Exploring the Shell

This chapter describes the shell and its role in the UNIX system and explains its features and capabilities. It discusses shell variables and explains their use and the way they are defined. The chapter also introduces more shell metacharacters and ways to make the shell ignore their special meanings and explains UNIX startup files, internal processes, and process management. This chapter continues the introduction of new commands (utilities) so you can build your vocabulary of UNIX commands.

In This Chapter

9.1 THE UNIX SHELL

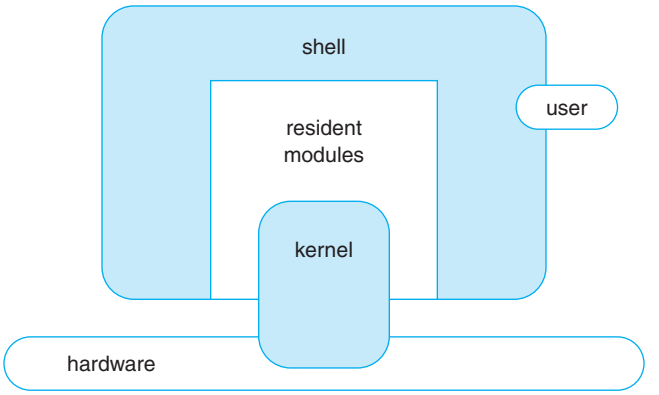
The UNIX operating system consists of two parts: the *kernel* and the *utilities*. The *kernel* is the heart of the UNIX system and is memory resident (which means that it stays in the memory from the time you boot the system until the system is shut down). All the routines that communicate directly with the hardware are concentrated in the kernel, which is relatively small in comparison with the rest of the operating system.

In addition to the kernel, other essential modules are also memory resident. These modules perform important functions such as input/output control, file management, memory management, and processor time management. Additionally, UNIX maintains several memory-resident tables for housekeeping purposes, to keep track of the system's status.

The rest of the UNIX system resides on the disk and is loaded into the memory only when necessary. Most of the UNIX commands you know are programs (called utilities) that reside on the disk. For those programs, when you type a command (request the program to be executed), the specified program is loaded into the memory.

You communicate with the operating system through a shell, and hardware-dependent operations are managed by the kernel. Figure 9.1 shows the components of the UNIX operating system.

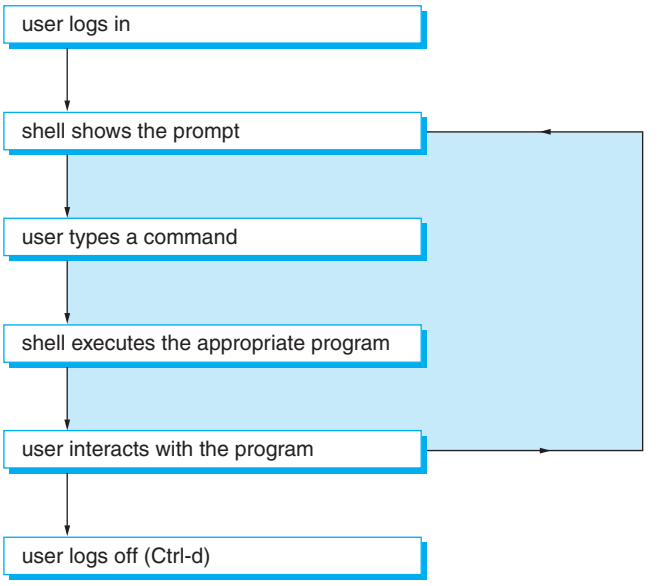
Figure 9.1
The Components of the UNIX Operating System



The shell is itself a program (a utility program). It loads into the memory whenever you log in to the system. When the shell is ready to receive commands, it displays a prompt. The shell itself does not carry out most of the commands that you type; it examines each command and starts the appropriate UNIX program (utility) to carry out the requested action. The shell determines what program to start (the name of the program is the same as the command you type). For example, when you type **ls** and press **[Return]** to list current directory files, the shell finds and starts a program called ls. The shell treats your application programs the same way: you type the program's name as a command, and the shell executes the program for you. Figure 9.2 shows the user interaction with the shell program.

See page x for an explanation of the icons used to highlight information in this chapter.

Figure 9.2
User Interaction with the Shell Program



9.1.1 Starting the Shell

In Chapters 2 and 3, the shell program was explained and you learned that the shell is the main method through which a user interacts with the UNIX, and also that there is more than one shell provided with each UNIX system. But how does the shell get started?

The shell is started after the user is successfully logged in to the system and remains active until the user logs out. Each user on the system has a default shell. The default shell for each user is specified in the system password file. This file is called `/etc/passwd`. The `passwd` file is the system password file and contains, among other things, user ID for each user, an encrypted copy of each user's password, and the name of the program to run immediately after a user logs in to the system. This program does not have to be one of the shell programs; however, it usually is.

When you log in, the system determines what shell to run by looking up your entry in the `/etc/passwd` file. The last field of each entry is the name of the program to run as the default shell. Table 9.1 shows the name of the shell programs and their corresponding shell names.

Table 9.1
The Shells and Shell Program Names

Shell Program Name	Prompt Sign	Shell Name
`/bin/sh`	$	Bourne shell
`/bin/ksh`	$	Korn shell
`/bin/bash`	$	Bourne Again shell
`/bin/csh`	%	C shell
`/bin/tcsh`	%	TC shell

Built-In Shell Commands

The shell command interpreters (*sh*, *ksh*, *bash*, etc.) have special built-in functions (programs), which are interpreted by the shell as commands. These commands are part of the shell itself and are recognized and executed internally. You already know some of the built-in commands such as **cd**, **pwd**, and others. Many of these built-in commands are implemented by more than one of the shells, and some are unique to a particular shell. You can obtain the full list of the built-in commands by typing the following command line:

```
$ man shell_builtins [Return] . . .
```
Display the shell built-in commands.

Table 9.2 lists the shell built-in commands that are covered in this chapter and shows their availability under different shells.

Table 9.2
The Built-In Shell Commands

Command	Built Into		
	Bourne Shell	Korn Shell	Bourne Again Shell
alias		ksh	bash
echo	sh	ksh	bash
history		ksh	bash
kill	sh	ksh	bash
set	sh	ksh	bash
unalias		ksh	bash
unset	sh	ksh	bash

9.1.2 Understanding the Shell's Major Functions

The standard UNIX system comes with more than 200 utility programs. One of these programs is sh, the shell itself.

The shell is the most frequently used utility program on the UNIX system. It is a sophisticated program that manages the dialogue between the user and the UNIX system. You interact with it repeatedly during work sessions. The shell is a regular executable C/C++ program that is usually stored in the `/bin` directory. When you log in, an interactive shell is invoked automatically. However, you can invoke another copy of the shell by typing **sh** (or **ksh** or **bash**, depending on what shells are available in your system) at the **$** prompt.

The shell includes the following major features. You are already familiar with some of these features, and the rest of them are explored in this chapter.

Command Execution Command (program) execution is a major function of the shell. Just about anything you type at the prompt is interpreted by the shell. When you press

[Return] at the end of the command line, the shell starts analyzing your command; if there are filename substitution characters or input/output redirection signs, it takes care of them, and then executes the appropriate program.

Filename Substitution If filename substitution (also called *filename generation*) is specified on the command line, the shell first performs the substitution and then executes the program. (The filename substitution characters—metacharacters * and **?**—were discussed in Chapter 8.)

I/O Redirection The input/output redirection is handled by the shell. Again, the shell program itself is not involved, and the redirection is set up before the command execution. If input or output redirection is specified on the command line, the shell opens the file and connects it to the standard input or standard output of the program respectively. This topic was discussed in Chapter 8.

Pipes Pipes, also called *pipelines,* let you connect simple programs together to perform a more complex task. The vertical line on the keyboard (¦) is the pipe operator.

Environment Control The shell lets you customize your environment to suit your needs. By setting the appropriate variables, you may change your `HOME` directory, prompt sign, or other aspects of your working environment.

Background Processing The background processing capability of the shell enables you to run programs in the background while doing other jobs in the foreground. This is helpful for time-consuming, noninteractive programs.

Shell Scripts Commonly used sequences of the shell commands can be stored in files called *shell scripts.* The name of the file later can be used to execute the stored program, enabling you to execute the stored commands with a single command. The shell also includes language constructs that allow you to build shell scripts that perform more complex jobs. Shell scripts are discussed in Chapter 12.

9.1.3 Displaying Information: The echo Command

You can use the **echo** command to display messages. It displays its arguments on your terminal, the standard output device. Without the argument, it produces an empty line and by default appends a new line to the end of the output. For example, if you type **echo hello there** and press **[Return]** at the prompt, you will see the following:

```
hello there
$_
```

The argument string can be any number of characters long. However, if your string contains any metacharacters, the string must be enclosed in quotation marks. (This topic is discussed later in this chapter.)

echo Options

Table 9.3 lists the **echo** command options and explains how they are used.

Table 9.3
The **echo** Command Options

Option	Operation
-n	Disables output of the trailing new line.
-e	Enables interpretations of the backslash escaped characters.

-e Option This option enables the interpretation of the escaped character such as **\n**. The **echo** command usually is set with this option as the default option. The **echo** command is implemented differently in each shell. Particularly for the *bash* shell, you must use the **-e** option in order for *bash* to recognize the escape characters.

Table 9.4 shows the characters that you can use as part of a string to control the format of the message. These characters are preceded by a backslash (\) and are interpreted by the shell to produce the desired output. They are also called *escape characters.*

Table 9.4
The Escape Characters

Escape Character	Meaning
\a	Audible alert (bell)
\b	Backspace
\c	Inhibit the terminating newline
\f	Form feed
\n	Carriage return and a line feed (newline)
\r	Carriage return without the line feed
\t	Horizontal tab
\v	Vertical tab

The backslash itself is a shell metacharacter. Therefore, if it is used in your string, it must be enclosed in quotation marks.

The following command sequences show how to use the **echo** command, and the result of incorporating the escape characters in the argument string:

```
$ echo Hi, this is a test. [Return]  . . . . . .  Show a simple message on the
                                                   screen.
Hi, this is a test.
$ echo Hi, "\n" this is a test. [Return] . . . .  Show the same message in two
                                                   lines.
Hi,
this is a test.
$_ . . . . . . . . . . . . . . . . . . . . . . .  Prompt.
```

1. **\n must be enclosed in quotation marks to be interpreted as the newline command.**
2. **If the echo command does not recognize the escape characters, use the -e option.**

```
$ echo -e Hi, "\n" this is a test [Return] . . . .   Using the -e option.
$ echo Hi, "\n" this is a test. > test [Return]. .   This time save the output in a
                                                     file.
$ cat test [Return]. . . . . . . . . . . . . . . .   Confirm the contents of test.
Hi,
this is a test.
$_ . . . . . . . . . . . . . . . . . . . . . . . .   Prompt.
$ echo Hi, "\n" this is a test. "\c" [Return]. . .   This time do not produce the
                                                     blank line at the end of the
                                                     message.
Hi,
this is a test.$
```

The prompt sign (**$**) appears right after the word *test.* That is the effect of **\c** in the argument string.

In the following command line, the extra spaces between the words are intentional:

```
$ echo This is a test. [Return] . . .   See what happens to the
                                        blanks.
This is a test.
```

The shell interprets this command line with four arguments, and each argument is separated with a space in the output.

```
$ echo "This is a test." [Return] . .   See the wonderful effects of
                                        quotation marks; now the
                                        blank spaces are preserved.

This is a test.
$_  . . . . . . . . . . . . . . . . .   And the prompt.
```

9.1.4 Removing Metacharacters' Special Meanings

The shell metacharacters have special meanings to the shell. But sometimes you want to override those meanings. The shell provides you with a set of characters that remove the meanings of the metacharacters. This process of removing the special meanings of the metacharacters is called *quoting* or *escaping.* The set of quoting characters is as follows:

- backslash \
- double quotation mark ”
- single quotation mark ’

Table 9.5 lists the characters that can be used for quoting.

The **echo** *command is used in most of the following examples to demonstrate how this process works. However, the use of quoting is applicable to other commands when you have to use any of the special characters as part of the command's argument.*

Table 9.5
Set of Quoting Characters

Quoting Character	Meaning
" " (double quotation marks)	Everything between "and" is taken literally, except for **$**, ` (grave accent quotation mark) and " (double quotation marks).
' ' (single quotation marks)	Everything between 'and' is taken literally, except for ' (single back quotation mark).
\ (backslash)	Any character following the \ is taken literally.

Backslash The backslash (\) is used to cause the character that follows it to be interpreted as an ordinary alphanumeric character. For example, **?** is a file substitution character (wild card) and has a special meaning to the shell. But **\?** is interpreted as the actual question mark.

Delete a file called `temp?` from your current directory:

```
$ rm temp? [Return]. . . . . .  Remove temp?.
```

The shell interprets this command as deleting all files whose filenames consist of `temp` and one character after it. Therefore, it deletes any file that matches this pattern, such as `temp`, `temp1`, `temp2`, `tempa`, `tempo`, and so on.

But if all you wanted to delete was a single file called `temp?`, then you would have typed the following:

```
$ rm temp\? [Return] . . . . . .  Try again, using \? to
                                  represent the ?
```

This time the shell scans the command line, finds the \, ignores the special meaning of the question mark, and passes the filename `temp?` to the **rm** program.

Display the metacharacters:

```
$ echo \< \> \" \' \$ \? \& \| \\ [Return].  Let's display them all.
< > "' $ ? & | \
$_ . . . . . .  Ready for the next
                command.
```

To remove the special meaning of the backslash, precede it with a backslash.

Double Quotation Marks You can use the double quotation marks (") to override the meaning of most of the special characters. Any special character between a pair of double quotation marks loses its special meaning, except the dollar sign (before a variable name), the single quotation mark, and the double quotation marks. (You use the backslash to remove their special meanings.)

Double quotation marks also preserve the white-space characters (i.e., the blank space, tab, and newline characters). The use of double quotation marks for this purpose was demonstrated in the **echo** command examples.

The following command sequences show the application of the double quotation marks:

```
$ echo > [Return] . . . . . .  Display the > sign.
syntax error: 'newline or;' unexpected
$_  . . . . . .  And the prompt appears.
```

The shell interprets your command as redirecting the output of the **echo** command to a file. It looks for the filename and because none is specified, it responds with a cryptic error message.

```
$ echo ">" [Return] . . . . . .  Enclose the argument with
                                 double quotation marks. The
                                 > is displayed.
>
$ ls -C [Return]  . . . . . .  Check your current directory files.
memos   myfirst   REPORTS
$ echo * [Return] . . . . . .  Use a metacharacter as the
                                 argument.
memos   myfirst   REPORTS
```

The shell substitutes * with the names of all the files in your current directory.

```
$ echo "*" [Return] . . . . . .  Now use the double quotation
                                 marks.
*
$_  . . . . . .  And the prompt appears.
```

No substitution occurs for the character between the double quotation marks. Therefore, the special meaning of the * is removed.

Display the message “The UNIX System”.

```
$ echo "\"The UNIX System\"" [Return]
"The UNIX system"
$_  . . . . . .  Prompt.
```

A backslash is necessary before a double quotation mark to override the special meaning of the double quotation marks.

Single Quotation Marks Single quotation marks (’) work very much like the double quotation marks. Any special character between a pair of single quotation marks loses its special meaning, except the single quotation mark. (You use \ to remove its special meaning.)

Single quotation marks also preserve the white-space characters. The string between the single quotation marks becomes a single argument, and the space character no longer has its special meaning as argument separator.

The open and close quotation marks for this purpose are the same character, the forward quotation mark. Do not use the back quotation mark (the grave accent character `). This distinction is very important. The shell interprets the string inside the grave accent marks as an executable command.

The grave accent mark (`), which is discussed in more detail later in the chapter, is the lowercase character on the tilde (~) key at the upper left of the keyboard.

Display special characters using a pair of single quotation marks.

```
$ echo ' < > " $ ? & ¦'[Return]. .   Use the echo command and
                                     single quotation marks.
< > " $ ? & ¦
$_ . . . . . . . . . . . . . . . .   The prompt is back.
```

The spaces between the characters are preserved.

9.2 SHELL VARIABLES

The shell program handles the user interface and acts as a command interpreter. In order for the shell to service all your requests (executing commands, manipulating files, etc.), it needs to have certain information and to keep track of that information (i.e., your HOME directory, terminal type, and prompt sign).

This information is stored in what are called the *shell variables. Variables* are named items you set to specific values to control or customize your environment. The shell supports two types of variables: environment variables and local variables.

Environment Variables Environment variables are also known as *standard variables*; they have names that are known to the system. They are used to keep track of the essential things the system needs and are usually defined by the system administrator. For example, the standard variable *TERM* is assigned to your terminal type:

```
TERM=ansi
```

Local Variables Local variables are user defined; they are entirely under your control. You can define, change, or delete them as you wish.

9.2.1 Displaying and Removing Variables: The set and unset Commands

You can use the **set** command to find out what shell variables are set for your shell to use.

Type **set** at the prompt and press **[Return]**, and the shell displays the list of the variables. Your list will be similar to but not exactly like that shown in Figure 9.3.

The names of the standard variables on the left of the equal **(=)** sign are shown in uppercase letters in Figure 9.3. This is not a requirement; you can use lowercase, uppercase, or any mixture for variable names. You can use characters, digits, and the underscore character in variable values, but the first letter must be a character, not a digit.

On the right side of the equal sign is the value assigned to a variable.

You must specify the exact variable name (including capitalization) when referring to a variable.

You use the **unset** command to remove an unwanted variable. If you have a variable *XYZ=10,* and you want to remove it, type **unset XYZ** and press **[Return]**.

Figure 9.3
The Output of the **set** Command

```
$ set
HOME=/usr/students/david
IFS=
LOGNAME=david
LOGTTY=/dev/tty06
MAIL=/usr/mail/students/david
MAILCHECK=600
PATH=:/bin:/usr/bin
PS1="$"
PS2=">"
TERM=ansi
TZ=EST=EDT
$_
```

9.2.2 Assigning Values to Variables

You can create your own variables, and you can also modify the values assigned to standard variables. You assign a value to a variable by writing the variable name, followed by an equal (=) sign (the assignment operator), followed by the value you want to assign to the variable, like this:

```
age=32
```

or

```
SYSTEM=UNIX
```

The shell treats every value that you assign to a variable as a string of characters. In the preceding example, the value of the variable *age* is the string 32, and not the number 32. If your string contains embedded white-space characters (**space**, **tab**, etc.) you must enclose the entire string in a pair of double quotation marks, like this:

```
message="Save your files, and log off" [Return]
```

1. **A shell variable name must begin with a lowercase or uppercase letter and not a digit.**
2. **There are no spaces on either side of the equal sign.**

9.2.3 Displaying the Values of Shell Variables

To access the value stored in a shell variable, you must precede the name of the variable with a **$**. Using the previous example, *age* is the name of the variable, and **$age** is 32, the value stored in the age variable.

You use the **echo** command to display the value assigned to the shell variable.

set *displays a list of variables;* **echo** *shows the specified variable.*

Use the **echo** command to display text and the values of the shell variables:

```
$ age=32 [Return] . . . . . .  Assign the value 32 to age.
$ echo Hi, nice day [Return]  . . . . . .  Display the argument string.
Hi, nice day
$ echo age [Return] . . . . . .  Display the argument, the word age.
age
$ echo $age [Return] . . . . . .  Now the argument is $age, the value stored
                                  in age.
32
$ echo You are $age years old. [Return] . .  Add some text to obtain more meaningful
                                             output.
You are 32 years old.
$_  . . . . . .  Ready for the next command.
```

The shell variables are frequently used as command arguments on a command line, as in the following:

```
$ all=-IFa [Return] . . . . . .  Create a variable called all and assign the
                                 value (string) -lFa (hyphen, lowercase
                                 letter l, upper-case letter F, lowercase
                                 letter a) to it.
$ ls $all myfirst [Return]  . . . . . .  Use the variable as part of a command
                                         line.
command's output
$_  . . . . . .  The prompt is displayed.
```

Variable names are preceded by **$**. Thus the shell substitutes the value **-laF** for the variable *all*. After substitution, the command becomes **ls -lFa myfirst**.

Observe the outputs of the following commands. They show the subtle differences in the way the variables are interpreted when they are between quotation marks.

```
$ age=32 [Return] . . . . . .  32 is assigned to the variable age.
$ echo $age "$age" '$age' [Return]  . . . .  Display age.
32 32 $age
$_  . . . . . .  Prompt.
```

9.2.4 Understanding the Standard Shell Variables

The values assigned to the standard shell variables are usually set by the system administrator. Thus, when you log in, the shell refers to these variables to keep track of things in your environment. You can change the value of these variables. However, the changes are temporary and apply only to the current session. The next time you log in, you have to set them again. If you want the changes to be permanent, place them in a file called `.profile`. The `.profile` file is explained later in this chapter.

HOME

When you log in, the shell assigns the full pathname of your `HOME` directory to the variable *HOME*. The `HOME` variable is used by several UNIX commands to locate the `HOME`

directory. For example, the **cd** command with no argument checks this variable to determine the pathname to the HOME directory and then sets the system to your HOME directory.

To experiment with the HOME variable, try the following command sequences:

```
$ echo $HOME [Return] . . . . . Show your HOME directory pathname.
/usr/david
$ pwd [Return] . . . . . Show the current directory pathname.
/usr/david/source . . . . . source subdirectory in david.
$ cd [Return] . . . . . No argument is specified. The default is
                        your HOME directory.
$ pwd [Return] . . . . . Check your current directory. You are in
                        david, your HOME directory.
/usr/david
$ HOME=/usr/david/memos/important [Return] . . Change your HOME directory pathname.
                        Now your HOME directory is important.
$ cd [Return] . . . . . Change to your HOME directory.
$ pwd [Return] . . . . . Display your current directory; your
                        current directory is important.
/usr/david/memos/important
$_ . . . . . And the prompt appears.
```

IFS

The Internal Field Separator (*IFS*) variable is set to a list of characters that are interpreted by the shell as separators of command line elements. For example, to get a long list of the files in your directory, you type **ls -l** and press **[Return]**. The space character in your command separates the command word (**ls**) from its option (**-l**).

Other separator characters assigned to the *IFS* variable are the tab character **[Tab]** and the newline character **[Return].**

The IFS characters are invisible (nonprintable) characters, so you do not see them on the right side of the equal sign. But they are there!

```
$ echo $IFS [Return] . . . . Display characters assigned to IFS.
$_
```

A blank line and prompt is back. The characters assigned to IFS are nonprintable.

Change the **IFS** field separator to the **:** (colon). The original **IFS** setting is saved before it is changed and later is restored to the original characters.

```
$ cd $HOME [Return] . . . . Notice here the field separator is a space
                        between cd and $HOME.
$ old_IFS=$IFS [Return] . . Save IFS characters.
$ IFS=":" [Return] . . . . . Change filed separator to :.
$ cd:$HOME [Return] . . . . Notice that the : is used as a field
                        separator instead of the space.
$ IFS=$old_IFS [Return] . . Restore the field separators back to the
                        original ones.
$_ . . . . . And the prompt is back.
```

It is not advisable to change the field separators. You usually do this to accommodate special circumstances. We are going to use this capability in sample programs in Chapter 13.

MAIL

The *MAIL* variable is set to the filename of the file that receives your mail. Mail sent to you is stored in this file, and the shell periodically checks the contents of this file to notify you if there is mail for you. For example, to set your mailbox to `/usr/david/mbox`, you would type **MAIL=/usr/david/mbox** and press **[Return]**.

MAILCHECK

The *MAILCHECK* variable specifies how often the shell is to check for arrival of mail in the file set in the *MAIL* variable. The default for *MAILCHECK* is 600 (seconds).

PATH

The *PATH* variable is set to the directory names that the shell searches for the location of the commands (programs) in the directory structure. For example, **PATH=:/bin:/ usr/bin**.

The directories in the path string are separated by colons. If the very first character in the path string is a colon, the shell interprets that as **.:** (dot, colon), meaning that your current directory is first on the list and is searched first.

UNIX usually stores the executable files in a directory called `bin`. You can create your own `bin` directory and store your executable files in it. If you add your `bin` directory (or any other name you call it) to the *PATH* variable, the shell looks there for any commands that it cannot find in the standard directories.

Suppose all of your executable files are located in a subdirectory called `mybin` that is located in your `HOME` directory. To add it to the *PATH* variable, you type: **PATH=:/ bin:/usr/bin:$HOME/mybin** and press **[Return]**.

PS1

The Prompt String 1 (*PS1*) variable is set to the string used as your prompt sign. The Bourne shell primary prompt sign is set to the dollar sign (**$**).

If you are tired of seeing the **$** prompt, you can easily change it by assigning a new value to the shell variable PS1.

```
$ PS1=Here: [Return] . . . . . Change your prompt to Here:
Here:_  . . . . . There you are.
Here: PS1="Here: " [Return] . . . . Add an extra blank space to the
                                    end.
Here: _ . . . . . It looks nicer!
```

If your prompt string has embedded spaces, then it must be enclosed in quotation marks.

```
Here: PS1="Next Command:"[Return] . . Change the prompt sign again.
Next Command:_. . . . . And it is changed.
Next Command: PS1="$ " [Return] . . Change back to the old $
                                    prompt.
$_. . . . . And the $ prompt returns.
```

PS2

The Prompt String 2 (*PS2*) variable is set to the prompt sign that is displayed whenever you press **[Return]** before completion of the command line and the shell expects the

rest of the command. You can change the *PS2* variable the same way you change the *PS1* variable. The Bourne shell secondary prompt defaults to the greater than sign (**>**).

The following command sequences show examples of the second prompt:

```
$ echo "Good news, UNIX [Return]. .   The command line is not
                                      complete; thus the PS2 prompt
                                      sign (>) is displayed.
> is on CDs." [Return]. . . . . .    Now the command line is
                                      complete.
Good news, UNIX is on CDs.
$ ls \ [Return] . . . . . .          The command line is not
                                      complete. This is signaled by
                                      the backslash.
> . . . . . .                        The shell displays the second
                                      prompt sign, and waits for the
                                      rest of the command.
> -l [Return] . . . . . .           Now the command line is
                                      complete. The shell puts it
                                      together as ls -l and executes it.
$_. . . . . .                        And the prompt is back.
```

CDPATH

The *CDPATH* variable is set to a list of absolute pathnames, similar to the *PATH* variable. The *CDPATH* affects the operation of the **cd** (change directory) command. If this variable is not defined, then **cd** searches your working directory to find the filename that matches its argument. If the subdirectory does not exist in your working directory, then UNIX displays an error message. If this variable is defined, **cd** searches for the specified directory according to the pathnames assigned to the *CDPATH* variable. If the directory is found, it becomes your working directory.

For example, if you type **CDPATH=:$HOME:$HOME/memos** and press **[Return]**, the next time you use the **cd** command, it will start searching from your current directory, then your `HOME` directory, and eventually the `memos` directory to find a match to the filename specified as the **cd** command argument.

SHELL

The *SHELL* variable is set to the full pathname of your login shell:

```
SHELL=/bin/sh
```

TERM

The *TERM* variable sets your terminal type:

```
TERM=ansi
```

TZ

The *TZ* variable sets the time zone that you are in:

```
TZ=EST
```

It is usually set by the system administrator.

9.3 MORE METACHARACTERS

As you remember from Chapter 8, metacharacters, or special characters, are interpreted and processed in a special manner by the shell. So far, we have discussed the file substitution and redirection metacharacters. This section explores some more of the metacharacters.

9.3.1 Executing the Commands: Using the Grave Accent Mark

The grave accent mark (`) before and after a command tell the shell to execute the enclosed command and to insert the command's output at the same point on the command line. It is also called *command substitution.* The format is as follows:

```
`command`
```

where *command* is the name of the command to be executed.

The grave accent mark character is found on the key at the far left of the keyboard, just below the **[Esc]** *key.*

The following command sequences show examples of command substitution:

```
$ echo The date and time is: `date` [Return] . .   Command date is executed.
The date and time is: Mon Nov 28 14:14:14 EDT 2005
$_. . . . . . . . . . . . . . . . . . . . . . . .   The prompt is back.
```

The shell scans the command line, finds the grave accent mark, and executes the command date. It replaces the **`date`** on the command line with the output from the date command and executes the echo command.

```
$ echo "List of filenames in your current directory:\n" `ls -C` > LIST [Return]
$ cat LIST [Return] . . . . . . . . . . . . . . .   Check what you have stored in
                                                    LIST.
List of filenames in your current directory:
memos   myfirst    REPORT
$_. . . . . . . . . . . . . . . . . . . . . . . .   Ready for the next command.
```

9.3.2 Sequencing the Commands: Using the Semicolon

You can enter a series of commands on a command line, separated by semicolons. The shell executes them in sequence from left to right.

To experiment with the semicolon metacharacter, try the following:

```
$ date ; pwd ; ls -C [Return] . . . . . . . . .   Three commands in sequence.
Mon Nov 28 14:14:14 EST 2005
/usr/david
memos myfirst REPORT
```

```
$ ls -C > list ; date > today; pwd [Return]. .   Three commands in sequence, with the
                                                 output of two commands redirected to
                                                 files.
/usr/david
$ cat list [Return]. . . . . . . . . . . . . .   Check content of list.
memos   myfirst   REPORT
$ cat today [Return] . . . . . . . . . . . . .   Check contents of today.
Mon Nov 28 14:14:14 EST 2001
$_ . . . . . . . . . . . . . . . . . . . . . .   Your favorite prompt sign appears.
```

9.3.3 Grouping the Commands: Using Parentheses

You can group commands together by placing them between a pair of parentheses. The group of commands can be redirected as if they were a single command.

To experiment with the parentheses as metacharacters, try the following:

```
$ (ls -C ; date ; pwd) > outfile [Return]. . .   Three commands in sequence, grouped
                                                 together, with the output redirected to
                                                 a file.
$ cat outfile [Return] . . . . . . . . . . . .   Check contents of outfile.
memos    myfirst    REPORT
Mon Nov 28 14:14:14 EST 2005
/usr/david
$_ . . . . . . . . . . . . . . . . . . . . . .   And the prompt appears.
```

9.3.4 Background Processing: Using the Ampersand

UNIX is a multitasking system; it allows you to execute several programs concurrently. Usually, you type a command and within a few seconds the output of the command is displayed on the terminal. What if you run a command that takes minutes to execute? In that case, you have to wait for the command to finish executing before you can proceed with the next job. However, you do not need to wait during all those unproductive minutes. The shell metacharacter ampersand (**&**) provides you the means to run programs in the background, as long as the background programs do not require input from the keyboard. If you enter a command followed by **&**, then that command is sent to the background for execution, and your terminal is free for the next command.

The following examples show applications of the ampersand metacharacter:

```
$ sort data > sorted & [Return]. . . . . . . .   Sort data and store the results in
                                                 sorted.
1348 . . . . . . . . . . . . . . . . . . . . .   Process ID is displayed.
$ date [Return] . . . . . . . . . . . . . . . .  The prompt is immediately displayed,
                                                 ready for the next command.
```

The output of the **sort** command is redirected to another file in order to prevent **sort** from sending its output to the terminal while you are doing other tasks.

1. *The background command process ID (PID) number identifies the background process and can be used to terminate it or obtain its status.*
2. *You can specify more than one background command on a single command line.*

```
$ date & pwd & ls -C & [Return] . . . . . .  Create three background processes;
                                             three PID numbers are displayed.
2215
2217
2216
$ echo "the foreground process" [Return]  . . Run the echo command in foreground.
Mon Nov 28 14:14:14 EST 2005  . . . . . .    Output from the background process
                                             date is displayed.
the foreground process  . . . . . .          Output from the foreground process
                                             echo is displayed.
/usr/david  . . . . . .                      Output from the background process
                                             pwd is displayed.
$_  . . . . . .                              The prompt appears and then output
                                             from the background process ls -C is
                                             displayed.
memos  myfirst  REPORT
$_  . . . . . .                              Ready for the next command.
```

By default, the output of the background commands is displayed on your terminal. Thus, the output of the foreground program is interleaved with the output of the background program and produces quite a confusing display. You can prevent the confusion by redirecting the output of background commands to files.

9.3.5 Chaining the Commands: Using the Pipe Operator

The shell lets you use the standard output of one process as standard input to another process. You use the pipe metacharacter, |, between the commands. The general format is as follows:

```
command A | command B
```

where command A output is introduced as input to command B. You can chain a sequence of commands together, creating what is called a *pipeline.* Let's look at some examples that give you an appreciation of this very useful and flexible shell capability.

Type **ls -l|lp** and press **[Return]** to send the output of the **ls -l** command to the printer.

To count the number of the files in your current directory, do the following:

```
$ ls -C [Return]  . . . . . .  Let's see the files in your current
                               directory.
memos myfirst REPORT
$ ls -C > count [Return]  . .  Now save the list of your files in count.
$ wc -w count [Return]  . . .  Count the number of words. You have
                               three files in your current directory.
3
ls -C | wc -w [Return]  . . .  Use the pipe operator to obtain the
                               number of files in your current directory.
3
```

The output of the command **ls -C** (list of the files in your current directory) is passed as input to the **wc -w** command.

To save the number of users logged in the system in a file, do the following:

```
$ echo "Number of logged-in users:" `who ¦ wc -l` > outfile [Return]
$ cat outfile [Return]. . . . . .  Check what is stored in outfile.
Number of logged-in users: 20
```

In the previous command, the shell scans the command line, finds the grave accent marks, executes the **who ¦ wc -l** commands, and passes the output of **who** to **wc** as input data. If there are 20 users logged on the system, then the output is 20. The shell replaces the **`who ¦ wc -l`** with 20. Then the shell executes the **echo** command that reads *Number of the logged in users: 20* and stores the output in `outfile`.

9.4 MORE UNIX UTILITIES

These utilities give you more flexibility and control in day-to-day usage of the system. Also, some of the utilities are used in script file (program) examples in Chapters 12 and 13.

Under Linux, the **--help** and **--version** options are available for most of the commands in this chapter. Please make a habit of using the **--help** option and read the help page to familiarize yourself with the other available options. Regardless of your UNIX system, you can always use the **man** command to obtain the full usage explanation for any of these commands.

9.4.1 Timing a Delay: The sleep Command

The **sleep** command causes the process executing it to go to sleep for a specified number of seconds. You can use **sleep** to delay the execution of a command for a period of time. For example, if you type **sleep 120 ; echo "I am awake!"** and press **[Return]**, the **sleep** command is executed and causes a two-minute delay; then (after two minutes) the **echo** command is executed, and the string argument *I am awake!* is displayed on the screen.

9.4.2 Displaying the PID: The ps Command

You can use the **ps** (process status) command to obtain the status of the active processes in the system. When used without any options, it displays information about your active processes. This information is arranged in four columns (see Figure 9.4) with the following column headings:

- PID: the process ID number
- TTY: your terminal number that controls the process
- TIME: time duration (in seconds) that your process is running
- COMMAND: the name of the command

Figure 9.4
The **ps** Command Output Format

```
$ ps
PID        TTY        TIME       COMMAND
24059      tty11      0:05       sh
24259      tty11      0:02       ps
$_
```

ps Options

Only two of the **ps** options are discussed in this book—the **-a** option and the **-f** option—and they are summarized in Table 9.6.

Table 9.6
The **ps** Command Options

Option	Operation
-a	Displays the status of all the active processes, not just the user's.
-f	Displays a full list of information, including the full command line.

-a Option The **-a** option displays status information for all active processes. Without this option, only the user's active processes are displayed.

-f Option The **-f** option displays a full list of information including the complete command line under the command column.

Figure 9.5 shows the output of the **ps** command using both **-a** and **-f** options.

Figure 9.5
Output of the **ps** Command with **-a** and **-f** Options

```
$ PS -a -f
PID      TTY      TIME    COMMAND
24059    tty11    0:05    sh
24259    tty11    0:02    ps
24059    tty12    0:05    ksh
24259    tty12    0:02    ps
$_
```

To find the process number of a process running in the background, use the following command sequence:

```
$ ( sleep 120 ; echo "Had a nice long sleep") & [Return]
24259 . . . . . The background process ID number.
$ ps [Return] . . . . . Show your processes' status.
PID  TTY  TIME  COMMAND
24059 tty11 0:05 sh . . . . . The login shell.
24070 tty11 0:00 sleep 120  . . The sleep command.
24150 tty11 0:00 echo . . . . . The echo command.
24259 tty11 0:02 ps . . . . . The ps command.
$ Had a nice long sleep . . . . Output from the background process.
$_. . . . . And the prompt appears.
```

The **sleep** command delays the execution of the **echo** command for two minutes. The **&** at the end of the command line places the commands in the background.

Separate the commands with semicolons, and group them together by placing them between parentheses.

9.4.3 Keep On Running: The nohup Command

When you log out, your background processes are terminated. The **nohup** command causes your background processes to be immune to terminating signals. This is useful when you want your programs to continue processing after you have logged out.

If you type **nohup (sleep 120 ; echo "job done") &** and press **[Return]** and then log out, your command process will continue in the background. Where is the **echo** command output displayed? Since you are logged out, the process is not associated with any terminal, and the output is automatically saved in a file called `nohup.out`.

When you log in, you can check the contents of this file to determine the output of your background processes. Alternatively, you can always redirect the output of your background programs to specified files.

To experiment with the **nohup** command, try the following command sequence:

```
$ nohup (sleep 120 ; echo "job done") & [Return]
  . . . . . . . . . . . . . . . . . Create a background job.
12235 . . . . . . . . . . . . . . . Background job PID.
$ [Ctrl-d]  . . . . . . . . . . . . Log out and wait a few minutes.
login: david [Return] . . . . . . Log in again.
password: . . . . . . . . . . . . Enter your password; it is not echoed
                                   to the screen.
$ cat nohup.out [Return]  . . . . Check the contents of nohup.out file.
job done
$_  . . . . . . . . . . . . . . . Ready for the next command.
```

9.4.4 Terminating a Process: The kill Command

Not all programs behave normally all the time. A program might be in an infinite loop or be waiting for resources that are not available. Sometimes an unruly program locks your keyboard, and then you are in real trouble! UNIX provides you with the **kill** command to terminate the unwanted process (a *process* is a running program). The **kill** command sends a signal to the specified process. The signal is an integer number indicating the kill type (UNIX is a morbid language), and the process is identified by the process ID number (PID). In order to use the **kill** command, you must know the PID of the process that you intend to terminate.

Signals Signals range from 1 to 15 and are mostly implementation dependent. However, 15 is usually the default signal value and causes the receiving process to terminate.

Some processes protect themselves from the **kill** signals. You use the signal value 9 (sure kill) to terminate these.

You can use the **kill** command with the **-l** option to obtain a list of the signals in your system:

```
$ kill -l [Return] . . . . . . . . . Display list of the signals.
```

The following command sequences illustrate the use of the **kill** command and its signals.

To issue a simple **kill** command, try the following:

```
$ (sleep 120 ; echo Hi) & [Return] . Create a background process.
22515. . . . . . . . . . . . . . . . Process ID number.
```

```
$ ps [Return]  . . . . . .Check the process's status. It is there:
PDI  TTY  TIME  COMMAND
24059 tty11 0:05 sh  . . . . . .The login shell.
22515 tty11 0:00 sleep 120   . .The sleep command.
24259 tty11 0:02 ps  . . . . . .The ps command.
$ kill 22515 [Return]  . . . . .Terminate the background process.
job  terminated
Hi   . . . . . .Output of the echo command.
$ ps [Return]  . . . . . .Check again. The background process
                          is terminated.
PDI  TTY  TIME  COMMAND
24059 tty11 0:05 sh  . . . . . .The login shell.
24259 tty11 0:02 ps  . . . . . .The ps command.
$_   . . . . . .And the prompt.
```

No signal number is specified. The default is signal number 15, which causes the receiving process to terminate.

To make sure an unruly process has been terminated, try the following:

```
$ (sleep 120 ; echo Hi) & [Return] . . . Create a background process.
22515  . . . . . .Process ID number.
$ kill 22515 [Return]  . . . . . .A simple kill.
$ ps [Return]  . . . . . .Check the process's status. It is still there:
PID  TTY  TIME  COMMAND
24059 tty11 0:05 sh  . . . . . .The login shell.
22515 tty11 0:00 sleep 120 . . . . . .The sleep command.
24259 tty11 0:02 ps  . . . . . .The ps command.
$ kill -9 22515 [Return] . . . . . .A sure kill; signal value 9 is specified.
$ ps [Return] . . . . . .Check again. Sure enough, the background
                         process is terminated.
PID  TTY  TIME  COMMAND
24059 tty11 0:05 sh  . . . . . .The login shell.
24259 tty11 0:02 ps  . . . . . .The ps command.
```

You can terminate only your own processes. The system administrator is authorized to terminate anybody's processes.

To terminate all of your processes, do the following:

```
$ (sleep 120 ; echo "sleep tight" ; sleep 120) & [Return]
11234  . . . . . .Sleep PID.
$ kill -9 0 [Return]  . . . . . You are logged out.
```

The PID 0 (zero) causes all processes associated with your shell to be terminated. That includes your login shell itself. Accordingly, when you use the 0 signal, you are logged out.

9.4.5 Splitting the Output: The tee Command

Sometimes you will want to look at the output of a program on the screen and also store the output in a file for later reference or obtain a hard copy of the output on the printer.

You can always do that this way: First run the command and view the output on the screen; then, using the redirection operator, save the output in a file or send it to the printer.

Alternatively, you can use the **tee** command to get the same result in less time and with less typing. The **tee** command is usually used with the pipe operator. For example, when you type **sort phone.list ¦ tee phone.sort** and press **[Return]**, the pipe operator passes the output of the **sort** command (the sorted `phone.list`) to **tee**. Then **tee** displays it on the terminal and also saves it in `phone.sort`, the specified file.

This is an indispensable command when you want to capture the user/program dialog while running an interactive program.

Try viewing the contents of your current directory and saving the output in a file as follows:

```
$ ls -C ¦ tee dir.list [Return]  .  Display the current directory file-
                                    names and also save the output in
                                    dir.list.
memos      myfile      REPORT
$ cat dir.list [Return]  . . . . .  Check the contents of dir.list.
memos      myfile      REPORT
$_ . . . . . . . . . . . . . . . .  The prompt appears.
```

The output of **ls -C** is piped to **tee**. The **tee** command shows its input on the screen (displays the input on the default output device) and also saves it in a file called `dir.list`.

tee Options

Table 9.7 summarizes the two options of the **tee** command.

Table 9.7
The **tee** Command Options

Option	Operation
-a	Appends output to a file without overwriting an existing file.
-i	Ignores interrupts; does not respond to the interrupt signals.

To view the list of the users currently on the system and save the list in an existing file called `dir.list`, type **who ¦ tee -a dir.list** and press **[Return]**. If `dir.list` exists, then the output of the **who** command is added to the end of the file. If `dir.list` does not exist, it is created.

Linux Alternative Options for the tee Command

Use the Linux alternative options for the **tee** command, as suggested in the following command lines:

```
$ tee --version [Return]  . . . .  Display version information.
$ tee --help [Return] . . . . . .  Display the help page.
```

Read the help page and familiarize yourself with other options available for the **tee** *command.*

9.4.6 File Searching: The grep Command

You can use the **grep** command to search for a specified pattern in a file or list of files. The pattern used by the **grep** command is called *regular expression*, hence the strange name of the command (*Global Regular Expression Print*).

grep is a file searching and selection command. You specify the filename and the pattern to be looked for in the file, and when **grep** finds a match, the line containing the specified pattern is displayed on the terminal. If no file is specified, the system searches through the input from the standard input device.

Look for the word *UNIX* in `myfile`.

```
$ cat myfile [Return]. . . . . Check contents of myfile.
I wish there were a better way to learn
UNIX. Something like having a daily UNIX pill.
$ grep UNIX myfile [Return]. . . Find the lines that contain the word
                                  UNIX.
UNIX. Something like having a daily UNIX pill.
```

You can specify more than one file or use file substitution (wild cards) in filenames.

Look for the string **"#include <private.h>"** in all the C source files:

- Type **grep "#include <private.h>" *.c** and press **[Return]** to look for the pattern in all files with extension c in the current directory.

The pattern is a string with embedded spaces and metacharacters, so it is enclosed in quotation marks.

If you specify more than one file to be searched, **grep** displays the name of the file preceding each line of output.

grep Options

If you do not specify any option, **grep** displays lines in the specified file(s) that contain a match for the specified pattern. The options give you more control over the output and the way the pattern search is done. Table 9.8 summarizes the **grep** options.

Assuming you have the following three files in your current directory, the command sequences show examples of **grep** using options. You can create these files using the vi editor or the **cat** command. To get organized, create these files under the `Chapter9` directory.

FILE1	FILE2	FILE3
UNIX	unix	Unix system
11122	11122	11122
BBAA	CCAA	AADD
unix system		

Search for the word *UNIX*:

```
$ grep UNIX FILE1 [Return] . . . Search for the word UNIX in FILE1.
UNIX
$_ . . . . . . . . . . . . . . . . And the prompt appears.
```

grep matches the exact pattern (whether in uppercase or lowercase letters), so it finds the word *UNIX* and not *unix* or *Unix*.

Table 9.8
The **grep** Command Options

Option		Operation
UNIX	Linux Alternative	
-c	**--count**	Displays only the count of the matching lines in each file that contains the match.
-i	**--ignore-case**	Ignores the distinction between lower- and uppercase letters in the search pattern.
-l	**--files-with-matches**	Displays the names of the files with one or more matching lines, not the lines themselves.
-n	**--line-number**	Displays a line number before each output line.
-v	**--revert-match**	Displays only those lines that do not match the pattern.
	--help	Displays help page and exits.
	--version	Displays version information and exits.

Specify more than one file as argument and use the **-i** option:

```
$ grep -i UNIX FILE? [Return]  . Use the -i option.
FILE1: UNIX
FILE1: unix system
FILE2: unix
FILE3: Unix system
$_ . . . . . . . . . . . . . . . Your prompt appears.
```

The **-i** option tries to match the specified letter pattern, regardless of case. Thus, the specified pattern *UNIX* matches *unix, Unix,* and so on.

The name of the file is displayed when you specify more than one file as the argument.

Show the lines that do not contain the word *UNIX*:

```
$ grep -vi UNIX FILE1 [Return] . Use options -i and -v.
11122
BBAA
$_ . . . . . . . . . . . . . . . The prompt appears.
```

Display how many lines in each file do not contain *11*:

```
$ grep -vc 11 FILE? [Return] . . Show a count of the lines in FILE1,
                                 FILE2, and FILE3 that do not
                                 contain 11.
FILE1:3
FILE2:2
FILE3:2
$_ . . . . . . . . . . . . . . . The prompt appears.
```

Find out whether user david is logged in:

```
$ who ¦ grep -i david [Return] . Use grep with the pipe operator.
$_ . . . . . . . . . . . . . . . The prompt appears.
```

The pipe makes the output of the **who** command the standard input to **grep**. Thus, **grep** scans the output of **who** for lines containing the pattern *david.* In this example, **grep** did not produce any output. Thus `david` is not in the system.

Linux Alternative Options for the grep Command

Use the Linux alternative options for the **grep** command, as suggested in the following command lines:

`$ grep --ignore-case UNIX file? [Return]`	Same as **grep -i UNIX file?**.
`$ grep --revert-match --ignore-case UNIX file? [Return]` .	Same as **grep -vi UNIX file?**.
`$ grep --revert-match --count file? [Return]`	**Same as grep -vc 11 file?**.
`$ grep --version [Return]`	Display version information.
`$ grep --help [Return]`	Display the help page.

9.4.7 Sorting Text Files: The sort Command

You can use the **sort** command to sort the contents of a file into alphabetical or numerical order. By default, the output is displayed on your terminal, but you can specify a filename as the argument or redirect the output to a file.

The **sort** command sorts the specified file on a line-by-line basis. If the first characters on two lines are the same, it compares the second characters to determine the order of the sort. If the second characters are the same, it compares the third characters, and this process goes on until two characters differ or the line ends. If two lines are identical, then it does not matter which one is placed first.

This command sorts files alphabetically, but the order of the sort might be different from one computer to another, depending on the computer's code set. The most commonly used code set in UNIX systems is ASCII.

Many options can be used to control the sort order, but let's start with a simple example to explore the **sort** basic functions.

Suppose you have a file called `junk` in your working directory. Figure 9.6 shows the contents of `junk`. Figure 9.7 shows the output of the **sort** command, after sorting the contents of `junk`.

Figure 9.6
The `junk` File

```
This is line one
this is line two
 this is a line starting with a space character
4: this is a line starting with a number
11: this is another line starting with a number
End of junk
```

Figure 9.7
The Sorted `junk` File

```
$ sort junk
 this is a line starting with a space character
11: this is another line starting with a number
4: this is a line starting with a number
End of junk
This is line one
this is line two
$_
```

1. *ASCII values for nonalphanumeric characters (space, dash, backslash, etc.) are less than those for alphanumeric characters. Thus, in Figure 9.7, the line starting with a blank space is placed at the top of the file.*
2. *Uppercase letters are sorted before lowercase letters. Thus, in our example,* `This` *appears before* `this`.
3. *Numbers are sorted by the first digit. Thus 11 appears before 4.*

sort Options

The **sort** example showed that the result of the **sort** command, the sorted output, is probably not what you would consider sorted. The **sort** command options give you freedom to sort files in a variety of orders. Table 9.9 summarizes some of the more useful options.

Table 9.9
The **sort** Command Options

Option	Operation
-b	Ignores leading blanks.
-d	Uses dictionary order for sorting. Ignores punctuation and control characters.
-f	Ignores the distinction between lowercase and uppercase letters.
-n	Numbers are sorted by their arithmetic values.
-o	Stores the output in the specified file.
-r	Reverses the order of the sort from ascending to descending order.

-b Option The **-b** option causes **sort** to ignore the leading blanks (tabs and space characters). These characters are usually delimiters (field separators) in your file, and when you use this option, **sort** does not consider them in sort comparison.

-d Option The **-d** option, used for dictionary sorting, uses only letters, digits, and blanks (spaces and tabs) in the **sort** comparison. It ignores the punctuation and control characters.

-f Option The **-f** option considers all lowercase characters as uppercase characters; it ignores the distinction between them in **sort** comparison.

-n Option The **-n** option causes numbers to be sorted by their arithmetic values rather than by their first digit. This includes ascribing minus signs and decimal points to their arithmetic values.

-o Option The **-o** option places the output in a specified file instead of the standard output.

-r Option The **-r** option reverses the order of the sort, such as sorting from *z* to *a* instead of *a* to *z*.

Using the `junk` *file again, let's see the effects of the options on the sorted output:*

```
$ sort -fn junk [Return]  . . . . . Sort junk using the
                                    -f and -n options.
this is a line starting with a space character
End of junk
This is line one
this is line two
4: this is a line starting with a number
11: this is another line starting with a number
$_  . . . . . . . . . . . . . . . . . . . . The prompt appears.
$ sort -f -r -o sorted junk [Return]  . Sort junk using the
                                        -f, -r, and -o options
                                        and save it in sorted.
$ cat sorted [Return] . . . . . . . . . Display sorted.
this is line two
This is line one
End of junk
4: this is a line starting with a number
11: this is another line starting with a number
this is a line starting with a space character
$_  . . . . . . . . . . . . . . . . . . . . The prompt appears.
```

A filename (`sorted`) *is specified with the* **-o** *option. Thus the output is saved in* `sorted`*, and* **cat** *is used to display the contents of* `sorted`.

9.4.8 Sorting on a Specified Field

Real files seldom contain what is in the example file `junk`. Usually files you want to sort contain lists of people, items, addresses, phone numbers, mailing lists, and so on. By default, **sort** sorts on a line-by-line basis, but you probably will want to sort files by a particular field, such as last name or area code.

You can direct the **sort** command to look at a specified field for **sort** comparison, provided that the file is set up accordingly. You specify the desired field by a number that indicates how many fields **sort** must skip to get to the field by which you want the file sorted. You set up your file by breaking each line into fields. No extra effort is needed, because in most list files each line is already divided into fields.

Create a file called `phone.list`, which contains a list of people and their phone numbers, such as the example in Figure 9.8, and then we will use the file to explore the other capabilities of the **sort** command.

Figure 9.8
Original `phone.list` File and Sorted `phone.list` File

```
$ cat phone.list
David Brown              (333) 111-1111
Emma Redd                (222) 222-2222
Tom Swanson              (111) 333-3333
Jim Schmid               (444) 444-4444
Bridget Erwin            (666) 555-5555
Mary Moffett             (555) 666-6666
Amir Afzal               (777) 777-7777

$ sort phone.list
Amir Afzal               (777) 777-7777
Bridget Erwin            (666) 555-5555
David Brown              (333) 111-1111
Emma Redd                (222) 222-2222
Jim Schmid               (444) 444-4444
Mary Moffett             (555) 666-6666
Tom Swanson              (111) 333-3333
$
```

Each line in `phone.list` consists of four fields, and the fields are separated by space or tab characters. Thus, in line one, *David* is field 1, *Brown* is field 2, and so on.

You can create the `junk` file using the vi editor or the **cat** command.

In sorting `phone.list`, no particular field is specified. Thus, the list is sorted on a line-by-line basis.

You may not want to sort the file by the first names. To sort the file in order of the last names (field 2), you must instruct **sort** to skip one field (first name) before it starts the sorting process. You specify the number of fields **sort** is to skip as part of the command argument.

To sort `phone.list` by the last names (field 2), type **sort +1 phone.list** and press **[Return]**. Figure 9.9 shows the result of sorting the file by the last names.

The **+1** argument indicates that **sort** must skip the first field (first name) before starting the sort process.

Figure 9.9
Sorted (by Last Names) `phone.list` File

```
$ sort +1 phone.list
Amir Afzal               (777) 777-7777
David Brown              (333) 111-1111
Bridget Erwin            (666) 555-5555
Mary Moffett             (555) 666-6666
Emma Redd                (222) 222-2222
Jim Schmid               (444) 444-4444
Tom Swanson              (111) 333-3333
$_
```

If you specify +2, then sort skips the first and second fields and starts from the third field (in this case, area code).

To sort `phone.list` by the third field, ignoring blanks, type **sort -b +2 phone.list** and press **[Return]**. Figure 9.10 shows the results.

Figure 9.10
Sorted (by Area Code) `phone.list` File

```
$ sort -b +2 phone.list
Tom Swanson              (111) 333-3333
Emma Redd                (222) 222-2222
David Brown              (333) 111-1111
Jim Schmid               (444) 444-4444
Mary Moffett             (555) 666-6666
Bridget Erwin            (666) 555-5555
Amir Afzal               (777) 777-7777
$_
```

9.5 STARTUP FILES

When you log in, the login program verifies your user ID and password against the list of authorized users stored in the password file. If the login attempt is successful, the login program brings your `HOME` directory up on the system, sets up your user ID and group ID, and finally starts your shell. Before displaying its prompt sign, the shell checks for two special files. These two files are called *profile files*, and they contain shell scripts (programs) that the shell can execute.

9.5.1 System Profile

The *system profile* file is stored in `/etc/profile`. The first thing your shell does is execute this file. It typically contains commands that display the message of the day, set up systemwide environment variables, and so on. This file is usually created and maintained by the system administrator, and only the superusers can modify it.

Figure 9.11 shows an example of a system profile file. The shell executes the commands in this file, so it displays the current date and time, then the message of the day (stored in the `/etc/motd file`), and, finally, the recent news items.

Figure 9.11
An Example of a Simple Profile File

```
$ cat /etc/profile
date
cat /etc/mtd
news
mesg n
stty erase ^H
export erase
$_
```

1. *Usually, the system* `profile` *file is complex and incorporates some system administration commands and requires some programming.*
2. *You can look at your system* `profile` *file by typing* **cat /etc/profile [Return]**. *Usually, this file is a read only file. You can read it, but cannot edit it.*

9.5.2 User Profile

Each time you log in, the shell checks for a startup file called `.profile` in your `HOME` directory. If the file is found, then the shell commands in `.profile` are executed. Whether or not you have a `.profile` file in your `HOME` directory, the shell continues its process and displays its prompt.

Figure 9.12 shows an example of a `.profile` file. Usually you have a `.profile` file, courtesy of the system administrator. You can modify the existing `.profile` file or create a new one using the **cat** or **vi** utilities.

Figure 9.12

An Example of the `.profile` File

```
$ cat .profile
   echo "welcome to my super Duper UNIX"
   TERM=ansi
   PS1="David Brown:"
   export TERM PS1
          calendar
   du
$_
```

1. The **echo** command displays its argument, *welcome to my super Duper UNIX.*
2. The standard variable *TERM* (terminal type) is set to *ansi.*
3. The standard variable *PSI* (primary prompt sign) is set to *David Brown.*
4. The **export** command makes the variables *TERM* and *PSI* available (exported to all programs).
5. The **calendar** and **du** commands are explained in Chapter 14.

1. *The name of the file is* `.profile`. *The filename starts with a dot; it is a hidden file.*
2. *The* `.profile` *file must be located in your* `HOME` *directory. This is the only place that the shell checks.*
3. *The* `.profile` *file is one of the startup files you can use to customize your UNIX environment. Other startup files exist in UNIX, such as the* `.exrc` *file that customizes the vi editor (discussed in Chapter 6) and the* `.mailrc` *file that customizes your mail environment (discussed in Chapter 10).*
4. *You do not have to have a* `.profile` *file in your* `HOME` *directory. Your system works without it. However, your shell usually inherits the setup from the* `/etc/profile` *file.*

More on the export Command

The **export** command makes a list of shell variables available to subshells. When you log in, the standard variables (and variables you may have defined) are known to your login shell. However, if you run a new shell, these variables are not known to the new shell.

For example, if you want to make the variables *VAR1* and *VAR2* available to the new shell, you specify the variable names as arguments for the **export** command. To see what variables are already exported, type **export** without any arguments.

Make variables available to other shell programs:

```
$ export VAR1 VAR2 [Return] . .  Export VAR1 and VAR2.
$ export [Return] . . . . . .    Check which variables are exported.
VAR1
VAR2
$_  . . . . . . . . . . . . . .  List of variables, then the prompt
                                 appears.
```

9.6 THE KORN AND BOURNE AGAIN SHELLS

Chapter 3 introduced the Korn shell (*ksh*) and Bourne Again shell (*bash*). Now we explain a few of the features that differ from those of the standard shell (*sh*). Once more, it is strongly recommended to use the **man** command and learn more about specific features of each of these shells. That is as easy as typing one of the following commands:

```
$ man sh [Return] . . .  Display manual pages for standard shell.
$ man ksh [Return]  . .  Display manual pages for Korn shell.
$ man bash [Return] . .  Display manual pages for Bourne Again shell.
```

9.6.1 The Shell Variables

The Korn shell (*ksh*) and Bourne Again shell *(bash)* uses many of the same variables used by the standard shell (*sh*). You can define variables, redefine variables, get their values, and in general manipulate variables to customize your environment just like the *sh* shell. The following are some important variables that are used by the Korn shell and Bourne Again shell:

ENV The *ENV* variable is set to the absolute pathname of the environment file that is read by *ksh* at startup. In the following example, *ENV* is set to the pathname (`$HOME/mine/my_env`) that tells *ksh* where to find the environment file:

```
ENV=$HOME/mine/my_env
```

HISTSIZE The *HISTSIZE* variable is set to the number of commands you intend to keep in your commands `history` list file. The default size is 128, but you can set it to any number of entries you wish. For example, the following command line sets the number of entries in your `history` list file to 100:

```
HISTSIZE=100
```

TMOUT The *TMOUT* variable is set to the number of seconds you want the system to wait before timing out if you do not type a command. The shell timeout logs you off if you do not provide any input within the given number of seconds. For example, the following command line sets timeout to 60 seconds:

```
TMOUT=60
```

VISUAL The *VISUAL* variable is used with command editing. If it is set to **vi**, *shell* gives you *vi-style* editing capabilities on your command lines. For example, the following command line sets the command line editing to the vi editor:

```
VISUAL=vi
```

Use the **set** *command, just like in the sh shell, to see the current shell variables and their values.*

9.6.2 The Shell Options

The Korn and Bourne Again shells provide a number of options that turn some of their special features *on* or *off*. To turn an option on, you use the **set** command with **-o** (option) followed by the option name. To list your options, simply type **set -o**.

noclobber The **noclobber** option prevents you from clobbering your files; that is, it prevents you from overwriting an existing file if you redirect output from a command. This can save you from losing important data by inadvertently overwriting your files.

Suppose a file called `xyz` exists in your current directory. The following command sequences demonstrate the use of the **noclobber** option.

```
$ set -o noclobber [Return] . .  Set the noclobber option.
$ who > xyz . . . . . .          Redirect the output to xyz.
xyz: file exists  . . . . . .    Warning message appears.
$_  . . . . . .                  The prompt appears.
```

If you really want to overwrite an existing file, *ksh* will oblige. Type a ¦ pipe symbol after the redirection operator.

```
$ who > ¦ xyz [Return]  . . . .  The xyz file is overwritten.
$_  . . . . . .                  The prompt appears.
```

To turn off the **noclobber** option, you use the **set** command with the **+o** option. For example,

```
$ set +o noclobber [Return] . .  The noclobber option is turned off.
$_  . . . . . .                  The prompt appears.
```

ignoreeof The **ignoreeof** option prevents you from accidentally logging yourself off by typing **[Ctrl-d]**. (You already know that **[Ctrl-d]** typed at the beginning of the command line terminates your shell, and logs you off from the system.)

If you set this option, you should use the **exit** *command to log off.*

```
$ set -o ignoreeof [Return] . . . .  Turn on the ignoreeof option.
$ [Ctrl-d]  . . . . . .               [Ctrl-d] is ignored. You are
                                      not logged off.
$ set +o ignoreeof [Return]  . . . .  Turn off the ignoreeof
                                      option.
$_  . . . . . .                       The prompt appears.
```

9.6.3 Command Line Editing

The Korn shell lets you edit your command line, or any of the commands in your `history` file, using the special line version of the vi editor. This feature enhances the use of the `history` file. It enables you to correct and modify the previous commands and makes it easier to search for a specific command in your commands `history` file. In short, it is good to know this command.

Turning on the Command Line Editing Option

You can turn on the command line editing option by using the **set** command or by setting the *EDITOR* or *VISUAL* variables to the pathname of your editor command. The effect of any of the following three commands is the same:

`$ set -o vi [Return]`	Turns on the command line editing option.
`$ EDITOR=/usr/bin/vi [Return]`	Turns on the command line editing option.
`$ VISUAL=/usr/bin/vi [Return]`	Turns on the command line editing option.

Using the vi-Style Command Line Editor

Assuming that you have turned on the command line editing feature, and that it is set to the vi editor, this section describes how you can use this very useful feature.

The *ksh* command line editor works on your current command line and your `history` file (explained later in this chapter). When you are entering a command, you are in the vi input (*text*) mode. This is opposite of the vi editor initial mode when you edit a file. You press the **[Return]** key to execute your command. As in the vi editor, you can switch to command mode at any time by pressing the **[Esc]** key. While in the *command mode,* the vi editor commands are available to change, delete, and correct your command line.

Now, type a command line and do not press the **[Return]** key. Instead, press the **[Esc]** key. This puts you in vi command mode.

```
$ This is a test to use the command line editing [Esc]
```

Let's say you forgot to type the **echo** command at the beginning of this command line. You can use vi editor keys to get to the beginning of the line. In this case, you type **$0** and the cursor will go to the letter T. You type **i** (insert) to return to vi text mode and so on.

1. *The command line vi editor is a special built-in vi editor.*
2. *You can use the* **j** *(down) and* **k** *(up) keys to access the commands from the* `history` *file.*
3. *You can use the* **l** *(right) and* **h** *(left) keys to move the cursor left or right on the command line.*

Remember, vi is in the input mode when you are entering a command. You must press the [Esc] key to change to the command mode before you can use the vi command keys such as k or j.

Table 9.10 lists some of the available commands with the built-in vi editor.

Table 9.10
The Built-In vi Editor Commands

Key	Operation
h and **1**	Moves cursor left and right one character on the command line.
k and **j**	Moves up and down one entry in the `history` list.
b and **w**	Moves cursor left and right one word on the command line.
$	Moves cursor to the end of the line.
x	Deletes the current character.
dw	Deletes the current word.
I and **i**	Inserts text.
A and **a**	Appends text.
R and **r**	Replaces text.

You can also invoke the *real* vi editor to use all the commands and features of it to edit the command line.

```
$ cp xyz xyz.bak . . .  Suppose you want to modify this command line.
[Esc] . . . . . .       Press the [Esc] key to get into the command mode.
v . . . . . .           Press the v key to invoke the real vi editor.
```

Now, you are using the vi editor with a file consisting of one line, your current command line. You can use vi to edit your command or add new commands. When you leave vi, the Korn shell will execute your commands.

You can use the vi editor in this manner to create a multiline sequence of commands to be executed.

9.6.4 The alias Command

You can use the **alias** command for shortening the names of frequently used commands or changing command names to names easier for you to remember. For example:

```
$ alias del=rm [Return]  . . . . . Now del is the alias for the rm
                                   command.
$ del xyz  . . . . . . . . . . . . Delete the file called xyz.
```

1. *Now you can type* **del** *instead of the* **rm** *command.*
2. *The* **rm** *command is not changed and you can still use it.*
3. *Aliases are defined the same way you define variables using the* **=** *sign.*

Set *ll* (*ell-ell,* for *long list*) as an alias for the command **ls -al**. Then you can type **ll** to get the listing of your current directory in long format.

```
$ alias ll="ls -al" [Return] . . . . Now ll is an alias for the ls -al
                                     command.
$_ . . . . . . . . . . . . . . . . . The prompt appears.
```

1. **As with the shell variables, there must be no spaces on either side of the = sign.**
2. **Also, if the assignment text includes spaces (like the above example, a command name and options), it must be enclosed in quotation marks.**

You can use the **alias** command with no argument to display the aliases that are set in your system:

```
$ alias [Return]  . . . . . Display a list of alias names.
alias ll ls -al . . . . . All alias names for your system are
                          displayed.
$_  . . . . . Back to the shell prompt.
```

You can use the **unalias** command to remove an alias name.

```
$ unalias ll [Return]  . . . . Remove alias name ll.
$ alias [Return]  . . . . . See if it is removed.
$_  . . . . . It is removed. The prompt is back.
```

9.6.5 Commands History List: The history Command

The Korn and Bourne Again shells have a commands history feature that keeps a list of all the commands you enter during your sessions. Using this very popular feature, you are able to list your previous commands, search for a particular command that you have issued, or easily edit and redo your previous commands.

The **history** command is one of the utilities that works on the commands `history` list. The **history** command or sometimes just letter **h** (usually an alias for the **fc** command) is used. If the **history** command is not working in your system (*ksh* or *bash*), try the **fc** command and its options or create your own alias for the **history** command. The **fc** command is explained later in this chapter and some of the examples will show you how to create aliases for the **history**, and **fc** commands.

To display the last few commands you have entered, do the following:

```
$ history [Return]  . . . . . Issue the command.
101 Who am i
102 ls -l
103 pwd
104 vi myfirst
105 rm myfirst
106 history
$_  . . . . . The prompt appears.
```

1. *The preceding example displays six lines of commands. The number of command lines that* ksh *or* bash *keep track of is controlled by the* HISTSIZE *variable.*
2. *The last item on the list is the last command you issued, and your earlier commands are higher on the list.*

The default `history` file is called `.sh_history` for *ksh* and `.bash.history` for *bash*, and is created by the system in your `HOME` directory. You may use another filename by setting the variable *HISTFILE* to the pathname of your desired `history` filename. For example:

```
HISTFILE=$HOME/history/my_hist
```

Restarting the History File

The history of your commands is kept from one session to another and consequently the entry numbers will become large and cumbersome to retype or refer to. If you want to restart your `history` file, remove your `.sh_history` or file from your `HOME` directory. The next time you log in, the system creates a new `.sh_history` or `.bash.history` file and the entries start from the first command you issue in that session.

Practicing the History Command

You can display the commands `history` list by starting it from a specified command in the list:

```
$ history 104 [Return] . . . . . Start from command number 104.
     104  vi myfirst
     105  cat myfirst
     106  history
     107  history 104
$ history vi [Return] . . . . . Start from first occurrence of vi in
                                the list.
     104  vi myfirst
     105  cat myfirst
     106  history
     107  history 104
     108  history vi
$_
```

Use the line editor to edit a command from the commands history list:

```
$ set -o vi
$ history [Return] . . . . . List the commands.
     104  vi myfirst
     105  cat myfirst
     106  history
     107  history 104
     108  history vi
     109  history
$_ . . . . . Prompt is back.
[Esc] . . . . . Press the [Esc] key.
j . . . . . Press the letter j to get to go one
                                command up on the list.
history vi
G . . . . . Show the last command in the
                                history file.
history
105G . . . . . Show the 105th command in
                                the history file.
cat myfirst
[Return] . . . . . Execute the command.
```

Your `history` *file probably will be different from the examples shown here. Your* `history` *file does not remain the same, and each command that you type is added to the list of commands already in the* `history` *file. In the following examples no specific* `history` *file is assumed.*

9.6.6 Redoing Commands (ksh): The r (redo) Command

You can use the **r** (*redo*) command to redo the last command you have issued. For example, suppose your last command is **rm myfirst** and you type the following:

```
$ r [Return]  . . . . . Repeats the last command.
rm myfirst  . . . . . . Your last command is repeated.
```

In this case, your last command is executed. The following command sequences show the **r** (*redo*) command options and features.

You can repeat other commands from the `history` file by adding the specific command name as an argument to the **r** command:

```
$ r vi [Return] . . . . . Repeat vi from the history list.
vi myfirst  . . . . . . . The first occurrence of the vi command in the
                          history is executed.
```

You can repeat any command from the `history` file by indicating the specific command entry number:

```
$ r 102 [Return]  . . . . Repeat command number 102 from the
                          history list.
ls -l . . . . . . . . . . The specified command is executed.
```

You can also repeat commands from the `history` file by indicating the number of entries you want to go back in the list.

```
$ r -3 [Return] . . . . . Go back three entries in the history list.
ls -l . . . . . . . . . . The specified command is executed.
```

9.6.7 Commands History List: The fc Command

The **fc** command provides capability to list, edit, and reexecute commands that were previously entered and were saved in the commands `history` list. For example, the following command line lists the previous commands from the `history` file:

```
$ fc -l [Return]  . . . . Same as the history command.
```

Figure 9.13 shows the output of the **fc** command. However, your command `history` list almost certainly would be different.

Figure 9.13
Output of the **fc** Command

```
$ fc -l
101  Who a m I
102  ls -l
103  pwd
104  vi myfirst
105  cat myfirst
106  history
107  fc -l
$_
```

1. *In the commands* `history` *list, each command is referenced by a number, and the list usually starts from 1.*
2. *When the number of commands reaches the value in* `HISTSIZE` *(default is 128), the shell may wrap the numbers, starting the next command with 1.*
3. *If the* `HISTSIZE` *variable is set when the shell is invoked, then this file is used to store the commands history.*
4. *If the* `HISTSIZE` *variable is set when the shell is invoked, then this number dictates the number of entries into the commands history. The default is 128.*

fc Options

The **fc** command is rich with options that provide you with many possibilities for editing and re-executing previous commands. Table 9.11 lists few of these options. Please use the **man** command to see a detailed list of the options.

Table 9.11
The **fc** Command Options

Option	Operation
-l	Lists the commands, with each command preceded by the command number.
-n	Suppresses command numbers when listing with **-l**.
-r	Reverses the order of the commands listed with **-l**.
-s	Re-executes the command without invoking an editor.

The following examples show the usage of the **fc** command options:

```
$ fc -l [Return]. . . . . Display the commands history list.
$ fc -l -n [Return] . . . . . Display the commands history list
                              without the command numbers.
$ fc -l -r [Return] . . . . . Display the command history list in
                              reverse order.
$ fc -s [Return]. . . . . Execute the previous command.
$ fc -s 107 [Return]. . . . . Execute command number 107 from the
                              history list.
$ fc -s vi [Return] . . . . . Execute the first occurrence of vi in the
                              history list.
$ fc -s c [Return]. . . . . Execute the first occurrence of the
                              command starting with the letter c.
```

Creating Aliases for the fc Command

The following command lines are suggestions to create aliases for the **fc** command. In case your system does not provide the **history** command, the following two commands creates aliases that behave like the **history** command:

```
$ alias r='fc -e -' [Return]. . . . Same as the r command.
$ alias history='fc -l' [Return]. . Same as the history command.
```

You can choose any names you want for the aliases and not necessarily the ones that duplicate the **history** command. For example:

```
$ alias hr='fc -e -' [Return] . . .   Same as the history r
                                      command.
$ alias h='fc -l' [Return] . . . . .  Same as the history
                                      command.
```

Now you can type a command such as:

```
$ h [Return] . . . . . . . . . . . .  Same as fc -l.
$ r 107 [Return] . . . . . . . . . .  Same as fc -s 107.
```

You can remove any of the aliases by using the **unalias** command.

9.6.8 Login and Startup

Like the standard shell (*sh*), the Korn shell (*ksh*) or Bourne Again shell (*bash*) reads the `.profile` file in your `HOME` directory when you log in. It executes the commands you want to run at login time, and initializes the variables that will be in effect for your login session. The `.profile` file typically includes commands such as **date**, **who**, and **calendar**, which provide information at login, terminal settings, and variable definitions that you want to export to the environment.

In addition to the `.profile` file (in your `HOME` directory), *ksh* or *bash* also reads your environment file. The environment file does not have a predefined name or location that *ksh* or *bash* looks for. You define its name and location with the *ENV* variable in your `.profile` file. For example, if your `.profile` file contains the line

```
ENV=$HOME/mine/my_env
```

shell will look for your environment file in the file named `my_env` in a subdirectory called `mine` in your `HOME` directory. Although it is not necessary, it is a good practice to call your environment file `.kshrc` or `.bashrc` (a hidden file) in your `HOME` directory.

```
ENV=$HOME/.kshrc
```

or

```
ENV=$HOME/.bashrc
```

1. *If your login shell is* ksh *you can specify all the shell variables and options in the* `.profile` *file in your* `HOME` *directory.*
2. *If your login shell is not* ksh *or* bash*, define all the specific shell variables an options in a file specified by the* ENV *variable. For the system to read your environment file, you must have the* ENV *variable defined in your* `.profile` *file.*

Figure 9.14 shows (using the **cat** command) an example of an environment file called `.kshrc`. This `.kshrc` file contains commands to set the vi-style command line editing, to turn on the **noclobber** and **ignoreeof** options, to set the `history` file size to 10 entries, and to set the *TMOUT* option to 600. You can create a similar file using the vi editor.

Figure 9.14
An Example of an Environment File Called `.kshrc`

```
$ cat .kshrc
   set -o vi
   set -o noclobber
   set -o ignoreeof
   HISTSIZE=10
   TMOUT=600
$_
```

9.6.9 Adding Event Numbers to the Prompt

Sometimes, it is useful to know the event number the shell gives to each command you enter. You can change your prompt to include this information. For example, if you type

```
PS1="!$"
```

the exclamation mark (**!**) will tell the system to read the last event number from your `history` file, add one to it, and display it. The prompt will continue displaying the event numbers as you type your commands.

Change your prompt to display the event numbers:

```
$ PS1="! $" [Return]  . . . . . Change the prompt.
6 $_  . . . . . . . . . . . . . The new prompt appears.
```

The new prompt indicates that the next command you enter will have the event number 6.

```
$ PS1="[!] $ "[Return]  . . . . Change the prompt this way.
[6] $_  . . . . . . . . . . . . The new prompt appears.
```

By adding your prompt definition to the `.kshrc` *file, you can make it appear each time you log in.*

9.6.10 Formatting the Prompt Variable (bash)

In addition to displaying static character strings in the prompts, *bash* provides a list of predefined special characters that can be used in formatting the prompt. These special characters place things such as the current time into the prompt. Table 9.12 lists some of these special character codes.

The following command lines changes the prompt display using the prompt special character codes:

```
$ PS1="[\!]$ " [Return] . . . . Display the command number.
[72]$_
```

Table 9.12
Special Character Codes to Format the Prompt

Character	Meaning
\!	Displays the history number of the current command.
\$	Displays a $ in the prompt unless the user directory is root. When user is root, it displays a #.
\d	Displays the current date.
\s	Displays the name of the shell that is running.
\t	Displays the current time.

```
$ PS1="[\d]$ " [Return] . . . .   Display the current date between
                                  brackets.
[Thu Nov 2]$_

$ PS1="\d $ " [Return]  . . . .   Display the current date.
Thu Nov 2 $_
$ PS1="[\!][\t]$ " [Return] . .   Display the command number and
                                  current time.
[79][20:33:41]$_
```

Note that the **[ ]** are not part of the command syntax or special codes. They are used here to make the prompt look nicer and more readable.

```
$PS1="\s$" [Return] . . . . . .   Display the shell name.
bash$_
```

9.7 UNIX PROCESS MANAGEMENT

In Chapter 3, we introduced the process of booting the system. Now let's go deeper into the UNIX internal process and see how it manages the running of programs.

In this chapter, you have encountered the word *process*. The execution of a program is called a *process:* you call it a *program,* but when your program is loaded into the memory for execution, UNIX calls it a *process.*

To keep track of the processes in the system, UNIX creates and maintains a process table for each process in the system. Among other things, the process table contains the following information:

- Process number
- Process status (ready/waiting)
- Event number that the process is waiting for
- System data area address

A process is created by a system routine called **fork**. A running process calls **fork**, and in response UNIX duplicates that process, creating two identical copies. The process that calls the **fork** routine is called the *parent,* and the copy of the parent created by **fork** is called the *child.* UNIX differentiates between the parent and the child by giving them different process IDs (PIDs).

The following steps are involved in managing a process:

- The parent calls **fork**, thus starting the process.
- Calling **fork** is a system call. UNIX gets control, and the address of the calling process is recorded in the process table's system data area. This is what is called the *return address,* so the parent process knows where to start later when it gets control again.
- **fork** duplicates (copies) the process and returns control to the parent.
- The parent receives the PID of the child, a positive integer number, and the child receives the return code zero. (A negative code indicates an error.)
- The parent receiving a positive PID calls another system routine called **wait** and goes to sleep. Now the parent is waiting for the child process to finish (in UNIX terminology, waiting for the child to die).
- The child process gets control and begins to execute. It checks the return code; because the return code is zero, the child process calls another system routine called **exec**. The **exec** routine responds by overlaying the child process area with the new program.
- The new program's first instruction is executed. When the new program gets to the end of the instruction, it calls yet another system routine called **exit**, and the child process dies. The death of the child awakens the parent, and the parent process takes over.

This process is depicted in Figures 9.15 through 9.18. An example is in order to shed some light on this apparently confusing process. Imagine that the shell (the sh program) is running, and you type a command, say **ls**. Let's explore the steps UNIX takes to run your command.

The shell is the parent process, and when created, the **ls** program becomes the child process. The parent process (shell) calls **fork**. The **fork** routine duplicates the parent (shell) process, and if the creation of the child process is successful, assigns the child process a PID and adds it to the system process table. Next, the parent receives the child PID, the child receives code zero, and control is returned to the parent. The shell calls the **wait** routine and goes to the wait state (goes to sleep). Meanwhile, the child gets control and calls **exec** to overlay the child process area with the new program—in this case **ls**, the command you typed. Now **ls** carries out the command. It lists your current directory filenames, and when it is finished processing, it calls **exit**. Thus the child dies. The death of the child generates an event signal. The parent process (shell) is waiting for this event. It is awakened and gets control. The shell program continues, starting execution from the same address it was at before going to sleep (recall that this address was stored in the process table system data area as return address), and the prompt is displayed.

What happens if the child is a background process? In that case, the parent (shell) does not call the **wait** routine; it continues in the foreground, and you see the prompt right away.

What creates the first parent and child processes? When UNIX is booted, the **init** *process is activated. Next,* **init** *creates one system process for each terminal. Thus,* **init** *is the original ancestor to all the processes in the system. For example, if your system supports 64 concurrent terminals, then* **init** *creates 64 processes. When you log in to one of these processes, the login process executes the shell. Later, when you log out (when the shell dies),* **init** *creates a new login process.*

Figure 9.15
Events Happening When **fork** is Called

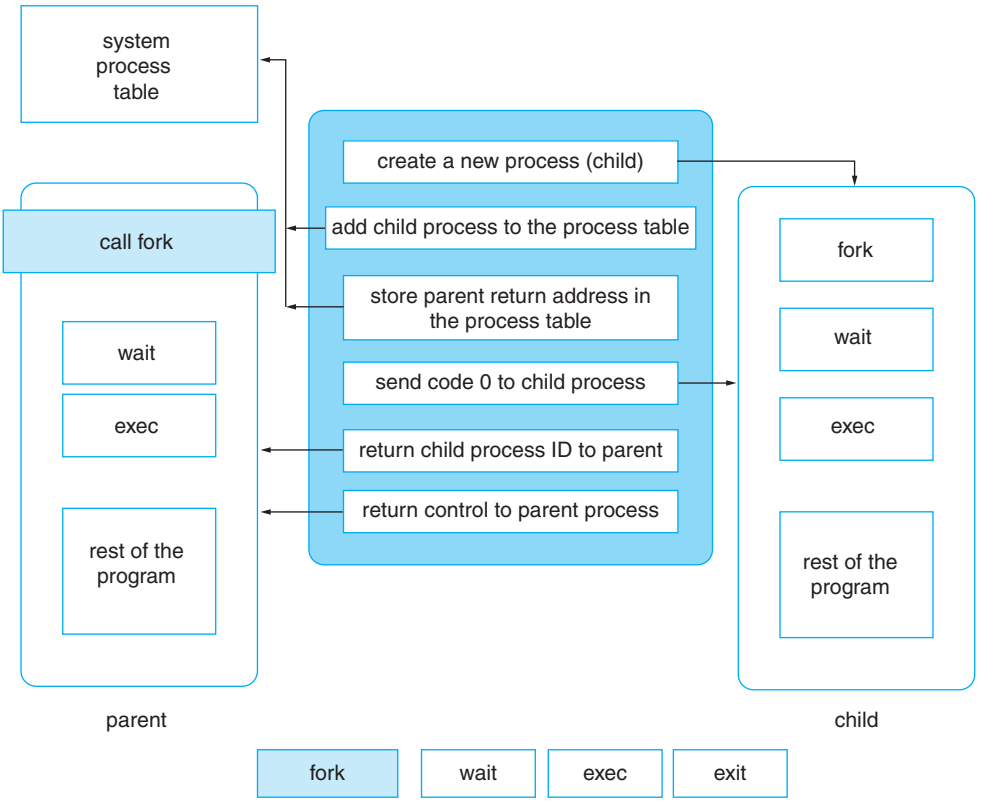


Figure 9.16
Events Happening After **wait** is Called

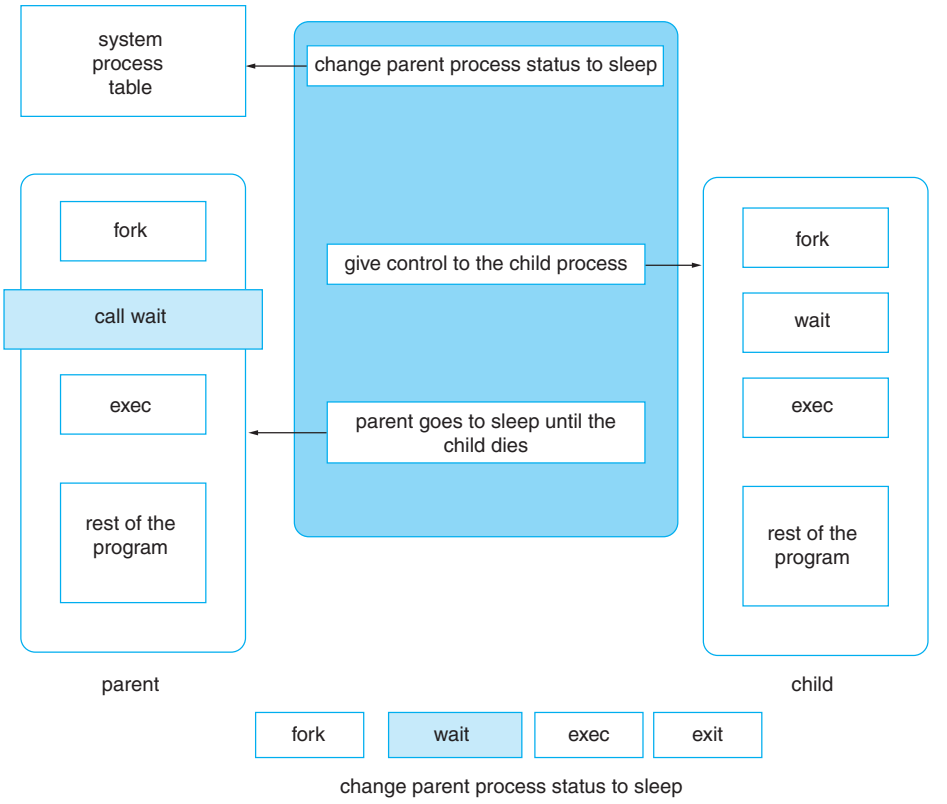


Figure 9.17
Events Happening When **exec** is Called

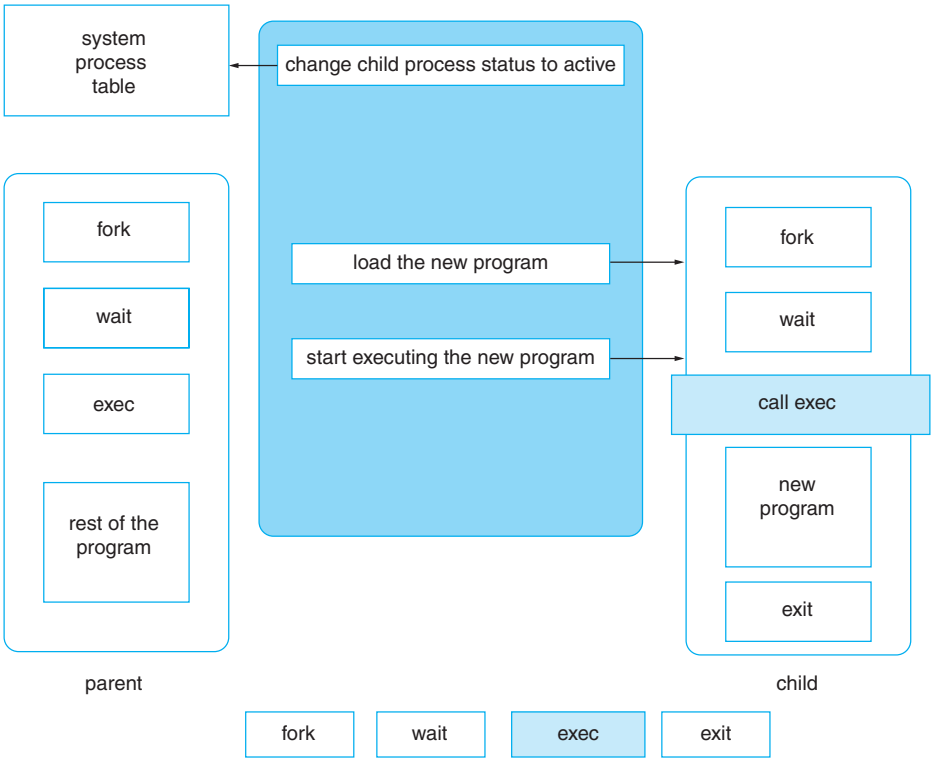
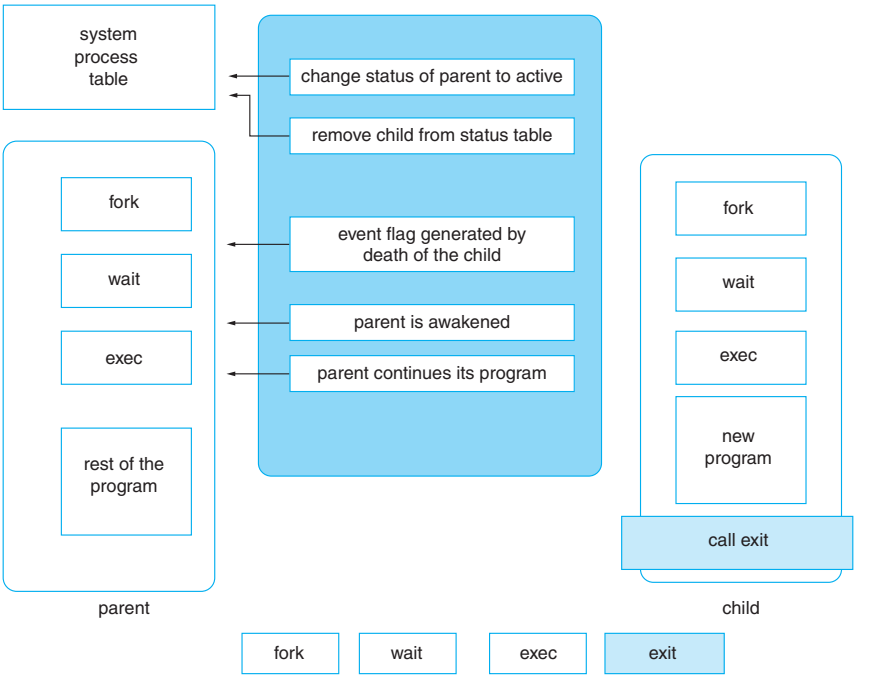


Figure 9.18
Events Happening When Child Calls **exit**



COMMAND SUMMARY

The following UNIX commands have been discussed in this chapter.

alias
This command creates aliases (names) for commands.

echo
This command displays (echoes) its arguments on the output device.

Escape Character	Meaning
\a	Audible alert (bell)
\b	Backspace
\c	Inhibit the terminating newline
\f	Form feed
\n	Carriage return and a line feed (newline)
\r	Carriage return without the line feed
\t	Horizontal tab
\v	Vertical tab

export
This command makes a specified list of variables available to other shells.

fc
This command provides capability to list, edit and re-execute commands that previously were entered and were saved in the history file.

Option	Operation
-l	Lists the commands, with each command preceded by the command number.
-n	Suppresses command numbers when listing with **-l**.
-r	Reverses the order of the commands listed with **-l**.
-s	Re-executes the command without invoking an editor.

grep (Global Regular Expression Print)

This command searches for a specified pattern in file(s). If the specified pattern is found, the line containing the pattern is displayed on your terminal.

Option		Operation
UNIX	Linux Alternative	
-c	**--count**	Displays only the count of the matching lines in each file that contains the match.
-i	**--ignore-case**	Ignores the distinction between lower and uppercase letters in the search pattern.
-l	**--files-with-matches**	Displays the names of the files with one or more matching lines, not the lines themselves.
-n	**--line-number**	Displays a line number before each output line.
-v	**--revert-match**	Displays only those lines that do not match the pattern.
	--help	Displays help page and exits.
	--version	Displays version information and exits.

history

This command is a Korn and Bourne Again shell feature that keeps a list of all the commands you enter during your sessions.

kill

This command terminates an unwanted or unruly process. You have to specify the process ID number. The process ID 0 kills all programs associated with your terminal.

nohup

This command prevents the termination of the background process when you log out.

ps (process status)

This command displays the process ID of the programs associated with your terminal.

Option	Operation
-a	Displays the status of all the active processes, not just the user's.
-f	Displays a full list of information, including the full command line.

r (*redo*)
This command is a Korn shell command that repeats the last command or commands from the `history` file.

set
This command displays the environmental/shell variables on the output device. The command **unset** removes the unwanted variables.

sleep
This command causes the process to go to sleep (wait) for the specified time in seconds.

sort
This command sorts text file(s) in different orders.

Option	Operation
-b	Ignores leading blanks.
-d	Uses dictionary order for sorting. Ignores punctuation and control characters.
-f	Ignores the distinction between lowercase and uppercase letters.
-n	Numbers are sorted by their arithmetic values.
-o	Stores the output in a specified file.
-r	Reverses the order of the sort from ascending to descending.

tee
This command splits the output. One copy is displayed on your terminal, the output device, and another copy is saved in a file.

Option	Operation
-a	Appends output to a file without overwriting an existing file.
-i	Ignores interrupts; does not respond to the interrupt signals.

REVIEW EXERCISES

1. What are the major functions of the shell?
2. What is the name of your system shell program, and where is it stored?
3. What are the metacharacters? How does the shell interpret them?
4. What are the quoting characters?
5. What are the shell variables?
6. What is the command to display the environment/shell variables?
7. What is the command to remove a variable?
8. Name some of the environment/standard variables.
9. What are the variables, and what role do they play?
10. How do you run a program in the background?
11. How do you terminate a background process?
12. What is the process ID number, and how do you know the process ID of a particular process?
13. What is the pipe operator, and what does it do?
14. How do you prevent termination of your background process after you log off?
15. What is the command for searching for a specified pattern in a file?
16. How do you delay the execution of a process?
17. What is the operator that groups the commands together?
18. What is the startup file?
19. What is the `.profile` file, and what is the `profile` file?
20. What are the parent and child in reference to UNIX process management?
21. What is a process?
22. What is the command to activate the command line editor?
23. What variable is set to change the size of the `history` file?
24. How do you make your `history` file start from event 1?
25. Which file does the Korn shell read at startup?
26. What is the command to repeat your last command?
27. What is the command to repeat event number 105 from your `history` file?
28. What is the command to set an alias for a command name?
29. What is the command to export a list of variables to the other shells?
30. What is the command to list aliases?
31. What is the command to display a list of the files in your directory and save a copy in another file?
32. What is the command to obtain a detailed description of the **alias** command?
33. What is the effect of setting the shell option **noclobber**?
34. What is the effect of setting the shell option **ignoreeof**?

Terminal Session

In this terminal session, you will practice the commands explained in this chapter. The following exercises are only some suggestions of how to use the commands. Use your own examples, and devise different scenarios to master the use of these commands.

1. Use the **echo** command to produce the following outputs:
   - **a.** Hello There
   - **b.** Hello

     There
   - **c.** "Hello There"
   - **d.** These are some of the metacharacters:

     ? * [ ] & ( ) ; > <
   - **e.** Filename: file? Option: all
2. Use the **echo** command and other commands to produce the following outputs:
   - **a.** Display the contents of your current directory. Have a header that shows a short prompt and the current date and time before listing your directory.
   - **b.** Show the message "I woke up" with a two-minute time delay.
3. Change your primary prompt sign.
4. Create a variable called *name* and store your first and last names in it.
5. Display the contents of the variable name.
6. Check whether you have a `.profile` file in your `HOME` directory.
7. Create a `.profile` file or modify your existing one to produce the following output each time you log in:

   ```
   Hello there
   I am at your service David Brown
   Current Date and Time: [the current date and time]
   Next Command:
   ```
8. Create a background process, check its process ID, and then terminate it.
9. Create a background process. Use the **nohup** command to prevent the termination of the background process.
10. Create a phone list. Let's say you gather the names and phone numbers of ten of your classmates. Use the **sort** command to sort this list in different orders: by first name, by last name, by phone number, in reverse order, and so on.
11. Use **grep** and its options to find a particular name in your phone list.
12. Use the **kill** command to log off.
13. If your shell is the Korn shell (*ksh* or *bash*), set up the following variables and practice the *ksh* commands:
    - **a.** Set the `history` file size to 50.
    - **b.** Activate the command line editor.
    - **c.** Use the built-in vi editor commands to access commands in your `history` file.
    - **d.** Use the built-in vi editor commands to edit/change the command line.

e. Repeat your last command.

f. Display the list of your previous commands.

g. Repeat the first command in the `history` file.

h. Start a new `history` file.

i. Create an alias for the delete command with the confirmation option (**rm -i**).

j. Create a `.kshrc` file that contains the setup for *ks*h variables and aliases.

14. Change your prompt to show the command number.

15. Change your prompt to show the name of the shell.

16. Create an alias called **ls** for **ls -l**.

17. Create an alias called **rm** for **rm -i**.

18. Set the online editor shell option. Check it to see if it works.

19. Set the **noclobber** shell option. Check it to see if it works.

20. Set the **ignoreeof** shell option. Check it to see if it works.