CHAPTER 4

The UNIX Shells

MOTIVATION

A shell is a program that sits between you and the raw UNIX operating system. There are four shells that are commonly supported by UNIX vendors: the Bourne shell (sh), the Korn shell (ksh), the C shell (csh), and the Bourne Again shell (bash). All of these shells share a common core set of operations that make life in the UNIX system a little easier. For example, all of the shells allow the output of a process to be stored in a file or "piped" to another process. They also allow the use of wildcards in filenames, so it's easy to say things like "list all of the files whose names end with the suffix '.c'." This chapter describes all of the common core shell facilities; Chapters 5 through 8 describe the special features of each individual shell.

PREREQUISITES

In order to understand this chapter, you should have already read Chapter 1 and Chapter 2. Some of the utilities that I mention are described fully in Chapter 3. It also helps if you have access to a UNIX system so that you can try out the various features that I discuss.

OBJECTIVES

In this chapter, I'll explain and demonstrate the common shell features, including I/O redirection, piping, command substitution, and simple job control.

PRESENTATION

The information is presented in the form of several sample UNIX sessions. If you don't have access to a UNIX account, march through the sessions anyway, and perhaps you'll be able to try them out later.

UTILITIES

The chapter introduces the following utilities, listed in alphabetical order:

chsh	kill	ps
echo	nohup	sleep

SHELL COMMANDS

Also introduced are the following shell commands, listed in alphabetical order:

echo	kill	umask
eval	login	wait
exec	shift	
exit	tee	

INTRODUCTION

A shell is a program that is an interface between a user and the raw operating system. It makes basic facilities such as multitasking and piping easy to use, as well as adding useful file-specific features, like wildcards and I/O redirection. There are four common shells in use:

- the Bourne shell (sh)
- the Korn shell (ksh)
- the C shell (csh)
- the Bourne Again shell (bash)

The shell that you use is a matter of taste, power, compatibility, and availability. For example, the C shell is better than the Bourne shell for interactive work, but slightly worse in some respects for script programming. The Korn shell was designed to be upward compatible with the Bourne shell, and it incorporates the best features of the Bourne and C shells, plus some more of its own. Bash also takes a "best of all worlds" approach, including features from all the other major shells. The Bourne shell comes with absolutely every version of UNIX. The others come with most versions these days, but you might not always find your favorite. Bash probably ships with the fewest versions of UNIX, but is available for the most. (Chapter 8 has information on downloading Bash if it doesn't come with your particular version of UNIX.)

SHELL FUNCTIONALITY

This chapter describes the common core of functionality that all four shells provide. Figure 4.1 illustrates the relationships among the shells. A hierarchy diagram is a useful way to illustrate the features shared by the four shells—so nice, in fact, that I use the same kind of hierarchy chart (Figure 4.2) throughout the rest of the book. The remainder of this chapter describes each component of the hierarchy in detail.

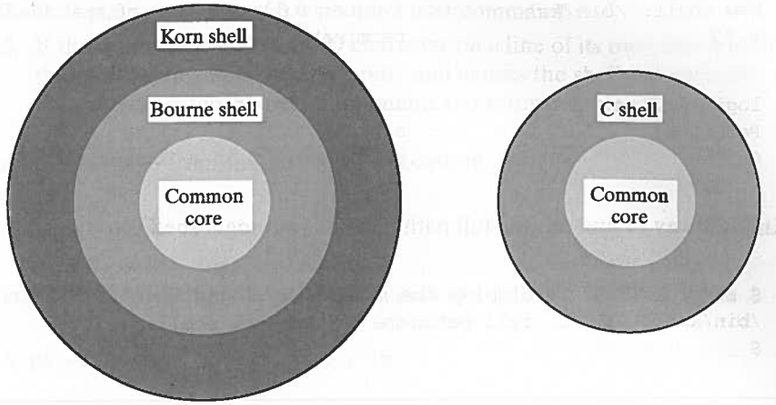

FIGURE 4.1
The relationship of shell functionality.

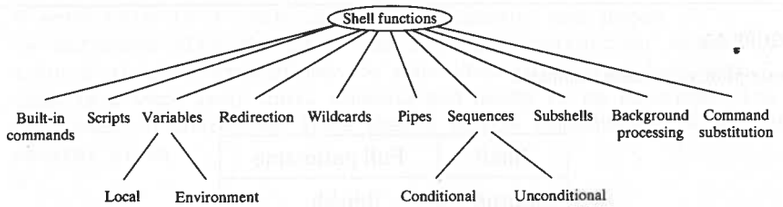

FIGURE 4.2
Core shell functionality.

SELECTING A SHELL

When you are provided a UNIX account, the system administrator chooses a shell for you. To find out which shell was chosen for you, look at your prompt. If you have a "%" prompt, you're probably in a C shell. The other shells use "$" as the default prompt. When I wrote this chapter, I used a Bourne shell, but it really doesn't matter, since the facilities that I'm about to describe are common to all four shells. However, when studying the later chapters, you will want to select the particular shell described in each.

To change your default login shell, use the **chsh** utility, which works as shown in Figure 4.3. In order to use **chsh**, you must know the full pathnames of the four shells. The names are shown in Figure 4.4. In the following example, I changed my default login shell from a Bourne shell to a Korn shell:

```
% chsh                    ...change the login shell from sh to ksh.
Changing login shell for glass
Old shell: /bin/sh        ...pathname of old shell is displayed.
```

```
New shell: /bin/ksh          ...enter full pathname of new shell.
% ^D                         ...terminate login shell.

login: glass                 ...log back in again.
Password:                    ...secret.
$ _                          ...this time I'm in a Korn shell.
```

Another way to find out the full pathname of your login shell is to type the following:

```
$ echo $SHELL   ...display the name of my login shell.
/bin/ksh        ...full pathname of the Korn shell.
$ _
```

Utility: **chsh**

chsh allows you to change your default login shell. It prompts you for the full pathname of the new shell, which is then used as your shell for subsequent logins.

FIGURE 4.3
Description of the **chsh** command.

Shell	Full pathname
Bourne	/bin/sh
Korn	/bin/ksh
C	/bin/csh
Bash	/bin/bash

FIGURE 4.4
Common shell locations.

This example illustrated the *echo* shell command and a shell variable called SHELL. Both of these new facilities—echoing and variables—are discussed later in the chapter.

SHELL OPERATIONS

When a shell is invoked, either automatically during a login or manually from a keyboard or script, it follows a preset sequence:

1. It reads a special start-up file, typically located in the user's home directory, that contains some initialization information. Each shell's start-up sequence is different, so I'll leave the specific details to later chapters.

2. It displays a prompt and waits for a user command.
3. If the user enters a Control-D character on a line of its own, this is interpreted by the shell as meaning "end of input" and causes the shell to terminate; otherwise, the shell executes the user's command and returns to step 2.

Commands range from simple utility invocations, such as

```
$ ls
```

to complex-looking pipeline sequences, such as

```
$ ps -ef | sort | ul -tdumb | lp
```

If you ever need to enter a command that is longer than a line on your terminal, you may terminate a portion of the command with a backslash (\) character, and the shell then allows you to continue the command on the next line:

```
$ echo this is a very long shell command and needs to \
be extended with the line continuation character. Note \
that a single command may be extended for several lines.
this is a very long shell command and needs to be extended with the line
continuation character. Note that a single command may be extended for
several lines.
$ _
```

EXECUTABLE FILES VERSUS BUILT-IN COMMANDS

Most UNIX commands invoke utility programs that are stored in the directory hierarchy. Utilities are stored in files that have execute permission. For example, when you type

```
$ ls
```

the shell locates the executable program called "ls," which is typically found in the "/bin" directory, and executes it. (The way that the shell finds a utility is described later in the chapter.) In addition to its ability to locate and execute utilities, the shell contains several built-in commands, which it recognizes and executes internally. I'll describe two of the most useful ones, *echo* and *cd*, now.

Displaying Information: echo

The built-in echo command displays its arguments to standard output. It works as shown in Figure 4.5. All of the shells we will see contain this built-in function, but you may also invoke the utility called **echo** (usually found in /bin) instead. This is sometimes useful, since some arguments and subtle behavior may vary among the different

built-ins and it can be confusing if you write scripts in more than one of these shells. (We'll look at writing shell scripts shortly.)

Shell Command: **echo** {*arg*}*
echo is a built-in shell command that displays all of its arguments to standard output. By default, it appends a newline to the output.

FIGURE 4.5
Description of the *echo* shell command.

Changing Directories: cd

The built-in *cd* command changes the current working directory of the shell to a new location and was described fully in Chapter 2.

METACHARACTERS

Some characters receive special processing by the shell and are known as *metacharacters*. All four shells share a core set of common metacharacters, whose meanings are shown in Figure 4.6. When you enter a command, the shell scans it for metacharacters and

Symbol	Meaning
>	Output redirection; writes standard output to a file.
>>	Output redirection; appends standard output to a file.
<	Input redirection; reads standard input from a file.
*	File substitution wildcard; matches zero or more characters.
?	File substitution wildcard; matches any single character.
[...]	File substitution wildcard; matches any character between brackets.
`command`	Command substitution; replaced by the output from *command*.
|	Pipe symbol; sends the output of one process to the input of another.
;	Used to sequence commands.
||	Conditional execution; executes a command if the previous one failed.
&&	Conditional execution; executes a command if the previous one succeeded.

FIGURE 4.6
Shell metacharacters.

(...)	Groups commands.
&	Runs a command in the background.
#	All characters that follow, up to a newline, are ignored by the shell and programs (i.e., signifies a comment).
$	Expands the value of a variable.
\	Prevents special interpretation of the next character.
<< *tok*	Input redirection; reads standard input from script, up to *tok*.

FIGURE 4.6 (*Continued*)

processes them specially. When all metacharacters have been processed, the command is finally executed. To turn off the special meaning of a metacharacter, precede it with a \ character. Here's an example:

```
$ echo hi > file          ...store output of echo in "file".
$ cat file                ...look at the contents of "file".
hi
$ echo hi \> file         ...inhibit > metacharacter.
hi > file                 ...> is treated like other characters.
$ _                       ...and output comes to terminal instead
```

This chapter describes the meaning of each metacharacter in the order in which it was listed in Figure 4.6.

REDIRECTION

The shell redirection facility allows you to do the following:

- store the output of a process to a file (*output redirection*)
- use the contents of a file as input to a process (*input redirection*)

Let's have a look at each facility, in turn.

Output Redirection

Output redirection is handy because it allows you to save a process' output into a file so it can be listed, printed, edited, or used as input to a future process. To redirect output, use either the > or >> metacharacter. The sequence

```
$ command > fileName
```

sends the standard output of *command* to the file with name *fileName*. The shell creates the file with name *fileName* if it doesn't already exist or overwrites its previous contents if it already exists. If the file already exists and doesn't have write permission, an error occurs. In the next example, I created a file called "alice.txt" by redirecting the output of the **cat** utility. Without parameters, **cat** simply copies its standard input—which in this case is the keyboard—to its standard output:

```
$ cat > alice.txt                    ...create a text file.
In my dreams that fill the night,
I see your eyes,
^D                                   ...end-of-input.
$ cat alice.txt                      ...look at its contents.
In my dreams that fill the night,
I see your eyes,
$ _
```

The sequence

```
$ command >> fileName
```

appends the standard output of *command* to the file with name *fileName*. The shell creates the file with name *fileName* if it doesn't already exist. In the following example, I appended some text to the existing "alice.txt" file:

```
$ cat > alice.txt                ...append to the file.
And I fall into them,
Like Alice fell into Wonderland.
^D                                   ...end-of-input.
$ cat alice.txt                  ...look at the new contents.
In my dreams that fill the night,
I see your eyes,
And I fall into them,
Like Alice fell into Wonderland.
$ _
```

By default, both forms of output redirection leave the standard error channel connected to the terminal. However, both shells have variations of output redirection that allow them to redirect the standard error channel. The C, Korn, and Bash shells also provide protection against accidental overwriting of a file due to output redirection. (These facilities are described in later chapters.)

Input Redirection

Input redirection is useful because it allows you to prepare a process' input and store it in a file for later use. To redirect input, use either the < or << metacharacter. The sequence

```
$ command < fileName
```


executes *command*, using the contents of the file *fileName* as its standard input. If the file doesn't exist or doesn't have read permission, an error occurs. In the following example, I sent myself the contents of "alice.txt" via the **mail** utility:

```
$ mail glass < alice.txt                ...send myself mail.
$ mail                                       ...look at my mail.
Mail version SMI 4.0 Sat Oct 13 20:32:29 PDT 1990  Type ? for help.
>N  1 glass@utdallas.edu Mon Feb  2 13:29   17/550
& 1                                          ...read message #1.
From: Graham Glass <glass@utdallas.edu>
To: glass@utdallas.edu
In my dreams that fill the night,
I see your eyes,
And I fall into them,
Like Alice fell into Wonderland
& q                                          ...quit mail.
$ _
```

When the shell encounters a sequence of the form

```
$ command << word
```

it copies its standard input up to, but not including, the line starting with *word* into a buffer and then executes *command*, using the contents of the buffer as its standard input. This facility, which is utilized almost exclusively to allow shell programs (*scripts*) to supply the standard input to other commands as in-line text, is revisited in more detail later on in the chapter.

FILENAME SUBSTITUTION (WILDCARDS)

All shells support a wildcard facility that allows you to select files from the file system that satisfy a particular name pattern. Any word on the command line that contains at least one of the wildcard metacharacters is treated as a pattern and is replaced by an alphabetically sorted list of all the matching filenames. This act of pattern replacement is called *globbing*. The wildcards and their meanings are as shown in Figure 4.7.

Wildcard	Meaning
*	Matches any string, including the empty string.
?	Matches any single character.
[..]	Matches any one of the characters between the brackets. A range of characters may be specified by separating a pair of characters by a dash.

FIGURE 4.7
Shell wildcards.

You may prevent the shell from processing the wildcards in a string by surrounding the string with single quotes (apostrophes) or double quotes. (See "Quoting" later in the chapter for more details.) A / character in a filename must be matched explicitly. Here are some examples of wildcards in action:

```
$ ls -FR                ...recursively list my current directory.
a.c   b.c   cc.c  dir1/  dir2/

dir1:
d.c  e.e

dir2:
f.d  g.c
$ ls *.c                ...any text followed by ".c".
a.c   b.c   cc.c
$ ls ?.c                ...one character followed by ".c".
a.c  b.c
$ ls [ac]*              ...any string beginning with "a" or "c".
a.c   cc.c
$ ls [A-Za-z]*          ...any string beginning with a letter.
a.c   b.c   cc.c
$ ls dir*/*.c           ...all ".c" files in "dir*" directories.
dir1/d.c  dir2/g.c
$ ls */*.c              ...all ".c" files in any subdirectory.
dir1/d.c  dir2/g.c
$ ls *2/?.? ?.?
a.c   b.c   dir2/f.d  dir2/g.c
$ _
```

The result of a pattern that has no matches is shell specific. Some shells have a mechanism for turning off wildcard replacement.

PIPES

The shell allows you to use the standard output of one process as the standard input of another process by connecting the processes together via the pipe (|) metacharacter. The sequence

```
$ command1 | command2
```

causes the standard output of *command1* to "flow through" to the standard input of *command2*. Any number of commands may be connected by pipes. A sequence of commands chained together in this way is called a *pipeline*. Pipelines support one of the basic UNIX philosophies, which is that large problems can often be solved by a chain of smaller processes, each performed by a relatively small, reusable utility. The standard

error channel is not piped through a standard pipeline, although some shells support this capability.

In the following example, I piped the output of the **ls** utility to the input of the **wc** utility to count the number of files in the current directory (see Chapter 2 for a description of **wc**):

```
$ ls                              ...list the current directory.
a.c   b.c   cc.c   dir1   dir2
$ ls | wc -w                      ...count the entries.
       5
$ _
```

Figure 4.8 is an illustration of a pipeline.

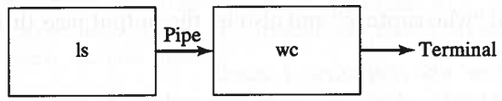

FIGURE 4.8
A simple pipeline.

In the next example, I piped the contents of the "/etc/passwd" file into the **awk** utility to extract the first field of each line. The output of **awk** was then piped to the sort utility, which sorted the lines alphabetically. The result was a sorted list of every user on the system. The commands are as follows (the **awk** utility is described fully in Chapter 3):

```
$ head -4 /etc/passwd       ...look at the password file.
root:eJ2S10rVe8mCg:0:1:Operator:/:/bin/csh
nobody:*:65534:65534::/:
daemon:*:1:1::/:
sys:*:2:2::/:/bin/csh
$ cat /etc/passwd | awk -F: '{ print $1 }' | sort
audit
bin
daemon
glass
ingres
news
nobody
root
sync
sys
tim
uucp
$ _
```

Figure 4.9 is an illustration of a pipeline that sorts.

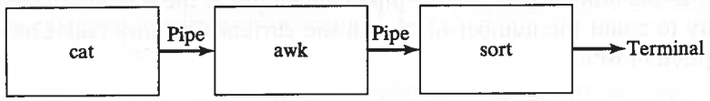

FIGURE 4.9

A pipeline that sorts.

There's a very handy utility called **tee** that allows you to copy the output of a pipe to a file while still allow sing that output to flow down the pipeline. As you might have guessed, the name of this utility comes from the "T" connections that plumbers use. Figure 4.10 shows how **tee** works. In the following example, I copied the output of **who** to a file called "who.capture" and also let the output pass through to **sort**:

```
$ who | tee who.capture | sort
ables    ttyp6   May  3 17:54 (gw.waterloo.com)
glass    ttyp0   May  3 18:49 (bridge05.utdalla)
posey    ttyp2   Apr 23 17:44 (blackfoot.utdall)
posey    ttyp4   Apr 23 17:44 (blackfoot.utdall)
$ cat who.capture          ...look at the captured data.
glass    ttyp0   May  3 18:49 (bridge05.utdalla)
posey    ttyp2   Apr 23 17:44 (blackfoot.utdall)
posey    ttyp4   Apr 23 17:44 (blackfoot.utdall)
ables    ttyp6   May  3 17:54 (gw.waterloo.com)
$
```

Utility: **tee** -ia {fileName}+

The **tee** utility copies its standard input to the specified files and to its standard output. The **-a** option causes the input to be appended to the files, rather than overwriting them. The **-i** option causes interrupts to be ignored.

FIGURE 4.10

Description of the **tee** command.

Notice that the output is captured directly from the **who** utility before the list is sorted.

COMMAND SUBSTITUTION

A command surrounded by grave accents (`) is executed, and its standard output is inserted in the command in its place. Any newlines in the output are replaced with spaces, as in the following examples:

```
$ echo the date today is `date`
the date today is Mon Feb 2 00:41:55 CST 1998
$ _
```

It's possible to do some crafty things by combining pipes and command substitution. For example, the **who** utility (described in Chapter 9) produces a list of all the users on the system, and the **wc** utility (described in Chapter 2) counts the number of words or lines in its input. By piping the output of **who** to the **wc** utility, it's possible to count the number of users on the system:

```
$ who              ...look at the output of who.
posey    ttyp0   Jan 22 15:31    (blackfoot:0.0)
glass    ttyp3   Feb  3 00:41    (bridge05.utdalla)
huynh    ttyp5   Jan 10 10:39    (atlas.utdallas.e)
$ echo there are `who | wc -l` users on the system
there are 3 users on the system
$ _
```

The output of command substitution may be used as part of another command. For example, the **vi** utility allows you to specify, on the command line, a list of files to be edited. These files are then visited by the editor one after the other. The **grep** utility, described in Chapter 3, has a **-l** option which returns a list of all the files on the command line that contain a specified pattern. By combining these two features via command substitution and using the following single command, it's possible to specify that **vi** be invoked for all files ending in ".c" that contain the pattern "debug":

```
$ vi `grep -l debug *.c`
```

SEQUENCES

If you enter a series of simple commands or pipelines separated by semicolons, the shell will execute them in sequence, from left to right. This facility is useful for type-ahead (and think-ahead) addicts who like to specify an entire sequence of actions at once. Here's an example:

```
$ date; pwd; ls    ...execute three commands in sequence.
Mon Feb  2 00:11:10 CST 1998
/home/glass/wild
a.c   b.c   cc.c  dir1  dir2
$ _
```

Each command in a sequence may be individually I/O redirected, as shown in the following example:

```
$ date > date.txt; ls; pwd > pwd.txt
a.c       b.c       cc.c      date.txt  dir1      dir2
```

```
$ cat date.txt                  ...look at output of date.
Mon Feb  2 00:12:16 CST 1998
$ cat pwd.txt                   ...look at output of pwd.
/home/glass
$ _
```

Conditional Sequences

Every UNIX process terminates with an exit value. By convention, an exit value of 0 means that the process completed successfully, and a nonzero exit value indicates failure. All built-in shell commands return 1 if they fail. You may construct sequences that make use of this exit value as follows:

- If you specify a series of commands separated by && tokens, the next command is executed only if the previous command returns an exit code of 0.
- If you specify a series of commands separated by || tokens, the next command is executed only if the previous command returns a nonzero exit code.

The && and || metacharacters therefore mirror the operation of their counterpart C operators.

For example, if the C compiler **cc** compiles a program without fatal errors, it creates an executable program called "a.out" and returns an exit code of 0; otherwise, it returns a nonzero exit code. The following conditional sequence compiles a program called "myprog.c" and executes the "a.out" file only if the compilation succeeds:

```
$ cc myprog.c && a.out
```

The following example compiles a program called "myprog.c" and displays an error message if the compilation fails:

```
$ cc myprog.c || echo compilation failed.
```

Exit codes are discussed in more detail toward the end of the chapter.

GROUPING COMMANDS

Commands may be grouped by placing them between parentheses, which causes them to be executed by a child shell (*subshell*). The commands in a given group share the same standard input, standard output, and standard error channels, and the group may be redirected and piped as if it were a simple command. Here are some examples:

```
$ date; ls; pwd > out.txt       ...execute a sequence
Mon Feb  2 00:33:12 CST 1998    ...output from date.
a.c       b.c                   ...output from ls.
$ cat out.txt                   ...only pwd was redirected.
/home/glass
$ (date; ls; pwd) > out.txt     ...group and then redirect.
```



```
$ cat out.txt                   ...all output was redirected.
Mon Feb  2 00:33:28 CST 1998
a.c
b.c
/home/glass
$ _
```

BACKGROUND PROCESSING

If you follow a simple command, pipeline, sequence of pipelines, or group of commands with the & metacharacter, a subshell is created to execute the commands as a background process that runs concurrently with the parent shell and does not take control of the keyboard. Background processing is therefore very useful for performing several tasks simultaneously, as long as the background tasks do not require keyboard input. In windowed environments, it's more common to run each command within its own window than to run many commands in one window using the background facility. When a background process is created, the shell displays some information that may be employed to control the process at a later stage. The exact format of this information is shell specific.

In the next example, I executed a **find** command in the foreground to locate the file called "a.c". This command took quite a while to execute, so I decided to run the next **find** command in the background. The shell displayed the background process' unique process ID number and then immediately gave me another prompt, allowing me to continue my work. Note that the output of the background process continued to be displayed at my terminal, which was inconvenient. In the next few sections, I'll show you how you can use the process ID number to control the background process and how to prevent background processes from messing up your terminal. The following are the commands to locate the file "a.c":

```
$ find . -name a.c -print       ...search for "a.c".
./wild/a.c
./reverse/tmp/a.c
$ find . -name b.c -print &     ...search in the background.
27174                           ...process ID number
$ date                          ...run "date" in the foreground.
./wild/b.c                  ...output from background "find".
Mon Feb  2 18:10:42 CST 1998        ...output from date.
$ ./reverse/tmp/b.c             ...more from background "find"
      ...came after we got the shell prompt so we don't
      ...get another one.
```

You may specify several background commands on a single line by separating each command with an ampersand, as shown in the following example:

```
$ date & pwd &          ...create two background processes.
27310                           ...process ID of "date".
27311                           ...process ID of "pwd".
```

```
/home/glass                          ...output from "date".
$ Mon Feb  2 18:37:22 CST 1998       ...output from "pwd".
$ _
```

REDIRECTING BACKGROUND PROCESSES

Redirecting Output

To prevent the output of a background process from arriving at your terminal, redirect its output to a file. In the next example, I redirected the standard output of the **find** command to a file called "find.txt". As the command was executing, I watched it grow via the ls command:

```
$ find . -name a.c -print > find.txt &
27188                          ...process ID of "find".
$ ls -l find.txt               ...look at "find.txt".
-rw-r--r--  1 glass              0 Feb  3 18:11 find.txt
$ ls -l find.txt               ...watch it grow.
-rw-r--r--  1 glass             29 Feb  3 18:11 find.txt
$ cat find.txt                       ...list "find.txt".
./wild/a.c
./reverse/tmp/a.c
$ _
```

Another alternative is to mail the output to yourself, as shown in the following example:

```
$ find . -name a.c -print | mail glass &
27193
$ cc program.c                                  ...do other useful work.
$ mail                                          ...read my mail.
Mail version SMI 4.0 Sat Oct 13 20:32:29 PDT 1990  Type ? for help.
>N  1 glass@utdallas.edu Mon Feb  3 18:12   10/346
& 1
From: Graham Glass <glass@utdallas.edu>
To: glass@utdallas.edu
./wild/a.c                          ...the output from "find".
./reverse/tmp/a.c
& q
$ _
```

Some utilities also produce output on the standard error channel, which must be redirected *in addition* to standard output. The next chapter describes in detail how this is done, but here is an example of how it is done in the Bourne and Korn shells just in case you're interested:

```
$ man ps > ps.txt &                  ...save documentation in background.
27203
$ Reformatting page.  Wait           ...shell prompt comes here.
```



```
done                                 ...standard error messages.
man ps > ps.txt 2>&1 &               ...redirect error channel too.
27212
$ _                                  ...all output is redirected.
```

Redirecting Input

When a background process attempts to read from a terminal, the terminal automatically sends it an error signal, which causes it to terminate. In the next example, I ran the **chsh** utility in the background. It immediately issued the "Login shell unchanged" message and terminated, never bothering to wait for any input. I then ran the **mail** utility in the background, which similarly issued the message "No message !?!". The commands are as follows:

```
$ chsh &                          ...run "chsh" in background.
27201
$ Changing NIS login shell for glass on csservr1.
Old shell: /bin/sh
New shell: Login shell unchanged.    ...didn't wait.

mail glass &                      ...run "mail" in background.
27202
$ No message !?!                  ...don't wait for keyboard input.
```

SHELL PROGRAMS: SCRIPTS

Any series of shell commands may be stored inside a regular text file for later execution. A file that contains shell commands is called a *script*. Before you can run a script, you must give it execute permission by using the **chmod** utility. Then, to run the script, you only need to type its name. Scripts are useful for storing commonly used sequences of commands and range in complexity from simple one-liners to full-blown programs. The control structures supported by the languages built into the shells are sufficiently powerful to enable scripts to perform a wide variety of tasks. System administrators find scripts particularly useful for automating repetitive administrative tasks, such as warning users when their disk usage goes beyond a certain limit.

When a script is run, the system determines which shell the script was written for and then executes the shell, using the script as its standard input. The system decides which shell the script is written for by examining the first line of the script. Here are the rules that it uses:

- If the first line is just a #, then the script is interpreted by the shell from which it was executed as a command.
- If the first line is of the form #! *pathName*, then the executable program *pathName* is used to interpret the script.
- If neither of the first two rules applies, then the script is interpreted by a Bourne shell.

If a # appears on any line other than the first, all characters up to the end of that line are treated as a comment. Scripts should be liberally commented in the interests of maintainability.

When you write your own scripts, I recommend that you use the #! form to specify which shell the script is designed for, as that form is unambiguous and doesn't require the reader to be aware of the default rules.

Here is an example that illustrates the construction and execution of two scripts, one for the C shell and the other for the Korn shell:

```
$ cat > script.csh              ...create the C shell script.
#!/bin/csh
# This is a sample C shell script.
echo -n the date today is       # in csh, -n omits newline
date        # output today's date.
^D                              ...end-of-input.
$ cat > script.ksh              ...create the Korn shell script.
#!/bin/ksh
# This is a sample Korn shell script.
echo "the date today is \c"     # in ksh, \c omits the nl
date        # output today's date.
^D                                      ...end-of-input.
$ chmod +x script.csh script.ksh        ...make them executable.
$ ls -lF script.csh script.ksh          ...look at attributes.
-rwxr-xr-x  1 glass         138 Feb  1 19:46 script.csh*
-rwxr-xr-x  1 glass         142 Feb  1 19:47 script.ksh*
$ script.csh                    ...execute the C shell script.
the date today is Sun Feb  1 19:50:00 CST 1998
$ script.ksh                    ...execute the Korn shell script.
the date today is Sun Feb  1 19:50:05 CST 1998
$ _
```

The ".csh" and ".ksh" extensions of my scripts are used only for clarity; scripts can be called anything at all and don't even need an extension.

Note the usage of "\c" and "-n" in the preceding examples of the **echo** command. Different versions of "/bin/echo" use one or the other to omit the newline. It may also depend on the shell being used: If the shell has a built-in *echo* function, then the specifics of "/bin/echo" won't matter. You'll want to experiment with your particular shell and echo combination; it isn't quite as simple as I implied in the preceding comments.

SUBSHELLS

When you log into a UNIX system, you execute an initial login shell. This shell executes any simple commands that you enter. However, there are several circumstances under which your current (*parent*) shell creates a new (*child*) shell to perform some tasks:

- When a grouped command such as (ls; pwd; date) is executed, the parent shell creates a child shell to execute the grouped commands. If the command is not executed in the background, the parent shell sleeps until the child shell terminates.
- When a script is executed, the parent shell creates a child shell to execute the commands in the script. If the script is not executed in the background, the parent shell sleeps until the child shell terminates.
- When a background job is executed, the parent shell creates a child shell to execute the background commands. The parent shell continues to run concurrently with the child shell.

A child shell is called a *subshell*. Just like any other UNIX process, a subshell has its own current working directory, so *cd* commands executed in a subshell do not affect the working directory of the parent shell:

```
$ pwd                   ...display my login shell's current dir.
/home/glass
$ (cd /; pwd)           ...the subshell moves and executes pwd.
/                       ...output comes from the subshell.
$ pwd                   ...my login shell never moved.
/home/glass
$ _
```

Every shell contains two data areas: an environment space and a local variable space. A child shell inherits a copy of its parent's environment space and a clean local variable space, as shown in Figure 4.11.

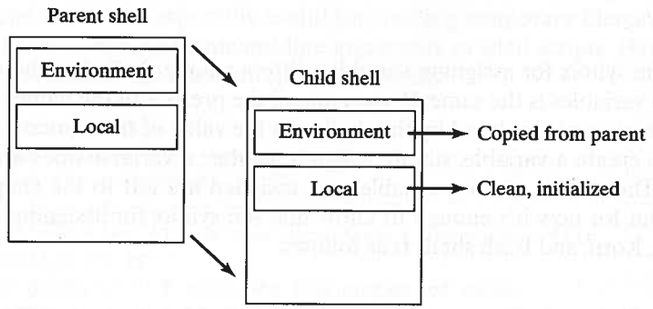

FIGURE 4.11

Child shell data spaces.

VARIABLES

A shell supports two kinds of variables: *local* and *environment* variables. Both hold data in a string format. The main difference between them is that when a shell invokes a subshell, the child shell gets a copy of its parent shell's environment variables, but not its local variables. Environment variables are therefore used for transmitting useful information between parent shells and their children.

Every shell has a set of predefined environment variables that are usually initialized by the start-up files described in later chapters. Similarly, every shell has a set of predefined local variables that have special meanings to the shell. Other environment and local variables may be created as needed and are particularly useful in writing scripts. Figure 4.12 shows a list of the predefined environment variables that are common to all shells.

Name	Meaning
$HOME	the full pathname of your home directory
$PATH	a list of directories to search for commands
$MAIL	the full pathname of your mailbox
$USER	your username
$SHELL	the full pathname of your login shell
$TERM	the type of your terminal

FIGURE 4.12
Predefined shell variables.

The syntax for assigning variables differs among shells, but the way that you access the variables is the same: If you append the prefix $ to the name of a variable, this token sequence is replaced by the shell with the value of the named variable.

To create a variable, simply assign it a value; a variable does not have to be declared. The details of how variables are assigned are left to the chapters on specific shells, but for now it's enough to know that the syntax for assigning a variable in the Bourne, Korn, and Bash shells is as follows:

```
variableName=value              ...place no spaces around the =.
```

In the following example, I displayed the values of some common shell environment variables:

```
$ echo HOME = $HOME, PATH = $PATH        ...list two variables.
HOME = /home/glass, PATH = /bin:/usr/bin:/usr/sbin
$ echo MAIL = $MAIL                      ...list another.
MAIL = /var/mail/glass
$ echo USER = $USER, SHELL = $SHELL, TERM=$TERM
USER = glass, SHELL = /bin/sh, TERM=vt100
$ _
```


The next example illustrates the difference between local and environment variables. I assigned values to two local variables and then made one of them an environment variable by using the Bourne shell *export* command (described fully in Chapter 5). I then created a child Bourne shell and displayed the values of the variables that I had assigned in the parent shell. Note that the value of the environment variable was copied into the child shell, but the value of the local variable was not. Finally, I typed a *Control-D* to terminate the child shell and restart the parent shell, and then I displayed the original variables. The commands are as follows:

```
$ firstname=Graham          ...set a local variable.
$ lastname=Glass            ...set another local variable.
$ echo $firstname $lastname   ...display their values.
Graham Glass
$ export lastname           ...make "lastname" an environment var.
$ sh                        ...start a child shell; the parent sleeps.
$ echo $firstname $lastname   ...display values again.
Glass                       ...note that firstname wasn't copied.
$ ^D                        ...terminate child; the parent awakens.
$ echo $firstname $lastname   ...they remain unchanged.
Graham Glass
$ _
```

Figure 4.13 shows several common built-in variables that have a special meaning. The first special variable is especially useful for creating temporary filenames, and the rest are handy for accessing command-line arguments in shell scripts. Here's an example that illustrates all of the common special variables:

```
$ cat script.sh                                    ...list the script.
echo the name of this script is $0
echo the first argument is $1
echo a list of all the arguments is $*
echo this script places the date into a temporary file
echo called $1.$$
date > $1.$$          # redirect the output of date.
ls $1.$$              # list the file.
rm $1.$$              # remove the file.
$ script.sh paul ringo george john     ...execute it.
the name of this script is script.sh
the first argument is paul
a list of all the arguments is paul ringo george john
this script places the date into a temporary file
called paul.24321
paul.24321
$ _
```

Name	Meaning
$$	The process ID of the shell.
$0	The name of the shell script (if applicable).
$1..$9	$n refers to the nth command line argument (if applicable).
$*	A list of all the command-line arguments.

FIGURE 4.13
Special built-in shell variables.

QUOTING

Oftentimes, you want to inhibit the shell's wildcard replacement, variable substitution, or command substitution mechanisms. The shell's quoting system allows you to do just that. Here's the way it works:

- Single quotes (') inhibit wildcard replacement, variable substitution, and command substitution.
- Double quotes (") inhibit wildcard replacement only.
- When quotes are nested, only the outer quotes have any effect.

The following example illustrates the difference between the two kinds of quotes:

```
$ echo 3 * 4 = 12          ...remember, * is a wildcard.
3 a.c b.c c.c 4 = 12
$ echo "3 * 4 = 12"          ...double quotes inhibit wildcards.
3 * 4 = 12
$ echo '3 * 4 = 12'          ...single quotes inhibit wildcards.
3 * 4 = 12
$ name=Graham
```

By using single quotes (apostrophes) around the text, we inhibit all wildcard replacement, variable substitution, and command substitution:

```
$ echo 'my name is $name - date is `date`'
my name is $name and the date is `date`
```

By using double quotes around the text, we inhibit wildcard replacement, but allow variable and command substitution:

```
$ echo "my name is $name - date is `date`"
my name is Graham - date is Mon Feb  2 23:14:56 CST 1998
$ _
```

TERMINATION AND EXIT CODES

Every UNIX process terminates with an exit value. By convention, an exit value of 0 means that the process completed successfully, and a nonzero exit value indicates failure. All built-in commands return 1 if they fail. In the Bourne, Korn, and Bash shells, the special shell variable $? always contains the value of the previous command's exit code. In the C shell, the $status variable holds the exit code. In the following example, the **date** utility succeeded, whereas the **cc** and **awk** utilities failed:

```
$ date                                  ...date succeeds.
Sat Feb  2 22:13:38 CST 2002
$ echo $?                               ...display its exit value.
0                                       ...indicates success.
$ cc prog.c                  ...compile a non-existent program.
cpp: Unable to open source file 'prog.c'.
$ echo $?                               ...display its exit value.
1                                       ...indicates failure.
$ awk                                   ...use awk illegally.
awk: Usage: awk [-Fc] [-f source | 'cmds'] [files]
$ echo $?                               ...display its exit value.
2                                       ...indicates failure.
$ _
```

Any script that you write should always explicitly return an exit code. To terminate a script, use the built-in *exit* command, which works as shown in Figure 4.21. If a shell doesn't include an explicit *exit* statement, the exit value of the last command is returned by default. The script in the following example returned an exit value of 3:

```
$ cat script.sh                         ...look at the script.
echo this script returns an exit code of 3
exit 3
$ script.sh                             ...execute the script.
this script returns an exit code of 3
$ echo $?                               ...look at the exit value.
3
$ _
```