The Shell

Redirection and Pipes

The Command Line

Special Characters and File Name Arguments

Redirection and the Standard Input/Output

Pipes

Redirecting and Piping the Stnadard Errori

Jobs: background, kills, and interruptions

7. The Shell

The shell is a command interpreter that provides a line-oriented interactive interface between the user and the operating system. You enter commands on a command line that are then interpreted by the shell and sent as instructions to the operating system. This interpretive capability of the shell provides very powerful features. For example, the shell has a set of metacharacters that can generate file names. It can redirect input and output. It can also run operations in the background, freeing you to perform other tasks. In the following chapters we shall see how you can edit the shell's command line, create aliases for Unix commands, keep a history of prior commands that you can execute and edit, and create shell scripts of Unix commands as well as configure the shell using system variables.

Several different shells have been developed for Unix, including the Bourne shell, the C-shell, the Korn shell, the BASH shell, the TCSH shell, and the Z-shell. The Bourne shell was one of the first designed for Unix. It contains all the basic operations found in all Unix systems and was widely used for early releases of System V. The C-shell was created by Bill Joy for BSD Unix and incorporates many new features such as history and aliasing. The Korn shell was later developed by Dave Korn as an extension of the Bourne shell, incorporating many of the C-shell features as well as several of its own. The Korn shell has all the same commands as the Bourne shell, even the same prompt. The Bourne shell and C-shell share the same core commands such as `ls` and `cp`, but differ in more complex features such as shell programming. The BASH shell (Bourne-Again Shell) integrates many of the features of all three shells but retains the command line syntax of the Bourne shell. The TCSH shell is an enhancement of the C-shell, and the Z-shell is an enhancement of the Korn shell.

Recent versions of Unix will have most of these shells. You can freely move from one type of shell to another, taking advantage of the different features in each. The Bourne shell is invoked with the sh command, the C-shell with the csh command, the Korn shell with the ksh command, and the BASH shell with the bash command, the TCSH shell with the tcsh command, and the Z-shell with the zsh command. You leave each shell with the Ctrl-d or exit commands.

You really need to use only one type of shell to do your work. In fact, if you have an earlier version of Unix you may be limited to only one of the shells. Early versions of System V have only the Bourne shell. BSD Unix uses the C-shell as well as the Bourne shell. Though this chapter discusses features common to all shells, later chapters branch off into features unique to either the Korn shell or C-shell. It will help to first know which shell you will be using, and then to examine that one in detail.

This chapter will discuss the basic features of the command line: entering commands, metacharacters, redirection, pipes, and job control. All of these, with the exception of job control, are core operations found in all three shells. Technically it is a discussion of the Bourne shell, but the operations discussed are the same for the C-shell, BASH shell, and Korn shell as well as the TCSH shell and Z-shell.

The Command Line

When the shell prompt appears, you are logged into the system. The prompt indicates the beginning of the command line. This is the line on which you enter your commands. You enter a command by typing in the command name and its arguments at the prompt. The command is not executed, however, until you press the enter key. In the next example the user has entered in the date command. The `date` command displays the date. The command is first typed in and then the enter key is hit to execute the command.

```
$ date
Sun July 5 11:42:36 PST 2006
```

There is a syntax that the shell follows for interpreting the command line. The first word entered on a command line must be the name of a command. The next words are options and arguments for the command. Each word on the command line must be separated by one or more spaces or tabs. The shell first reads the command name and then checks to see if there is an actual Unix command by that name. If there is no such command, the shell issues an error message.

`$` *Command Options Arguments*

Options modify the command's execution. An option is represented by a single character and is preceded by a dash. `-F` is an option for the `ls` command. With this option, the `ls` command displays directory names with a preceding slash so that you can easily identify them. Options are entered on the command line before the arguments. In the case of the `cp` command, the `-i` option for checking the overwrite condition is entered before the file name arguments. Options are, of course, optional. You can enter the `ls` or `cp` commands without any options.

Arguments are usually file names. Depending upon the command, you may or may not have to enter in arguments. Some commands, such as `ls`, do not require any arguments. Others, such as `cp`, always require an exact number of arguments. In the case of the `cp` command, two arguments are required: the name of the source file and the name of the copy file. If the number of arguments does not match the number required by the command, then the shell issues an error message.

`$ ls`	*Command without options*
`$ ls -F`	*Command with options*
`$ cp -i mydata newdata`	*Command with options and arguments*

The line on which you enter commands is actually an editable buffer of text. Before you hit the enter key, you can perform editing commands on the text of the command you have just typed. The basic editing capabilities are limited but they do provide a way to correct mistakes. The BASH, TCSH, and Z-shell all have more advanced and easy to use editing features not available in the Bourne or C-shell (see

Chapter 10). Table 7-1 lists the different command line editing capabilities.

The erase keys, such as Ctrl-h and the backspace or delete keys, allow you to erase the character just typed in. With this character-erasing capability, you can backspace over your entire line if you wish, erasing what you have entered.

```
$ datl      If you make a mistake and enter the wrong character,
$ dat       use the Ctrl-h or backspace key to erase the character
$ date      and then continue typing in the command.
```

Sometimes you may need to erase the entire line and start over. The Ctrl-u key erases the entire text on the command line and lets you start over again at the prompt.

```
$ dfater
$           Having entered text, hitting the Ctrl-u key erases the line
```

There may be times when you mistakenly execute the wrong command. Some commands can be very complex and take some time to execute. You can interrupt and stop such commands with the the interrupt keys, Ctrl-c or delete.

You are not limited to one command per line. You can place more than one command on the same line or you can use several lines to enter in a single command. You can enter more than one command on the same line by separating the commands with a semicolon. In the next example, the `ls` command and the `cp` command are entered in on the same line.

```
$ ls -F ; cp -i mydata newdata
```

Should you ever need to, you can enter a command on several lines by entering a backslash just before you hit the enter key. The backslash escapes the enter key, effectively continuing the same command line into the next line. In the next example, the `cp` command is entered in on three lines. However, the first two lines end in a backslash, effectively making all three one command line.

```
$ cp -i \
mydata \
newdata
```

*Metacharacters and File Name Arguments: *, ?, [ ]*

Many of the arguments used for Unix commands are file names. There may be situations where you may know only part of the file name, or you may want to enter several file names that have the same extension or prefix. Unix provides a set of metacharacters that search out, match on, and generate a list of file names. These metacharacters are the

asterisk, question mark, and brackets (`*`, `?`, `[ ]`) and are also referred to as wildcard characters. They are used by the shell as matching operators to search for files given a partial name and generate a list of file names found. The shell will then replace the partial file name argument with the list of matched file names. This list of files names can then become the arguments for commands such as `ls` that can operate on many files. Table 7-1 lists the different shell metacharacters.

The asterisk metacharacter is designed to match on file names given only a part of the name such as a prefix or suffix. If you want files that begin with a certain set of characters you can match them by just adding a `*` at the end. The set of characters form a pattern to be searched for in file names. You can place the asterisk before or after a set of characters, or both. If the asterisk is placed before the pattern, then file names that end in that pattern are searched for, whereas if the asterisk is placed after the pattern, then file names that begin with that pattern are searched for. Any file name that matches is copied into a list of file names generated by this operation. In the next example, all file names beginning with the pattern "art" are searched for and a list generated. Then all file names ending with the pattern "ter" are searched for and a list generated.

```
$ ls
art1 art2 article arts myart story1 letter
$ ls art*
art1 art2 article arts
$ ls *ter
letter
$
```

The asterisk is often used to generate lists of files that have a specific extension. You can give a file name an extension as part of its name when you create it. An extension begins with a period and followed by a one or more characters. For example, C program files have an extension **.c** as in **main.c** or **lib.c**. The extension has no special status. It is only part of the characters making up the file name. In the next example, the asterisk is used to list only those files with a **.c** extension. The asterisk placed before the **.c** constitutes the argument for `ls`.

```
$ ls *.c
lib.c main.c
```

The `rm` command will erase files and when you use `*` in the file name argument you can easily erase several files at once. If you use the `*` by itself as the file name argument, then all your files in that directory will be erased. In the next example, the `rm` command erases all files beginning with the pattern "art'. Then the `rm *` command erases all files in the working directory.

```
$ ls *
art1 art2 article arts myart myletter yourletter
$ rm art*
$ ls *
myart myletter yourletter
$ rm *
$ ls
$
```

You need to be careful whenever you use the * metacharacter in an rm command. With a misplaced * in an rm command without the ii option, you could accidentally erase all the files in your working directory. The first command in the next example erases only those files with a .c extension. The second command, however, erases all files in the working directory. The difference between the two commands is the space between the asterisk and the period, * .c in the second command. A space in a command line functions as a delimiter, separating arguments. The asterisk is considered one argument and the .c another. The asterisk by itself matches all files and, when used as an argument with the rm command, instructs rm to erase all your files.

```
$ rm *.c
$ rm * .c
```

Whereas the * will match any number of characters, the question mark, ?, is designed to match only a single character. Otherwise it functions much like the *. You can have several ? in your pattern, placing them at the beginning, end, or in the middle of the characters you are searching for. For example, suppose that you want to match on the files **art1** and **artA**, but not on article. Whereas the asterisk would match file names of any length, the question mark limits the file names matched to just one extra character. In the next example, the user matches files that begin with the word "art" followed by a single differing letter. Then the user searches for a pattern with three possible differing characters.

```
$ ls
art1 artA article
$ ls art?
art1 artA
$ ls ?y?ar?
myarts mylark Syart1
```

The ? can be combined with other metacharacters to construct more precise matching operations. For example, suppose you want to find all files that have a single character extension. You could use the asterisk to match the file name proper, and the ? to match on the single character extension: *.?. In the next example, the all files that have a single character extension are displayed.

```
$ ls *.?
main.c  lib.a
```

Whereas the * and ? metacharacters specify incomplete portions of a file name, the brackets metacharacters allows you to specify a set of valid characters to search for. Characters placed within a set of brackets metacharacters, [], specify a set of characters to be matched. Any letter placed within the brackets will be matched in the file name. For example, suppose you want to list files beginning with "art" but only ending in a 1 or an A. You are not interested in file names ending in 2 or 'B' or any other character. In the next example the user lists only those file names beginning with "art" and ending in either a '1' or an 'a'.

```
$ ls
art1 art2 art3 artA artB artD article
$ ls art[1A]
art1 artA
```

Rather than listing characters one by one, you can specify a range of characters. A dash placed between the upper and lower bound of a set of characters, selects all characters within that range. The range is usually determined by the characters set in use. In an ASCII character, set the range a-g will select all lower-case alphabetic characters from 'a' through 'g'. In the next example, a file beginning with the pattern art and ending in characters '1' through '3' are selected. Then those ending in characters 'b' through 'e' are matched.

```
$ ls art[1-3]
art1 art2 art3
$ ls art[B-E]
artB artD
```

Combining the brackets with other metacharacters, you can create very flexible matching patterns. For example, suppose you only want to list file names using a specific set of possible extensions. If you wanted to list file names ending in either a .c or .o extension, but no other extension, you can use a combination of the asterisk and the brackets: *.[co]. The asterisk matches all file names, and the brackets match only file names with extension 'c' or 'o'.

```
$ ls *.[co]
main.c  main.o  calc.c
```

The shell will always treat the *, ?, and [] symbols as metacharacters, even if they happen to be part of a valid file name. For example suppose you have a file name called **answers?**, where the ? character is part of the name. Each time you entered **answers?** on the command line, the shell will take the ? as a metacharacter and match all files beginning with "answers" and having one extra letter, such as **answers1 answersF**. You can, have the shell treat any of the metacharacters as simple characters by quoting

them. You can quote a metacharacter by preceding it with a backslash. Whenever you have a metacharacter as part of the name of a file, you would need to quote the metacharacter in order to reference the file. In the case of the **answers?** file, you need to quote the ? with a preceding backslash, **answers\?**.

```
$ ls answers\?
answers?
```

You can, of course, combine a quoted character with metacharacters in your file name. In the next example, the user lists all files beginning with **answers?** that have an extension.

```
$ ls answers\?.*
answers?.quiz  answers?.mid  answers?.final
```

TABLE 7-1

Command Line and Metacharacters

Command execution

Enter	Execute a command line.
;	Separate commands on the same command line.
command\ *opts args*	Continue entering a command on the next line by entering a backslash before Enter
`*command*`	Execute a command.
backspace	Erase the previous character
Ctrl-h	Erase the previous character
Ctrl-u	Erase the command line and start over.
Ctrl-c	Interrupt and stop a command execution.

Metacharacters for file name generation.

*	Match any set of characters.
?	Match any single characters.
[]	Match a class of possible characters.
\	Quote the following character. Used to quote metacharacters. Allow you to use the character literally.

Redirection and the Standard Input/Output: >, >>, <

In Unix, there is a difference between the physical implementation and logical organization of a file. Unix files are accessed physically as randomly arranged blocks. However, all files are organized logically as a continuous stream of bytes. Aside from special system calls, the physical implementation remains hidden from the user. The user deals with the logical organization. All files appear to the user as a continuous stream of bytes, accessible one after the other. Because all files have this same byte stream organization, any file can be easily copied or appended to another. In this sense, there is only one standard type of file in Unix, the byte stream file.

This same byte stream organization of a file is extended to input and output operations. The data in input and output operations are organized like a file. Data that are input through the keyboard are placed in a data stream arranged as a continuous set of bytes. Data output from a command or program are also placed in a data stream and arranged as a continuous set of bytes. This input data stream is referred to in Unix as the standard input, and the output data stream is called the standard output. Figure 7.1 shows the basic relationship between the standard input and the keyboard device, and between the standard output and the screen device.

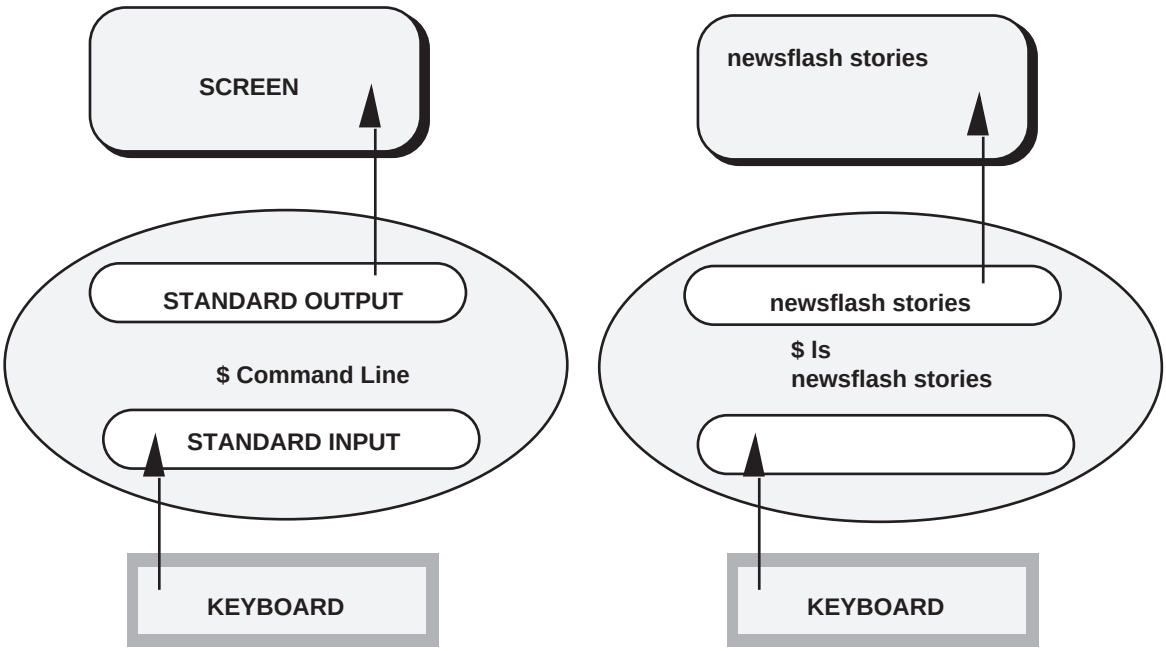

Figure 7.1. The standard output and standard input.

Because the standard input and standard output have the same organization as that of a file, they can easily interact with files. Unix has a redirection capability that allows you to easily move data in and out of files. For example, instead of displaying the output on a screen, you can save it to a file with a simple redirection operation. You can also redirect the standard input away from the keyboard to a file, so that input is read from a file instead of your keyboard. You can even redirect the output of a command as input for another. The redirection operator > directs the standard output to a file and the < operator reads the standard input from a file. The pipe operator, |, takes the standard output of one program and channels it as standard input for another.

When a Unix command that is executed produces output, this output is placed in the standard output data stream. The default destination for a the standard output data stream is a device, the screen. Devices, such as the keyboard and screen, are treated as files. Devices receive and send out streams of bytes with the same organization as that of a byte stream file. The screen is a device that displays a continuous stream of bytes. By default, the standard output sends its data to the screen device, which will then display the data. For example, the `ls` command will generate a list of all file names and send it to the standard output (as shown in Figure 2). This stream of bytes in the standard output is then directed to the screen device. The list of file names is then printed on the screen. The `cat` command also sends output to the standard output. The contents of a file are copied to the standard output whose default destination is the screen. The contents of the file are then displayed on the screen.

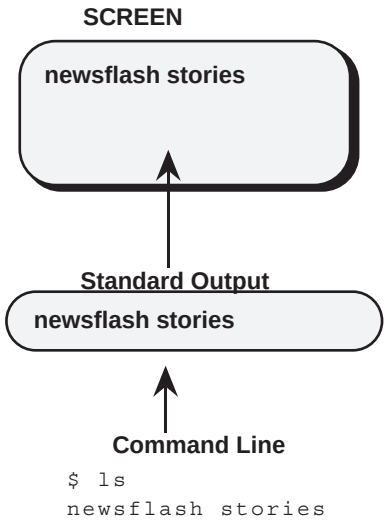

Figure 7.2. The standard output.

Redirecting the Standard Output: > and >>

Unix has a redirection capability that allows you to redirect the standard output from its default destination to any other file or to another device you should choose. Suppose that instead of displaying information on your screen, you would want to save it in a file. For example, the `ls` command will display the list of files in your working directory on the screen. Suppose, that instead of displaying this list on the screen, you want to save it to a file. Redirection allows you to do just that. The `ls` command actually outputs data to the standard output, and the default destination for the standard output is the screen. You can redirect the standard output to another destination, a file, rather than to its default destination, the screen. The output redirection operator is the greater-than sign, `>`. To redirect output, you place the greater-than sign and the name of a file on the command line after the Unix command. Table 7-2 lists the different ways you can use the redirection operators. In the next example, the output of the `ls` command is redirected from the screen device to the file mylist.

```
$ ls > mylist
```

Redirection File Creation: >

The fact that the redirection operator and file name are placed after the command whose output is being redirected can mislead you to think that the redirection is executed after the command. In fact, it is executed before the command. The redirection operation creates the file and sets up the redirection before it receives any data from the standard output. In effect, the command generating the output is executed only after the redirected file has been created. The name you give to that file can cause problems if there already exists file with that same name. The original file will be destroyed and replaced by the one set up by the redirection operation. In the next example, the output of the `ls` command is redirected from the screen device to a file. First, the `ls` command lists files,and then, in the next command, `ls` redirects its file list to the **listf** file. Then the `cat` command displays the list of files saved in **listf**. Notice that the list of files in **listf** includes the **listf** file name. As shown in figure 7.3, the list of file names generated by the `ls` command will include the name of the file created by the redirection operation, in this case, **listf**. The **listf** file is first created by the redirection operation, and then the `ls` command lists it along with other files. This file list output by `ls` is then redirected to the listf file, instead of being printed on the screen.

```
$ ls
mydata intro preface
$ ls > listf
$ cat listf
mydata intro listf preface
```

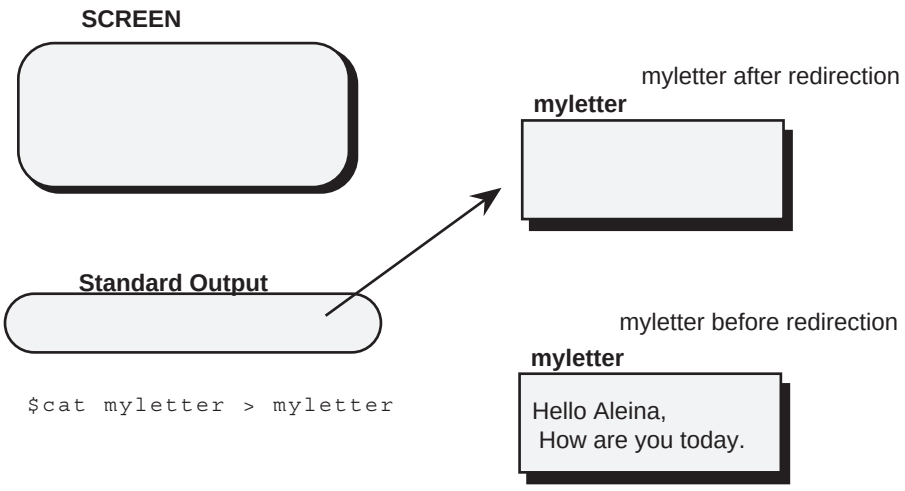

Figure 7.3. Redirection file creation

Errors occur when you try to use the same file name for both an input file for the command and the redirected destination file. In this case, since the redirection operation is executed first, the input file, since it exists, is destroyed and replaced by a file of the same name. When the command is executed it finds an input file that is empty.

In the `cat` command below, the file **myletter** is both the name of the destination file for the redirected output and of the input file for the `cat` operation. As shown in figure 7.4, the redirection operation is executed first, destroying the **myletter** file and replacing it with a new and empty **myletter** file. Then the `cat` operation is executed and attempts to read all the data in the **myletter** file. However, there is now nothing in the **myletter** file.

```
$ cat myletter > myletter
```

In shells other than Bourne, you can set the `noclobber` option to prevent overwriting an existing file with the redirection operation. In that case, the redirection operation on an existing file will fail. You can overcome the `noclobber` option by placing an exclamation point after the redirection operator. The next example sets the `noclobber` option for the Korn shell and then forces the overwriting of the **oldletter** file if it already exists.

```
$ set -o noclobber
$ cat myletter >! oldletter
```

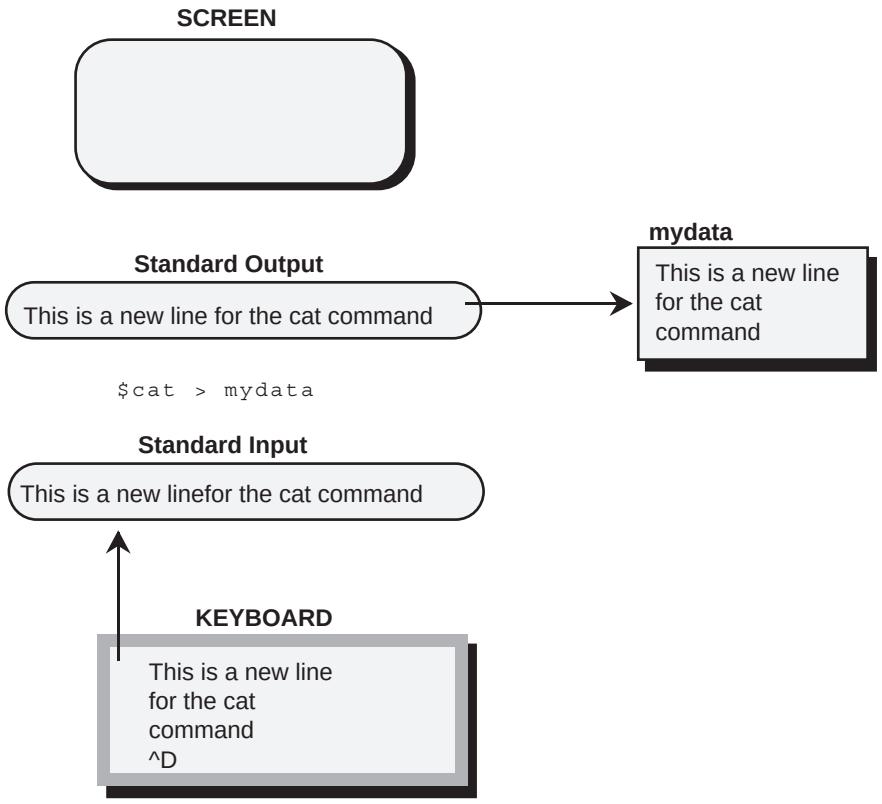

Figure 7.4. A file that already exists will be erased by a redirection operation. Notice that there is nothing in the standard output. myletter has already been destroyed before the cat command is executed.

Appending the Standard Output: >>

A very useful variation on the > redirection operator is the >> append operator ,which allows you to append the standard output to an existing file. Instead of destroying the existing file, the data in the standard output are added at the end of that file. In the next example, the first command redirects a listing of files in the reports directory to the **dirlist** file. Then the second command redirects a listing of files in the stories directory to that same **dirlist** file, using the >> operator to append this list at the end. The >> operator does not destroy the original **dirlist** as the > operator would. **dirlist** now contains a listing of the reports directory followed by a listing of the stories directory.

```
$ ls reports > dirlist
$ ls stories >> dirlist
```

The Standard Input

Just as data can be sent to a device, data can also be read from a device or file. Data read from a device or file are placed in the standard input and can then be used as input for a Unix command. The default device for the standard input is the keyboard. Characters typed into the keyboard are placed in the standard input, and the standard input is then directed to the Unix command.

Many Unix commands are designed to read from the standard input. For example, the `cat` command without a file name argument will read data from standard input. When you type in data on the keyboard, each character typed will be placed in the standard input and directed to the `cat` command. The `cat` command will then send the character to the standard output. The standard output is directed to the screen device, which will then display the character on the screen.

When you try this you will find that you will enter one line and then that line will immediately be displayed on the screen. This is due to the line buffering method used in many Unix systems. Line buffering requires that a user type in an entire line before any input is sent to the standard input. The `cat` command will receive from the standard input a whole line at a time. It will immediately display the line. This has the effect of displaying each line as it is entered. In the next example, the user executes the `cat` command without any arguments. The `cat` command will then receive input from the standard input. As the user types in a line, it is sent to the standard input which the `cat` command then reads and sends to the standard output.

```
$ cat
This is a new line
This is a new line
for the cat
for the cat
command
command
^D
$
```

The `cat` operation will continue until Ctrl-d is entered in on a line by itself. Ctrl-d is the end-of-file character for any Unix file. In a sense, the user is actually entering a file at the keyboard and ending the file with the end-of-file character. Remember, that the standard input, as well as the standard output, have the same format as any Unix file.

If you combine the `cat` command with redirection, you have an easy way of saving to a file what you have typed in. As shown in figure 7.5, the output of the `cat` operation is redirected to the **mydata** file. The **mydata** file will now contain all the data typed in at the keyboard. The `cat` command, in this case, has no file arguments. It will receive its data from the standard input, the keyboard device. The redirection operator redirects the output of the `cat` command to the file **mydata**. The `cat` command has no direct contact with any files. It is receiving input from the standard input and sending

output to the standard output.

```
$ cat > mydata
This is a new line
for the cat
command
^D
$
```

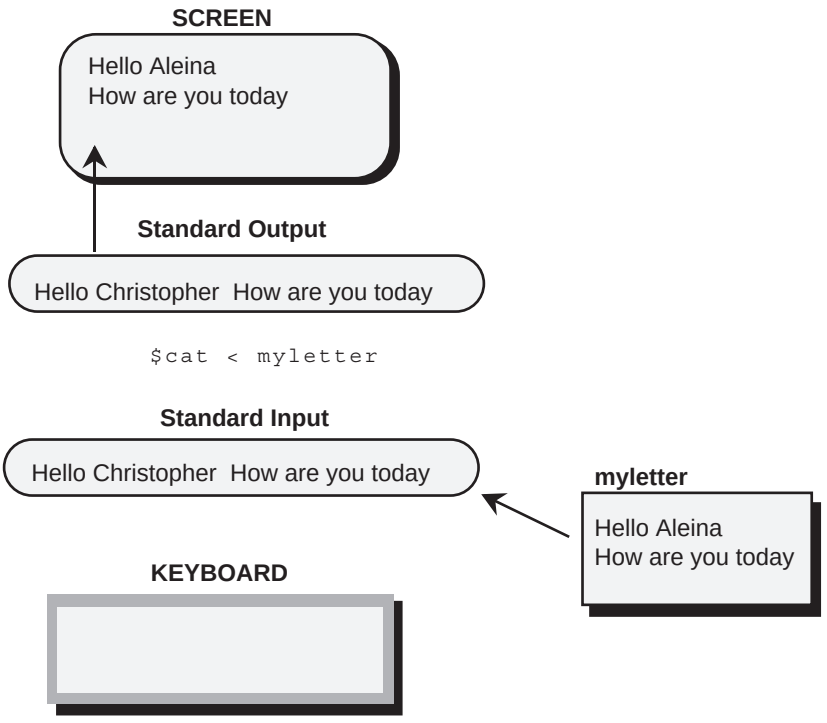

Figure 7.5. Redirecting from the standard input to a file.

Redirecting the Standard Input: <

You can redirect the standard input to read from a device or file other than the keyboard (the default device). This means that the standard input may read data from a file, rather than the keyboard. The operator for redirecting the standard input is the less-than sign, <. The file or device from which you are reading data is placed after the < operator. In figure 7.6, the standard input is redirected to receive input from the **myletter** file rather than the keyboard device. The contents of **myletter** are read into the standard input by the redirection operation. Then the `cat` command reads the standard input and

displays the contents of **myletter**.

```
$ cat < myletter
```

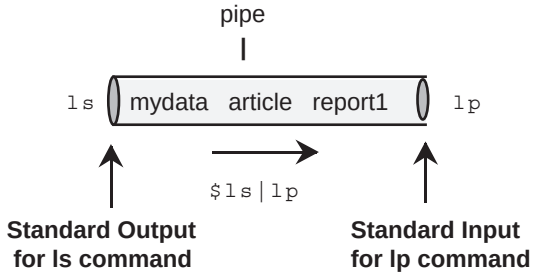

Figure 7.6. Redirecting the standard input.

It is perfectly acceptable to use the redirection operations for both standard input and standard output with the same command line. In the next example, the `cat` command has no file name arguments. Without file name arguments, the `cat` command receives input from the standard input and sends output to the standard output. However, the standard input has been redirected to receive its data from the **myletter** file and the standard output has been redirected to place its data in the **newletter** file. In effect, a copy of **myletter** is made called **newletter**.

```
$ cat < myletter > newletter
```

Pipes: |

There are tasks that require several Unix commands, in which the data output from one command is passed on and used as input for another command. In this case, the standard output of a command is sent to another command, not to a destination file as with redirection. For example, suppose you want to send a list of your file names to the printer to be printed. You need two commands to do this: the `ls` command to generate a list of file names and the `lp` command to send the list to the printer. In effect, you need to take the output of the `ls` command and use it as input for the `lp` command. You can think of the data flowing from one command to another. You can form such a connection in Unix with what is called a pipe. The pipe operator is the vertical bar character, |. You place the pipe operator between two commands. The pipe operation receives the output from the command placed before the pipe, and sends this data as input to the command placed after the pipe. In effect, a connection is formed between them in which the standard output of one command becomes the standard input for another. As shown in figure 7.7, you can connect the `ls` command and the `lp` command with a pipe. The list of file names output by the `ls` command is piped into the `lp` command.

```
$ ls | lp
```

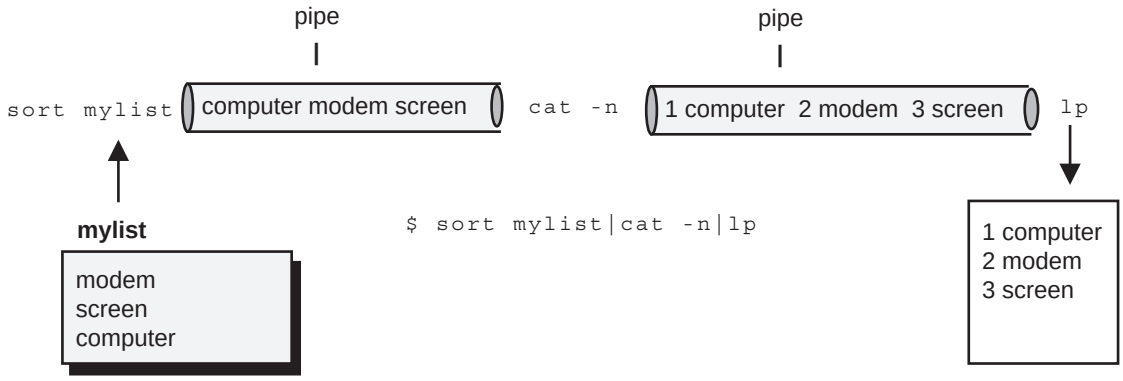

Figure 7.7. Piping the output of one command into another command.

Combining the pipe operation with other shell features such as metacharacters lets you perform very specialized operations. The next example prints only files with a .c extension. The `ls` command is used with the asterisk metacharacter and the ".c" characters to generate a list of file names with the .c extension. Then this list is piped to the `lp` command.

```
$ ls *.c | lp
```

It is important to keep in mind the difference between redirection and pipes. Whereas redirection places output in a file, pipes send output to another Unix command. Keep in mind the difference between a file and a command. A file is a storage medium that holds data. You can save data in it or read data from it. A command is a program that executes instructions. A command may read or save data in a file, but a command is not a file itself. For this reason, a redirection operation operates on files, not on commands. Redirection can send data from a program to a file but it cannot send data from a program to another program. For example, if you want to print the output of the `cat` command, you have to send it to the `lp` command. Redirection would only save the output to a file. In the following example, `cat` takes input from the keyboard and pipes the output to the `lp` command. The `cat` command is executed before the `lp` command, so you first enter your data for the `cat` command in the keyboard, ending with the end-of-file character, Ctrl-d.

```
$ cat | lp
This text will
be printed
^D
$
```

It is possible, though not advisable, to simulate a pipe operation through a series of redirection operations. The output of the first command could be redirected to a file, and then another command could use that same file as redirected input. The next example uses two redirection operations in two separate commands to print a list of file names. The same task was performed above using a single pipe operation.

```
$ ls > tempfile
$ lp < tempfile
```

The pipe operation literally takes the standard output of one command and uses it as standard input for another command. There is one important difference. With few exceptions, pipes will pass the standard output on one line at a time. If you have several commands connected by pipes, each pipe reads one line of output and passes it on to the next command; the pipe then reads the next line of output. Commands are being handed a line of output much like water in a bucket brigade. Certain filters such as the sort filter are exceptions to this process and require that all lines of output be read from a pipe before sending anything on to the next pipe.

Although up to this point we have been using a list of file names as input, it is important to note that pipes operate on whatever the standard output of a command might be. The contents of whole files or even several files can be piped from one command to another. In the next example, the `cat` command reads and outputs the contents of the **mydata** file, which is then piped to the `lp` command.

```
$ cat mydata | lp
```

As another example, suppose you want to print a file with a line number inserted before each line. Most Unix shells provides `cat` with an `-n` option, which outputs the contents of a file, adding line numbers. To print your file with line numbers, you need to first use the `cat` command with the `-n` option to output the contents of the file with line numbers added. You then need to pipe this output to the `lp` command for printing. In the next example, the contents of the **mydata** file, are printed with line numbers.

```
% cat -n  mydata | lp
```

A pipe is also used commonly with the more or pg commands, which display output a page at a time. more and pg are programs that can read the standard output and display it screen by screen, allowing you to move forward or backward through the output. You can even specify several files at once and pipe their output to the more command, examining all the files. In the next example, both **mydata** and preface are

piped to the more command for screen-by-screen examination.

```
$ cat mydata preface | more
```

Unix has many commands that generate modified output. For example, the sort command takes the contents of a file and generates a version with each line sorted in alphabetic order. It works best with files that are lists of items. Commands like sort that output a modified version of the input are referred to as filters. Filters are discussed in detail in Chapter 11. They are often used with pipes. In the next example, a sorted version of **mylist** is generated and piped into the `pg` command for display on the screen. The original file, **mylist** , has not been changed and is not itself sorted. Only the output of sort in the standard output is sorted.

```
$ sort mylist | pg
```

You can create complex tasks by using several pipes on the command line connecting different commands. The output of one command is piped into another command, which, in turn, is piped into still another command. For example, suppose you have a file with a list of items that you want to print out both numbered and in alphabetical order. To print the numbered and sorted list you can first generate a sorted version with the sort command and then pipe that output to the `cat` command. The `cat` command with the `-n` option then takes as its input the sorted list and generates as its output a numbered sorted list which can then be piped to the `lp` command for printing. In figure 7.8 a sorted and numbered version of the **mylist** file is printed.

```
$ sort mylist | cat -n | lp
```

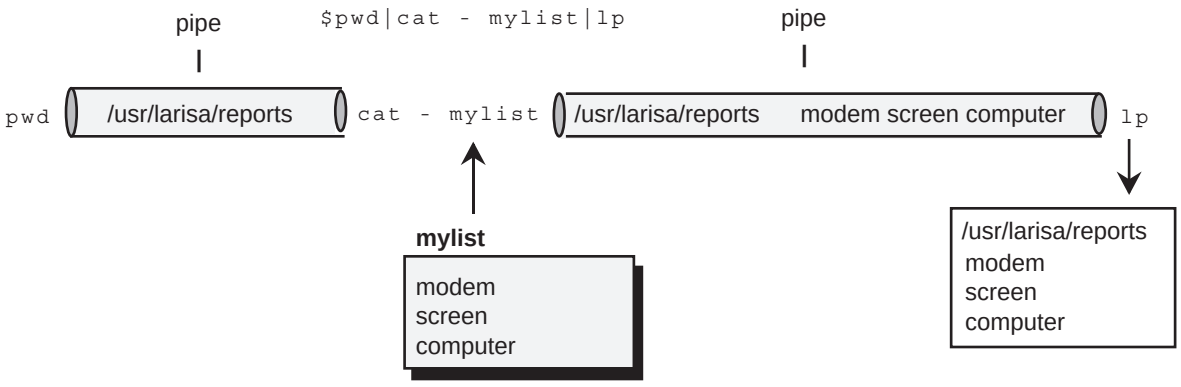

Figure 7.8. Piping through several commands.

Though many commands are designed to read from the standard input, others specify input using file name arguments only. Some commands, like `cat`, do both. The standard input piped into such a command can be specified using the dash, -, as the file

name argument. When you use the dash as an argument for a command it represents the standard input. As an example, suppose you would like to print a file with the name of its directory at the top. The `pwd` command outputs a directory name and the `cat` command outputs the contents of a file. In this case, the `cat` command needs to take as its input both the file and the standard input piped in from the `pwd` command. The `cat` command will have two arguments: the standard input as represented by the dash and the file name of the file to be printed, `cat - mylist`. In figure 7.9, the `pwd` command generates the directory name and pipes it into the `cat` command. For the `cat` command, this piped-in standard input now contains the directory name. As represented by the dash, the standard input is the first argument to the `cat` command. The `cat` command then copies the directory name and the contents of the mylist file to the standard output ,which is then piped to the `lp` command for printing. If you want to print the directory name at the end of the file instead, simply make the dash the last argument and the file name the first argument, `cat mylist -`.

```
$ pwd | cat - mylist | lp
```

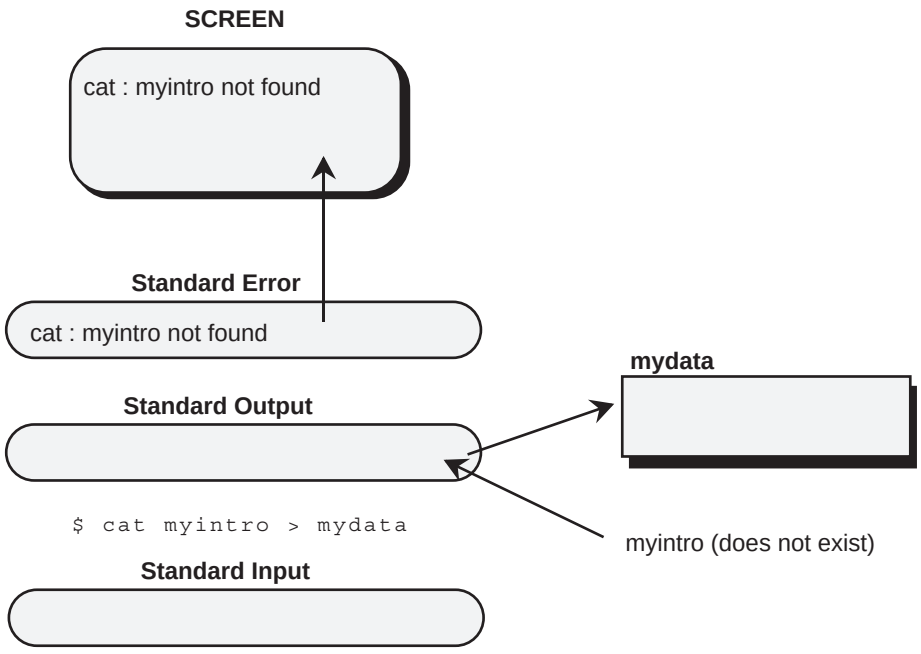

Figure 7.9. Controlling piped-in standard input using the dash as an argument.

Pipes and Redirection: tee

There may be situations where you would want to both save the output of a command to a file and also send it on as input to another command. This means that you would have to both pipe and redirect the output at the same time. You can do this by piping the output to the tee command. The tee command is designed to be used with pipes. It copies the standard input that it receives to a file, taking as its argument the name of that file. In effect, it redirects the output of the previous command to a file. At the same time, the standard input is output and can be piped to another command. In effect, the standard output from the previous command is passed on to the next command. It is as if the standard output were split into two copies, one being redirected to a file and the other continuing on its way to the next command. The next example copies the contents of the file **mylist** to the file **newlist** and displays it on the screen.

```
$ cat mylist | tee newlist
```

The tee command is very helpful when you are modifying output and you would like to save the modified output in a file and also see what the modifications look like. In the next example, the file **mylist** is again sorted and the sorted output is piped to the tee command. The tee command then both saves the sorted output in a file called **sfile** and also displays it on the screen.

```
$ sort mylist | tee sfile
computer
modem
screen
$
```

As another example, suppose you need to both save your output in a file and print it. You can use the tee command to copy the output to another file while allowing the standard output to be piped into the `lp` command. In the next example, the output of the sort command is first piped to tee which copies the output to the file **sfile**. The output itself is then piped into the `lp` command to be printed.

```
$ sort mylist | tee sfile | lp
```

You can combine piped operations with redirection of the standard input or output. A standard output redirection specifies a destination for the standard output. The redirection operation is placed at the end of the piping operation and constitutes a destination for the final output. Once the redirection operation has saved the output there is no output to be piped into another command. This means that, although redirection can take place at the end of a series of piping operations, it cannot take place within piping operations. The next example is a valid use of pipes and redirection. The output of the sort command is piped to the `cat` command with the `-n` option to number lines, and then the result is saved in the **nfile** file.

```
$ sort mylist | cat -n > nfile
```

What if you need to both save the result in **nfile** and to print it out? You cannot do something like:

```
sort mylist | cat -n > nfile | lp     ERROR
```

The only way to both save the output in a file and print it is to use the tee command.

```
$ sort mylist | cat -n | tee nfile | lp
```

You can use tee anywhere in the piping sequence. The next example saves a sorted version of the list while printing the numbered version.

```
$ sort mylist | tee sfile | cat -n | lp
```

Redirecting and Piping the Standard Error: >&, 2>, |&

When you execute commands, it is possible for an error to occur. You may give the wrong number of arguments or some kind of system error could take place. When an error occurs, the system will issue an error message. Usually such error messages are displayed on the screen along with the standard output. However, Unix distinguishes between the standard output and error messages. Error messages are placed in yet another standard byte stream called the standard error. In the next example, the `cat` command is given as its argument the name of a file that does not exist, **myintro**. In this case, the `cat` command will simply issue an error.

```
$ cat myintro
cat : myintro not found
$
```

Because error messages are in a separate data stream from the standard output, this means that if you have redirected the standard output to a file, error messages will still appear on the screen for you to see. Though the standard output may be redirected to a file, the standard error is still directed to the screen. In the next example, the standard output of the `cat` command is redirected to the file **mydata**. However, the standard error, containing the error messages, is still directed toward the screen (figure 7.10).

```
$ cat myintro > mydata
cat : myintro not found
$
```

Figure 7.10. The standard error displayed on the screen.

Like the standard output, you can also redirect the standard error. This means that you can save your error messages in a file for future reference. This is helpful if you need to save a record of the error messages. Like the standard output, the standard error's default destination is the screen device. Using special redirection operators, you can redirect the standard error to any file or device that you choose. If you redirect the standard error, the error messages will not be displayed on the screen. You can examine them later by viewing the contents of the file in which you saved them.

Redirection of the standard error works differently in the Bourne and C-shells. The C-shell modifies the redirection operation to include error messages into the standard output, saving both the standard error and the standard output in the same destination file. In the Bourne shell, the standard error is handled as a separate data stream and its contents are redirected into a separate file from that of the standard output. Redirection of the C-shell standard error is simpler than that of the Bourne shell, and for that reason we will first discuss the C-shell. Redirection operators for both the C-shell and the Bourne shell are listed in Table 7-2.

The C-shell has a special operator for redirecting the standard error, the greater-than sign followed by an ampersand, `>&`. Use of this operator will redirect any error messages into the standard output and save them in the same destination file as the standard output. In the next example, the `cat` command has as its argument the name of a file that does not exist, **nodata**. The resulting error message is redirected to the file **mydata**. The message is not displayed on the screen but, instead, saved in a file. To see the error message, simply display the contents of the **mydata** file. If the **nodata** file had actually existed, then **mydata** would hold the contents of that file instead of error messages. In the C-shell, error messages are simply channeled into the standard output and end up wherever the standard output is directed.

```
% cat nodata >& mydata
% cat mydata
cat : nodata not found
%
```

In the C-shell, you can also pipe error messages with the standard output to other commands. In the C-shell, the operator for piping the standard error is `|&`. The standard error is combined with the standard output and both become input for the next Unix command. In the next example, any errors with the `cat` command will be piped with the standard output to the printer. If the file **nodata** does not exist, then the resulting error message will be printed.

```
% cat nodata |& lp
```

Redirection of the standard error in the Bourne and BASH shells is not as simple. It relies on a special feature of Bourne shell redirection. In the Bourne and BASH shells, all the standard byte streams can be referenced in redirection operations with numbers. The numbers 0, 1, and 2 reference the standard input, standard output, and standard error respectively. By default an output redirection, >, operates on the standard output, 1. You can modify the output redirection to operate on the standard error by preceding the output redirection operator with the number 2, `2>`. In the next example, the `cat` command again will generate an error. The error message is redirected to the standard byte stream represented by number 2, the standard error.

```
$ cat nodata 2> myerrors
$ cat myerrors
cat : nodata not found
$
```

You can also append the standard error to a file by using the number 2 and the redirection append operator, >>. In the next example, the user appends the standard error to the **myerrors** file. **myerrors** then functions as a log of errors.

```
$ cat nodata 2>> myerrors
$ cat compls 2>> myerrors
$ cat myerrors
cat : nodata not found
cat : compls not found
$
```

To both redirect the standard output as well as the standard error, you will need a separate redirection operation and file for each. In the next example, the standard output is redirected to the file **mydata**, and the standard error is redirected to **myerrors**. If **nodata** were to actually exist, then **mydata** would hold a copy of its contents.

```
$ cat nodata 1> mydata 2> myerrors
$ cat myerrors
cat : nodata not found
$
```

If, however, you want to save a record of your errors in the same file as that used for the redirected standard output, you need to redirect the standard error into the standard output. In the Bourne shell you can reference a standard byte stream by preceding its number with an ampersand. `&1` references the standard output. You can use such a reference in a redirection operation to make a standard byte stream a destination file. The redirection operation `2>&1` redirects the standard error into the standard output. In effect, the standard output becomes the destination file for the standard error. Conversely the redirection operation `1>&2` would redirect the standard

input into the standard error. In the next example, both the contents of the standard error and the standard output will be saved in the same file, **mydata**.

```
$ cat nodata 1> mydata 2>&1
```

TABLE 7-2

Redirection and Pipes

Redirection

command > *filename*	Redirect the standard output to a file or device, creating the file if it does not exist and overwriting the file if it does exist.
command < *filename*	Redirect the standard input from a file or device to a program.
command >> *filename*	Redirect the standard output to a file or device, appending the output to the end of the file.
command >! *filename*	In the C-shell and Korn shell, force the overwriting of a file if it already exists. This overrides the noclobber option.
command 2> *filename*	Redirect the standard error to a file or device in the Bourne shell. *command* 2>> *filename* Redirect and append the standard error to a file or device in the Bourne shell.
command 2>&1	Redirect the standard error to the standard output in the Bourne shell. *command* >& *filename* Redirect the standard error to a file or device in the C-shell.

Pipes

command | *command*	Pipe the standard output of one command as input for another command.
command |& *command*	Pipe the standard error as input to another command in the C-shell.

Jobs: Background, Kills, and Interruptions

In Unix, you can control the execution of a command: running it in the background, interrupting it and starting it up later from where you left off, or canceling it before it finishes. Background operations are particularly useful for long jobs. Instead of waiting at the terminal until a command has finished execution, you can place it in the background and then continue on with other work, executing other Unix commands. For example, you could edit one file while other files are being printed. The ability to cancel a background command can save you a lot of unnecessary expense. If, say, you had executed a command to print out all your files and then realized you had some very large files you did not want to print out, you could reference that execution of the print

command and cancel it. Interrupting commands is rarely used sometimes unintentionally executed. You can, if you want, interrupt an editing session to send mail and then return to your editing session, continuing from where you left off. The background commands as well as commands to cancel and interrupt jobs are listed in Table 7-3.

In Unix, a command is considered a process, a task to be performed. A Unix system can execute several processes at the same time, just as Unix can handle several users at the same time. There are commands to examine and control processes, although they are often reserved for system administration operations. Processes actually include not only the commands a user executes, but also all the tasks the system must perform to keep Unix running.

The commands that users execute are often called jobs in order to distinguish them from system processes. When the user executes a command, it becomes a job to be performed by the system. The shell provides a set of job control operations that allow the user to control the execution of these jobs. You can place a job in the background, cancel a job, or interrupt one.

The job control feature is built into the BASH, Korn, TCSH, C-shell and Z-shell, but not into the Bourne shell. Development of the Bourne shell preceded the invention of job control in Unix. The Bourne shell does have a few limited job control features such as the background option, but referencing and canceling jobs have to be done through the more complex system process operations. If you are using the Bourne shell, you can switch to another shell to perform job control operations. Also, some systems may have the jsh shell which is a shell, that provides job control and was meant to complement the Bourne shell.

Background and Foreground: &, fg, bg

To execute a command in the background, you place an ampersand, &, on the command line at the end of the command. Upon pressing enter, a user job number and a system process number is displayed. The user job number is the number by which the user references the job and is displayed within brackets. The system process number is the number by which the system identifies the job. In the next example, the command to print the file **mydata** is placed in the background.

```
$ lp mydata &
[1]  534
$
```

You can put the execution of as many commands as you want into the background. Each is classified as a job and given a name and a job number. The command jobs will list the jobs being run in the background. Each entry in the list will display the job number in brackets, whether it is stopped or running, and the name of the job. The + sign indicates the job currently being processed, and the - sign indicates the next job to be executed. In the next example, two commands have been placed in the

background. The jobs command then lists those jobs, showing which one is currently being executed.

```
$ lp intro &
[1]  547
$ cat *.c > myprogs &
[2]  548
$ jobs
[1]  +  Running  lp intro
[2]  -  Running  cat *.c > myprogs
$
```

You can even use a single command line to place several commands at once in the background by entering the commands separated by an ampersand, `&`. In this case, the `&` both separates commands on the command line and executes them in the background. In the next example, the first command, to sort and redirect all files with an .l extension, is placed in the background. On the same command line, the second command to print all files with a .c extension is also placed in the background. Notice that the two commands each end with an `&`. The jobs command then lists the sort and `lp` commands as separate operations.

```
$ sort *.l > larts & lp *.c &
[1]  534
[2]  567
$ jobs
[1]  +  Running  sort *.l > larts
[2]  -  Running  lp
$
```

If you have jobs running in the background, then whenever you press enter to execute a command you have entered on the command line, the system will then tell you what background jobs are still running as well as those that have just been completed. The system will not interrupt any operation, such as editing, to notify you about a job ending. If you want to be notified immediately when a certain job ends, no matter what you are doing on the system, you can use the notify command to instruct the system to tell you. The notify command takes as its argument a job number preceded by the percent sign, `%`. When that job is finished, the system will interrupt what you are doing to notify you that the job has ended. The next example tells the system to notify the user when job 2 has finished.

```
% notify %2
```

You can, if you wish, bring a job out of the background to the foreground, executing it as your current command. You do this with the foreground command, `fg`. If there is only one job in the background, the `fg` command with no arguments will bring it to the foreground. If there is more than one job in the background, then `fg` requires the

job's number. You place the job number after the `fg` command, and precede it with a percent sign. In the next example, the second job is brought back into the foreground. You will not receive a prompt until the job that is now in the foreground and executing is finished. When the command is finished executing, the prompt will appear. Then you can execute another command.

```
$ fg %2
cat *.c > myprogs
$
```

There is also a `bg` command that places a job in the background. This command is usually used for interrupted jobs sot it is discussed later when examining interruptions.

Canceling Jobs: kill

If you want to cancel a job that is running in the background, you can force a job to end with the `kill` command. The `kill` command takes as its argument either the user job number or the system process number. The user job number needs to be preceded by a percent sign, `%`, and the number of the job. You can find out the job number from the jobs command. In the next example, the `jobs` command lists the background jobs. Then, the second job is canceled with the kill command and the user job number.

```
$ jobs
[1]  +  Running  lp intro
[2]  -  Running  cat *.c > myprogs
$ kill %2
$
```

You can also use the system process number to cancel a job. In the Bourne shell, that is the only way to cancel a job. You can obtain the system process number with the ps command. The `ps` command displays a great deal more information than the `jobs` command. The next example lists the processes a user is running. The PID is the system process number, also known as the process id. TTY is the terminal identifier. The time is how long the process has taken so far. COMMAND is the name of the process.

```
$ ps
PID  TTY  TIME COMMAND
523  tty24     0:05      sh
567  tty24     0:01      lp
570  tty24     0:00      ps
```

You can then reference the system process number in a kill command. Use the process number without any preceding percent sign. The next example kills process 567.

```
$ kill 567
```

Interruptions: Ctrl-z

You can interrupt a job and stop it with the Ctrl-z command. This places the job to the side until the job is restarted. The job is not ended. It merely remains suspended until you wish to continue with it. You can continue with the job in either the foreground or the background using the `fg` or `bg` commands. The `fg` command will restart an interrupted job in the foreground. The `bg` command will place the interrupted job in the background.

The interrupt operation is often used to place a job currently running in the foreground into the background. First, you interrupt the job with Ctrl-z; then you place it in the background with the `bg` command. In the next example, the current command to list and redirect **.c** files is first interrupted with a Ctrl-z. Then, that job is placed in the background.

```
$ cat *.c > myprogs
^Z
$ bg
```

There is a situation in Vi where users, at times, accidentally stop Vi, making the editing session a stopped job. While in the Vi editor, you may make the mistake of entering a Ctrl-z instead of a `Shift-ZZ` to end your session (a good reason to use `:wq` instead of `ZZ`). The Ctrl-z will interrupt the Vi editor and return the user to the Unix prompt. The editing session has not ended. It has only been interrupted. Often, you only detect such a mistake when you try to log out. The system will not allow you to log out while an interrupted job remains. To log out you need to first restart the interrupted job with the `fg` command. In the case of the Vi editor interruption, the `fg` command will place you back in the Vi editor. Then a `ZZ` editor command will end the vi editor job and you can log out. In the next example, the jobs command shows that there are stopped jobs. The `fg` command then brings the job to the foreground.

```
$ jobs
[1]  +  Stopped  vi mydata
$ fg %1
```

Delayed execution: at

In Unix, you can instruct your system to execute specified commands at a certain time. Instead of placing a job immediately in the background and executing it right away, you can use the `at` command to specify a later time when you want it executed. You can then log out and the system then keep track of what commands to execute and when to execute them.

The `at` command takes as its argument a time and a date. The time is the time at which you want the commands executed. It consists of a number specifying the hour followed by the keywords "am" or "pm". The date is optional. If no date is specified, then today's date is assumed. When you press Enter to execute the at command, it will then read in Unix commands from the standard input. These are the commands that will be executed at the specified time. You can enter these commands at the keyboard, ending the standard input with a Ctrl-d, or you can enter the commands into a file that you redirect through the standard input to the `at` command. In the next example, the user decides to execute a command at 4 am.

```
$ at 4am
lp stories/*
^D
$
```

In the next example, the user decides to places several commands in a file called `schedcmds` and then redirect the contents of this file as input to an at command. The `at` command will execute the commands at 6 pm.

```
schedcmds
lp stories/*
cat *.c > myprogs
```

```
$ at 6pm < schedcmds
```

You have a great deal of leeway in specifying the time and data. `at` assumes a 24-hour sequence for the time unless modified by the keywords am or pm. You can specify the minutes in an hour by separating the hour and minutes with a colon. Six-thirty is written as 6:30. The at command also recognizes a series of keywords that specify certain dates and times. The keyword noon will specify 12pm. You can use the keyword midnight instead of 12am. In the next examples, the user executes commands using a minute specification and then the keyword noon.

```
$ at 7:35pm < schedcmds
$ at noon < schedcmds
```

The date can be specified as a day of the month or a day of the week. The day of the month consists of the number of the day and a keyword representing the month. Months can be represented by three letter abbreviations. For example January is written as Jan. The day of the month follows the month's name. If there is no name, then the current month is assumed. Feb 14 will specify the fourteenth of February. 21 by itself

specifies the twenty-first day of the current month. In the next example, the user first executes commands on the fifteenth of this month and then on the twenty-ninth of October.

```
$ at 7:35pm  15 < schedcmds
$ at noon Oct 29 < schedcmds
```

If you want to run a job during your current week, you only have to use the name of the weekday. Entering tuesday as your date will run your commands on Tuesday. You can also use the keywords today and tomorrow for your date. In the next examples, the user executes commands on Friday and tomorrow.

```
$ at 7:35pm  friday < schedcmds
$ at noon tomorrow < schedcmds
```

With either the time or date you can specify an increment. For example, you could have commands executed one week from today or two months from Friday. You specify an increment using the + operator followed by a keyword denoting a segment of time. Segments-of-time keywords are minutes, hours, days, weeks, months, or years. The plural 's' can be left off to denote one segment; week is one week. The increment is added to the time or date that you specify. For example, to run commands one month from the nineteenth of the current month you enter 19 + month for the date. One week from tomorrow is tomorrow + week. Two weeks from today is today +2 weeks. In the next example, the user executes commands three weeks from Tuesday and five months from today.

```
$ at 7:35pm  tuesday +3 weeks < schedcmds
$ at noon today +5 months < schedcmds
```

You can easily list or cancel any at jobs you have set up. Each time you execute an at command, the Unix commands you specify for late execution are queued and listed as an at job. You can obtain a list of your at jobs by entering the `at` command with the `-l` option. Each job will have a number with which you can reference it.

```
$ at -l
749263402.a    Tue Sept 26 20:15:00  2006
749263403.a    Sat Sept 23 12:00:00  2006
```

You can cancel your at jobs using the `-r` option. To cancel a specific job you need to enter the job's number after the `-r` option. In the next example, the user cancels at job 749263402.a.

```
$ at -r 749263402.a
$ at -l
749263403.a    Sat Sept 23 12:00:00  2006
```

The `at` command ordinarily does not notify you when a job has been executed. You can use the -m option to request that you be notified by mail when an at job finishes execution. You can specify a particular job number to receive mail for just that job. In the next example, you will be notified by mail when at job 749263403.a has executed.

```
$ at -m 749263403.a
```

TABLE 7-3

Background Jobs and At Jobs

`&`	Execute a command in the background
`fg` *%jobnum*	Bring a command from the background to the foreground or resume an interrupted program.
`bg`	Place a command in the foreground into the background
Ctrl-z	Interrupt and stop the currently running program. The program remains stopped and waiting in the background for you resume it.
`notify` *%jobnum*	Notify you when a job ends.
`kill` *%jobnum*	Cancel a job running in the background.
`kill` *processnum*	Cancel a job running in the background.
`jobs`	List all background jobs. The jobs command is not available in the Bourne shell, unless it is using the BASH shell.
`ps`	List all currently running processes including background jobs.
`at` *time date*	Execute commands at a specified time and date. The time can be entered with hours and minutes and qualified as am or pm. hour:minutes am pm The date is specified as a day of the month or day of the week. The month can be represented by a three letter abbreviation: Jan, Feb, etc. *month day* Days of the week are specified by their names. monday tuesday wednesday Keywords can be used to specify the date and time. `am` `pm` `now` `noon` `midnight` `today` `tomorrow` You can increment from a date or time by a time segment using the + operator. A number after the + operator specifies how many time segments. *date +num time-segment*

Time segments

hours minutes days weeks months years

The next keyword increments by a time segment from the current date or time:

`next` *time-segment*

`next` *week*

options

`-l` *jobnum*	List current at jobs.
`-r` *jobnum*	Cancel a job.
`-m` *jobnum*	Be notified by mail when job finishes.

Chapter Summary: the Shell

The shell is a command interpreter that provides an interface between the user and the operating system. You enter commands on a command line that are then interpreted by the shell and sent as instructions to the operating system. The shell has very powerful features such as metacharacters, redirection, pipes, and job control, history, and aliasing.

The shell has three metacharacters that you can use to generate file names. The `*`, `?`, and `[]` metacharacters allow you to generate a list of file names for use as arguments on the command line. The `*` will match any possible sequence of characters. The `?` matches on any one character, and the `[]` matches a specified set of characters. You can even combine the metacharacters to compose complex matches.

In Unix, files, devices, and input and output from commands all have the same structure, a byte stream. All input for a command is placed in a data stream called the standard input, and all output is placed in a data stream called the standard output. Because the standard input and standard output have the same structure as that of files, they can easily interface with a file. Using redirection operators, you can redirect the standard input or the standard output from and to a file. With the > redirection operator you can redirect the standard output from a command to a file. With the < redirection operator you can redirect the standard input to be read from a file. You can even use the redirection append operator, >>, to append standard output to a file that already exits.

Just as you can redirect the standard output, you can also redirect the standard error. Error messages are placed in yet another data stream called the standard error. The standard error is redirected differently in the C-shell. In the C-shell you use the `>&` operator to redirect the standard error; in the Bourne, BASH, Z, and Korn shells you use the `2>` operator to redirect the standard error.

Since the input and output for commands has the same standard format, you can easily use the output of one command as input for another. Pipes allow you to take the standard output of one command and pipe it to another command as standard input. On the same command line you can string together several commands each receiving their input from the output of another command.

When you execute a command, it is treated by Unix as a job to be performed.

You can instruct Unix to execute a job in the background, allowing you to continue executing other commands. Placing the `&` background operator at the end of the command line instructs the system to run this command in the background. You can list the jobs that you have in the background using the `jobs` command. With the `fg` command, you can bring a job in the background into the foreground. You can also cancel background jobs using the kill command, and interrupt jobs using the Ctrl-z command.