this: the stand-alone **nice** interprets **nice -5** to mean a *positive* increment of 5, whereas the shell built-in **nice** interprets this same form to mean a *negative* increment of 5.

Implementations of **ps** have become hopelessly complex over the last decade. Several vendors have abandoned the attempt to define meaningful displays and made their **ps**es completely configurable. With a little customization work, almost any desired output can be produced. As a case in point, the **ps** used by Linux is a trisexual and hermaphroditic version that understands multiple option sets and uses an environment variable to tell it what universe it's living in.

Do not be alarmed by all this complexity: it's there mainly for developers, not for system administrators. Although you will use **ps** frequently, you only need to know a few specific incantations.

On Linux and AIX, you can obtain a useful overview of all the processes running on the system with **ps aux**. The **a** option means to show all processes, **x** means to show even processes that don't have a control terminal, and **u** selects the "user oriented" output format. Here's an example of **ps aux** output on a machine running Red Hat (AIX output for the same command differs slightly):

```
redhat$ ps aux
  USER     PID  %CPU%MEM  VSZ   RSS   TTY  STAT TIME  COMMAND
   root      1   0.1  0.2  3356  560   ?    S    0:00  init [5]
   root      2   0    0       0  0     ?    SN   0:00  [ksoftirqd/0]
   root      3   0    0       0  0     ?    S<   0:00  [events/0]
   root      4   0    0       0  0     ?    S<   0:00  [khelper]
   root      5   0    0       0  0     ?    S<   0:00  [kacpid]
   root     18   0    0       0  0     ?    S<   0:00  [kblockd/0]
   root     28   0    0       0  0     ?    S    0:00  [pdflush]
…
   root    196   0    0       0  0     ?    S    0:00  [kjournald]
   root   1050   0    0.1  2652  448   ?    S<s  0:00  udevd
   root   1472   0    0.3  3048  1008  ?    S<s  0:00  /sbin/dhclient -1
   root   1646   0    0.3  3012  1012  ?    S<s  0:00  /sbin/dhclient -1
   root   1733   0    0       0  0     ?    S    0:00  [kjournald]
   root   2124   0    0.3  3004  1008  ?    Ss   0:00  /sbin/dhclient -1
   root   2182   0    0.2  2264  596   ?    Ss   0:00  syslogd -m 0
   root   2186   0    0.1  2952  484   ?    Ss   0:00  klogd -x
    rpc   2207   0    0.2  2824  580   ?    Ss   0:00  portmap
rpcuser   2227   0    0.2  2100  760   ?    Ss   0:00  rpc.statd
   root   2260   0    0.4  5668  1084  ?    Ss   0:00  rpc.idmapd
   root   2336   0    0.2  3268  556   ?    Ss   0:00  /usr/sbin/acpid
   root   2348   0    0.8  9100  2108  ?    Ss   0:00  cupsd
   root   2384   0    0.6  4080  1660  ?    Ss   0:00  /usr/sbin/sshd
   root   2399   0    0.3  2780  828   ?    Ss   0:00  xinetd -stayalive
   root   2419   0    1.1  7776  3004  ?    Ss   0:00  sendmail: accept
…
```

Command names in brackets are not really commands at all but rather kernel threads scheduled as processes. The meaning of each field is shown in Table 5.4 on the next page.

Another useful set of arguments for Linux and AIX is **lax**, which provides more technical information. The **a** and **x** options are as above (show every process), and

Table 5.4 Explanation of ps aux output

Field	Contents
USER	Username of the process's owner
PID	Process ID
%CPU	Percentage of the CPU this process is using
%MEM	Percentage of real memory this process is using
VSZ	Virtual size of the process
RSS	Resident set size (number of pages in memory)
TTY	Control terminal ID
STAT	Current process status: R = Runnable D = In uninterruptible sleep S = Sleeping (< 20 sec) T = Traced or stopped Z = Zombie Additional flags: W = Process is swapped out < = Process has higher than normal priority N = Process has lower than normal priority L = Some pages are locked in core s = Process is a session leader
TIME	CPU time the process has consumed
COMMAND	Command name and arguments[a]

a. Programs can modify this info, so it's not necessarily an accurate representation of the actual command line.

l selects the "long" output format. **ps lax** is also slightly faster to run than **ps aux** because it doesn't have to translate every UID to a username—efficiency can be important if the system is already bogged down.

Shown here in an abbreviated example, **ps lax** includes fields such as the parent process ID (PPID), nice value (NI), and the type of resource on which the process is waiting (WCHAN).

```
redhat$ ps lax
   F  UID    PID   PPID PRI NI  VSZ   RSS  WCHAN  STAT TIME  COMMAND
   4    0      1      0  16  0  3356   560 select S    0:00  init [5]
   1    0      2      1  34 19     0     0 ksofti SN   0:00  [ksoftirqd/0
   1    0      3      1   5-10     0     0 worker S<   0:00  [events/0]
   1    0      4      3   5-10     0     0 worker S<   0:00  [khelper]
   5    0   2186      1  16  0  2952   484 syslog Ss   0:00  klogd -x
   5   32   2207      1  15  0  2824   580 -      Ss   0:00  portmap
   5   29   2227      1  18  0  2100   760 select Ss   0:00  rpc.statd
   1    0   2260      1  16  0  5668  1084 -      Ss   0:00  rpc.idmapd
   1    0   2336      1  21  0  3268   556 select Ss   0:00  acpid
   5    0   2384      1  17  0  4080  1660 select Ss   0:00  sshd
   1    0   2399      1  15  0  2780   828 select Ss   0:00  xinetd -sta
   5    0   2419      1  16  0  7776  3004 select Ss   0:00  sendmail: a
...
```

Under Solaris and HP-UX, **ps -ef** is a good place to start. The **e** option selects all processes, and the **f** option sets the output format. (**ps -ef** also works on AIX and Linux systems; note the dash.)

```
solaris$ ps -ef
 UID     PID  PPID   C    STIME    TTY    TIME   COMD
 root      0     0  80   Dec 21    ?      0:02   sched
 root      1     0   2   Dec 21    ?      4:32   /etc/init-
 root      2     0   8   Dec 21    ?      0:00   pageout
 root    171     1  80   Dec 21    ?      0:02   /usr/lib/sendmail-bd
trent   8482  8444  35  14:34:10  pts/7   0:00   ps-ef
trent   8444  8442 203  14:32:50  pts/7   0:01   -csh
…
```

The columns in the **ps -ef** output are explained in Table 5.5.

Table 5.5 Explanation of ps -ef output

Field	Content	Field	Content
UID	Username of the owner	STIME	Time the process was started
PID	Process ID	TTY	Control terminal
PPID	PID of the parent process	TIME	CPU time consumed
C	CPU use/scheduling info	COMD	Command and arguments

Like **ps lax** in the Linux and AIX worlds, **ps -elf** shows additional gory details on Solaris and HP-UX systems:

```
% ps -elf
  F  S  UID   PID  PPID   C   P   NI   ADDR       SZ    WCHAN     TIME   COMD
 19  T  root    0     0  80   0   SY   f00c2fd8   0               0:02   sched
  8  S  root    1     0  65   1   20   ff26a800   88    ff2632c8  4:32   init-
  8  S  root  142     1  41   1   20   ff2e8000   176   f00cb69   0:00   syslogd
…
```

The STIME and TTY columns have been omitted to fit this page; they are identical to those produced with **ps -ef**. Nonobvious fields are described in Table 5.6 on the next page.

5.8 Dynamic monitoring with top, prstat, and topas

Since commands like **ps** offer only a one-time snapshot of your system, it is often difficult to grasp the big picture of what's really happening. **top** is a free utility that runs on many systems and provides a regularly updated summary of active processes and their use of resources. On AIX, an equivalent utility is **topas**, and on Solaris the analogous tool is **prstat**.

Table 5.6 Explanation of ps -elf output

Field	Contents
F	Process flags; possible values vary by system (rarely useful for sysadmins)

S	Process status:
	O = Currently running S = Sleeping (waiting for event)
	R = Eligible to run T = Stopped or being traced
	Z = Zombie D = Uninterruptible sleep (disk, usually)

C	Process CPU utilization/scheduling info
P	Scheduling priority (internal to the kernel, different from nice value)
NI	Nice value or SY for system processes
ADDR	Memory address of the process
SZ	Size (in pages) of the process in main memory
WCHAN	Address of the object the process is waiting for

For example:

```
ubuntu$ top
top - 16:37:08 up  1:42,  2 users,  load average: 0.01, 0.02, 0.06
Tasks:  76 total,   1 running,  74 sleeping,   1 stopped,   0 zombie
Cpu(s):  1.1% us,  6.3% sy,  0.6% ni, 88.6% id,  2.1% wa,  0.1% hi,  1.3% si
Mem:   256044k  total,  254980k  used,     1064k  free,    15944k  buffers
Swap:  524280k  total,        0k  used,   524280k  free,   153192k  cached

  PID  USER  PR   NI   VIRT   RES   SHR  S  %CPU  %MEM  TIME+    COMMAND
 3175  root  15    0  35436   12m  4896  S   4.0   5.2  01:41.9  X
 3421  root  25   10  29916   15m  9808  S   2.0   6.2  01:10.5  rhn-applet-gui
    1  root  16    0   3356   560   480  S   0.0   0.2  00:00.9  init
    2  root  34   19      0     0   0    S   0.0     0  00:00.0  ksoftirqd/0
    3  root   5  -10      0     0   0    S   0.0     0  00:00.7  events/0
    4  root   5  -10      0     0   0    S   0.0     0  00:00.0  khelper
    5  root  15  -10      0     0   0    S   0.0     0  00:00.0  kacpid
   18  root   5  -10      0     0   0    S   0.0     0  00:00.0  kblockd/0
   28  root  15    0      0     0   0    S   0.0     0  00:00.0  pdflush
   29  root  15    0      0     0   0    S   0.0     0  00:00.3  pdflush
   31  root  13  -10      0     0   0    S   0.0     0  00:00.0  aio/0
   19  root  15    0      0     0   0    S   0.0     0  00:00.0  khubd
   30  root  15    0      0     0   0    S   0.0     0  00:00.2  kswapd0
  187  root   6  -10      0     0   0    S     0     0  00:00.0  kmirrord/0
  196  root  15    0      0     0   0    S     0     0  00:01.3  kjournald
...
```

By default, the display updates every 10 seconds. The most CPU-consumptive processes appear at the top. **top** also accepts input from the keyboard and allows you to send signals and to **renice** processes, so you can observe how your actions affect the overall condition of the machine.

Root can run **top** with the **-q** option to goose it up to the highest possible priority. This option can be very useful when you are trying to track down a process that has already brought the system to its knees.

5.9 The /proc filesystem

The Linux versions of **ps** and **top** read their process status information from the **/proc** directory, a pseudo-filesystem in which the kernel exposes a variety of interesting information about the system's state. Despite the name **/proc** (and the name of the underlying filesystem type, "proc"), the information is not limited to process information—a variety of status information and statistics generated by the kernel are represented here. You can even modify some parameters by writing to the appropriate **/proc** file. See page 421 for some examples.

Although some of the information is easier to access through front-end commands such as **vmstat** and **ps**, some of the less popular information must be read directly from **/proc**. It's worth poking around in this directory to familiarize yourself with everything that's there. **man proc** also lists some useful tips and tricks.

Because the kernel creates the contents of **/proc** files on the fly (as they are read), most appear to be empty when listed with **ls -l**. You'll have to **cat** or **more** the contents to see what they actually contain. But be cautious—a few files contain or link to binary data that can confuse your terminal emulator if viewed directly.

Process-specific information is divided into subdirectories named by PID. For example, **/proc/1** is always the directory that contains information about **init**. Table 5.7 lists the most useful per-process files.

Table 5.7 Process information files in Linux /proc (numbered subdirectories)

File	Contents
cmd	Command or program the process is executing
cmdline[a]	Complete command line of the process (null-separated)
cwd	Symbolic link to the process's current directory
environ	The process's environment variables (null-separated)
exe	Symbolic link to the file being executed
fd	Subdirectory containing links for each open file descriptor
maps	Memory mapping information (shared segments, libraries, etc.)
root	Symbolic link to the process's root directory (set with **chroot**)
stat	General process status information (best decoded with **ps**)
statm	Memory usage information

a. May be unavailable if the process is swapped out of memory.

The individual components contained within the **cmdline** and **environ** files are separated by null characters rather than newlines. You can filter their contents through **tr "\000" "\n"** to make them more readable.

The **fd** subdirectory represents open files in the form of symbolic links. File descriptors that are connected to pipes or network sockets don't have an associated filename. The kernel supplies a generic description as the link target instead.

The **maps** file can be useful for determining what libraries a program is linked to or depends on.

Solaris and AIX also have a **/proc** filesystem, but it does not include the extra status and statistical information found on Linux. A group of tools known collectively as the proc utilities display some useful information about running processes. For instance, the **procsig** command in AIX and its Solaris equivalent **psig** print the signal actions and handlers for a given process. Table 5.8 shows the most useful proc utilities and their functions.

Table 5.8 Commands for reading /proc information in AIX and Solaris

Solaris[a]	AIX	Description
pcred [*pid* | *core*]	**proccred** [*pid*]	Prints/sets real, effective, and saved UID/GID
pldd [**-F**] [*pid* | *core*]	**procldd** [*pid*]	Shows library dependencies (like **ldd**)
psig [*pid*]	**procsig** [*pid*]	Lists signal actions and handlers
pfiles [*pid*]	**procfiles** [*pid*]	Prints open files
pwdx [*pid*]	**procwdx** [*pid*]	Prints the current working directory
pwait [*pid*]	**procwait** [*pid*]	Waits for a process to exit

a. Some of the Solaris proc tools accept a core file as input. This is primarily a debugging tool.

HP-UX does not have a **/proc** filesystem or equivalent.

5.10 STRACE, TRUSS, AND TUSC: TRACE SIGNALS AND SYSTEM CALLS

It can sometimes be hard to figure out what a process is actually doing. You may have to make educated guesses based on indirect data from the filesystem and from tools such as **ps**.

Linux lets you directly observe a process with the **strace** command, which shows every system call the process makes and every signal it receives. A similar command for Solaris and AIX is **truss**. The HP-UX equivalent is **tusc**; however, **tusc** must be separately installed.

You can even attach **strace** or **truss** to a running process, snoop for a while, and then detach from the process without disturbing it.[9]

9. Well, usually. **strace** can interrupt system calls. The monitored process must then be prepared to restart them. This is a standard rule of UNIX software hygiene, but it's not always observed.

Although system calls occur at a relatively low level of abstraction, you can usually tell quite a bit about a process's activity from the output. For example, the following log was produced by **strace** run against an active copy of **top**:

```
redhat$ sudo strace -p 5810
gettimeofday({1116193814, 213881}, {300, 0})          = 0
open("/proc", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 7
fstat64(7, {st_mode=S_IFDIR|0555, st_size=0, ...})    = 0
fcntl64(7, F_SETFD, FD_CLOEXEC)                       = 0
getdents64(7, /* 36 entries */, 1024)                 = 1016
getdents64(7, /* 39 entries */, 1024)                 = 1016
stat64("/proc/1", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
open("/proc/1/stat", O_RDONLY)                        = 8
read(8, "1 (init) S 0 0 0 0 -1 4194560 73"..., 1023)  = 191
close(8)                                              = 0
...
```

Not only does **strace** show you the name of every system call made by the process, but it also decodes the arguments and shows the result code the kernel returns.

strace is packed with goodies, most of which are documented in the man page. For example, the **-f** flag follows forked processes, which is useful for tracing daemons such as **httpd** that spawn many children. The **-e file** option displays only file operations, a feature that's especially handy for discovering the location of evasive configuration files.

In this example, **top** starts by checking the current time. It then opens and stats the **/proc** directory and reads the directory's contents, thereby obtaining a list of running processes. **top** goes on to stat the directory representing the **init** process and then opens **/proc/1/stat** to read the **init**'s status information.

Here's an even simpler example (the **date** command) using **truss** on Solaris:

```
solaris$ truss date
...
time()                                    = 1242507670
brk(0x00024D30)                           = 0
brk(0x00026D30)                           = 0
open("/usr/share/lib/zoneinfo/US/Mountain", O_RDONLY) = 3
fstat64(3, 0xFFBFFAF0)                    = 0
read(3, " T Z i f\0\0\0\0\0\0\0\0".., 877)  = 877
close(3)                                  = 0
ioctl(1, TCGETA, 0xFFBFFA94)              = 0
fstat64(1, 0xFFBFF9B0)                    = 0
write(1, " S a t   M a y   1 6   1".., 29)  = 29
Sat May 16 14:56:46 MDT 2009
_exit(0)
```

Here, after allocating memory and opening library dependencies (not shown), **date** uses the **time** system call to read the system time, opens the appropriate time zone file to determine the appropriate offset, and prints the date and time stamp by calling the **write** system call.

5.11 Runaway processes

See page 1131 for more information about runaway processes.

Runaway processes come in two flavors: user processes that consume excessive amounts of a system resource, such as CPU time or disk space, and system processes that suddenly go berserk and exhibit wild behavior. The first type of runaway is not necessarily malfunctioning; it might simply be a resource hog. System processes are always supposed to behave reasonably.

You can identify processes that use excessive CPU time by looking at the output of **ps** or **top**. If it's obvious that a user process is consuming more CPU than is reasonable, investigate the process. It can also be useful to look at the number of processes waiting to run. Use the **uptime** command to show the load averages (average numbers of runnable processes) over 1, 5, and 15-minute intervals.

There are two reasons to find out what a process is trying to do before tampering with it. First, the process may be both legitimate and important. It's unreasonable to kill processes at random just because they happen to use a lot of CPU. Second, the process may be malicious or destructive. In this case, you've got to know what the process was doing (e.g., cracking passwords) so that you can fix the damage.

Processes that make excessive use of memory relative to the system's physical RAM can cause serious performance problems. You can check the memory size of processes by using **top**. The VIRT column shows the total amount of virtual memory allocated by each process, and the RES column shows the portion of that memory that is currently mapped to specific memory pages (the "resident set"). On Linux systems, applications that use the video card (such as the X server) get a bad rap because video memory is included in the memory usage computations.

Both of these numbers can include shared resources such as libraries, and that makes them potentially misleading. A more direct measure of process-specific memory consumption is found in the DATA column, which is not shown by default. To add this column to **top**'s display, type the **f** key once **top** is running and select DATA from the list. The DATA value indicates the amount of memory in each process's data and stack segments, so it's relatively specific to individual processes (modulo shared memory segments). Look for growth over time as well as absolute size.

Runaway processes that produce output can fill up an entire filesystem, causing numerous problems. When a filesystem fills up, lots of messages will be logged to the console and attempts to write to the filesystem will produce error messages.

The first thing to do in this situation is to determine which filesystem is full and which file is filling it up. The **df -k** command shows filesystem use. Look for a filesystem that's 100% or more full.[10] Use the **du** command on the identified filesystem to find which directory is using the most space. Rinse and repeat with **du**

10. Most filesystem implementations reserve a portion (about 5%) of the storage space for "breathing room," but processes running as root can encroach on this space, resulting in a reported usage that is greater than 100%.

until the large files are discovered. If you can't determine which process is using the file, try using the **fuser** and **lsof** commands (covered in detail on page 144) for more information.

You may want to suspend all suspicious-looking processes until you find the one that's causing the problem, but remember to restart the innocents when you are done. When you find the offending process, remove the files it was creating. Sometimes it's smart to compress the file with **gzip** and rename it in case it contains useful or important data.

5.12 Recommended reading

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel (3rd Edition)*. Sebastopol, CA: O'Reilly Media, 2006.

McKusick, Marshall Kirk, and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Reading, MA: Addison-Wesley Professional, 2004.

5.13 Exercises

E5.1 Explain the relationship between a file's UID and a running process's real UID and effective UID. Besides file access control, what is the purpose of a process's effective UID?

E5.2 Suppose that a user at your site has started a long-running process that is consuming a significant fraction of a machine's resources.

a) How would you recognize a process that is hogging resources?

b) Assume that the misbehaving process might be legitimate and doesn't deserve to die. Show the commands you would use to suspend the process temporarily while you investigate.

c) Later, you discover that the process belongs to your boss and must continue running. Show the commands you'd use to resume the task.

d) Alternatively, assume that the process needs to be killed. What signal would you send, and why? What if you needed to guarantee that the process died?

E5.3 Find a process with a memory leak (write your own program if you don't have one handy). Use **ps** or **top** to monitor the program's memory use as it runs.

☆ E5.4 Write a simple Perl script that processes the output of **ps** to determine the total VSZ and RSS of the processes running on the system. How do these numbers relate to the system's actual amount of physical memory and swap space?

Chapter 31. background jobs

Table of Contents

31.1. background processes

jobs

Stuff that runs in background of your current shell can be displayed with the **jobs** command. By default you will not have any **jobs** running in background.

```
root@rhel53 ~# jobs
root@rhel53 ~#
```

This **jobs** command will be used several times in this section.

control-Z

Some processes can be **suspended** with the **Ctrl-Z** key combination. This sends a **SIGSTOP** signal to the **Linux kernel**, effectively freezing the operation of the process.

When doing this in **vi(m)**, then **vi(m)** goes to the background. The background **vi(m)** can be seen with the **jobs** command.

```
[paul@RHEL4a ~]$ vi procdemo.txt

[5]+  Stopped                 vim procdemo.txt
[paul@RHEL4a ~]$ jobs
[5]+  Stopped                 vim procdemo.txt
```

& ampersand

Processes that are started in background using the **&** character at the end of the command line are also visible with the **jobs** command.

```
[paul@RHEL4a ~]$ find / > allfiles.txt 2> /dev/null &
[6] 5230
[paul@RHEL4a ~]$ jobs
[5]+  Stopped                 vim procdemo.txt
[6]-  Running                 find / >allfiles.txt 2>/dev/null &
[paul@RHEL4a ~]$
```

jobs -p

An interesting option is **jobs -p** to see the **process id** of background processes.

```
[paul@RHEL4b ~]$ sleep 500 &
```

```
[1] 4902
[paul@RHEL4b ~]$ sleep 400 &
[2] 4903
[paul@RHEL4b ~]$ jobs -p
4902
4903
[paul@RHEL4b ~]$ ps `jobs -p`
  PID TTY      STAT   TIME COMMAND
 4902 pts/0    S      0:00 sleep 500
 4903 pts/0    S      0:00 sleep 400
[paul@RHEL4b ~]$
```

fg

Running the **fg** command will bring a background job to the foreground. The number of the background job to bring forward is the parameter of **fg**.

```
[paul@RHEL5 ~]$ jobs
[1]   Running                 sleep 1000 &
[2]-  Running                 sleep 1000 &
[3]+  Running                 sleep 2000 &
[paul@RHEL5 ~]$ fg 3
sleep 2000
```

bg

Jobs that are **suspended** in background can be started in background with **bg**. The **bg** will send a **SIGCONT** signal.

Below an example of the sleep command (suspended with **Ctrl-Z**) being reactivated in background with **bg**.

```
[paul@RHEL5 ~]$ jobs
[paul@RHEL5 ~]$ sleep 5000 &
[1] 6702
[paul@RHEL5 ~]$ sleep 3000

[2]+  Stopped                 sleep 3000
[paul@RHEL5 ~]$ jobs
[1]-  Running                 sleep 5000 &
[2]+  Stopped                 sleep 3000
[paul@RHEL5 ~]$ bg 2
[2]+ sleep 3000 &
[paul@RHEL5 ~]$ jobs
[1]-  Running                 sleep 5000 &
[2]+  Running                 sleep 3000 &
[paul@RHEL5 ~]$
```

31.2. practice : background processes

1. Use the **jobs** command to verify whether you have any processes running in background.

2. Use **vi** to create a little text file. Suspend **vi** in background.

3. Verify with **jobs** that **vi** is suspended in background.

4. Start **find / > allfiles.txt 2>/dev/null** in foreground. Suspend it in background before it finishes.

5. Start two long **sleep** processes in background.

6. Display all **jobs** in background.

7. Use the **kill** command to suspend the last **sleep** process.

8. Continue the **find** process in background (make sure it runs again).

9. Put one of the **sleep** commands back in foreground.

10. (if time permits, a general review question...) Explain in detail where the numbers come from in the next screenshot. When are the variables replaced by their value ? By which shell ?

```
[paul@RHEL4b ~]$ echo $$ $PPID
4224 4223
[paul@RHEL4b ~]$ bash -c "echo $$ $PPID"
4224 4223
[paul@RHEL4b ~]$ bash -c 'echo $$ $PPID'
5059 4224
[paul@RHEL4b ~]$ bash -c `echo $$ $PPID`
4223: 4224: command not found
```

31.3. solution : background processes

1. Use the **jobs** command to verify whether you have any processes running in background.

```
jobs (maybe the catfun is still running?)
```

2. Use **vi** to create a little text file. Suspend **vi** in background.

```
vi text.txt
(inside vi press ctrl-z)
```

3. Verify with **jobs** that **vi** is suspended in background.

```
[paul@rhel53 ~]$ jobs
[1]+  Stopped                 vim text.txt
```

4. Start **find / > allfiles.txt 2>/dev/null** in foreground. Suspend it in background before it finishes.

```
[paul@rhel53 ~]$ find / > allfiles.txt 2>/dev/null
   (press ctrl-z)
[2]+  Stopped                 find / > allfiles.txt 2> /dev/null
```

5. Start two long **sleep** processes in background.

```
sleep 4000 & ; sleep 5000 &
```

6. Display all **jobs** in background.

```
[paul@rhel53 ~]$ jobs
[1]-  Stopped                 vim text.txt
[2]+  Stopped                 find / > allfiles.txt 2> /dev/null
[3]   Running                 sleep 4000 &
[4]   Running                 sleep 5000 &
```

7. Use the **kill** command to suspend the last **sleep** process.

```
[paul@rhel53 ~]$ kill -SIGSTOP 4519
[paul@rhel53 ~]$ jobs
[1]   Stopped                 vim text.txt
[2]-  Stopped                 find / > allfiles.txt 2> /dev/null
[3]   Running                 sleep 4000 &
[4]+  Stopped                 sleep 5000
```

8. Continue the **find** process in background (make sure it runs again).

```
bg 2 (verify the job-id in your jobs list)
```

9. Put one of the **sleep** commands back in foreground.

```
fg 3 (again verify your job-id)
```

10. (if time permits, a general review question...) Explain in detail where the numbers come from in the next screenshot. When are the variables replaced by their value ? By which shell ?

```
[paul@RHEL4b ~]$ echo $$ $PPID
4224 4223
[paul@RHEL4b ~]$ bash -c "echo $$ $PPID"
4224 4223
[paul@RHEL4b ~]$ bash -c 'echo $$ $PPID'
5059 4224
[paul@RHEL4b ~]$ bash -c `echo $$ $PPID`
4223: 4224: command not found
```

The current bash shell will replace the $$ and $PPID while scanning the line, and before executing the echo command.

```
[paul@RHEL4b ~]$ echo $$ $PPID
4224 4223
```

The variables are now double quoted, but the current bash shell will replace $$ and $PPID while scanning the line, and before executing the bach -c command.

```
[paul@RHEL4b ~]$ bash -c "echo $$ $PPID"
4224 4223
```

The variables are now single quoted. The current bash shell will **not** replace the $$ and the $PPID. The bash -c command will be executed before the variables replaced with their value. This latter bash is the one replacing the $$ and $PPID with their value.

```
[paul@RHEL4b ~]$ bash -c 'echo $$ $PPID'
5059 4224
```

With backticks the shell will still replace both variable before the embedded echo is executed. The result of this echo is the two process id's. These are given as commands to bash -c. But two numbers are not commands!

```
[paul@RHEL4b ~]$ bash -c `echo $$ $PPID`
4223: 4224: command not found
```

```
haldaemon:x:68:
xfs:x:43:
gdm:x:42:
rich:x:500:
mama:x:501:
katie:x:502:
jessica:x:503:
mysql:x:27:
test:x:504:
sharing:x:505:test,rich
#
```

When changing the name of a group, the GID and group members remain the same, only the group name changes. Because all security permissions are based on the GID, you can change the name of a group as often as you wish without adversely affecting file security.

Decoding File Permissions

Now that you know about users and groups, it's time to decode the cryptic file permissions you've seen when using the `ls` command. This section describes how to decipher the permissions and where they come from.

Using File Permission Symbols

If you remember from Chapter 3, the `ls` command allows you to see the file permissions for files, directories, and devices on the Linux system:

```
$ ls -l
total 68
-rw-rw-r-- 1 rich rich   50 2010-09-13 07:49 file1.gz
-rw-rw-r-- 1 rich rich   23 2010-09-13 07:50 file2
-rw-rw-r-- 1 rich rich   48 2010-09-13 07:56 file3
-rw-rw-r-- 1 rich rich   34 2010-09-13 08:59 file4
-rwxrwxr-x 1 rich rich 4882 2010-09-18 13:58 myprog
-rw-rw-r-- 1 rich rich  237 2010-09-18 13:58 myprog.c
drwxrwxr-x 2 rich rich 4096 2010-09-03 15:12 test1
drwxrwxr-x 2 rich rich 4096 2010-09-03 15:12 test2
```

```
$
```

The first field in the output listing is a code that describes the permissions for the files and directories. The first character in the field defines the type of the object:

- `-` for files
- d for directories
- `l` for links
- c for character devices
- b for block devices
- n for network devices

After that, there are three sets of three characters. Each set of three characters defines an access permission triplet:

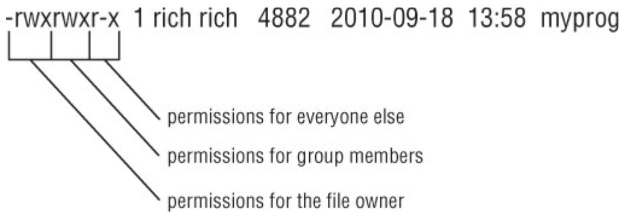
- `r` for read permission for the object
- `w` for write permission for the object
- `x` for execute permission for the object

If a permission is denied, a dash appears in the location. The three sets relate the three levels of security for the object:

- The owner of the object
- The group that owns the object
- Everyone else on the system

This is broken down in Figure 6.1.

Figure 6.1 The Linux file permissions


-rwxrwxr-x 1 rich rich 4882 2010-09-18 13:58 myprog
permissions for everyone else
permissions for group members
permissions for the file owner

The easiest way to discuss this is to take an example and decode the file permissions one by one:

```
-rwxrwxr-x 1 rich rich 4882 2010-09-18 13:58 myprog
```

The file `myprog` has the following sets of permissions:

- `rwx` for the file owner (set to the login name rich)
- `rwx` for the file group owner (set to the group name rich)
- `r-x` for everyone else on the system

These permissions indicate that the user login name rich can read, write, and execute the file (considered full permissions). Likewise, members in the group rich can also read, write, and execute the file. However, anyone else not in the rich group can only read and execute the file; the *w* is replaced with a dash, indicating that write permissions are not assigned to this security level.

Default File Permissions

You may be wondering about where these file permissions come from. The answer is *umask*. The `umask` command sets the default permissions for any file or directory you create:

```
$ touch newfile

$ ls -al newfile

-rw-r--r--    1 rich     rich            0 Sep 20 19:16 newfile

$
```

The `touch` command created the file using the default permissions assigned to my user account. The `umask` command shows and sets the default permissions:

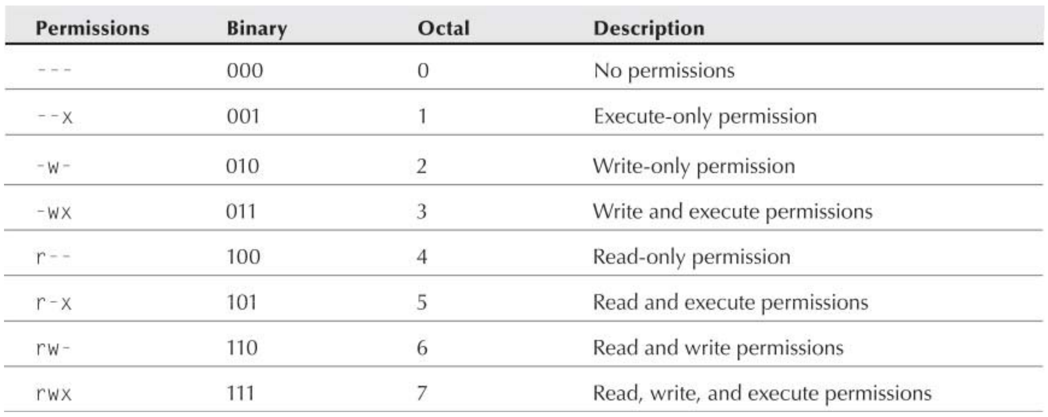
```
$ umask

0022

$
```

Unfortunately, the `umask` command setting isn't overtly clear, and trying to understand exactly how it works makes things even muddier. The first digit represents a special security feature called the *`sticky bit`*. We'll talk more about that later on in this chapter in the "Sharing Files" section.

The next three digits represent the octal values of the `umask` for a file or directory. To understand how `umask`works, you first need to understand octal mode security settings.

`Octal mode` security settings take the three rwx permission values and convert them into a 3-bit binary value, represented by a single octal value. In the binary representation, each position is a binary bit. Thus, if the read permission is the only permission set, the value becomes `r--`, relating to a binary value of 100, indicating the octal value of 4. Table 6.5 shows the possible combinations you'll run into.

Table 6.5 Linux File Permission Codes



Permissions	Binary	Octal	Description
`---`	000	0	No permissions
`--x`	001	1	Execute-only permission
`-w-`	010	2	Write-only permission
`-wx`	011	3	Write and execute permissions
`r--`	100	4	Read-only permission
`r-x`	101	5	Read and execute permissions
`rw-`	110	6	Read and write permissions
`rwx`	111	7	Read, write, and execute permissions

Octal mode takes the octal permissions and lists three of them in order for the three security levels (user, group, and everyone). Thus, the octal mode value 664 represents read and write permissions for the user and group, but read-only permission for everyone else.

Now that you know about octal mode permissions, the `umask` value becomes even more confusing. The octal mode shown for the default umask on my Linux system is 0022, but the file I created had an octal mode permission of 644. How did that happen?

The umask value is just that, a mask. It masks out the permissions you don't want to give to the security level. Now we have to dive into some octal arithmetic to figure out the rest of the story.

The umask value is subtracted from the full permission set for an object. The full permission for a file is mode 666 (read/write permission for all), but for a directory it's 777 (read/write/execute permission for all).

Thus, in the example, the file starts out with permissions 666, and the umask of 022 is applied, leaving a file permission of 644.

The umask value is normally set in the `/etc/profile` startup file (see Chapter 5). You can specify a different default umask setting using the `umask` command:

```
$ umask 026
$ touch newfile2
$ ls -l newfile2
-rw-r-----    1 rich     rich            0 Sep 20 19:46 newfile2
$
```

By setting the umask value to 026, the default file permissions become 640, so the new file now is restricted to read-only for the group members, and everyone else on the system has no permissions to the file.

The umask value also applies to making new directories:

```
$ mkdir newdir
$ ls -l
drwxr-x--x    2 rich     rich         4096 Sep 20 20:11 newdir/
$
```

Because the default permissions for a directory are 777, the resulting permissions from the umask are different from those of a new file. The 026 umask value is subtracted from 777, leaving the 751 directory permission setting.

Changing Security Settings

If you've already created a file or directory and need to change the security settings on it, there are a few different utilities available in Linux. This section shows you how to change the existing permissions, the default owner, and the default group settings for a file or directory.

Changing Permissions

The chmod command allows you to change the security settings for files and directories. The format of the chmodcommand is:

```
chmod options mode file
```

The mode parameter allows you to set the security settings using either octal or symbolic mode. The octal mode settings are pretty straightforward; just use the standard three-digit octal code you want the file to have:

```
$ chmod 760 newfile
$ ls -l newfile
-rwxrw----    1 rich     rich            0 Sep 20 19:16 newfile
$
```

The octal file permissions are automatically applied to the file indicated. The symbolic mode permissions are not so easy to implement.

Instead of using the normal string of three sets of three characters, the chmod command takes a different approach. The following is the format for specifying a permission in symbolic mode:

```
[ugoa…][[+-=][rwxXstugo…]
```

Makes perfectly good sense, doesn't it? The first group of characters defines to whom the new permissions apply:

- u for the user
- g for the group
- o for others (everyone else)
- a for all of the above

Next, a symbol is used to indicate whether you want to add the permission to the existing permissions (+), subtract the permission from the existing permission (–), or set the permissions to the value (=).

Finally, the third symbol is the permission used for the setting. You may notice that there are more than the normal rwx values here. The additional settings are:

- X to assign execute permissions only if the object is a directory or if it already had execute permissions
- s to set the UID or GID on execution
- t to save program text
- u to set the permissions to the owner's permissions
- g to set the permissions to the group's permissions
- o to set the permissions to the other's permissions

Using these permissions looks like this:

```
$ chmod o+r newfile
$ ls -l newfile
-rwxrw-r--    1 rich     rich            0 Sep 20 19:16 newfile
$
```

The o+r entry adds the read permission to whatever permissions the everyone security level already had.

```
$ chmod u-x newfile
$ ls -l newfile
-rw-rw-r--    1 rich     rich           0 Sep 20 19:16 newfile
$
```

The u-x entry removes the execute permission that the user already had. Note that the settings for the ls command indicate if a file has execution permissions by adding an asterisk to the file name.

The *options* parameters provide a few additional features to augment the behavior of the chmod command. The -Rparameter performs the file and directory changes recursively. You can use wildcard characters for the file name specified, changing the permissions on multiple files with just one command.

Changing Ownership

Sometimes you need to change the owner of a file, such as when someone leaves an organization or a developer creates an application that needs to be owned by a system account when it's in production. Linux provides two commands for doing that. The chown command makes it easy to change the owner of a file, and the chgrp command allows you to change the default group of a file.

The format of the chown command is:

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

You can specify either the login name or the numeric UID for the new owner of the file:

```
# chown dan newfile
# ls -l newfile
-rw-rw-r--    1 dan      rich           0 Sep 20 19:16 newfile
#
```

Simple. The chown command also allows you to change both the user and group of a file:

```
# chown dan.shared newfile
# ls -l newfile
-rw-rw-r--    1 dan      shared            0 Sep 20 19:16 newfile
#
```

If you really want to get tricky, you can just change the default group for a file:

```
# chown .rich newfile
# ls -l newfile
-rw-rw-r--    1 dan      rich           0 Sep 20 19:16 newfile
#
```

Finally, if your Linux system uses individual group names that match user login names, you can change both with just one entry:

```
# chown test. newfile
# ls -l newfile
-rw-rw-r--    1 test    test             0 Sep 20 19:16 newfile
#
```

The chown command uses a few different option parameters. The -R parameter allows you to make changes recursively through subdirectories and files, using a wildcard character. The -h parameter also changes the ownership of any files that are symbolically linked to the file.

Note

Only the root user can change the owner of a file. Any user can change the default group of a file, but the user must be a member of the groups the file is changed from and to.

The chgrp command provides an easy way to change just the default group for a file or directory:

```
$ chgrp shared newfile
$ ls -l newfile
-rw-rw-r--    1 rich     shared          0 Sep 20 19:16 newfile
$
```

Now any member in the shared group can write to the file. This is one way to share files on a Linux system. However, sharing files among a group of people on the system can get tricky. The next section discusses how to do this.

Sharing Files

As you've probably already figured out, creating groups is the way to share access to files on the Linux system. However, for a complete file-sharing environment, things are more complicated.

As you've already seen in the "Decoding File Permissions" section, when you create a new file, Linux assigns the file permissions of the new file using your default UID and GID. To allow others access to the file, you need to either change the security permissions for the everyone security group or assign the file a different default group that contains other users.

This can be a pain in a large environment if you want to create and share documents among several people. Fortunately, there's a simple solution for how to solve this problem.

There are three additional bits of information that Linux stores for each file and directory:

- **The set user id (SUID):** When a file is executed by a user, the program runs under the permissions of the file owner.

- **The set group id (SGID):** For a file, the program runs under the permissions of the file group. For a directory, new files created in the directory use the directory group as the default group.
- **The sticky bit:** The file remains (sticks) in memory after the process ends.

The SGID bit is important for sharing files. By enabling the SGID bit, you can force all new files created in a shared directory to be owned by the directory's group and now the individual user's group.

The SGID is set using the chmod command. It's added to the beginning of the standard three-digit octal value (making a four-digit octal value), or you can use the symbol s in symbolic mode.

If you're using octal mode, you'll need to know the arrangement of the bits, shown in Table 6.6.

Table 6.6 The chmod SUID, SGID, and Sticky Bit Octal Values

Binary	Octal	Description
000	0	All bits are cleared.
001	1	The sticky bit is set.
010	2	The SGID bit is set.
011	3	The SGID and sticky bits are set.
100	4	The SUID bit is set.
101	5	The SUID and sticky bits are set.
110	6	The SUID and SGID bits are set.
111	7	All bits are set.

So, to create a shared directory that always sets the directory group for all new files, all you need to do is set the SGID bit for the directory:

```
$ mkdir testdir
$ ls -l
drwxrwxr-x    2 rich     rich         4096 Sep 20 23:12 testdir/
$ chgrp shared testdir
$ chmod g+s testdir
$ ls -l
drwxrwsr-x    2 rich     shared       4096 Sep 20 23:12 testdir/
$ umask 002
$ cd testdir
$ touch testfile
$ ls -l
total 0
-rw-rw-r--    1 rich     shared          0 Sep 20 23:13 testfile
```

```
$
```

The first step is to create a directory that you want to share using the `mkdir` command. Next, the `chgrp` command is used to change the default group for the directory to a group that contains the members who need to share files. Finally, the SGID bit is set for the directory to ensure that any files created in the directory use the shared group name as the default group.

For this environment to work properly, all of the group members need to have their umask values set to make files writable by group members. In the preceding example, the `umask` is changed to `002` so that the files are writable by the group.

After all that's done, any member of the group can go to the shared directory and create a new file. As expected, the new file uses the default group of the directory, not the user account's default group. Now any user in the shared group can access this file.

Summary

This chapter discussed the command line commands you need to know to manage the Linux security on your system. Linux uses a system of user IDs and group IDs to protect access to files, directories, and devices. Linux stores information about user accounts in the `/etc/passwd` file and information about groups in the `/etc/group` file. Each user is assigned a unique numeric user ID, along with a text login name to identify the user in the system. Groups are also assigned unique numerical group IDs, and text group names. A group can contain one or more users to allowed shared access to system resources.

There are several commands available for managing user accounts and groups. The `useradd` command allows you to create new user accounts, and the `groupadd` command allows you to create new group accounts. To modify an existing user account, use the `usermod` command. Similarly, the `groupmod` command is used to modify group account information.

Linux uses a complicated system of bits to determine access permissions for files and directories. Each file contains three security levels of protection: the file's owner, a default group that has access to the file, and a level for everyone else on the system. Each security level is defined by three access bits: read, write, and execute. The combination of three bits is often referred to by the symbols `rwx`, for read, write, and execute. If a permission is denied, its symbol is replaced with a dash (such as `r--` for read-only permission).

The symbolic permissions are often referred to as octal values, with the three bits combined into one octal value and three octal values representing the three security levels. The `umask` command is used to set the default security settings for files and directories created on the system. The system administrator normally sets a default umask value in the `/etc/profile` file, but you can use the `umask` command to change your umask value at any time.

The `chmod` command is used to change security settings for files and directories. Only the file's owner can change permissions for a file or directory. However, the root user can change the security settings for any file or directory on the system.

- If the word appears after the current cursor location, it jumps to the first location where the text appears.
- If the word doesn't appear after the current cursor location, it wraps around the end of the file to the first location in the file where the text appears (and indicates this with a message).
- It produces an error message stating that the text was not found in the file.

To continue searching for the same word, press the forward slash character and then press the Enter key, or you can use the n key, for *next*.

The substitute command allows you to quickly replace (substitute) one word for another in the text. To get to the substitute command you must be in command line mode. The format for the substitute command is:

```
:s/old/new/
```

The vim editor jumps to the first occurrence of the text `old` and replaces it with the text `new`. There are a few modifications you can make to the substitute command to substitute more than one occurrence of the text:

- `:s/old/new/g` to replace all occurrences of `old` in a line
- `:`*`n`*`,`*`ms`*`/old/new/g` to replace all occurrences of `old` between line numbers n and m
- `:%s/old/new/g` to replace all occurrences of `old` in the entire file
- `:%s/old/new/gc` to replace all occurrences of `old` in the entire file, but prompt for each occurrence

As you can see, for a command line text editor, vim contains quite a few advanced features. Because every Linux distribution includes it, it's a good idea to at least know the basics of the vim editor so that you can always edit scripts, no matter where you are or what you have available.

The emacs Editor

The emacs editor is an extremely popular editor that appeared before even Unix was around. Developers liked it so much they ported it to the Unix environment, and now it's been ported to the Linux environment. The emacs editor started out life as a console editor, much like vi, but has made the migration to the graphical world.

The emacs editor still provides the original console mode editor, but now it also has the ability to use a graphical X Windows window to allow editing text in a graphical environment. Typically, when you start the emacs editor from a command line, the editor will determine if you have an available X Window session and start in graphical mode. If you don't, it will start in console mode.

This section describes both the console mode and graphical mode emacs editors so that you'll know how to use either one if you want (or need) to.

Using emacs on the Console

The console mode version of emacs is another editor that uses lots of key commands to perform editing functions. The emacs editor uses key combinations involving the Control key (the Ctrl key on the PC keyboard) and the Meta key. In most PC terminal emulator packages, the Meta key is mapped to the PC's Alt key. The official emacs documents abbreviate the Ctrl key as C- and the Meta key as M-, Thus, if you enter a Ctrl-x key combination, the document shows C-x. This chapter will do the same so as not to confuse you.

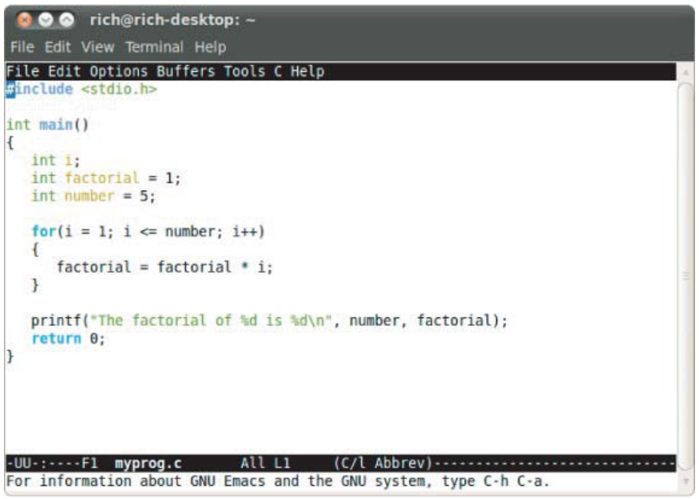
The Basics of emacs

To edit a file using emacs, from the command line, enter:

```
$ emacs myprog.c
```

The emacs console mode window appears with a short introduction and help screen. Don't be alarmed; as soon as you press a key, emacs loads the file into the active buffer and displays the text, as shown in Figure 9.2.

<u>Figure 9.2</u> Editing a file using the emacs editor in console mode


rich@rich-desktop: ~
File Edit View Terminal Help
File Edit Options Buffers Tools C Help

```
#include <stdio.h>

int main()
{
   int i;
   int factorial = 1;
   int number = 5;

   for(i = 1; i <= number; i++)
   {
      factorial = factorial * i;
   }

   printf("The factorial of %d is %d\n", number, factorial);
   return 0;
}
```

```
-UU-:----F1  myprog.c       All L1     (C/l Abbrev)----------------------------
For information about GNU Emacs and the GNU system, type C-h C-a.
```

You'll notice that the top of the console mode window shows a typical menu bar. Unfortunately, you won't be able to use the menu bar in console mode, only in graphical mode.

Note

If you have a graphical desktop but you prefer to use emacs in console mode instead of X Windows mode, use the -nwoption on the command line.

Unlike the vim editor, where you have to move into and out of insert mode to switch between entering commands and inserting text, the emacs editor has only one mode. If you type a printable character, emacs inserts it at the current cursor position. If you type a command, emacs executes the command.

To move the cursor around the buffer area, you can use the arrow keys and the PageUp and PageDown keys, assuming that emacs detected your terminal emulator correctly. If not, there are commands for moving the cursor around:

- `C-p` to move up one line (the previous line in the text).
- `C-b` to move left (back) one character.
- `C-f` to move right (forward) one character.
- `C-n` to move down one line (the next line in the text).

There are also commands for making longer jumps with the cursor within the text:

- `M-f` moves right (forward) to the next word.
- `M-b` moves left (backward) to the previous word.
- `C-a` moves to the beginning of the current line.
- `C-e` moves to the end of the current line.
- `M-a` moves to the beginning of the current sentence.
- `M-e` moves to the end of the current sentence.
- `M-v` moves back one screen of data.
- `C-v` moves forward one screen of data.
- `M-<` to move the first line of the text.
- `M->` to move to the last line of the text.

There are several commands you should know for saving the editor buffer back into the file, and exiting emacs:

- `C-x C-s` to save the current buffer contents to the file.
- `C-z` to exit emacs but keep it running in your session so that you can come back to it.
- `C-x C-c` to exit emacs and stop the program.

You'll notice that two of these features require two key commands. The `C-x` command is called the *extend command*. This provides yet another whole set of commands to work with.

Editing Data

The emacs editor is pretty robust about inserting and deleting text in the buffer. To insert text, just move the cursor to the location where you want to insert the text and start typing. To delete text, emacs uses the Backspace key to delete the character before the current cursor position and the Delete key to delete the character at the current cursor location.

The emacs editor also has commands for killing text. The difference between deleting text and killing text is that when you kill text, emacs places it in a temporary area where you can retrieve it (see the “Copying and Pasting” section). Deleted text is gone forever.

There are a few commands for killing text in the buffer:

- `M-Backspace` to kill the word before the current cursor position
- `M-d` to kill the word after the current cursor position
- `C-k` to kill from the current cursor position to the end of the line
- `M-k` to kill from the current cursor position to the end of the sentence

The emacs editor also includes a fancy way of mass-killing text. Just move the cursor to the start of the area you want to kill and press either the `C-@` or `C-Spacebar` keys. Then move the cursor to the end of the area you want to kill and press the `C-w` command keys. All of the text between the two locations is killed.

If you happen to make a mistake when killing text, the `C-u` command will undo the kill command, and return the data the state it was in before you killed it.

Copying and Pasting

You've seen how to cut data from the emacs buffer area; now it's time to see how to paste it somewhere else. Unfortunately, if you use the vim editor, this process may confuse you when you use the emacs editor.

In an unfortunate coincidence, pasting data in emacs is called *yanking*. In the vim editor, copying is called yanking, which is what makes this a difficult thing to remember if you happen to use both editors.

After you kill data using one of the kill commands, move the cursor to the location where you want to paste the data, and use the `C-y` command. This yanks the text out of the temporary area and pastes it at the current cursor position. The `C-y` command yanks the text from the last kill command. If you've performed multiple kill commands, you can cycle through them using the `M-y` command.

To copy text, just yank it back into the same location you killed it from and then move to the new location and use the `C-y` command again. You can yank text back as many times as you desire.

Searching and Replacing

Searching for text in the emacs editor is done by using the `C-s` and `C-r` commands. The `C-s` command performs a forward search in the buffer area from the current cursor position to the end of the buffer, whereas the `C-r`command performs a backward search in the buffer area from the current cursor position to the start of the buffer.

When you enter either the `C-s` or `C-r` command, a prompt appears in the bottom line, querying you for the text to search. There are two types of searches that emacs can perform.

In an *incremental* search, the emacs editor performs the text search in real-time mode as you type the word. When you type the first letter, it highlights all of the occurrences of that letter in the buffer. When you type the second letter, it highlights all of the occurrences of the two-letter combination in the text, and so on until you complete the text you're searching for.

In a *non-incremental* search, press the Enter key after the `C-s` or `C-r` commands. This locks the search query into the bottom line area and allows you to type the search text in full before searching.

To replace an existing text string with a new text string, you have to use the `M-x` command. This command requires a text command, along with parameters.

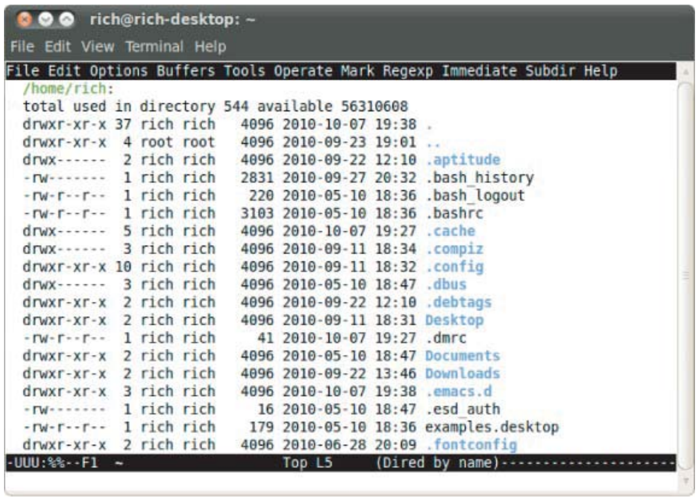
The text command is `replace-string`. After typing the command, press the Enter key, and emacs will query you for the existing text string. After entering that, press the Enter key again, and emacs will query you for the new replacement text string.

Using Buffers in emacs

The emacs editor allows you to edit multiple files at the same time by having multiple buffer areas. You can load files into a buffer and switch between buffers while editing.

To load a new file into a buffer while you're in emacs, use the `C-x C-f` key combination. This is the emacs Find a File mode. It takes you to the bottom line in the window and allows you to enter the name of the file you want to start to edit. If you don't know the name or location of the file, just press the Enter key. This brings up a file browser in the edit window, as shown in Figure 9.3.

Figure 9.3 The emacs Find a File mode browser



```
File Edit Options Buffers Tools Operate Mark Regexp Immediate Subdir Help
  /home/rich:
  total used in directory 544 available 56310608
  drwxr-xr-x 37 rich rich   4096 2010-10-07 19:38 .
  drwxr-xr-x  4 root root   4096 2010-09-23 19:01 ..
  drwx------  2 rich rich   4096 2010-09-22 12:10 .aptitude
  -rw-------  1 rich rich   2831 2010-09-27 20:32 .bash_history
  -rw-r--r--  1 rich rich    220 2010-05-10 18:36 .bash_logout
  -rw-r--r--  1 rich rich   3103 2010-05-10 18:36 .bashrc
  drwx------  5 rich rich   4096 2010-10-07 19:27 .cache
  drwx------  3 rich rich   4096 2010-09-11 18:34 .compiz
  drwxr-xr-x 10 rich rich   4096 2010-09-11 18:32 .config
  drwx------  3 rich rich   4096 2010-05-10 18:47 .dbus
  drwxr-xr-x  2 rich rich   4096 2010-09-22 12:10 .debtags
  drwxr-xr-x  2 rich rich   4096 2010-09-11 18:31 Desktop
  -rw-r--r--  1 rich rich     41 2010-10-07 19:27 .dmrc
  drwxr-xr-x  2 rich rich   4096 2010-05-10 18:47 Documents
  drwxr-xr-x  2 rich rich   4096 2010-09-22 13:46 Downloads
  drwxr-xr-x  3 rich rich   4096 2010-10-07 19:38 .emacs.d
  -rw-------  1 rich rich     16 2010-05-10 18:47 .esd_auth
  -rw-r--r--  1 rich rich    179 2010-05-10 18:36 examples.desktop
  drwxr-xr-x  2 rich rich   4096 2010-06-28 20:09 .fontconfig
-UUU:%%--F1  ~                 Top L5     (Dired by name)--------------------
```

From here, you can browse to the file you want to edit. To traverse up a directory level, go to the double dot entry, and press the Enter key. To traverse down a directory, go to the directory entry and press the Enter key. When you've found the file you want to edit, just press the Enter key, and emacs will load it into a new buffer area.

You can list the active buffer areas by pressing the `C-x C-b` extended command combination. The emacs editor splits the editor window and displays a list of buffers in the bottom window. There are always two buffers that emacs provides in addition to your main editing buffer:

- A scratch area called *scratch*
- A message area called *Messages*

The scratch area allows you to enter LISP programming commands as well as enter notes to yourself. The message area shows messages generated by emacs while operating. If any errors occur while using emacs, they will appear in the message area.

There are two ways to switch to a different buffer area in the window:

- `C-x o` to switch to the buffer listing window. Use the arrow keys to move to the buffer area you want and press the Enter key.

- `C-x b` to type in the name of the buffer area you want to switch to.

When you select the option to switch to the buffer listing window, emacs will open the buffer area in the new window area. The emacs editor allows you to have multiple windows open in a single session. The following section discusses how to manage multiple windows in emacs.

Using Windows in Console Mode emacs

The console mode emacs editor was developed many years before the idea of graphical windows appeared. However, it was advanced for its time, in that it could support multiple editing windows within the main emacs window.

You can split the emacs editing window into multiple windows by using one of two commands:

- `C-x 2` splits the window horizontally into two windows.
- `C-x 3` splits the window vertically into two windows.

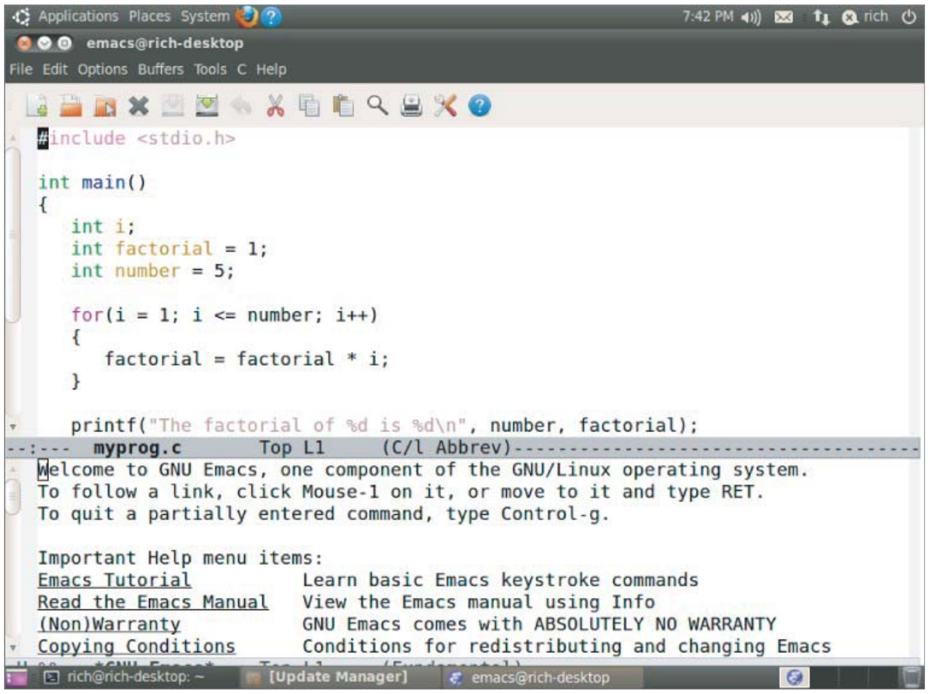
To move from one window to another, use the `C-x o` command. You'll notice that when you create a new window, emacs uses the buffer area from the original window in the new window. Once you move into the new window, you can use the`C-x C-f` command to load a new file, or one of the commands to switch to a different buffer area in the new window.

To close a window, move to it and use the `C-x 0` (that's a zero) command. If you want to close all of the windows except the one you're in, use the `C-x 1` (that's a numerical one) command.

Using emacs in X Windows

If you use emacs from an X Windows environment (such as the KDE or GNOME desktops), it will start in graphical mode, as shown in Figure 9.4.

Figure 9.4 The emacs graphical window



```
#include <stdio.h>

int main()
{
   int i;
   int factorial = 1;
   int number = 5;

   for(i = 1; i <= number; i++)
   {
      factorial = factorial * i;
   }

   printf("The factorial of %d is %d\n", number, factorial);
--:---  myprog.c       Top L1     (C/l Abbrev)------------------------------------
Welcome to GNU Emacs, one component of the GNU/Linux operating system.
To follow a link, click Mouse-1 on it, or move to it and type RET.
To quit a partially entered command, type Control-g.

Important Help menu items:
Emacs Tutorial          Learn basic Emacs keystroke commands
Read the Emacs Manual   View the Emacs manual using Info
(Non)Warranty           GNU Emacs comes with ABSOLUTELY NO WARRANTY
Copying Conditions      Conditions for redistributing and changing Emacs
```

If you've already used emacs in console mode, you should be fairly familiar with the X Windows mode. All of the key commands are available as menu bar items. The emacs menu bar contains the following items:

- **File:** Allows you to open files in the window, create new windows, close windows, save buffers, and print buffers.
- **Edit:** Allows you to cut and copy selected text to the clipboard, paste clipboard data to the current cursor position, search for text, and replace text.
- **Options:** Provides settings for many more emacs features, such as highlighting, word wrap, cursor type, and setting fonts.
- **Buffers:** Lists the current buffers available and allows you to easily switch between buffer areas.
- **Tools:** Provides access to the advanced features in emacs, such as the command line interface access, spell checking, comparing text between files (called diff), sending an e-mail message, calendar, and the calculator.
- **Help:** Provides the emacs manual online for access to help on specific emacs functions.

In addition to the normal graphical emacs menu bar items, there is often a separate item specific to the file type in the editor buffer. Figure 9.4 shows opening a C program, so emacs provided a C menu item, allowing advanced settings for highlighting C syntax, and compiling, running, and debugging the code from a command prompt.

The graphical emacs window is an example of an older console application making the migration to the graphical world. Now that many Linux distributions provide graphical desktops (even on servers that don't need them), graphical editors are becoming more commonplace. Both of the popular Linux desktop environments (KDE and GNOME) have also provided graphical text editors specifically for their environments, which are covered in the rest of this chapter.

The KDE Family of Editors

If you're using a Linux distribution that uses the KDE desktop (see Chapter 1), there are a couple of options for you when it comes to text editors. The KDE project officially supports two different text editors:

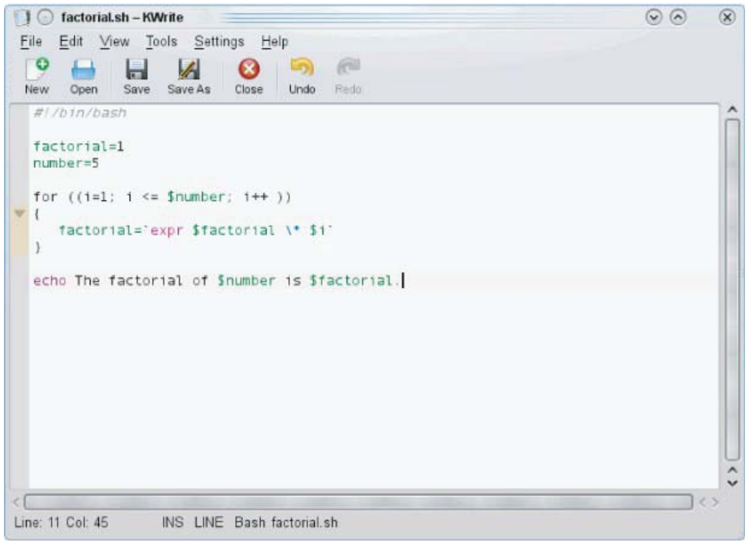
- **KWrite:** A single-screen text-editing package
- **Kate:** A full-featured, multi-window text-editing package

Both of these editors are graphical text editors that contain many advanced features. The Kate editor provides more advanced features, plus extra niceties not often found in standard text editors. This section describes each of the editors and shows some of the features that you can use to help with your shell script editing.

The KWrite Editor

The basic editor for the KDE environment is KWrite. It provides simple word-processing–style text editing, along with support for code syntax highlighting and editing. The default KWrite editing window is shown in Figure 9.5.

Figure 9.5 The default KWrite window editing a shell script program



```
#!/bin/bash

factorial=1
number=5

for ((i=1; i <= $number; i++ ))
{
   factorial=`expr $factorial \* $i`
}

echo The factorial of $number is $factorial.
```

You can't tell from Figure 9.5, but the KWrite editor recognizes several types of programming languages and uses color coding to distinguish constants, functions, and comments. Also, notice that the `for` loop has an icon that links the opening and closing braces. This is called a *folding marker*. By clicking the icon, you can collapse the function into a single line. This is a great feature when working through large applications.

Chapter 10

Basic Script Building

In This Chapter

- Basic script building
- Using multiple commands
- Creating a script file

Now that we've covered the basics of the Linux system and the command line, it's time to start coding. This chapter discusses the basics of writing shell scripts. You'll need to know these basic concepts before you can start writing your own shell script masterpieces.

Using Multiple Commands

So far you've seen how to use the command line interface (CLI) prompt of the shell to enter commands and view the command results. The key to shell scripts is the ability to enter multiple commands and process the results from each command, even possibly passing the results of one command to another. The shell allows you to chain commands together into a single step.

If you want to run two commands together, you can enter them on the same prompt line, separated with a semicolon:

```
$ date ; who
Mon Feb 21 15:36:09 EST 2011
Christine tty2         2011-02-21 15:26
Samantha tty3         2011-02-21 15:26
Timothy  tty1         2011-02-21 15:26
user     tty7         2011-02-19 14:03 (:0)
user     pts/0        2011-02-21 15:21 (:0.0)


$
```

Congratulations, you just wrote a shell script! This simple script uses just two bash shell commands. The `date`command runs first, displaying the current date and time, followed by the output of the who command, showing who is currently logged on to the system. Using this technique, you can string together as many commands as you wish, up to the maximum command line character count of 255 characters.

While using this technique is fine for small scripts, it has a major drawback in that you have to enter the entire command at the command prompt every time you want to run it. Instead of having to manually enter the commands onto a command line, you can combine the commands into a simple text file. When you need to run the commands, just simply run the text file.

Creating a Script File

To place shell commands in a text file, first you'll need to use a text editor (see Chapter 9) to create a file, then enter the commands into the file.

When creating a shell script file, you must specify the shell you are using in the first line of the file. The format for this is:

```
#!/bin/bash
```

In a normal shell script line, the pound sign (#) is used as a comment line. A comment line in a shell script isn't processed by the shell. However, the first line of a shell script file is a special case, and the pound sign followed by the exclamation point tells the shell what shell to run the script under (yes, you can be using a bash shell and run your script using another shell).

After indicating the shell, commands are entered onto each line of the file, followed by a carriage return. As mentioned, comments can be added by using the pound sign. An example looks like this:

```
#!/bin/bash
# This script displays the date and who's logged on
```

```
date
who
```

And that's all there is to it. You can use the semicolon and put both commands on the same line if you want to, but in a shell script, you can list commands on separate lines. The shell will process commands in the order in which they appear in the file.

Also notice that another line was included that starts with the pound symbol and adds a comment. Lines that start with the pound symbol (other than the first `#!` line) aren't interpreted by the shell. This is a great way to leave comments for yourself about what's happening in the script, so when you come back to it two years later you can easily remember what you did.

Save this script in a file called `test1`, and you are almost ready. There are still a couple of things to do before you can run your new shell script file.

If you try running the file now, you'll be somewhat disappointed to see this:

```
$ test1
bash: test1: command not found
$
```

The first hurdle to jump is getting the bash shell to find your script file. If you remember from Chapter 5, the shell uses an environment variable called PATH to find commands. A quick look at the PATH environment variable demonstrates our problem:

```
$ echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/usr/bin
:/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user/bin $
```

The PATH environment variable is set to look for commands only in a handful of directories. To get the shell to find the `test1` script, we need to do one of two things:

- Add the directory where our shell script file is located to the PATH environment variable.
- Use an absolute or relative filepath to reference our shell script file in the prompt.

Tip

Some Linux distributions add the `$HOME/bin` directory to the PATH environment variable. This creates a place in every user's HOME directory to place files where the shell can find them to execute.

For this example, we'll use the second method to tell the shell exactly where the script file is located. Remember that to reference a file in the current directory, you can use the single dot operator in the shell:

```
$ ./test1
bash: ./test1: Permission denied
$
```

Now the shell found the shell script file just fine, but there's another problem. The shell indicated that you don't have permission to execute the file. A quick look at the file permissions should show what's going on here:

```
$ ls -l test1
-rw-r--r--    1 user     user           73 Sep 24 19:56 test1
$
```

When the new `test1` file was created, the `umask` value determined the default permission settings for the new file. Because the `umask` variable is set to 022 (see Chapter 6), the system created the file with only read/write permissions for the file's owner.

The next step is to give the file owner permission to execute the file, using the `chmod` command (see Chapter 6):

```
$ chmod u+x test1
$ ./test1
Mon Feb 21 15:38:19 EST 2011
Christine tty2         2011-02-21 15:26
Samantha tty3         2011-02-21 15:26
Timothy  tty1         2011-02-21 15:26
user     tty7         2011-02-19 14:03 (:0)
user     pts/0        2011-02-21 15:21 (:0.0) $
```

Success! Now all of the pieces are in the right places to execute the new shell script file.

Displaying Messages

Most shell commands produce their own output, which is displayed on the console monitor where the script is running. Many times, however, you will want to add your own text messages to help the script user know what is happening within the script. You can do this with the `echo` command. The `echo` command can display a simple text string if you add the string following the command:

```
$ echo This is a test
This is a test
$
```

Notice that by default you don't need to use quotes to delineate the string you're displaying. However, sometimes this can get tricky if you are using quotes within your string:

```
$ echo Let's see if this'll work
Lets see if thisll work
$
```

The `echo` command uses either double or single quotes to delineate text strings. If you use them within your string, you need to use one type of quote within the text and the other type to delineate the string:

```
$ echo “This is a test to see if you're paying attention”
This is a test to see if you're paying attention
$ echo ‘Rich says “scripting is easy”.’
Rich says “scripting is easy”.
$
```

Now all of the quotation marks appear properly in the output.

You can add `echo` statements anywhere in your shell scripts where you need to display additional information:

```
$ cat test1
#!/bin/bash
# This script displays the date and who's logged on
echo  The time and date are:
date
echo “Let's see who's logged into the system:”
who
$
```

When you run this script, it produces the following output:

```
$ ./test1
The time and date are:
Mon Feb 21 15:41:13 EST 2011
Let's see who's logged into the system:
Christine tty2         2011-02-21 15:26
Samantha tty3         2011-02-21 15:26
Timothy  tty1         2011-02-21 15:26
user     tty7         2011-02-19 14:03 (:0)
user     pts/0        2011-02-21 15:21 (:0.0)
$
```

That's nice, but what if you want to echo a text string on the same line as a command output? You can use the -nparameter for the echo statement to do that. Just change the first echo statement line to this:

```
echo -n “The time and date are: ”
```

You'll need to use quotes around the string to ensure that there's a space at the end of the echoed string. The command output begins exactly where the string output stops. The output will now look like this:

```
$ ./test1
The time and date are: Mon Feb 21 15:42:23 EST 2011
Let's see who's logged into the system:
Christine tty2         2011-02-21 15:26
Samantha tty3         2011-02-21 15:26
Timothy  tty1         2011-02-21 15:26
user     tty7         2011-02-19 14:03 (:0)
user     pts/0        2011-02-21 15:21 (:0.0)
$
```

Perfect! The echo command is a crucial piece of shell scripts that interact with users. You'll find yourself using it in many situations, especially when you want to display the values of script variables. Let's look at that next.

Using Variables

Just running individual commands from the shell script is useful, but this has its limitations. Often you'll want to incorporate other data in your shell commands to process information. You can do this by using *variables*. Variables allow you to temporarily store information within the shell script for use with other commands in the script. This section shows how to use variables in your shell scripts.

Environment Variables

You've already seen one type of Linux variable in action. Chapter 5 described the environment variables available in the Linux system. You can access these values from your shell scripts as well.

The shell maintains environment variables that track specific system information, such as the name of the system, the name of the user logged in to the system, the user's system ID (called UID), the default home directory of the user, and the search path used by the shell to find programs. You can display a complete list of active environment variables available by using the `set` command:

```
$ set
BASH=/bin/bash
...
HOME=/home/Samantha
HOSTNAME=localhost.localdomain
HOSTTYPE=i386
IFS=$‘ \t\n’
IMSETTINGS_INTEGRATE_DESKTOP=yes
IMSETTINGS_MODULE=none
LANG=en_US.utf8
LESSOPEN=‘|/usr/bin/lesspipe.sh %s’
LINES=24
LOGNAME=Samantha
...
```

You can tap into these environment variables from within your scripts by using the environment variable's name preceded by a dollar sign. This is demonstrated in the following script:

```
$ cat test2
#!/bin/bash
# display user information from the system.
echo “User info for userid: $USER”
echo UID: $UID
echo HOME: $HOME
$
```

The *$USER*, *$UID*, and *$HOME* environment variables are used to display the pertinent information about the logged-in user. The output should look something like this:

```
$chmod u+x test2
$ ./test2
User info for userid: Samantha
UID: 1001
HOME: /home/Samantha
$ $
```

Notice that the environment variables in the echo commands are replaced by their current values when the script is run. Also notice that we were able to place the *$USER* system variable within the double quotation marks in the first string, and the shell script was still able to figure out what we meant. There is a drawback to using this method, however. Look at what happens in this example:

```
$ echo “The cost of the item is $15”
The cost of the item is 5
```

That is obviously not what was intended. Whenever the script sees a dollar sign within quotes, it assumes you're referencing a variable. In this example the script attempted to display the variable *$1* (which was not defined), and then the number 5. To display an actual dollar sign, you must precede it with a backslash character:

```
$ echo “The cost of the item is \$15”
The cost of the item is $15
```

That's better. The backslash allowed the shell script to interpret the dollar sign as an actual dollar sign, and not a variable. The next section shows how to create your own variables in your scripts.

Note

You may also see variables referenced using the format ${variable}. The extra braces around the variable name are often used to help identify the variable name from the dollar sign.

User Variables

In addition to the environment variables, a shell script allows you to set and use your own variables within the script. Setting variables allows you to temporarily store data and use it throughout the script, making the shell script more like a real computer program.

User variables can be any text string of up to 20 letters, digits, or an underscore character. User variables are case sensitive, so the variable *Var1* is different from the variable *var1*. This little rule often gets novice script programmers in trouble.

Values are assigned to user variables using an equal sign. No spaces can appear between the variable, the equal sign, and the value (another trouble spot for novices). Here are a few examples of assigning values to user variables:

```
var1=10
var2=-57
var3=testing
var4=“still more testing”
```

The shell script automatically determines the data type used for the variable value. Variables defined within the shell script maintain their values throughout the life of the shell script but are deleted when the shell script completes.

Just like system variables, user variables can be referenced using the dollar sign:

```
$ cat test3
#!/bin/bash
# testing variables
```

```
days=10
guest="Katie"
echo "$guest checked in $days days ago"
days=5
guest="Jessica"
echo "$guest checked in $days days ago"
$
```

Running the script produces the following output:

```
$ chmod u+x test3
$ ./test3
Katie checked in 10 days ago
Jessica checked in 5 days ago
$
```

Each time the variable is referenced, it produces the value currently assigned to it. It's important to remember that when referencing a variable value you use the dollar sign, but when referencing the variable to assign a value to it, you do not use the dollar sign. Here's an example of what I mean:

```
$ cat test4
#!/bin/bash
# assigning a variable value to another variable


value1=10
value2=$value1
echo The resulting value is $value2
$
```

When you use the value of the *value1* variable in the assignment statement, you must still use the dollar sign. This code produces the following output:

```
$ chmod u+x test4
$ ./test4
The resulting value is 10
$
```

If you forget the dollar sign, and make the *value2* assignment line look like:

```
value2=value1
```

you get the following output:

```
$ ./test4
The resulting value is value1
$
```

Without the dollar sign the shell interprets the variable name as a normal text string, which is most likely not what you wanted.

The Backtick

One of the most useful features of shell scripts is the lowly back quote character, usually called the backtick (‘) in the Linux world. Be careful—this is not the normal single quotation mark character you are used to using for strings. Because it is not used very often outside of shell scripts, you may not even know where to find it on your keyboard. You should become familiar with it, because it's a crucial component of many shell scripts. Hint: On a U.S. keyboard, it is usually on the same key as the tilde symbol (~).

The backtick allows you to assign the output of a shell command to a variable. While this doesn't seem like much, it is a major building block in script programming.

You must surround the entire command line command with backtick characters:

```
testing=‘date’
```

The shell runs the command within the backticks and assigns the output to the variable testing. Here's an example of creating a variable using the output from a normal shell command:

```
$ cat test5
#!/bin/bash
# using the backtick character
testing=‘date’
echo “The date and time are: ” $testing
$
```

The variable testing receives the output from the `date` command, and it is used in the `echo` statement to display it. Running the shell script produces the following output:

```
$ chmod u+x test5
$ ./test5
The date and time are:  Mon Jan 31 20:23:25 EDT 2011
$
```

That's not all that exciting in this example (you could just as easily just put the command in the `echo`statement), but once you capture the command output in a variable, you can do anything with it.

Here's a popular example of how the backtick is used to capture the current date and use it to create a unique filename in a script:

```
#!/bin/bash
# copy the /usr/bin directory listing to a log file
today=‘date +%y%m%d’
ls /usr/bin -al > log.$today
```

The `today` variable is assigned the output of a formatted date command. This is a common technique used to extract date information for log filenames. The `+%y%m%d` format instructs the date command to display the date as a two-digit year, month, and day:

```
$ date +%y%m%d
110131
$
```

The script assigns the value to a variable, which is then used as part of a filename. The file itself contains the redirected output (discussed later in the “Redirecting Input and Output” section) of a directory listing. After running the script, you should see a new file in your directory:

```
-rw-r--r--    1 user     user          769 Jan 31 10:15 log.110131
```

The log file appears in the directory using the value of the *`$today`* variable as part of the filename. The contents of the log file are the directory listing from the `/usr/bin` directory. If the script is run the next day, the log filename will be `log.110201`, thus creating a new file for the new day.

Redirecting Input and Output

There are times when you'd like to save the output from a command instead of just having it displayed on the monitor. The bash shell provides a few different operators that allow you to *`redirect`* the output of a command to an alternative location (such as a file). Redirection can be used for input as well as output, redirecting a file to a command for input. This section describes what you need to do to use redirection in your shell scripts.

Output Redirection

The most basic type of redirection is sending output from a command to a file. The bash shell uses the greater-than symbol (>) for this:

```
command > outputfile
```

Anything that would appear on the monitor from the command instead is stored in the output file specified:

```
$ date > test6
$ ls -l test6
-rw-r--r--    1 user     user           29 Feb 10 17:56 test6
$ cat test6
Thu Feb 10 17:56:58 EDT 2011
$
```

The redirect operator created the file `test6` (using the default `umask` settings) and redirected the output from the date command to the `test6` file. If the output file already exists, the redirect operator overwrites the existing file with the new file data:

```
$ who > test6
$ cat test6
user     pts/0    Feb 10 17:55
$
```

Now the contents of the `test6` file contain the output from the who command.

Sometimes, instead of overwriting the file's contents, you may need to append output from a command to an existing file, for example if you're creating a log file to document an action on the system. In this situation, you can use the double greater-than symbol (>>) to append data:

```
$ date >> test6
$ cat test6
user     pts/0    Feb 10 17:55
Thu Feb 10 18:02:14 EDT 2011
$
```

The `test6` file still contains the original data from the who command processed earlier—plus now it contains the new output from the `date` command.

Input Redirection

Input redirection is the opposite of output redirection. Instead of taking the output of a command and redirecting it to a file, input redirection takes the content of a file and redirects it to a command.

The input redirection symbol is the less-than symbol (<):

```
command < inputfile
```

The easy way to remember this is that the command is always listed first in the command line, and the redirection symbol "points" to the way the data is flowing. The less-than symbol indicates that the data is flowing from the input file to the command.

Here's an example of using input redirection with the wc command:

```
$ wc < test6
      2      11      60
$
```

The wc command provides a count of text in the data. By default, it produces three values:

- The number of lines in the text
- The number of words in the text
- The number of bytes in the text

By redirecting a text file to the wc command, you can get a quick count of the lines, words, and bytes in the file. The example shows that there are 2 lines, 11 words, and 60 bytes in the `test6` file.

There's another method of input redirection, called *inline input redirection*. This method allows you to specify the data for input redirection on the command line instead of in a file. This may seem somewhat odd at first, but there are a few applications for this process (such as those shown in the "Performing Math" section later).

The inline input redirection symbol is the double less-than symbol (<<). Besides this symbol, you must specify a text marker that delineates the beginning and end of the data used for input. You can use any string value for the text marker, but it must be the same at the beginning of the data and the end of the data:

```
command << marker
data
marker
```

When using inline input redirection on the command line, the shell will prompt for data using the secondary prompt, defined in the *PS2* environment variable (see Chapter 5). Here's how this looks when you use it:

```
$ wc << EOF
> test string 1
> test string 2
> test string 3
> EOF
   3       9      42
$
```

The secondary prompt continues to prompt for more data until you enter the string value for the text marker. Thewc command performs the line, word, and byte counts of the data supplied by the inline input redirection.

Pipes

There are times when you need to send the output of one command to the input of another command. This is possible using redirection, but somewhat clunky:

```
$ rpm -qa > rpm.list
$ sort < rpm.list
abrt-1.1.14-1.fc14.i686
abrt-addon-ccpp-1.1.14-1.fc14.i686
abrt-addon-kerneloops-1.1.14-1.fc14.i686
abrt-addon-python-1.1.14-1.fc14.i686
abrt-desktop-1.1.14-1.fc14.i686
abrt-gui-1.1.14-1.fc14.i686
abrt-libs-1.1.14-1.fc14.i686
abrt-plugin-bugzilla-1.1.14-1.fc14.i686
abrt-plugin-logger-1.1.14-1.fc14.i686
abrt-plugin-runapp-1.1.14-1.fc14.i686
acl-2.2.49-8.fc14.i686


...
```

The `rpm` command manages the software packages installed on systems using the Red Hat Package Management system (RPM), such as the Fedora system as shown. When used with the `-qa` parameters, it produces a list of the existing packages installed, but not necessarily in any specific order. If you're looking for a specific package, or group of packages, it can be difficult to find it using the output of the `rpm` command.

Using the standard output redirection, the output was redirected from the `rpm` command to a file, called `rpm.list`. After the command finished, the `rpm.list` file contained a list of all the installed software packages on my system. Next, input redirection was used to send the contents of the `rpm.list` file to the `sort` command to sort the package names alphabetically.

That was useful, but again, a somewhat clunky way of producing the information. Instead of redirecting the output of a command to a file, you can redirect the output to another command. This process is called piping.

Like the backtick (‘), the symbol for piping is not used often outside of shell scripting. The symbol is two vertical lines, one above the other. However, the `pipe` symbol often looks like a single vertical line in print (**|**). On a U.S. keyboard, it is usually on the same key as the backslash (\). The pipe is put between the commands to redirect the output from one to the other:

```
command1 | command2
```

Don't think of piping as running two commands back to back. The Linux system actually runs both commands at the same time, linking them together internally in the system. As the first command produces output, it's sent immediately to the second command. No intermediate files or buffer areas are used to transfer the data.

Now, using piping you can easily pipe the output of the rpm command directly to the sort command to produce your results:

```
$ rpm -qa | sort
abrt-1.1.14-1.fc14.i686
abrt-addon-ccpp-1.1.14-1.fc14.i686
abrt-addon-kerneloops-1.1.14-1.fc14.i686
abrt-addon-python-1.1.14-1.fc14.i686
abrt-desktop-1.1.14-1.fc14.i686
abrt-gui-1.1.14-1.fc14.i686
abrt-libs-1.1.14-1.fc14.i686
abrt-plugin-bugzilla-1.1.14-1.fc14.i686
abrt-plugin-logger-1.1.14-1.fc14.i686
abrt-plugin-runapp-1.1.14-1.fc14.i686
acl-2.2.49-8.fc14.i686


...
```

Unless you're a (very) quick reader, you probably couldn't keep up with the output generated by this command. Because the piping feature operates in real time, as soon as the rpm command produces data, the sort command gets busy sorting it. By the time the rpm command finishes outputting data, the sort command already has the data sorted and starts displaying it on the monitor.

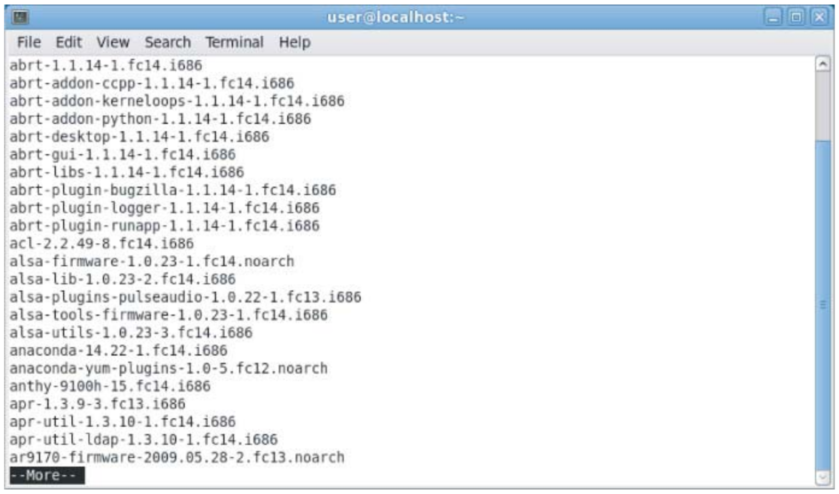
There's no limit to the number of pipes you can use in a command . You can continue piping the output of commands to other commands to refine your operation.

In this case, because the output of the sort command zooms by so quickly, you can use one of the text paging commands (such as less or more) to force the output to stop at every screen of data:

```
$ rpm -qa | sort | more
```

This command sequence runs the rpm command, pipes the output to the sort command, and then pipes that output to the more command to display the data, stopping after every screen of information. This now lets you pause and read what's on the display before continuing, as shown in Figure 10.1.

Figure 10.1 Using piping to send data to the more command



```
abrt-1.1.14-1.fc14.i686
abrt-addon-ccpp-1.1.14-1.fc14.i686
abrt-addon-kerneloops-1.1.14-1.fc14.i686
abrt-addon-python-1.1.14-1.fc14.i686
abrt-desktop-1.1.14-1.fc14.i686
abrt-gui-1.1.14-1.fc14.i686
abrt-libs-1.1.14-1.fc14.i686
abrt-plugin-bugzilla-1.1.14-1.fc14.i686
abrt-plugin-logger-1.1.14-1.fc14.i686
abrt-plugin-runapp-1.1.14-1.fc14.i686
acl-2.2.49-8.fc14.i686
alsa-firmware-1.0.23-1.fc14.noarch
alsa-lib-1.0.23-2.fc14.i686
alsa-plugins-pulseaudio-1.0.22-1.fc13.i686
alsa-tools-firmware-1.0.23-1.fc14.i686
alsa-utils-1.0.23-3.fc14.i686
anaconda-14.22-1.fc14.i686
anaconda-yum-plugins-1.0-5.fc12.noarch
anthy-9100h-15.fc14.i686
apr-1.3.9-3.fc13.i686
apr-util-1.3.10-1.fc14.i686
apr-util-ldap-1.3.10-1.fc14.i686
ar9170-firmware-2009.05.28-2.fc13.noarch
--More--
```

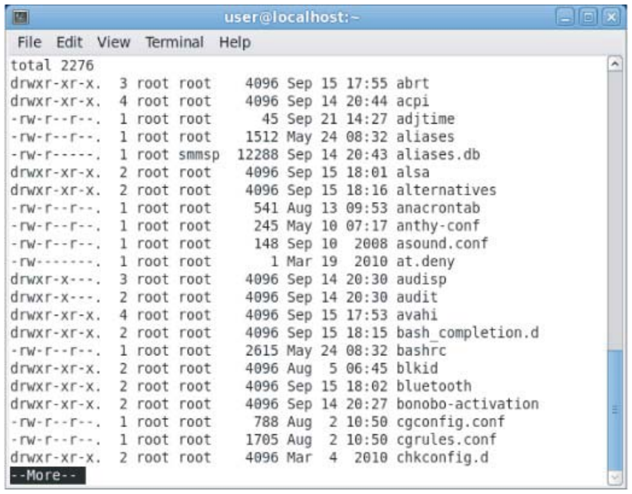
To get even fancier, you can use redirection along with piping to save your output to a file:

```
$ rpm -qa | sort > rpm.list
$ more rpm.list
abrt-1.1.14-1.fc14.i686
abrt-addon-ccpp-1.1.14-1.fc14.i686
abrt-addon-kerneloops-1.1.14-1.fc14.i686
abrt-addon-python-1.1.14-1.fc14.i686
abrt-desktop-1.1.14-1.fc14.i686
abrt-gui-1.1.14-1.fc14.i686
abrt-libs-1.1.14-1.fc14.i686
abrt-plugin-bugzilla-1.1.14-1.fc14.i686
abrt-plugin-logger-1.1.14-1.fc14.i686
abrt-plugin-runapp-1.1.14-1.fc14.i686
acl-2.2.49-8.fc14.i686
...
```

As expected, the data in the `rpm.list` file is now sorted!

By far one of the most popular uses of piping is piping the results of commands that produce long output to the `more` command. This is especially common with the `ls` command, as shown in Figure 10.2.

Figure 10.2 Using the more command with the ls command



```
total 2276
drwxr-xr-x.  3 root root    4096 Sep 15 17:55 abrt
drwxr-xr-x.  4 root root    4096 Sep 14 20:44 acpi
-rw-r--r--.  1 root root      45 Sep 21 14:27 adjtime
-rw-r--r--.  1 root root    1512 May 24 08:32 aliases
-rw-r-----.  1 root smmsp  12288 Sep 14 20:43 aliases.db
drwxr-xr-x.  2 root root    4096 Sep 15 18:01 alsa
drwxr-xr-x.  2 root root    4096 Sep 15 18:16 alternatives
-rw-r--r--.  1 root root     541 Aug 13 09:53 anacrontab
-rw-r--r--.  1 root root     245 May 10 07:17 anthy-conf
-rw-r--r--.  1 root root     148 Sep 10  2008 asound.conf
-rw-------.  1 root root       1 Mar 19  2010 at.deny
drwxr-x---.  3 root root    4096 Sep 14 20:30 audisp
drwxr-x---.  2 root root    4096 Sep 14 20:30 audit
drwxr-xr-x.  4 root root    4096 Sep 15 17:53 avahi
drwxr-xr-x.  2 root root    4096 Sep 15 18:15 bash_completion.d
-rw-r--r--.  1 root root    2615 May 24 08:32 bashrc
drwxr-xr-x.  2 root root    4096 Aug  5 06:45 blkid
drwxr-xr-x.  2 root root    4096 Sep 15 18:02 bluetooth
drwxr-xr-x.  2 root root    4096 Sep 14 20:27 bonobo-activation
-rw-r--r--.  1 root root     788 Aug  2 10:50 cgconfig.conf
-rw-r--r--.  1 root root    1705 Aug  2 10:50 cgrules.conf
drwxr-xr-x.  2 root root    4096 Mar  4  2010 chkconfig.d
--More--
```

The `ls -l` command produces a long listing of all the files in the directory. For directories with lots of files, this can be quite a listing. By piping the output to the `more` command, you force the output to stop at the end of every screen of data.

Performing Math

Another feature crucial to any programming language is the ability to manipulate numbers. Unfortunately, for shell scripts this process is a bit awkward. There a two different ways to perform mathematical operations in your shell scripts.

The expr Command

Originally, the Bourne shell provided a special command that was used for processing mathematical equations. The`expr` command allowed the processing of equations from the command line, but it is extremely clunky:

```
$ expr 1 + 5
6
```

The `expr` command recognizes a few different mathematical and string operators, shown in Table 10.1.

Table 10.1 The expr Command Operators

Operator	Description
`ARG1 | ARG2`	Return ARG1 if neither argument is null or zero; otherwise, return ARG2.
`ARG1 & ARG2`	Return ARG1 if neither argument is null or zero; otherwise, return 0.
`ARG1 < ARG2`	Return 1 if ARG1 is less than ARG2; otherwise, return 0.
`ARG1 <= ARG2`	Return 1 if ARG1 is less than or equal to ARG2; otherwise, return 0.
`ARG1 = ARG2`	Return 1 if ARG1 is equal to ARG2; otherwise, return 0.
`ARG1 != ARG2`	Return 1 if ARG1 is not equal to ARG2; otherwise, return 0.
`ARG1 >= ARG2`	Return 1 if ARG1 is greater than or equal to ARG2; otherwise, return 0.

`ARG1 > ARG2`	Return 1 if `ARG1` is greater than `ARG2`; otherwise, return 0.
`ARG1 + ARG2`	Return the arithmetic sum of `ARG1` and `ARG2`.
`ARG1 - ARG2`	Return the arithmetic difference of `ARG1` and `ARG2`.
`ARG1 * ARG2`	Return the arithmetic product of `ARG1` and `ARG2`.
`ARG1 / ARG2`	Return the arithmetic quotient of `ARG1` divided by `ARG2`.
`ARG1 % ARG2`	Return the arithmetic remainder of `ARG1` divided by `ARG2`.
`STRING : REGEXP`	Return the pattern match if `REGEXP` matches a pattern in `STRING`.
`match STRING REGEXP`	Return the pattern match if `REGEXP` matches a pattern in `STRING`.
`substr STRING POS LENGTH`	Return the substring `LENGTH` characters in length, starting at position `POS` (starting at 1).
`index STRING CHARS`	Return position in `STRING` where `CHARS` is found; otherwise, return 0.
`length STRING`	Return the numeric length of the string `STRING`.
`+ TOKEN`	Interpret `TOKEN` as a string, even if it's a keyword.
`(EXPRESSION)`	Return the value of `EXPRESSION`.

While the standard operators work fine in the `expr` command, the problem occurs when using them from a script or the command line. Many of the `expr` command operators have other meanings in the shell (such as the asterisk). Using them in the `expr` command produces odd results:

```
$ expr 5 * 2
expr: syntax error
$
```

To solve this problem, you need to use the shell escape character (the backslash) to identify any characters that may be misinterpreted by the shell before being passed to the `expr` command:

```
$ expr 5 \* 2
10
$
```

Now that's really starting to get ugly! Using the `expr` command in a shell script is equally cumbersome:

```
$ cat test6
#!/bin/bash
# An example of using the expr command
var1=10
var2=20
var3=‘expr $var2 / $var1’
echo The result is $var3
```

To assign the result of a mathematical equation to a variable, you have to use the backtick character to extract the output from the `expr` command:

```
$ chmod u+x test6
$ ./test6
The result is 2
$
```

Fortunately, the bash shell has an improvement for processing mathematical operators as you shall see in the next section.

Using Brackets

The bash shell includes the `expr` command to stay compatible with the Bourne shell; however, it also provides a much easier way of performing mathematical equations. In bash, when assigning a mathematical value to a variable, you can enclose the mathematical equation using a dollar sign and square brackets (`$[ operation ]`):

```
$ var1=$[1 + 5]
$ echo $var1
6
$ var2 = $[$var1 * 2]
$ echo $var2
12
$
```

Using brackets makes shell math much easier than with the `expr` command. This same technique also works in shell scripts:

```
$ cat test7
#!/bin/bash
var1=100
var2=50
var3=45
var4=$[$var1 * ($var2 - $var3)]
echo The final result is $var4
$
```

Running this script produces the output:

```
$ chmod u+x test7
$ ./test7
The final result is 500
$
```

Also, notice that when using the square brackets method for calculating equations you don't need to worry about the multiplication symbol, or any other characters, being misinterpreted by the shell. The shell knows that it's not a wildcard character because it is within the square brackets.

There's one major limitation to performing math in the bash shell script. Take a look at this example:

```
$ cat test8
#!/bin/bash
var1=100
var2=45
var3=$[$var1 / $var2]
echo The final result is $var3
$
```

Now run it and see what happens:

```
$ chmod u+x test8
$ ./test8
The final result is 2
$
```

The bash shell mathematical operators support only integer arithmetic. This is a huge limitation if you're trying to do any sort of real-world mathematical calculations.

Note

The z shell (zsh) provides full floating-point arithmetic operations. If you require floating-point calculations in your shell scripts, you might consider checking out the z shell (discussed in Chapter 22).

A Floating-Point Solution

There are several solutions for overcoming the bash integer limitation. The most popular solution uses the built-in bash calculator, called `bc`.

The Basics of bc

The bash calculator is actually a programming language that allows you to enter floating-point expressions at a command line and then interprets the expressions, calculates them, and returns the result. The bash calculator recognizes:

- Numbers (both integer and floating point)
- Variables (both simple variables and arrays)
- Comments (lines starting with a pound sign or the C language /* */ pair
- Expressions
- Programming statements (such as `if-then` statements)
- Functions

You can access the bash calculator from the shell prompt using the `bc` command:

```
$ bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, I
nc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type ‘warranty’.
12 * 5.4
64.8
3.156 * (3 + 5)
25.248
quit
$
```

The example starts out by entering the expression `12 * 5.4`. The bash calculator returns the answer. Each subsequent expression entered into the calculator is evaluated, and the result is displayed. To exit the bash calculator, you must enter `quit`.

The floating-point arithmetic is controlled by a built-in variable called *`scale`*. You must set this value to the desired number of decimal places you want in your answers or you won't get what you were looking for:

```
$ bc -q
3.44 / 5
0
scale=4
3.44 / 5
.6880
quit
$
```

The default value for the *`scale`* variable is zero. Before the *`scale`* value is set, the bash calculator provides the answer to zero decimal places. After you set the *`scale`* variable value to four, the bash calculator displays the answer to four decimal places. The `-q` command line parameter suppresses the lengthy welcome banner from the bash calculator.

In addition to normal numbers, the bash calculator also understands variables:

```
$ bc -q
var1=10
var1 * 4
40
var2 = var1 / 5
print var2
2
quit
$
```

Once a variable value is defined, you can use the variable throughout the bash calculator session. The `print`statement allows you to print variables and numbers.

Using bc in Scripts

Now you may be wondering how the bash calculator is going to help you with floating-point arithmetic in your shell scripts. Do you remember your friend the backtick character? Yes, you can use the backtick character to run a `bc`command and assign the output to a variable! The basic format to use is this:

```
variable=‘echo “options; expression” | bc’
```

The first portion, `options`, allows you to set variables. If you need to set more than one variable, separate them using the semicolon. The expression parameter defines the mathematical expression to evaluate using `bc`. Here's a quick example of doing this in a script:

```
$ cat test9
#!/bin/bash
var1=‘echo “ scale=4; 3.44 / 5” | bc’
echo The answer is $var1
$
```

This example sets the scale variable to four decimal places and then specifies a specific calculation for the expression. Running this script produces the following output:

```
$ chmod u+x test9
$ ./test9
The answer is .6880
$
```

Now that's fancy! You aren't limited to just using numbers for the expression value. You can also use variables defined in the shell script:

```
$ cat test10
#!/bin/bash
var1=100
var2=45
var3=‘echo “scale=4; $var1 / $var2” | bc’
echo The answer for this is $var3
$
```

The script defines two variables, which are used within the expression sent to the bc command. Remember to use the dollar sign to signify the value for the variables and not the variables themselves. The output of this script is as follows:

```
$ ./test10
The answer for this is 2.2222
$
```

And of course, once a value is assigned to a variable, that variable can be used in yet another calculation:

```
$ cat test11
#!/bin/bash
var1=20
var2=3.14159
var3=‘echo “scale=4; $var1 * $var1” | bc’
var4=‘echo “scale=4; $var3 * $var2” | bc’
echo The final result is $var4
$
```

This method works fine for short calculations, but sometimes you need to get more involved with your numbers. If you have more than just a couple of calculations, it gets confusing trying to list multiple expressions on the same command line.

There's a solution to this problem. The bc command recognizes input redirection, allowing you to redirect a file to the bc command for processing. However, this also can get confusing, as you'd need to store your expressions in a file.

The best method is to use inline input redirection, which allows you to redirect data directly from the command line. In the shell script, you assign the output to a variable:

```
variable=‘bc << EOF
options
statements
expressions
EOF
```

```
'
```

The EOF text string indicates the beginning and end of the inline redirection data. Remember that the backtick characters are still needed to assign the output of the bc command to the variable.

Now you can place all of the individual bash calculator elements on separate lines in the script file. Here's an example of using this technique in a script:

```
$ cat test12
#!/bin/bash

var1=10.46
var2=43.67
var3=33.2
var4=71

var5=‘bc << EOF
scale = 4
a1 = ( $var1 * $var2)
b1 = ($var3 * $var4)
a1 + b1
EOF
'

echo The final answer for this mess is $var5
$
```

Placing each option and expression on a separate line in your script makes things cleaner and easier to read and follow. The EOF string indicates the start and end of the data to redirect to the bc command. Of course, you need to use the backtick characters to indicate the command to assign to the variable.

You'll also notice in this example that you can assign variables within the bash calculator. It's important to remember that any variables created within the bash calculator are valid only within the bash calculator and can't be used in the shell script.

Exiting the Script

So far in our sample scripts, we terminated things pretty abruptly. When we were done with our last command, we just ended the script. There's a more elegant way of completing things available to us.

Every command that runs in the shell uses an *exit status* to indicate to the shell that it's done processing. The exit status is an integer value between 0 and 255 that's passed by the command to the shell when the command finishes running. You can capture this value and use it in your scripts.

Checking the exit Status

Linux provides the $? special variable that holds the exit status value from the last command that executed. You must view or use the $? variable immediately after the command you want to check. It changes values to the exit status of the last command executed by the shell:

```
$ date
Sat Jan 15 10:01:30 EDT 2011
$ echo $?
0
$
```

By convention, the exit status of a command that successfully completes is zero. If a command completes with an error, then a positive integer value is placed in the exit status:

```
$ asdfg
-bash: asdfg: command not found
$ echo $?
127
$
```

The invalid command returns an exit status of 127. There's not much of a standard convention to Linux error exit status codes. However, there are a few guidelines you can use, as shown in Table 10.2.

Table 10.2 Linux Exit Status Codes

Code	**Description**
0	Successful completion of the command
1	General unknown error
2	Misuse of shell command
126	The command can't execute
127	Command not found
128	Invalid exit argument
128+x	Fatal error with Linux signal x
130	Command terminated with Ctrl+C
255	Exit status out of range

An exit status value of 126 indicates that the user didn't have the proper permissions set to execute the command:

```
$ ./myprog.c
-bash: ./myprog.c: Permission denied
$ echo $?
126
$
```

Another common error you'll encounter occurs if you supply an invalid parameter to a command:

```
$ date %t
```

```
date: invalid date ‘%t’
$ echo $?
1
$
```

This generates the general exit status code of one, indicating an unknown error occurred in the command.

The exit Command

By default, your shell script will exit with the exit status of the last command in your script:

```
$ ./test6
The result is 2
$ echo $?
0
$
```

You can change that to return your own exit status code. The `exit` command allows you to specify an exit status when your script ends:

```
$ cat test13
#!/bin/bash
# testing the exit status
var1=10
var2=30
var3=$[ $var1 + var2 ]
echo The answer is $var3
exit 5
$
```

When you check the exit status of the script, you'll get the value used as the parameter of the `exit` command:

```
$ chmod u+x test13
$ ./test13
The answer is 40
$ echo $?
5
$
```

You can also use variables in the `exit` command parameter:

```
$ cat test14
#!/bin/bash
# testing the exit status
var1=10
var2=30
var3=$[ $var1 + var2 ]
exit $var3
$
```

When you run this command, it produces the following exit status:

```
$ chmod u+x test14
$ ./test14
$ echo $?
40
$
```

You should be careful with this feature, however, as the exit status codes can only go up to 255. Watch what happens in this example:

```
$ cat test14b
#!/bin/bash
# testing the exit status
var1=10
var2=30
var3=$[ $var1 * var2 ]
echo The value is $var3
exit $var3
$
```

Now when you run it, you get the following:

```
$ ./test14b
The value is 300
$ echo $?
44
$
```

The exit status code is reduced to fit in the 0 to 255 range. The shell does this by using modulo arithmetic. The*modulo* of a value is the remainder after a division. The resulting number is the remainder of the specified number divided by 256. In the case of 300 (the result value), the remainder is 44, which is what appears as the exit status code.

In Chapter 11, you'll see how you can use the `if-then` statement to check the error status returned by a command to see if the command was successful or not.

Summary

The bash shell script allows you to string commands together into a script. The most basic way to create a script is to separate multiple commands on the command line using a semicolon. The shell executes each command in order, displaying the output of each command on the monitor.

You can also create a shell script file, placing multiple commands in the file for the shell to execute in order. The shell script file must define the shell used to run the script. This is done in the first line of the script file, using the `#!` symbol, followed by the full path of the shell.

Within the shell script you can reference environment variable values by using a dollar sign in front of the variable. You can also define your own variables for use within the script, and assign values and even the output of a command by using the backtick character. The variable value can be used within the script by placing a dollar sign in front of the variable name.

The bash shell allows you to redirect both the input and output of a command from the standard behavior. You can redirect the output of any command from the monitor display to a file by using the greater-than symbol, followed by the name of the file to capture the output. You can append output data to an existing file by using two greater-than symbols. The less-than symbol is used to redirect input to a command. You can redirect input from a file to a command.

The Linux `pipe` command (the broken bar symbol) allows you to redirect the output of a command directly to the input of another command. The Linux system runs both commands at the same time, sending the output of the first command to the input of the second command without using any redirect files.

The bash shell provides a couple of ways for you to perform mathematical operations in your shell scripts. The`expr` command is a simple way to perform integer math. In the bash shell, you can also perform basic math calculations by enclosing equations in square brackets, preceded by a dollar sign. To perform floating-point arithmetic, you need to utilize the `bc` calculator command, redirecting input from inline data and storing the output in a user variable.

Finally, the chapter discussed how to use the exit status in your shell script. Every command that runs in the shell produces an exit status. The exit status is an integer value between 0 and 255 that indicates if the command completed successfully, and if not, what the reason may have been. An exit status of 0 indicates that the command completed successfully. You can use the `exit` command in your shell script to declare a specific exit status upon the completion of your script.

So far in your shell scripts, things have proceeded in an orderly fashion from one command to the next. In the next chapter, you'll see how you can use some logic flow control to alter which commands are executed within the script.

Chapter 11

Using Structured Commands

In This Chapter

- Working with the if-then statement
- The if-then-else statement
- Nesting ifs
- The test command
- Compound condition testing
- Advanced if-then features
- The code command
- Managing user accounts

In the shell scripts presented in Chapter 10, the shell processed each individual command in the shell script in the order it appeared. This works out fine for sequential operations, where you want all of the commands to process in the proper order. However, this isn't how all programs operate.

Many programs require some sort of logic flow control between the commands in the script. This means that the shell executes certain commands given one set of circumstances, but it has the ability to execute other commands given a different set of circumstances. There is a whole class of commands that allows the script to skip over or loop through commands based on conditions of variable values or the result of other commands. These commands are generally referred to as *structured commands*.

The structured commands allow you to alter the flow of operation of a program, executing some commands under some conditions while skipping others under other conditions. There are quite a few structured commands available in the bash shell, so we'll look at them individually. In this chapter, we look at the `if-then` statement.

Working with the if-then Statement

The most basic type of structured command is the `if-then` statement. The `if-then` statement has the following format:

```
if command
then
  commands
fi
```

If you're using `if-then` statements in other programming languages, this format may be somewhat confusing. In other programming languages, the object after the `if` statement is an equation that is evaluated for a TRUE or FALSEvalue. That's not how the bash shell `if` statement works.

The bash shell `if` statement runs the command defined on the `if` line. If the exit status of the command (see Chapter 10) is zero (the command completed successfully), the commands listed under the `then` section are executed. If the exit status of the command is anything else, the `then` commands aren't executed, and the bash shell moves on to the next command in the script.

Here's a simple example to demonstrate this concept:

```
$ cat test1
#!/bin/bash
# testing the if statement
if date
then
  echo “it worked”
fi
```

This script uses the `date` command on the `if` line. If the command completes successfully, the `echo` statement should display the text string. When you run this script from the command line, you'll get the following results:

```
$ ./test1
Sat Jan 23 14:09:24 EDT 2011
it worked
$
```

The shell executed the `date` command listed on the `if` line. Since the exit status was zero, it also executed the`echo` statement listed in the `then` section.

Here's another example:

```
$ cat test2
#!/bin/bash
# testing a bad command
if asdfg
then
  echo “it did not work”
fi
echo “we are outside of the if statement”
$
$ ./test2
./test2: line 3: asdfg: command not found
we are outside of the if statement
$
```

In this example, a command was deliberately used that will not work in the `if` statement line. Because this is a bad command, it will produce an exit status that's non-zero, and the bash shell skips the `echo` statement in the`then` section. Also notice that the error message generated from running the command in the `if` statement still appears in the output of the script. There will be times when you won't want this to happen. Chapter 14 discusses how this can be avoided.

You are not limited to just one command in the `then` section. You can list commands just as in the rest of the shell script. The bash shell treats the commands as a block, executing all of them when the command in the `if`statement line returns a zero exit status or skipping all of them when the command returns a non-zero exit status:

```
$ cat test3
#!/bin/bash
# testing multiple commands in the then section
```

```
testuser=rich
if grep $testuser /etc/passwd
then
  echo The bash files for user $testuser are:
  ls -a /home/$testuser/.b*
fi
```

The `if` statement line uses the `grep` comment to search the `/etc/passwd` file to see if a specific username is currently used on the system. If there's a user with that logon name, the script displays some text, and then lists the bash files in the user's *HOME* directory:

```
$ ./test3
rich:x:500:500:Rich Blum:/home/rich:/bin/bash
The files for user rich are:
/home/rich/.bash_history  /home/rich/.bash_profile
/home/rich/.bash_logout   /home/rich/.bashrc
$
```

However, if you set the *`testuser`* variable to a user that doesn't exist on the system, nothing happens:

```
$ ./test3
$
```

That's not all that exciting. It would be nice if we could display a little message saying that the username wasn't found on the system. Well, we can, using another feature of the `if-then` statement.

Note

You might see an alternative form of the `if-then` statement used in some scripts:

```
if command; then
  commands
fi
```

By putting a semicolon at the end of the command to evaluate, you can include the `then` statement on the same line, which looks more like how `if-then` statements are handled in some other programming languages.

The if-then-else Statement

In the `if-then` statement, you have only one option of whether or not a command is successful. If the command returns a non-zero exit status code, the bash shell just moves on to the next command in the script. In this situation, it would be nice to be able to execute an alternate set of commands. That's exactly what the `if-then-else` statement is for.

The `if-then-else` statement provides another group of commands in the statement:

```
if command
```

```
then
  commands
else
  commands
fi
```

When the command in the `if` statement line returns with an exit status code of zero, the commands listed in the`then` section are executed, just as in a normal `if-then` statement. When the command in the `if` statement line returns a non-zero exit status code, the bash shell executes the commands in the `else` section.

Now you can modify the test script to look like this:

```
$ cat test4
#!/bin/bash
# testing the else section
testuser=badtest
if grep $testuser /etc/passwd
then
  echo The files for user $testuser are:
  ls -a /home/$testuser/.b*
else
  echo “The user name $testuser does not exist on this system”
fi
$
$ ./test4
The user name badtest does not exist on this system
$
```

That's more user-friendly. Just like the `then` section, the `else` section can contain multiple commands. The `fi`statement delineates the end of the `else` section.

Nesting ifs

Sometimes you must check for several situations in your script code. Instead of having to write separate `if-then`statements, you can use an alternative version of the `else` section, called `elif`.

The `elif` continues an else section with another `if-then` statement:

```
if command1
then
  commands
elif command2
then
   more commands
fi
```

The `elif` statement line provides another command to evaluate, similar to the original `if` statement line. If the exit status code from the `elif` command is zero, bash executes the commands in the second `then` statement section.

You can continue to string `elif` statements together, creating one huge `if-then-elif` conglomeration:

```
if command1
then
    command set 1
elif command2
then
   command set 2
elif command3
then
   command set 3
elif command4
then
   command set 4
fi
```

Each block of commands is executed depending on which command returns the zero exit status code. Remember, the bash shell will execute the `if` statements in order, and only the first one that returns a zero exit status will result in the `then` section being executed. Later on in “The case Command” section, you'll see how to use the `case`command instead of having to nest lots of `if-then` statements.

The test Command

So far, all you've seen in the `if` statement line are normal shell commands. You might be wondering if the bash `if-then` statement has the ability to evaluate any condition other than the exit status code of a command.

The answer is no, it can't. However, there's a neat utility available in the bash shell that helps you evaluate other things, using the `if-then` statement.

The `test` command provides a way to test different conditions in an `if-then` statement. If the condition listed in the `test` command evaluates to true, the `test` command exits with a zero exit status code, making the `if-then`statement behave in much the same way that `if-then` statements work in other programming languages. If the condition is false, the `test` command exits with a`1`, which causes the `if-then` statement to fail.

The format of the test command is pretty simple:

```
test condition
```

The *condition* is a series of parameters and values that the `test` command evaluates. When used in an `if-then`statement, the `test` command looks like this:

```
if test condition
```

```
then
  commands
fi
```

The bash shell provides an alternative way of declaring the `test` command in an `if-then` statement:

```
if [ condition ]
then
  commands
fi
```

The square brackets define the condition that's used in the `test` command. Be careful; you *must* have a space after the first bracket and a space before the last bracket or you'll get an error message.

The `test` command can evaluate three classes of conditions:

- Numeric comparisons
- String comparisons
- File comparisons

The next sections describe how to use each of these classes of tests in your `if-then` statements.

Numeric Comparisons

The most common method for using the `test` command is to perform a comparison of two numeric values. Table 11.1shows the list of condition parameters used for testing two values.

Table 11.1 The test Numeric Comparisons

Comparison	Description
`n1 -eq n2`	Check if `n1` is equal to `n2`.
`n1 -ge n2`	Check if `n1` is greater than or equal to `n2`.
`n1 -gt n2`	Check if `n1` is greater than `n2`.
`n1 -le n2`	Check if `n1` is less than or equal to `n2`.
`n1 -lt n2`	Check if `n1` is less than `n2`.
`n1 -ne n2`	Check if `n1` is not equal to `n2`.

The numeric test conditions can be used to evaluate both numbers and variables. Here's an example of doing that:

```
$ cat test5
#!/bin/bash
# using numeric test comparisons
val1=10
val2=11


if [ $val1 -gt 5 ]
```

```
then
  echo “The test value $val1 is greater than 5”
fi


if [ $val1 -eq $val2 ]
then
  echo “The values are equal”
else
  echo “The values are different”
fi
```

The first test condition:

```
if [ $val1 -gt 5 ]
```

tests if the value of the variable *val1* is greater than 5. The second test condition:

```
if [ $val1 -eq $val2 ]
```

tests if the value of the variable *val1* is equal to the value of the variable *val2*. Run the script and watch the results:

```
$ ./test5
The test value 10 is greater than 5
The values are different
$
```

Both of the numeric test conditions evaluated as expected.

There is a limitation to the test numeric conditions, however. Try this script:

```
$ cat test6
#!/bin/bash
# testing floating point numbers
val1=’ echo “scale=4; 10 / 3 ” | bc‘
echo “The test value is $val1”
if [ $val1 -gt 3 ]
then
  echo “The result is larger than 3”
fi
$
$ ./test6
The test value is 3.3333
./test6: line 5: [: 3.3333: integer expression expected
$
```

This example uses the bash calculator to produce a floating-point value, stored in the *val1* variable. Next, it uses the `test` command to evaluate the value. Something obviously went wrong here.

In Chapter 10, you learned how to trick the bash shell into handling floating-point values; there's still a problem in this script. The `test` command wasn't able to handle the floating-point value that was stored in the *val1*variable.

Remember that the only numbers the bash shell can handle are integers. When you utilize the bash calculator, you just fool the shell into storing a floating-point value in a variable as a string value. This works perfectly fine if all you need to do is display the result, using an `echo` statement, but this doesn't work in numeric-oriented functions, such as our numeric test condition. The bottom line is that you're not able to use floating-point values in the `test` command.

String Comparisons

The `test` command also allows you to perform comparisons on string values. Performing comparisons on strings can get tricky, as you'll see. Table 11.2 shows the comparison functions you can use to evaluate two string values.

Table 11.2 The test Command String Comparisons

Comparison	Description
`str1 = str2`	Check if `str1` is the same as string `str2`.
`str1 != str2`	Check if `str1` is not the same as `str2`.
`str1 < str2`	Check if `str1` is less than `str2`.
`str1 > str2`	Check if `str1` is greater than `str2`.
`-n str1`	Check if `str1` has a length greater than zero.
`-z str1`	Check if `str1` has a length of zero.

The following sections describe the different string comparisons available.

String Equality

The equal and not equal conditions are fairly self-explanatory with strings. It's pretty easy to know when two string values are the same or not:

```
$cat test7
#!/bin/bash
# testing string equality
testuser=rich


if [ $USER = $testuser ]
then
  echo “Welcome $testuser”
fi
$
$ ./test7
Welcome rich
$
```

Also, using the not equals string comparison, will allow you to determine if two strings have the same value or not:

```
$ cat test8
#!/bin/bash
```

```
# testing string equality
testuser=baduser


if [ $USER != $testuser ]
then
  echo “This is not $testuser”
else
  echo “Welcome $testuser”
fi
$
$ ./test8
This is not baduser
$
```

The `test` comparison takes all punctuation and capitalization into account when comparing strings for equality.

String Order

Trying to determine if one string is less than or greater than another is where things start getting tricky. There are two problems that often plague shell programmers when trying to use the greater-than or less-than features of the `test` command:

- The greater-than and less-than symbols must be escaped, or the shell will use them as redirection symbols, with the string values as filenames.
- The greater-than and less-than order is not the same as that used with the sort command.

The first item can result in a huge problem that often goes undetected when programming your scripts. Here's a typical example of what sometimes happens to novice shell script programmers:

```
$ cat badtest
#!/bin/bash
# mis-using string comparisons


val1=baseball
val2=hockey


if [ $val1 > $val2 ]
then
  echo “$val1 is greater than $val2”
else
  echo “$val1 is less than $val2”
fi
$
```

```
$ ./badtest
baseball is greater than hockey
$ ls -l hockey
-rw-r--r--    1 rich     rich            0 Sep 30 19:08 hockey
$
```

By just using the greater-than symbol itself in the script, no errors are generated, but the results are wrong. The script interpreted the greater-than symbol as an output redirection. Thus, it created a file called hockey. Because the redirection completed successfully, the `test` command returns a zero exit status code, which the `if`statement evaluates as though things completed successfully!

To fix this problem, you need to properly escape the greater-than symbol:

```
$ cat test9
#!/bin/bash
# mis-using string comparisons


val1=baseball
val2=hockey


if [ $val1 \> $val2 ]
then
 echo “$val1 is greater than $val2”
else
  echo “$val1 is less than $val2”
fi
$
$ ./test9
baseball is less than hockey
$
```

Now that answer is more along the lines of what you would expect from the string comparison.

The second issue is a little more subtle, and you may not even run across it unless you are working with uppercase and lowercase letters. The `sort` command handles uppercase letters opposite to the way the `test` command considers them. Let's test this feature in a script:

```
$ cat test9b
#!/bin/bash
# testing string sort order
val1=Testing
val2=testing


if [ $val1 \> $val2 ]
then
  echo “$val1 is greater than $val2”
```

```
else
  echo “$val1 is less than $val2”
fi
$
$ ./test9b
Testing is less than testing
$ sort testfile
testing
Testing
$
```

Capitalized letters are treated as less than lowercase letters in the `test` command. However, when you put the same strings in a file and use the `sort` command, the lowercase letters appear first. This is due to the ordering technique each command uses. The `test` command uses standard ASCII ordering, using each character's ASCII numeric value to determine the sort order. The `sort` command uses the sorting order defined for the system locale language settings. For the English language, the locale settings specify that lowercase letters appear before uppercase letters in sorted order.

Caution

Notice that the `test` command uses the standard mathematical comparison symbols for string comparisons and text codes for numerical comparisons. This is a subtle feature that many programmers manage to get reversed. If you use the mathematical comparison symbols for numeric values, the shell interprets them as string values and may not produce the correct results.

String Size

The `-n` and `-z` comparisons are handy when trying to evaluate if a variable contains data or not:

```
$ cat test10
#!/bin/bash
# testing string length
val1=testing
val2=‘’

if [ -n $val1 ]
then
  echo “The string ‘$val1’ is not empty”
else
  echo “The string ‘$val1’ is empty”
fi

if [ -z $val2 ]
then
  echo “The string ‘$val2’ is empty”
```

```
else
  echo “The string ‘$val2’ is not empty”
fi


if [ -z $val3 ]
then
  echo “The string ‘$val3’ is empty”
else
  echo “The string ‘$val3’ is not empty”
fi
$
$ ./test10
The string ‘testing’ is not empty
The string ‘’ is empty
The string ‘’ is empty
$
```

This example creates two string variables. The `val1` variable contains a string, and the `val2` variable is created as an empty string. The following comparisons are made as shown below:

```
if [ -n $val1 ]
```

determines if the *val1* variable is non-zero in length, which it is, so the `then` section is processed:

```
if [ -z $var2 ]
```

determines if the *val2* variable is zero in length, which it is, so the `then` section is processed:

```
if [ -z $val3 ]
```

determines if the *val3* variable is zero in length. This variable was never defined in the shell script, so it indicates that the string length is still zero, even though it wasn't defined.

Caution

Empty and uninitialized variables can have catastrophic effects on your shell script tests. If you're not sure of the contents of a variable, it's always best to test if the variable contains a value using `-n` or `-z` before using it in a numeric or string comparison.

File Comparisons

The last category of test comparisons is quite possibly the most powerful and most used comparisons in shell scripting. The `test` command allows you to test the status of files and directories on the Linux filesystem. Table 11.3 lists these comparisons.

Table 11.3 The test Command File Comparisons

Comparison	**Description**
`-d file`	Check if `file` exists and is a directory.

`-e file`	Checks if `file` exists.
`-f file`	Checks if `file` exists and is a file.
`-r file`	Checks if `file` exists and is readable.
`-s file`	Checks if `file` exists and is not empty.
`-w file`	Checks if `file` exists and is writable.
`-x file`	Checks if `file` exists and is executable.
`-O file`	Checks if `file` exists and is owned by the current user.
`-G file`	Checks if `file` exists and the default group is the same as the current user.
`file1 -nt file2`	Checks if `file1` is newer than `file2`.
`file1 -ot file2`	Checks if `file1` is older than `file2`.

These conditions give you the ability to check files in your filesystem within your shell scripts, and they are often used in scripts that access files. Because they're used so much, let's look at each of these individually.

Checking Directories

The `-d` test checks if a specified filename exists as a directory on the system. This is usually a good thing to do if you're trying to write a file to a directory, or before you try to change to a directory location:

```
$ cat test11
#!/bin/bash
# look before you leap
if [ -d $HOME ]
then
  echo “Your HOME directory exists”
  cd $HOME
  ls -a
else
  echo “There is a problem with your HOME directory”
fi
$
$ ./test11
“Your HOME directory exists”
.      Documents      .gvfs             .pulse-cookie
..     Downloads      .ICEauthority     .recently-used.xbel
.aptitude             .esd_auth         .local
.sudo_as_admin_successful
.bash_history         examples          .desktop   .mozilla      Templates
.bash_logout          .fontconfig       Music      test11
.bashrc               .gconf            .nautilus  Videos
.cache                .gconfd           .openoffice.org
.xsession-errors
```

```
.config              .gksu.lock        Pictures
.xsession-errors.old
.dbus                .gnome2           .profile
Desktop              .gnome2_private Public
.dmrc                .gtk-bookmarks    .pulse
$
```

The sample code uses the -d test condition to see if the *$HOME* directory exists for the user. If it does, it proceeds to use the cd command to change to the *$HOME* directory and performs a directory listing.

Checking If an Object Exists

The -e comparison allows you to check if a file or directory object exists before you attempt to use it in your script:

```
$ cat test12
#!/bin/bash
# checking if a directory exists
if [ -e $HOME ]
then
  echo “OK on the directory, now to check the file”
  # checking if a file exists
  if [ -e $HOME/testing ]
  then
    # the file exists, append data to it
    echo “Appending date to existing file”
    date >> $HOME/testing
  else
    # the file does not exist, create a new file
    echo “Creating new file”
    date > $HOME/testing
  fi
else
  echo “Sorry, you do not have a HOME directory”
fi
$
$ ./test12
OK on the directory, now to check the file
Creating new file
$ ./test12
OK on the directory, now to check the file
```

```
Appending date to existing file
$
```

The first check uses the `-e` comparison to determine if the user has a *$HOME* directory. If so, the next `-e`comparison checks to determine if the testing file exists in the *$HOME* directory. If the file doesn't exist, the shell script uses the single greater-than redirect symbol, creating a new file with the output from the `date`command. The second time you run the shell script, it uses the double greater-than symbol, so it just appends the date output to the existing file.

Checking for a File

The `-e` comparison works for both files and directories. To be sure that the object specified is a file, you must use the `-f` comparison:

```
$ cat test13
#!/bin/bash
# check if a file
if [ -e $HOME ]
then
  echo “The object exists, is it a file?”
  if [ -f $HOME ]
  then
    echo “Yes, it is a file!”
  else
    echo “No, it is not a file!”
    if [ -f $HOME/.bash_history ]
    then
      echo “But this is a file!”
    fi
  fi
else
  echo “Sorry, the object does not exist”
fi
$
$ ./test13
The object exists, is it a file?
No, it is not a file!
But this is a file!
$
```

This little script does a whole lot of checking! First, it uses the `-e` comparison to test if *$HOME* exists. If it does, it uses `-f` to test if it's a file. If it isn't a file (which of course it isn't), we use the `-f` comparison to test if it's a file, which it is.

Can You Read It?

Before trying to read data from a file, it's usually a good idea to test if you can read from the file first. You do this with the `-r` comparison:

```
$ cat test14
#!/bin/bash
# testing if you can read a file
pwfile=/etc/shadow


# first, test if the file exists, and is a file
if [ -f $pwfile ]
then
  # now test if you can read it
  if [ -r $pwfile ]
  then
    tail $pwfile
  else
    echo “Sorry, I am unable to read the $pwfile file”
  fi
else
  echo “Sorry, the file $file does not exist”
fi
$
$ ./test14
Sorry, I am unable to read the /etc/shadow file
$
```

The `/etc/shadow` file contains the encrypted passwords for system users, so it's not readable by normal users on the system. The `-r` comparison determined that I didn't have read access to the file, so the test command failed and the bash shell executed the else section of the `if-then` statement.

Checking for Empty Files

You should use `-s` comparison to check if a file is empty, especially if you're trying to remove a file. Be careful because when the `-s` comparison succeeds, it indicates that a file has data in it:

```
$ cat test15
#!/bin/bash
# testing if a file is empty
file=t15test
touch $file
```

```
if [ -s $file ]
then
  echo “The $file file exists and has data in it”
else
  echo “The $file exists and is empty”
fi
date > $file
if [ -s $file ]
then
  echo “The $file file has data in it”
else
  echo “The $file is still empty”
fi
$
$./test15
The t15test exists and is empty
The t15test file has data in it
$
```

The `touch` command creates the file but doesn't put any data in it. After we use the `date` command and redirect the output to the file, the `-s` comparison indicates that there's data in the file.

Checking If You Can Write to a File

The `-w` comparison determines if you have permission to write to a file:

```
$ cat test16
#!/bin/bash
# checking if a file is writeable


logfile=$HOME/t16test
touch $logfile
chmod u-w $logfile
now=‘date +%Y%m%d-%H%M’


if [ -w $logfile ]
then
  echo “The program ran at: $now” > $logfile
  echo “The first attempt succeeded”
else
  echo “The first attempt failed”
fi
```

```
chmod u+w $logfile
if [ -w $logfile ]
then
  echo “The program ran at: $now” > $logfile
  echo “The second attempt succeeded”
else
  echo “The second attempt failed”
fi
$
$ ./test16
The first attempt failed
The second attempt succeeded
$ cat $HOME/t16test
The program ran at: 20110124-1602
$
```

This is a pretty busy shell script! First, it defines a log file in your *$HOME* directory, stores the filename of it in the variable *logfile*, creates the file, and then removes the write permission for the user, using the chmodcommand. Next, it creates the variable *now* and stores a timestamp, using the date command. After all of that, it checks if you have write permission to the new log file (which you just took away). Because you don't have write permission, you should see the failed message appear.

After that, the script uses the chmod command again to grant the write permission back for the user, and tries to write to the file again. This time the write is successful.

Checking If You Can Run a File

The -x comparison is a handy way to determine if you have execute permission for a specific file. While this may not be needed for most commands, if you run a lot of scripts from your shell scripts, it could come in handy:

```
$ cat test17
#!/bin/bash
# testing file execution
if [ -x test16 ]
then
  echo “You can run the script: ”
  ./test16
else
  echo “Sorry, you are unable to execute the script”
fi
$
$ ./test17
You can run the script:
The first attempt failed
The second attempt succeeded
$
$ chmod u-x test16
$
$ ./test17
Sorry, you are unable to execute the script
$
```

This example shell script uses the -x comparison to test if you have permission to execute the `test16` script. If so, it runs the script (notice that even in a shell script, you must have the proper path to execute a script that's not located in your *$PATH*). After successfully running the `test16` script the first time, change the permissions on it and try again. This time, the -x comparison fails, as you don't have execute permission for the`test16` script.

Checking Ownership

The `-O` comparison allows you to easily test if you're the owner of a file:

```
$ cat test18
#!/bin/bash
# check file ownership


if [ -O /etc/passwd ]
then
  echo “You are the owner of the /etc/passwd file”
else
  echo “Sorry, you are not the owner of the /etc/passwd file”
fi
$
$ ./test18
Sorry, you are not the owner of the /etc/passwd file
$
$ su
Password:
$
# ./test18
You are the owner of the /etc/passwd file
#
```

The script uses the `-O` comparison to test if the user running the script is the owner of the `/etc/passwd` file. The first time the script is run under a normal user account, so the test fails. The second time, we used the `su`command to become the root user, and the test succeeded.

Checking Default Group Membership

The -G comparison checks the default group of a file, and it succeeds if it matches the group of the default group for the user. This can be somewhat confusing because the -G comparison checks the default groups only and not all the groups the user belongs to. Here's an example of this:

```
$ cat test19
#!/bin/bash# check file group test


if [ -G $HOME/testing ]
then
  echo “You are in the same group as the file”
else
  echo “The file is not owned by your group”
fi
$
$ ls -l $HOME/testing
-rw-rw-r-- 1 rich rich 58 2011-01-30 15:51 /home/rich/testing
$
$ ./test19
You are in the same group as the file
$
$ chgrp sharing $HOME/testing
$
$ ./test19
The file is not owned by your group
$
```

The first time the script is run, the *$HOME/testing* file is in the `rich` group, and the -G comparison succeeds. Next, the group is changed to the `sharing` group, which the user is also a member of. However, the -G comparison failed, since it only compares the default groups, not any additional group memberships.

Checking File Date

The last set of comparisons deal with comparing the creation times of two files. This comes in handy when writing scripts to install software. Sometimes you don't want to install a file that is older than a file already installed on the system.

The `-nt` comparison determines if a file is newer than another file. If a file is newer, it will have a more recent file creation time. The `-ot` comparison determines if a file is older than another file. If the file is older, it will have an older file creation time:

```
$ cat test20
#!/bin/bash
# testing file dates


if [ ./test19 -nt ./test18 ]
then
```

```
  echo “The test19 file is newer than test18”
else
  echo “The test18 file is newer than test19”
fi
if [ ./test17 -ot ./test19 ]
then
 echo “The test17 file is older than the test19 file”
fi
$
$ ./test20
The test19 file is newer than test18
The test17 file is older than the test19 file
$
$ ls -l test17 test18 test19
-rwxrw-r-- 1 rich rich 167 2011-01-30 16:31 test17
-rwxrw-r-- 1 rich rich 185 2011-01-30 17:46 test18
-rwxrw-r-- 1 rich rich 167 2011-01-30 17:50 test19
$
```

The filepaths used in the comparisons are relative to the directory from where you run the script. This can cause problems if the files you're checking can be moved around. Another problem is that neither of these comparisons check if the file exists first. Try this test:

```
$ cat test21
#!/bin/bash
# testing file dates


if [ ./badfile1 -nt ./badfile2 ]
then
  echo “The badfile1 file is newer than badfile2”
else
  echo “The badfile2 file is newer than badfile1”
fi
$
$ ./test21
The badfile2 file is newer than badfile1
$
```

This little example demonstrates that if the files don't exist, the `-nt` comparison just returns a failed condition. It's imperative that you ensure the files exist before trying to use them in the `-nt` or `-ot` comparison.

Compound Condition Testing

The `if-then` statement allows you to use Boolean logic to combine tests. There are two Boolean operators you can use:

- `[ condition1 ] && [ condition2 ]`
- `[ condition1 ] || [ condition2 ]`

The first Boolean operation uses the AND Boolean operator to combine two conditions. Both conditions must be met for the `then` section to execute.

The second Boolean operation uses the OR Boolean operator to combine two conditions. If either condition evaluates to a true condition, the `then` section is executed.

```
$ cat test22
#!/bin/bash
# testing compound comparisons


if [ -d $HOME ] && [ -w $HOME/testing ]
then
  echo “The file exists and you can write to it”
else
  echo “I cannot write to the file”
fi
$
$ ./test22
I cannot write to the file
$ touch $HOME/testing
$
$ ./test22
The file exists and you can write to it
$
```

Using the AND Boolean operator, both of the comparisons must be met. The first comparison checks to see if the*$HOME* directory exists for the user. The second comparison checks to see if there's a file called `testing` in the user's *$HOME* directory, and if the user has write permissions for the file. If either of these comparisons fails, the `if` statement fails and the shell executes the `else` section. If both of the comparisons succeed, the `if`statement succeeds and the shell executes the `then` section.

Advanced if-then Features

There are two relatively recent additions to the bash shell that provide advanced features that you can use in `if-then` statements:

- Double parentheses for mathematical expressions
- Double square brackets for advanced string handling functions

The following sections describe each of these features in more detail.

Using Double Parentheses

The *double parentheses* command allows you to incorporate advanced mathematical formulas in your comparisons. The`test` command only allows for simple arithmetic operations in the comparison. The double parentheses command provides more mathematical symbols, which programmers, who have used other programming languages, may be used to using. The format of the double parentheses command is:

```
(( expression ))
```

The *expression* term can be any mathematical assignment or comparison expression. Besides the standard mathematical operators that the `test` command uses, Table 11.4 shows the list of additional operators available for use in the double parentheses command.

Table 11.4 The Double Parentheses Command Symbols

Symbol	Description
`val++`	Post-increment
`val--`	Post-decrement
`++val`	Pre-increment
`--val`	Pre-decrement
`!`	Logical negation
`~`	Bitwise negation
`**`	Exponentiation
`<<`	Left bitwise shift
`>>`	Right bitwise shift
`&`	Bitwise Boolean AND
`|`	Bitwise Boolean OR
`&&`	Logical AND
`||`	Logical OR

You can use the double parentheses command in an `if` statement, as well as a normal command in the script for assigning values:

```
$ cat test23
#!/bin/bash
# using double parenthesis

val1=10

if (( $val1 ** 2 > 90 ))
then
  (( val2 = $val1 ** 2 ))
  echo “The square of $val1 is $val2”
fi
```

```
$
$ ./test23
The square of 10 is 100
$
```

Notice that you don't need to escape the greater-than symbol in the expression within the double parentheses. This is yet another advanced feature provided by the double parentheses command.

Using Double Brackets

The *double bracket* command provides advanced features for string comparisons. The double bracket command format is:

```
[[ expression ]]
```

The double bracketed *expression* uses the standard string comparison used in the `test` command. However, it provides an additional feature that the `test` command doesn't, *pattern matching*.

In pattern matching, you can define a regular expression (discussed in detail in Chapter 19) that's matched against the string value:

```
$ cat test24
#!/bin/bash
# using pattern matching


if [[ $USER == r* ]]
then
  echo “Hello $USER”
else
  echo “Sorry, I do not know you”
fi
$
$ ./test24
Hello rich
$
```

The double bracket command matches the *$USER* environment variable to see if it starts with the letter r. If so, the comparison succeeds, and the shell executes the `then` section commands.

The case Command

Often you'll find yourself trying to evaluate the value of a variable, looking for a specific value within a set of possible values. In this scenario, you end up having to write a lengthy `if-then-else` statement, like this:

```
$ cat test25
#!/bin/bash
```

```
# looking for a possible value

if [ $USER = “rich” ]
then
  echo “Welcome $USER”
  echo “Please enjoy your visit”
elif [ $USER = barbara ]
then
  echo “Welcome $USER”
  echo “Please enjoy your visit”
elif [ $USER = testing ]
then
  echo “Special testing account”
elif [ $USER = jessica ]
then
  echo “Do not forget to logout when you're done”
else
  echo “Sorry, you are not allowed here”
fi
$
$ ./test25
Welcome rich
Please enjoy your visit
$
```

The `elif` statements continue the `if-then` checking, looking for a specific value for the single comparison variable.

Instead of having to write all of the `elif` statements to continue checking the same variable value, you can use the `case` command. The `case` command checks multiple values of a single variable in a list-oriented format:

```
case variable in
pattern1 | pattern2) commands1;;
pattern3) commands2;;
*) default commands;;
esac
```

The `case` command compares the variable specified against the different patterns. If the variable matches the pattern, the shell executes the commands specified for the pattern. You can list more than one pattern on a line, using the bar operator to separate each pattern. The asterisk symbol is the catch-all for values that don't match any of the listed patterns. Here's an example of converting the `if-then-else` program to using the `case` command:

```
$ cat test26
#!/bin/bash
```

```
# using the case command


case $USER in
rich | barbara)
  echo “Welcome, $USER”
  echo “Please enjoy your visit”;;
testing)
 echo “Special testing account”;;
jessica)
  echo “Do not forget to log off when you're done”;;
*)
  echo “Sorry, you are not allowed here”;;
esac
$
$ ./test26
Welcome, rich
Please enjoy your visit
$
```

The `case` command provides a much cleaner way of specifying the various options for each possible variable value.

Summary

Structured commands allow you to alter the normal flow of execution on the shell script. The most basic structured command is the `if-then` statement. This statement allows you to evaluate a command, and perform other commands based on the outcome of the command you evaluated.

You can expand the `if-then` statement to include a set of commands the bash shell executes if the specified command fails as well. The `if-then-else` statement allows you to execute commands only if the command being evaluated returns a non-zero exit status code.

You can also link `if-then-else` statements together, using the `elif` statement. The `elif` is equivalent to using an`else if` statement, providing for additional checking if the original command that was evaluated failed.

In most scripts, instead of evaluating a command, you'll want to evaluate a condition, such as a numeric value, the contents of a string, or the status of a file or directory. The `test` command provides an easy way for you to evaluate all of these conditions. If the condition evaluates to a true condition, the `test` command produces a zero exit status code for the `if-then` statement. If the condition evaluates to a false condition, the `test` command produces a non-zero exit status code for the `if-then` statement.

The square bracket is a special bash command that is a synonym for the `test` command. You can enclose a test condition in square brackets in the `if-then` statement to test for numeric, string, and file conditions.

The double parentheses command allows you to perform advanced mathematical evaluations using additional operators, and the double square bracket command allows you to perform advanced string pattern-matching evaluations.

Finally, the chapter discussed the `case` command, which is a shorthand way of performing multiple `if-then-else`commands, checking the value of a single variable against a list of values.

The next chapter continues the discussion of structured commands by examining the shell looping commands. The `for`and `while` commands allow you to create loops that iterate through commands for a given period of time.

Chapter 12

More Structured Commands

In This Chapter

- Looping with the for statement
- Iterating with the until statement
- Using the while statement
- Combining loops
- Redirecting loop output

In the previous chapter, you saw how to manipulate the flow of a shell script program by checking the output of commands and the values of variables. In this chapter, we'll continue to look at structured commands that control the flow of your shell scripts. You'll see how you can perform repeating processes, commands that can loop through a set of commands until an indicated condition has been met. This chapter discusses and demonstrates the `for`,`while`, and `until` bash shell looping commands.

The for Command

Iterating through a series of commands is a common programming practice. Often you need to repeat a set of commands until a specific condition has been met, such as processing all of the files in a directory, all of the users on a system, or all of the lines in a text file.

The bash shell provides the `for` command to allow you to create a loop that iterates through a series of values. Each iteration performs a defined set of commands using one of the values in the series. The following is the basic format of the bash shell `for` command:

```
for var in list

do

commands

done
```

You supply the series of values used in the iterations in the *list* parameter. There are several different ways that you can specify the values in the list.

In each iteration, the variable *var* contains the current value in the list. The first iteration uses the first item in the list, the second iteration the second item, and so on until all of the items in the list have been used.

The *commands* entered between the do and done statements can be one or more standard bash shell commands. Within the commands, the `$var` variable contains the current list item value for the iteration.

Note

If you prefer, you can include the **do** statement on the same line as the **`for`** statement, but you must separate it from the list items using a semicolon: `for var in list; do`.

We mentioned that there are several different ways to specify the values in the list. The following sections show the various ways to do that.

Reading Values in a List

The most basic use of the `for` command is to iterate through a list of values defined within the `for` command itself:

```
$ cat test1
#!/bin/bash
# basic for command

for test in Alabama Alaska Arizona Arkansas California Colorado
do
  echo The next state is $test
done
$ ./test1
The next state is Alabama
The next state is Alaska
The next state is Arizona
The next state is Arkansas
The next state is California
The next state is Colorado
$
```

Each time the `for` command iterates through the list of values provided, it assigns the `$test` variable the next value in the list. The `$test` variable can be used just like any other script variable within the `for` command statements. After the last iteration, the `$test` variable remains valid throughout the remainder of the shell script. It retains the last iteration value (unless you change its value):

```
$ cat test1b
#!/bin/bash
# testing the for variable after the looping
```

```
for test in Alabama Alaska Arizona Arkansas California Colorado
do
  echo “The next state is $test”
done
echo “The last state we visited was $test”
test=Connecticut
echo “Wait, now we're visiting $test”
$ ./test1b
The next state is Alabama
The next state is Alaska
The next state is Arizona
The next state is Arkansas
The next state is California
The next state is Colorado
The last state we visited was Colorado
Wait, now we're visiting Connecticut
$
```

The `$test` variable retained its value and also allowed us to change the value and use it outside of the `for`command loop, as any other variable would.

Reading Complex Values in a List

Things aren't always as easy as they seem with the `for` loop. There are times when you run into data that causes problems. Here's a classic example of what can cause problems for shell script programmers:

```
$ cat badtest1
#!/bin/bash
# another example of how not to use the for command

for test in I don't know if this'll work
do
  echo “word:$test”
done
$ ./badtest1
```

```
word:I
word:dont know if thisll
word:work
$
```

Ouch, that hurts. The shell saw the single quotation marks within the list values and attempted to use them to define a single data value, and it really messed things up in the process.

There are two ways to solve this problem:

- Use the escape character (the backslash) to escape the single quotation mark.
- Use double quotation marks to define the values that use single quotation marks.

Neither solution is all that fantastic, but each one does help solve the problem:

```
$ cat test2
#!/bin/bash
# another example of how not to use the for command

for test in I don\'t know if “this'll” work
do
 echo “word:$test”
done
$ ./test2
word:I
word:don't
word:know
word:if
word:this'll
word:work
$
```

In the first problem value, you added the backslash character to escape the single quotation mark in the `don't`value. In the second problem value, you enclosed the `this'll` value in double quotation marks. Both methods worked fine to distinguish the value.

Yet another problem you may run into is multi-word values. Remember that the `for` loop assumes that each value is separated with a space. If you have data values that contain spaces, you'll run into yet another problem:

```
$ cat badtest2
#!/bin/bash
```

```
# another example of how not to use the for command

for test in Nevada New Hampshire New Mexico New York North Carolina
do
  echo “Now going to $test”
done
$ ./badtest1
Now going to Nevada
Now going to New
Now going to Hampshire
Now going to New
Now going to Mexico
Now going to New
Now going to York
Now going to North
Now going to Carolina
$
```

Oops, that's not exactly what we wanted. The `for` command separates each value in the list with a space. If there are spaces in the individual data values, you must accommodate them using double quotation marks:

```
$ cat test3
#!/bin/bash
# an example of how to properly define values

for test in Nevada “New Hampshire” “New Mexico” “New York”
do
  echo “Now going to $test”
done
$ ./test3
Now going to Nevada
Now going to New Hampshire
Now going to New Mexico
Now going to New York
```

```
$
```

Now the `for` command can properly distinguish between the different values. Also, notice that when you use double quotation marks around a value, the shell doesn't include the quotation marks as part of the value.

Reading a List from a Variable

Often what happens in a shell script is that you accumulate a list of values stored in a variable and then need to iterate through the list. You can do this using the `for` command as well:

```
$ cat test4
#!/bin/bash
# using a variable to hold the list

list=“Alabama Alaska Arizona Arkansas Colorado”
list=$list“ Connecticut”

for state in $list
do
  echo “Have you ever visited $state?”
done
$ ./test4
Have you ever visited Alabama?
Have you ever visited Alaska?
Have you ever visited Arizona?
Have you ever visited Arkansas?
Have you ever visited Colorado?
Have you ever visited Connecticut?
$
```

The `$list` variable contains the standard text list of values to use for the iterations. Notice that the code also uses another assignment statement to add (or concatenate) an item to the existing list contained in the `$list`variable. This is a common method for adding text to the end of an existing text string stored in a variable.

Reading Values from a Command

Yet another way to generate values for use in the list is to use the output of a command. You use the backtick characters to execute any command that produces output, and then use the output of the command in the for command:

```
$ cat test5
#!/bin/bash
# reading values from a file

file="states"

for state in 'cat $file'
do
  echo "Visit beautiful $state"
done
$ cat states
Alabama
Alaska
Arizona
Arkansas
Colorado
Connecticut
Delaware
Florida
Georgia
$ ./test5
Visit beautiful Alabama
Visit beautiful Alaska
Visit beautiful Arizona
Visit beautiful Arkansas
Visit beautiful Colorado
Visit beautiful Connecticut
Visit beautiful Delaware
Visit beautiful Florida
Visit beautiful Georgia
```

```
$
```

This example uses the `cat` command in tick marks to display the contents of the file states. You'll notice that the states file includes each state on a separate line, not separated by spaces. The `for` command still iterates through the output of the `cat` command one line at a time, assuming that each state is on a separate line. However, this doesn't solve the problem of having spaces in data. If you list a state with a space in it, the `for` command will still take each word as a separate value. There's a reason for this, which we look at in the next section.

Note

The `test5` code example assigned the filename to the variable using just the filename without a path. This requires that the file be in the same directory as the script. If this isn't the case, you'll need to use a full pathname (either absolute or relative) to reference the file location.

Changing the Field Separator

The cause of this problem is the special environment variable IFS, called the *internal field separator*. The IFS environment variable defines a list of characters the bash shell uses as field separators. By default, the bash shell considers the following characters as field separators:

- A space
- A tab
- A newline

If the bash shell sees any of these characters in the data, it will assume that you're starting a new data field in the list. When working with data that can contain spaces (such as filenames), this can be annoying, as you saw in the previous script example.

To solve this problem, you can temporarily change the IFS environment variable values in your shell script to restrict the characters the bash shell recognizes as field separators. However, there is somewhat of an odd way of doing this. For example, if you want to change the IFS value to recognize only the newline character, you need to do this:

```
IFS=$‘\n’
```

Adding this statement to your script tells the bash shell to ignore spaces and tabs in data values. Applying this to the previous script yields the following:

```
$ cat test5b

#!/bin/bash

# reading values from a file


file=“states”


IFS=$‘\n’
```

```
for state in ‘cat $file’
do
  echo “Visit beautiful $state”
done
$ ./test5b
Visit beautiful Alabama
Visit beautiful Alaska
Visit beautiful Arizona
Visit beautiful Arkansas
Visit beautiful Colorado
Visit beautiful Connecticut
Visit beautiful Delaware
Visit beautiful Florida
Visit beautiful Georgia
Visit beautiful New York
Visit beautiful New Hampshire
Visit beautiful North Carolina
$
```

Now the shell script is able to use values in the list that contain spaces.

Caution

When working on long scripts, it's possible to change the IFS value in one place, and then forget about it and assume the default value elsewhere in the script. A safe practice to get into is to save the original IFS value before changing it and then restore it when you're done.

This technique can be coded like this:

```
IFS.OLD=$IFS
IFS=$‘\n’
<use the new IFS value in code>
IFS=$IFS.OLD
```

This ensures that the IFS value is returned to the default value for future operations within the script.

There are other excellent applications of the IFS environment variable. Say that you want to iterate through values in a file that are separated by a colon (such as in the `/etc/passwd` file). All you need to do is set the IFS value to a colon:

```
IFS=:
```

If you want to specify more than one IFS character, just string them together on the assignment line:

```
IFS=$‘\n’:;”
```

This assignment uses the newline, colon, semicolon, and double quotation mark characters as field separators. There's no limit to how you can parse your data using the IFS characters.

Reading a Directory Using Wildcards

Finally, you can use the `for` command to automatically iterate through a directory of files. To do this, you must use a wildcard character in the file or pathname. This forces the shell to use *file globbing*. File globbing is the process of producing file or path names that match a specified wildcard character.

This feature is great for processing files in a directory when you don't know all of the filenames:

```
$ cat test6
#!/bin/bash
# iterate through all the files in a directory

for file in /home/rich/test/*
do

  if [ -d “$file” ]
  then
    echo “$file is a directory”
  elif [ -f “$file” ]
  then
    echo “$file is a file”
  fi
done
$ ./test6
/home/rich/test/dir1 is a directory
/home/rich/test/myprog.c is a file
/home/rich/test/myprog is a file
/home/rich/test/myscript is a file
/home/rich/test/newdir is a directory
```

```
/home/rich/test/newfile is a file
/home/rich/test/newfile2 is a file
/home/rich/test/testdir is a directory
/home/rich/test/testing is a file
/home/rich/test/testprog is a file
/home/rich/test/testprog.c is a file
$
```

The `for` command iterates through the results of the `/home/rich/test/*` listing. The code tests each entry using the `test` command (using the square bracket method) to see if it's a directory, using the `-d` parameter, or a file, using the `-f` parameter (See Chapter 11).

Notice in this example that we did something different in the `if` statement tests:

```
if [ -d “$file” ]
```

In Linux, it's perfectly legal to have directory and filenames that contain spaces. To accommodate that, you should enclose the `$file` variable in double quotation marks. If you don't, you'll get an error if you run into a directory or filename that contains spaces:

```
./test6: line 6: [: too many arguments
./test6: line 9: [: too many arguments
```

The bash shell interprets the additional words as arguments within the `test` command, causing an error.

You can also combine both the directory search method and the list method in the same `for` statement by listing a series of directory wildcards in the `for` command:

```
$ cat test7
#!/bin/bash
# iterating through multiple directories

for file in /home/rich/.b* /home/rich/badtest
do
  if [ -d “$file” ]
  then
    echo “$file is a directory”
  elif [ -f “$file” ]
  then
    echo “$file is a file”
  else
   echo “$file doesn't exist”
  fi
done
$ ./test7
/home/rich/.backup.timestamp is a file
/home/rich/.bash_history is a file
/home/rich/.bash_logout is a file
/home/rich/.bash_profile is a file
/home/rich/.bashrc is a file
/home/rich/badtest doesn't exist
$
```

The `for` statement first uses file globbing to iterate through the list of files that result from the wildcard character; then it iterates through the next file in the list. You can combine any number of wildcard entries in the list to iterate through.

Caution

Notice that you can enter anything in the list data. Even if the file or directory doesn't exist, the `for` statement attempts to process whatever you place in the list. This can be a problem when working with files and directories. You have no way of knowing if you're trying to iterate through a nonexistent directory: It's always a good idea to test each file or directory before trying to process it.

The C-Style for Command

If you've done any programming using the C programming language, you're probably surprised by the way the bash shell uses the `for` command. In the C language, a `for` loop normally defines a variable, which it then alters automatically during each iteration. Typically, programmers use this variable as a counter and either increment or decrement the counter by one in each iteration. The bash `for` command can also provide this functionality. This section shows you how you can use a C-style `for` command in a bash shell script.

The C Language for Command

The C language `for` command has a specific method for specifying a variable, a condition that must remain true for the iterations to continue, and a method for altering the variable for each iteration. When the specified condition becomes false, the `for` loop stops. The condition equation is defined using standard mathematical symbols. For example, consider the following C language code:

```
for (i = 0; i < 10; i++)
{
  printf("The next number is %d\n", i);
}
```

This code produces a simple iteration loop, where the variable `i` is used as a counter. The first section assigns a default value to the variable. The middle section defines the condition under which the loop will iterate. When the defined condition becomes false, the `for` loop stops iterations. The last section defines the iteration process. After each iteration, the expression defined in the last section is executed. In this example, the `i` variable is incremented by one after each iteration.

The bash shell also supports a version of the `for` loop that looks similar to the C-style `for` loop, although it does have some subtle differences, including a couple of things that will confuse shell script programmers. Here's the basic format of the C-style bash `for` loop:

```
 for ((
variable
assignment ;
condition ;
iteration
process ))
```

The format of the C-style `for` loop can be confusing for bash shell script programmers, as it uses C-style variable references instead of the shell-style variable references. Here's what a C-style `for` command looks like:

```
for (( a = 1; a < 10; a++ ))
```

Notice that there are a couple of things that don't follow the standard bash shell `for` method:

- The assignment of the variable value can contain spaces.
- The variable in the condition isn't preceded with a dollar sign.
- The equation for the iteration process doesn't use the `expr` command format.

The shell developers created this format to more closely resemble the C-style `for` command. While this is great for C programmers, it can throw even expert shell programmers into a tizzy. Be careful when using the C-style `for`loop in your scripts.

Here's an example of using the C-style `for` command in a bash shell program:

```
$ cat test8
#!/bin/bash
# testing the C-style for loop


for (( i=1; i <= 10; i++ ))
do
  echo "The next number is $i"
done
```

```
$ ./test8
The next number is 1
The next number is 2
The next number is 3
The next number is 4
The next number is 5
The next number is 6
The next number is 7
The next number is 8
The next number is 9
The next number is 10
$
```

The `for` loop iterates through the commands using the variable defined in the for loop (the letter *i* in this example). In each iteration, the `$i` variable contains the value assigned in the `for` loop. After each iteration, the loop iteration process is applied to the variable, which in this example, increments the variable by one.

Using Multiple Variables

The C-style `for` command also allows you to use multiple variables for the iteration. The loop handles each variable separately, allowing you to define a different iteration process for each variable. While you can have multiple variables, you can only define one condition in the `for` loop:

```
$ cat test9
#!/bin/bash
# multiple variables

for (( a=1, b=10; a <= 10; a++, b-- ))
do
  echo “$a - $b”
done
$ ./test9
1 - 10
2 - 9
3 - 8
4 - 7
```

```
5 - 6
6 - 5
7 - 4
8 - 3
9 - 2
10 - 1
$
```

The a and b variables are each initialized with different values and different iteration processes are defined. While the loop increases the a variable, it decreases the b variable for each iteration.

The while Command

The `while` command is somewhat of a cross between the `if-then` statement and the `for` loop. The `while` command allows you to define a command to test and then loop through a set of commands for as long as the defined test command returns a zero exit status. It tests the `test` command at the start of each iteration. When the `test` command returns a non-zero exit status, the `while` command stops executing the set of commands.

Basic while Format

The format of the `while` command is:

```
while
test
command
do

other
commands
done
```

The *`test command`* defined in the `while` command is the exact same format as in `if-then` statements (see Chapter 11). As in the `if-then` statement, you can use any normal bash shell command, or you can use the `test` command to test for conditions, such as variable values.

The key to the `while` command is that the exit status of the *`test command`* specified must change, based on the commands run during the loop. If the exit status never changes, the `while` loop will get stuck in an infinite loop.

The most common use of the *`test command`* is to use brackets to check a value of a shell variable that's used in the loop commands:

```
$ cat test10
#!/bin/bash
# while command test

var1=10
while [ $var1 -gt 0 ]
do
  echo $var1
  var1=$[ $var1 - 1 ]
done
$ ./test10
10
9
8
7
6
5
4
3
2
1
$
```

The `while` command defines the test condition to check for each iteration:

```
while [ $var1 -gt 0 ]
```

As long as the test condition is true, the `while` command continues to loop through the commands defined. Within the commands, the variable used in the test condition must be modified, or else you'll have an infinite loop. In this example, we use shell arithmetic to decrease the variable value by one:

```
var1=$[ $var1 - 1 ]
```

The `while` loop stops when the test condition is no longer true.

Using Multiple Test Commands

In somewhat of an odd situation, the `while` command allows you to define multiple test commands on the while statement line. Only the exit status of the last test command is used to determine when the loop stops. This can cause some interesting results if you're not careful. Here's an example of what we mean:

```
$ cat test11

#!/bin/bash

# testing a multicommand while loop


var1=10


while echo $var1

   [ $var1 -ge 0 ]

do

  echo “This is inside the loop”

  var1=$[ $var1 - 1 ]

done

$ ./test11

10

This is inside the loop

9

This is inside the loop

8

This is inside the loop

7

This is inside the loop

6

This is inside the loop

5

This is inside the loop

4

This is inside the loop

3

This is inside the loop

2
```

```
This is inside the loop
1
This is inside the loop
0
This is inside the loop
-1
$
```

Pay close attention to what happened in this example. There were two test commands defined in the `while`statement:

```
while echo $var1
   [ $var1 -ge 0 ]
```

The first test simply displays the current value of the `var1` variable. The second test uses brackets to determine the value of the `var1` variable. Inside the loop, an `echo` statement displays a simple message, indicating that the loop was processed. Notice when you run the example how the output ends:

```
This is inside the loop
-1
$
```

The `while` loop executed the `echo` statement when the `var1` variable was equal to zero, and then decreased the `var1`variable value. Next, the test commands were executed for the next iteration. The `echo` test command was executed, displaying the value of the `var1` variable, which is now less than zero. It's not until the shell executes the `test`test command that the `while` loop terminates.

This demonstrates that in a multi-command `while` statement, all of the test commands are executed in each iteration, including the last iteration when the last test command fails. Be careful of this. Another thing to be careful of is how you specify the multiple test commands. Note that each test command is on a separate line!

The until Command

The `until` command works in exactly the opposite way from the `while` command. The `until` command requires that you specify a test command that normally produces a non-zero exit status. As long as the exit status of the test command is non-zero, the bash shell executes the commands listed in the loop. Once the test command returns a zero exit status, the loop stops.

As you would expect, the format of the `until` command is:

```
until test
commands
```

```
do

other commands
done
```

Similar to the `while` command, you can have more than one *test command* in the `until` command statement. Only the exit status of the last command determines if the bash shell executes the *other commands* defined.

The following is an example of using the `until` command:

```
$ cat test12
#!/bin/bash
# using the until command

var1=100

until [ $var1 -eq 0 ]
do
  echo $var1
  var1=$[ $var1 - 25 ]
done
$ ./test12
100
75
50
25
$
```

This example tests the `var1` variable to determine when the `until` loop should stop. As soon as the value of the variable is equal to zero, the `until` command stops the loop. The same caution as for the `while` command applies when you use multiple test commands with the `until` command:

```
$ cat test13
#!/bin/bash
# using the until command

var1=100
```

```
until echo $var1
   [ $var1 -eq 0 ]
do
  echo Inside the loop: $var1
  var1=$[ $var1 - 25 ]
done
$ ./test13
100
Inside the loop: 100
75
Inside the loop: 75
50
Inside the loop: 50
25
Inside the loop: 25
0
$
```

The shell executes the test commands specified and stops only when the last command is true.

Nesting Loops

A loop statement can use any other type of command within the loop, including other loop commands. This is called a*nested loop*. Care should be taken when using nested loops, as you're performing an iteration within an iteration, which multiplies the number of times commands are being run. Not paying close attention to this can cause problems in your scripts.

Here's a simple example of nesting a `for` loop inside another `for` loop:

```
$ cat test14
#!/bin/bash
# nesting for loops

for (( a = 1; a <= 3; a++ ))
do
  echo “Starting loop $a:”
  for (( b = 1; b <= 3; b++ ))
  do
    echo “   Inside loop: $b”
  done
done
$ ./test14
Starting loop 1:
  Inside loop: 1
  Inside loop: 2
  Inside loop: 3
Starting loop 2:
  Inside loop: 1
  Inside loop: 2
  Inside loop: 3
Starting loop 3:
  Inside loop: 1
  Inside loop: 2
  Inside loop: 3
$
```

The nested loop (also called the *inner loop*) iterates through its values for each iteration of the outer loop. Notice that there's no difference between the `do` and `done` commands for the two loops. The bash shell knows when the first `done` command is executed that it refers to the inner loop and not the outer loop.

The same applies when you mix loop commands, such as placing a `for` loop inside a `while` loop:

```
$ cat test15
#!/bin/bash
# placing a for loop inside a while loop

var1=5

while [ $var1 -ge 0 ]
do
  echo “Outer loop: $var1”
  for (( var2 = 1; $var2 < 3; var2++ ))
  do
    var3=$[ $var1 * $var2 ]
    echo "  Inner loop: $var1 * $var2 = $var3"
  done
  var1=$[ $var1 - 1 ]
done
$ ./test15
Outer loop: 5
 Inner loop: 5 * 1 = 5
 Inner loop: 5 * 2 = 10
Outer loop: 4
 Inner loop: 4 * 1 = 4
 Inner loop: 4 * 2 = 8
Outer loop: 3
 Inner loop: 3 * 1 = 3
 Inner loop: 3 * 2 = 6
Outer loop: 2
 Inner loop: 2 * 1 = 2
 Inner loop: 2 * 2 = 4
Outer loop: 1
 Inner loop: 1 * 1 = 1
 Inner loop: 1 * 2 = 2
Outer loop: 0
 Inner loop: 0 * 1 = 0
 Inner loop: 0 * 2 = 0
$
```

Again, the shell was able to distinguish between the `do` and `done` commands of the inner `for` loop from the same commands in the outer while loop.

If you really want to test your brain, you can even combine `until` and `while` loops:

```
$ cat test16
#!/bin/bash
# using until and while loops
```

```
var1=3

until [ $var1 -eq 0 ]
do
  echo “Outer loop: $var1”
  var2=1
  while [ $var2 -lt 5 ]
  do
    var3=‘echo “scale=4; $var1 / $var2” | bc’
    echo “   Inner loop: $var1 / $var2 = $var3”
    var2=$[ $var2 + 1 ]
  done
  var1=$[ $var1 - 1 ]
done
$ ./test16
Outer loop: 3
  Inner loop: 3 / 1 = 3.0000
  Inner loop: 3 / 2 = 1.5000
  Inner loop: 3 / 3 = 1.0000
  Inner loop: 3 / 4 = .7500
Outer loop: 2
  Inner loop: 2 / 1 = 2.0000
  Inner loop: 2 / 2 = 1.0000
  Inner loop: 2 / 3 = .6666
  Inner loop: 2 / 4 = .5000
Outer loop: 1
  Inner loop: 1 / 1 = 1.0000
  Inner loop: 1 / 2 = .5000
  Inner loop: 1 / 3 = .3333
  Inner loop: 1 / 4 = .2500
$
```

The outer `until` loop starts with a value of 3 and continues until the value equals 0. The inner `while` loop starts with a value of 1 and continues as long as the value is less than 5. Each loop must change the value used in the test condition, or the loop will get stuck infinitely.

Looping on File Data

Often, you must iterate through items stored inside a file. This requires combining two of the techniques covered:

- Using nested loops
- Changing the `IFS` environment variable

By changing the `IFS` environment variable, you can force the `for` command to handle each line in the file as a separate item for processing, even if the data contains spaces. Once you've extracted an individual line in the file, you may have to loop again to extract data contained within it.

The classic example of this is processing data in the `/etc/passwd` file. This requires that you iterate through the `/etc/passwd` file line by line and then change the `IFS` variable value to a colon so that you can separate out the individual components in each line.

The following is an example of doing just that:

```
#!/bin/bash
# changing the IFS value


IFS.OLD=$IFS
IFS=$‘\n’
for entry in ‘cat /etc/passwd’
do
  echo “Values in $entry –”
  IFS=:
  for value in $entry
  do
    echo “   $value”
  done
done
$
```

This script uses two different `IFS` values to parse the data. The first `IFS` value parses the individual lines in the `/etc/passwd` file. The inner `for` loop next changes the `IFS` value to the colon, which allows you to parse the individual values within the `/etc/passwd` lines.

When you run this script, you'll get output something like this:

```
Values in rich:x:501:501:Rich Blum:/home/rich:/bin/bash -
  rich
  x
  501
  501
  Rich Blum
  /home/rich
  /bin/bash
Values in katie:x:502:502:Katie Blum:/home/katie:/bin/bash -
  katie
  x
  506
  509
  Katie Blum
  /home/katie
  /bin/bash
```

The inner loop parses each individual value in the `/etc/passwd` entry. This is also a great way to process comma-separated data, a common way to import spreadsheet data.

Controlling the Loop

You might be tempted to think that once you start a loop, you're stuck until the loop finishes all of its iterations. This is not true. There are a couple of commands that help us control what happens inside of a loop:

- The `break` command
- The `continue` command

Each command has a different use in how to control the operation of a loop. The following sections describe how you can use these commands to control the operation of your loops.

The break Command

The break command is a simple way to escape out of a loop in progress. You can use the break command to exit out of any type of loop, including while and until loops.

There are several situations in which you can use the break command. This section shows each of these methods.

Breaking Out of a Single Loop

When the shell executes a break command, it attempts to break out of the loop that's currently processing:

```
$ cat test17
#!/bin/bash
# breaking out of a for loop

for var1 in 1 2 3 4 5 6 7 8 9 10
do
  if [ $var1 -eq 5 ]
  then
    break
  fi
  echo “Iteration number: $var1”
done
echo “The for loop is completed”
$ ./test17
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
The for loop is completed
$
```

The for loop should normally have iterated through all of the values specified in the list. However, when the if-then condition was satisfied, the shell executed the break command, which stopped the for loop.

This technique also works for while and until loops:

```
$ cat test18
#!/bin/bash
# breaking out of a while loop
```

```
var1=1

while [ $var1 -lt 10 ]
do
  if [ $var1 -eq 5 ]
  then
    break
  fi
  echo “Iteration: $var1”
  var1=$[ $var1 + 1 ]
done
echo “The while loop is completed”
$ ./test18
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
The while loop is completed
$
```

The `while` loop terminated when the `if-then` condition was met, executing the `break` command.

Breaking Out of an Inner Loop

When you're working with multiple loops, the `break` command automatically terminates the innermost loop you're in:

```
$ cat test19
#!/bin/bash
# breaking out of an inner loop

for (( a = 1; a < 4; a++ ))
do
  echo “Outer loop: $a”
  for (( b = 1; b < 100; b++ ))
  do
    if [ $b -eq 5 ]
    then
      break
    fi
    echo “   Inner loop: $b”
  done
done
$ ./test19
Outer loop: 1
  Inner loop: 1
  Inner loop: 2
  Inner loop: 3
  Inner loop: 4
Outer loop: 2
  Inner loop: 1
  Inner loop: 2
  Inner loop: 3
  Inner loop: 4
Outer loop: 3
  Inner loop: 1
  Inner loop: 2
  Inner loop: 3
  Inner loop: 4
$
```

The `for` statement in the inner loop specifies to iterate until the b variable is equal to 100. However, the `if-then` statement in the inner loop specifies that when the b variable value is equal to five, the `break` command is executed. Notice that even though the inner loop is terminated with the `break` command, the outer loop continues working as specified.

Breaking Out of an Outer Loop

There may be times when you're in an inner loop but need to stop the outer loop. The `break` command includes a single command line parameter value:

```
break
```

n

where *n* indicates the level of the loop to break out of. By default, *n* is 1, indicating to break out of the current loop. If you set *n* to a value of 2, the `break` command will stop the next level of the outer loop:

```
$ cat test20
#!/bin/bash
# breaking out of an outer loop

for (( a = 1; a < 4; a++ ))
do
  echo "Outer loop: $a"
  for (( b = 1; b < 100; b++ ))
  do
    if [ $b -gt 4 ]
    then
      break 2
    fi
    echo "   Inner loop: $b"
  done
done
$ ./test20
Outer loop: 1
  Inner loop: 1
  Inner loop: 2
  Inner loop: 3
  Inner loop: 4
$
```

Now when the shell executes the `break` command, the outer loop stops.

The continue Command

The `continue` command is a way to prematurely stop processing commands inside of a loop but not terminate the loop completely. This allows you to set conditions within a loop where the shell won't execute commands. Here's a simple example of using the `continue` command in a `for` loop:

```
$ cat test21
```

```
#!/bin/bash
# using the continue command

for (( var1 = 1; var1 < 15; var1++ ))
do
  if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
  then
    continue
  fi
  echo “Iteration number: $var1”
done
$ ./test21
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
Iteration number: 10
Iteration number: 11
Iteration number: 12
Iteration number: 13
Iteration number: 14
$
```

When the conditions of the `if-then` statement are met (the value is greater than five and less than 10), the shell executes the `continue` command, which skips the rest of the commands in the loop, but keeps the loop going. When the`if-then` condition is no longer met, things return back to normal.

You can use the `continue` command in `while` and `until` loops, but be extremely careful with what you're doing. Remember, when the shell executes the `continue` command, it skips the remaining commands. If you're incrementing your test condition variable in one of those conditions, bad things will happen:

```
$ cat badtest3
#!/bin/bash
# improperly using the continue command in a while loop
```

```
var1=0

while echo “while iteration: $var1”
   [ $var1 -lt 15 ]
do
  if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
  then
    continue
  fi
  echo “   Inside iteration number: $var1”
  var1=$[ $var1 + 1 ]
done
$ ./badtest3 | more
while iteration: 0
  Inside iteration number: 0
while iteration: 1
  Inside iteration number: 1
while iteration: 2
  Inside iteration number: 2
while iteration: 3
  Inside iteration number: 3
while iteration: 4
  Inside iteration number: 4
while iteration: 5
  Inside iteration number: 5
while iteration: 6
while iteration: 6
while iteration: 6
while iteration: 6
while iteration: 6
while iteration: 6
while iteration: 6
while iteration: 6
while iteration: 6
while iteration: 6
while iteration: 6
$
```

You'll want to make sure you redirect the output of this script to the `more` command so you can stop things. Everything seems to be going just fine until the `if-then` condition is met, and the shell executes the `continue`command. When the shell executes the `continue` command, it skips the remaining commands in the `while` loop. Unfortunately, that's where the `$var1` counter variable that is tested in the `while` test command is incremented. That meant that the variable wasn't incremented, as you can see from the continually displaying output.

As with the `break` command, the `continue` command allows you to specify what level of loop to continue with a command line parameter:

```
continue
```
n

where *n* defines the loop level to continue. Here's an example of continuing an outer `for` loop:

```
$ cat test22
#!/bin/bash
# continuing an outer loop

for (( a = 1; a <= 5; a++ ))
do
  echo “Iteration $a:”
  for (( b = 1; b < 3; b++ ))
  do
    if [ $a -gt 2 ] && [ $a -lt 4 ]
    then
      continue 2
    fi
    var3=$[ $a * $b ]
    echo “   The result of $a * $b is $var3”
  done
done
$ ./test22
Iteration 1:
  The result of 1 * 1 is 1
  The result of 1 * 2 is 2
Iteration 2:
  The result of 2 * 1 is 2
  The result of 2 * 2 is 4
Iteration 3:
Iteration 4:
  The result of 4 * 1 is 4
  The result of 4 * 2 is 8
Iteration 5:
  The result of 5 * 1 is 5
  The result of 5 * 2 is 10
$
```

The if-then statement:

```
if [ $a -gt 2 ] && [ $a -lt 4 ]
    then
      continue 2
    fi
```

uses the `continue` command to stop processing the commands inside the loop but continue the outer loop. Notice in the script output that the iteration for the value 3 doesn't process any inner loop statements, as the `continue`command stopped the processing, but continues with the outer loop processing.

Processing the Output of a Loop

Finally, you can either pipe or redirect the output of a loop within your shell script. You do this by adding the processing command to the end of the `done` command:

```
for file in /home/rich*
 do
  if [ -d “$file” ]
  then
    echo “$file is a directory”
  elif
    echo “$file is a file”
  fi
```

```
done > output.txt
```

Instead of displaying the results on the monitor, the shell redirects the results of the `for` command to the file`output.txt`.

Consider the following example of redirecting the output of a `for` command to a file:

```
$ cat test23
#!/bin/bash
# redirecting the for output to a file

for (( a = 1; a < 10; a++ ))
do
  echo “The number is $a”
done > test23.txt
echo “The command is finished.”
$ ./test23
The command is finished.
$ cat test23.txt
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9
$
```

The shell creates the file `test23.txt` and redirects the output of the `for` command only to the file. The shell displays the `echo` statement after the `for` command just as normal.

This same technique also works for piping the output of a loop to another command:

```
$ cat test24
#!/bin/bash
# piping a loop to another command
```

```
for state in “North Dakota” Connecticut Illinois Alabama Tennessee
do
  echo “$state is the next place to go”
done | sort
echo “This completes our travels”
$ ./test24
Alabama is the next place to go
Connecticut is the next place to go
Illinois is the next place to go
North Dakota is the next place to go
Tennessee is the next place to go
This completes our travels
$
```

The state values aren't listed in any particular order in the `for` command list. The output of the `for` command is piped to the `sort` command, which will change the order of the `for` command output. Running the script indeed shows that the output was properly sorted within the script.

Summary

Looping is an integral part of programming. The bash shell provides three different looping commands that you can use in your scripts. The `for` command allows you to iterate through a list of values, either supplied within the command line, contained in a variable, or obtained by using file globbing, to extract file and directory names from a wildcard character.

The `while` command provides a method to loop based on the condition of a command, using either ordinary commands or the test command, which allows you to test conditions of variables. As long as the command (or condition) produces a zero exit status, the `while` loop will continue to iterate through the specified set of commands.

The `until` command also provides a method to iterate through commands, but it bases its iterations on a command (or condition) producing a non-zero exit status. This feature allows you to set a condition that must be met before the iteration stops.

You can combine loops in shell scripts, producing multiple layers of loops. The bash shell provides the `continue`and `break` commands, which allow you to alter the flow of the normal loop process based on different values within the loop.

The bash shell also allows you to use standard command redirection and piping to alter the output of a loop. You can use redirection to redirect the output of a loop to a file or piping to redirect the output of a loop to another command. This provides a wealth of features with which you can control your shell script execution.