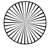
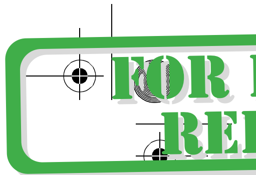
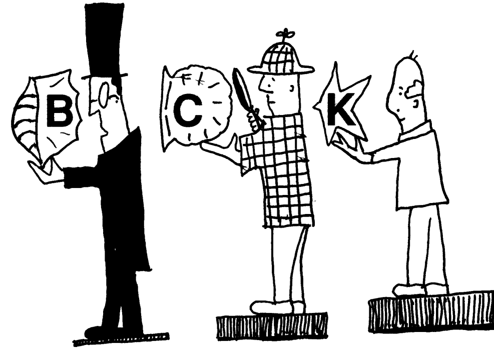




chapter

1

Introduction to UNIX Shells



1.1 Definition and Function

The shell is a special program used as an interface between the user and the heart of the UNIX operating system, a program called the *kernel*, as shown in Figure 1.1. The kernel is loaded into memory at boot-up time and manages the system until shutdown. It creates and controls processes, and manages memory, file systems, communications, and so forth. All other programs, including shell programs, reside out on the disk. The kernel loads those programs into memory, executes them, and cleans up the system when they terminate. The shell is a utility program that starts up when you log on. It allows users to interact with the kernel by interpreting commands that are typed either at the command line or in a script file.

When you log on, an interactive shell starts up and prompts you for input. After you type a command, it is the responsibility of the shell to (a) parse the command line; (b) handle wildcards, redirection, pipes, and job control; and (c) search for the command, and if found, execute that command. When you first learn UNIX, you spend most of your time executing commands from the prompt. You use the shell interactively.

If you type the same set of commands on a regular basis, you may want to automate those tasks. This can be done by putting the commands in a file, called a *script file*, and then executing the file. A shell script is much like a batch file: It is a list of UNIX commands typed into a file, and then the file is executed. More sophisticated scripts contain programming constructs for making decisions, looping, file testing, and so forth. Writing scripts not only requires learning programming constructs and techniques, but assumes that you have a good understanding of UNIX utilities and how they work. There are some utilities, such as *grep, sed*, and *awk,* that are extremely powerful tools used in scripts for the manipulation of command output and files. After you have become familiar with these tools and the programming constructs for your particular shell, you will be ready to start writing useful scripts. When executing commands from within a script, you are using the shell as a programming language.

quigley.book Page 2 Friday, September 28, 2001 10:00 AM

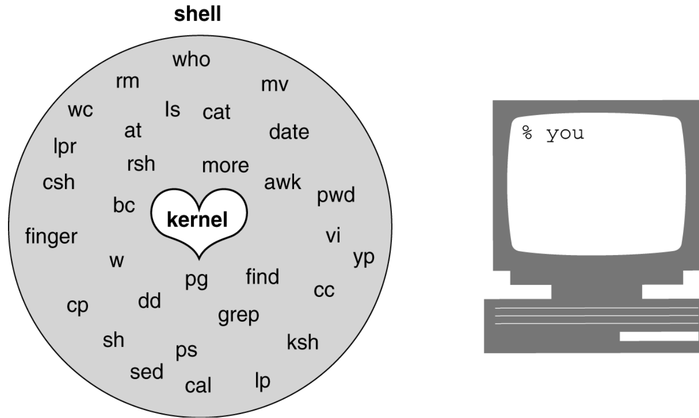


Figure 1.1 The kernel, the shell, and you.

1.1.1 The Three Major UNIX Shells

The three prominent and supported shells on most UNIX systems are the *Bourne* shell (AT&T shell), the *C* shell (Berkeley shell), and the *Korn* shell (superset of the Bourne shell). All three of these behave pretty much the same way when running interactively, but have some differences in syntax and efficiency when used as scripting languages.

The Bourne shell is the standard UNIX shell, and is used to administer the system. Most of the system administration scripts, such as the *rc start* and *stop* scripts and *shutdown* are Bourne shell scripts, and when in single user mode, this is the shell commonly used by the administrator when running as root. This shell was written at AT&T and is known for being concise, compact, and fast. The default Bourne shell prompt is the dollar sign ($).

The C shell was developed at Berkeley and added a number of features, such as command line history, aliasing, built-in arithmetic, filename completion, and job control. The C shell has been favored over the Bourne shell by users running the shell interactively, but administrators prefer the Bourne shell for scripting, because Bourne shell scripts are simpler and faster than the same scripts written in C shell. The default C shell prompt is the percent sign (%).

The Korn shell is a superset of the Bourne shell written by David Korn at AT&T. A number of features were added to this shell above and beyond the enhancements of the C shell. Korn shell features include an editable history, aliases, functions, regular expression wildcards, built-in arithmetic, job control, coprocessing, and special debugging features. The Bourne shell is almost completely upward-compatible with the Korn shell, so older Bourne shell programs will run fine in this shell. The default Korn shell prompt is the dollar sign ($).

1.1.2 The Linux Shells

Although often called "Linux" shells, Bash and TC shells are freely available and can be compiled on any UNIX system; in fact, the shells are now bundled with Solaris 8 and Sun's

quigley.book Page 3 Friday, September 28, 2001 10:00 AM

UNIX operating system. But when you install Linux, you will have access to the GNU shells and tools, and not the standard UNIX shells and tools. Although Linux supports a number of shells, the Bourne Again shell (*bash)* and the TC shell (*tcsh)* are by far the most popular. The Z shell is another Linux shell that incorporates a number of features from the Bourne Again shell, the TC shell, and the Korn shell. The Public Domain Korn shell (*pdksh*) a Korn shell clone, is also available, and for a fee you can get AT&T's Korn shell, not to mention a host of other unknown smaller shells.

To see what shells are available under your version of Linux, look in the file, */etc/shell*.

To change to one of the shells listed in */etc/shell*, type the *chsh* command and the name of the shell. For example, to change permanently to the TC shell, use the *chsh* command. At the prompt, type:

```
chsh /bin/tcsh
```

1.1.3 History of the Shell

The first significant, standard UNIX shell was introduced in V7 (seventh edition of AT&T) UNIX in late 1979, and was named after its creator, Stephen Bourne. The Bourne shell as a programming language is based on a language called Algol, and was primarily used to automate system administration tasks. Although popular for its simplicity and speed, it lacks many of the features for interactive use, such as history, aliasing, and job control. Enter *bash*, the Bourne Again shell, which was developed by Brian Fox of the Free Software Foundation under the GNU copyright license and is the default shell for the very popular Linux operating system. It was intended to conform to the IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools standard. *Bash* also offers a number of new features (both at the interactive and programming level) missing in the original Bourne shell (yet Bourne shell scripts will still run unmodified). It also incorporates the most useful features of both the C shell and Korn shell. It's big. The improvements over Bourne shell are: command line history and editing, directory stacks, job control, functions, aliases, arrays, integer arithmetic (in any base from 2 to 64), and Korn shell features, such as extended metacharacters, select loops for creating menus, the *let* command, etc.

The C shell, developed at the University of California at Berkeley in the late 1970s, was released as part of 2BSD UNIX. The shell, written primarily by Bill Joy, offered a number of additional features not provided in the standard Bourne shell. The C shell is based on the C programming language, and when used as a programming language, it shares a similar syntax. It also offers enhancements for interactive use, such as command line history, aliases, and job control. Because the shell was designed on a large machine and a number of additional features were added, the C shell has a tendency to be slow on small machines and sluggish even on large machines when compared to the Bourne shell.

The TC shell is an expanded version of the C shell. Some of the new features are: command line editing (*emacs* and *vi*), scrolling the history list, advanced filename, variable, and command completion, spelling correction, scheduling jobs, automatic locking and logout, time stamps in the history list, etc. It's also big.

With both the Bourne shell and the C shell available, the UNIX user now had a choice, and conflicts arose over which was the better shell. David Korn, from AT&T,

quigley.book Page 4 Friday, September 28, 2001 10:00 AM

invented the Korn shell in the mid-1980s. It was released in 1986 and officially became part of the SVR4 distribution of UNIX in 1988. The Korn shell, really a superset of the Bourne shell, runs not only on UNIX systems, but also on OS/2, VMS, and DOS. It provides upward-compatibility with the Bourne shell, adds many of the popular features of the C shell, and is fast and efficient. The Korn shell has gone through a number of revisions. The most widely used version of the Korn shell is the 1988 version, although the 1993 version is gaining popularity. Linux users may find they are running the free version of the Korn shell, called The Public Domain Korn shell, or simply *pdksh*, a clone of David Korn's 1988 shell. It is free and portable and currently work is underway to make it fully compatible with its namesake, Korn shell, and to make it POSIX compliant. Also available is the Z shell (*zsh*), another Korn shell clone with TC shell features, written by Paul Falsted, and freely available at a number of Web sites.

1.1.4 Uses of the Shell

One of the major functions of a shell is to interpret commands entered at the command line prompt when running interactively. The shell parses the command line, breaking it into words (called *tokens*), separated by whitespace, which consists of tabs, spaces, or a newline. If the words contain special metacharacters, the shell evaluates them. The shell handles file I/O and background processing. After the command line has been processed, the shell searches for the command and starts its execution.

Another important function of the shell is to customize the user's environment, normally done in shell initialization files. These files contain definitions for setting terminal keys and window characteristics; setting variables that define the search path, permissions, prompts, and the terminal type; and setting variables that are required for specific applications such as windows, text-processing programs, and libraries for programming languages. The Korn shell and C shell also provide further customization with the addition of history and aliases, built-in variables set to protect the user from clobbering files or inadvertently logging out, and to notify the user when a job has completed.

The shell can also be used as an interpreted programming language. Shell programs, also called scripts, consist of commands listed in a file. The programs are created in an editor (although on-line scripting is permitted). They consist of UNIX commands interspersed with fundamental programming constructs such as variable assignment, conditional tests, and loops. You do not have to compile shell scripts. The shell interprets each line of the script as if it had been entered from the keyboard. Because the shell is responsible for interpreting commands, it is necessary for the user to have an understanding of what those commands are. See Appendix A for a list of useful commands.

1.1.5 Responsibilities of the Shell

The shell is ultimately responsible for making sure that any commands typed at the prompt get properly executed. Included in those responsibilities are:

1. Reading input and parsing the command line.
2. Evaluating special characters.

quigley.book Page 5 Friday, September 28, 2001 10:00 AM

3. Setting up pipes, redirection, and background processing.
4. Handling signals.
5. Setting up programs for execution.

Each of these topics is discussed in detail as it pertains to a particular shell.

1.2 System Startup and the Login Shell

When you start up your system, the first process is called *init*. Each process has a process identification number associated with it, called the *PID*. Since *init* is the first process, its PID is 1. The *init* process initializes the system and then starts another process to open terminal lines and set up the standard input (*stdin*), standard output (*stdout*), and standard error (*stderr*), which are all associated with the terminal. The standard input normally comes from the keyboard; the standard output and standard error go to the screen. At this point, a login prompt would appear on your terminal.

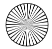
After you type your login name, you will be prompted for a password. The */bin/login* program then verifies your identity by checking the first field in the *passwd* file. If your username is there, the next step is to run the password you typed through an encryption program to determine if it is indeed the correct password. Once your password is verified, the *login* program sets up an initial environment consisting of variables that define the working environment that will be passed on to the shell. The *HOME, SHELL, USER,* and *LOGNAME* variables are assigned values extracted from information in the *passwd* file. The *HOME* variable is assigned your home directory; the *SHELL* variable is assigned the name of the login shell, which is the last entry in the *passwd* file. The *USER* and/or *LOGNAME* variables are assigned your login name. A *search path* variable is set so that commonly used utilities may be found in specified directories. When *login* has finished, it will execute the program found in the last entry of the *passwd* file. Normally, this program is a shell. If the last entry in the *passwd* file is */bin/csh*, the C shell program is executed. If the last entry in the *passwd* file is */bin/sh* or is null, the Bourne shell starts up. If the last entry is */bin/ksh*, the Korn shell is executed. This shell is called the *login shell*.

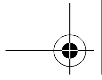
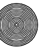
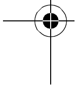
After the shell starts up, it checks for any systemwide initialization files set up by the system administrator and then checks your home directory to see if there are any shell-specific initialization files there. If any of these files exist, they are executed. The initialization files are used to further customize the user environment. After the commands in those files have been executed, a prompt appears on the screen. The shell is now waiting for your input.

1.2.1 Parsing the Command Line

When you type a command at the prompt, the shell reads a line of input and parses the command line, breaking the line into words, called tokens. Tokens are separated by spaces and tabs and the command line is terminated by a newline.[1] The shell then

1. The process of breaking the line up into tokens is called *lexical analysis*.



quigley.book Page 6 Friday, September 28, 2001 10:00 AM




checks to see whether the first word is a built-in command or an executable program located somewhere out on disk. If it is built-in, the shell will execute the command internally. Otherwise, the shell will search the directories listed in the path variable to find out where the program resides. If the command is found, the shell will fork a new process and then execute the program. The shell will sleep (or wait) until the program finishes execution and then, if necessary, will report the status of the exiting program. A prompt will appear and the whole process will start again. The order of processing the command line is as follows:

1. History substitution is performed (if applicable).
2. Command line is broken up into tokens, or words.
3. History is updated (if applicable).
4. Quotes are processed.
5. Alias substitution and functions are defined (if applicable).
6. Redirection, background, and pipes are set up.
7. Variable substitution (*$user*, *$name*, etc.) is performed.
8. Command substitution (echo for *today is 'date'*) is performed.
9. Filename substitution, called *globbing* (*cat abc.??*, *rm *.c*, etc.) is performed.
10. Program execution.

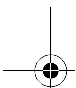
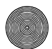
1.2.2 Types of Commands

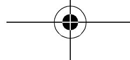
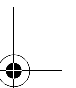
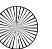
When a command is executed, it is an alias, a function, a built-in command, or an executable program on disk. Aliases are abbreviations (nicknames) for existing commands and apply to the C, TC, Bash, and Korn shells. Functions apply to the Bourne (introduced with AT&T System V, Release 2.0), Bash, and Korn shells. They are groups of commands organized as separate routines. Aliases and functions are defined within the shell's memory. Built-in commands are internal routines in the shell, and executable programs reside on disk. The shell uses the path variable to locate the executable programs on disk and forks a child process before the command can be executed. This takes time. When the shell is ready to execute the command, it evaluates command types in the following order:[2]

1. Aliases
2. Keywords
3. Functions (*bash*)
4. Built-in commands
5. Executable programs

If, for example, the command is *xyz* the shell will check to see if *xyz* is an alias. If not, is it a built-in command or a function? If neither of those, it must be an executable command residing on the disk. The shell then must search the path for the command.

2. Numbers 3 and 4 are reversed for Bourne and Korn(88) shells. Number 3 does not apply for C and TC shells.

quigley.book Page 7 Friday, September 28, 2001 10:00 AM

1.3 Processes and the Shell

A process is a program in execution and can be identified by its unique PID (process identification) number. The kernel controls and manages processes. A process consists of the executable program, its data and stack, program and stack pointer, registers, and all the information needed for the program to run. When you start the shell, it is a process. The shell belongs to a process group identified by the group's PID. Only one process group has control of the terminal at a time and is said to be running in the foreground. When you log on, your shell is in control of the terminal and waits for you to type a command at the prompt.

The shell can spawn other processes. In fact, when you enter a command at the prompt or from a shell script, the shell has the responsibility of finding the command either in its internal code (built-in) or out on the disk and then arranging for the command to be executed. This is done with calls to the kernel, called *system calls*. A system call is a request for kernel services and is the only way a process can access the system's hardware. There are a number of system calls that allow processes to be created, executed, and terminated. (The shell provides other services from the kernel when it performs redirection and piping, command substitution, and the execution of user commands.)

The system calls used by the shell to cause new processes to run are discussed in the following sections. See Figure 1.2.

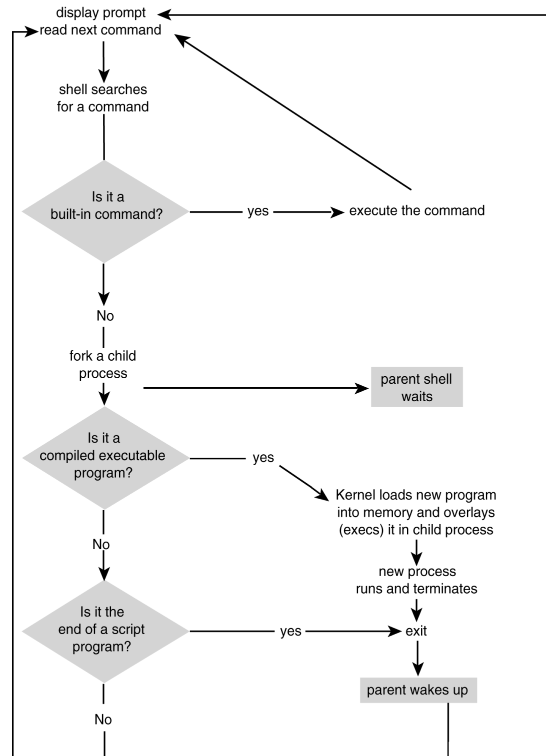


Figure 1.2 The shell and command execution.

quigley.book Page 8 Friday, September 28, 2001 10:00 AM

1.3.1 What Processes Are Running?

The *ps* Command. The *ps* command with its many options displays a list of the processes currently running in a number of formats. Example 1.1 shows all processes that are running by users on a Linux system. (See Appendix A for *ps* and its options.)

EXAMPLE 1.1

```
$ ps au  (BSD/Linux ps)  (use ps -ef for SVR4)
USER   PID %CPU %MEM  SIZE   RSS TTY STAT START  TIME COMMAND
ellie  456  0.0  1.3  1268   840   1 S    13:23  0:00 -bash
ellie  476  0.0  1.0  1200   648   1 S    13:23  0:00 sh /usr/X11R6/bin/sta
ellie  478  0.0  1.0  2028   676   1 S    13:23  0:00 xinit /home/ellie/.xi
ellie  480  0.0  1.6  1852  1068   1 S    13:23  0:00 fvwm2
ellie  483  0.0  1.3  1660   856   1 S    13:23  0:00 /usr/X11R6/lib/X11/fv
ellie  484  0.0  1.3  1696   868   1 S    13:23  0:00 /usr/X11R6/lib/X11/fv
ellie  487  0.0  2.0  2348  1304   1 S    13:23  0:00 xclock -bg #c0c0c0 -p
ellie  488  0.0  1.1  1620   724   1 S    13:23  0:00 /usr/X11R6/lib/X11/fv
ellie  489  0.0  2.0  2364  1344   1 S    13:23  0:00 xload -nolabel -bg gr
ellie  495  0.0  1.3  1272   848  p0 S    13:24  0:00 -bash
ellie  797  0.0  0.7   852   484  p0 R    14:03  0:00 ps au
root   457  0.0  0.4   724   296   2 S    13:23  0:00 /sbin/mingetty tty2
root   458  0.0  0.4   724   296   3 S    13:23  0:00 /sbin/mingetty tty3
root   459  0.0  0.4   724   296   4 S    13:23  0:00 /sbin/mingetty tty4
root   460  0.0  0.4   724   296   5 S    13:23  0:00 /sbin/mingetty tty5
root   461  0.0  0.4   724   296   6 S    13:23  0:00 /sbin/mingetty tty6
root   479  0.0  4.5 12092  2896   1 S    13:23  0:01 X :0
root   494  0.0  2.5  2768  1632   1 S    13:24  0:00 nxterm -ls -sb -fn
```

1.3.2 Creating Processes

The *fork* System Call. A process is created in UNIX with the *fork* system call. The *fork* system call creates a duplicate of the calling process. The new process is called the *child* and the process that created it is called the *parent.* The child process starts running right after the call to *fork*, and both processes initially share the CPU. The child process has a copy of the parent's environment, open files, real and user identifications, *umask*, current working directory, and signals.

When you type a command, the shell parses the command line and determines whether the first word is a built-in command or an executable command that resides out on the disk. If the command is built-in, the shell handles it, but if on the disk, the shell invokes the *fork* system call to make a copy of itself (Figure 1.3). Its child will search the path to find the command, as well as set up the file descriptors for redirection, pipes, command substitution, and background processing. While the child shell works, the parent normally sleeps. (See *wait,* below.)

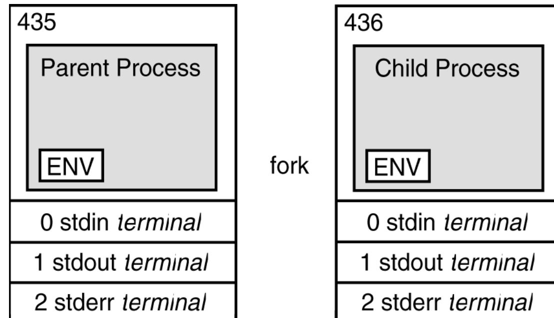


Figure 1.3 The *fork* system call.

The *wait* System Call. The parent shell is programmed to go to sleep (wait) while the child takes care of details such as handling redirection, pipes, and background processing. The *wait* system call causes the parent process to suspend until one of its children terminates. If *wait* is successful, it returns the PID of the child that died and the child's exit status. If the parent does not wait and the child exits, the child is put in a zombie state (suspended animation) and will stay in that state until either the parent calls *wait* or the parent dies.[3] If the parent dies before the child, the *init* process adopts any orphaned zombie process. The *wait* system call, then, is not just used to put a parent to sleep, but also to ensure that the process terminates properly.

The *exec* System Call. After you enter a command at the terminal, the shell normally forks off a new shell process: the child process. As mentioned earlier, the child shell is responsible for causing the command you typed to be executed. It does this by calling the *exec* system call. Remember, the user command is really just an executable program. The shell searches the path for the new program. If it is found, the shell calls the *exec* system call with the name of the command as its argument. The kernel loads this new program into memory in place of the shell that called it. The child shell, then, is overlaid with the new program. The new program becomes the child process and starts executing. Although the new process has its own local variables, all environment variables, open files, signals, and the current working directory are passed to the new process. This process exits when it has finished, and the parent shell wakes up.

The *exit* System Call. A new program can terminate at any time by executing the *exit* call. When a child process terminates, it sends a signal (*sigchild*) and waits for the parent to accept its exit status. The exit status is a number between 0 and 255. An exit status of zero indicates that the program executed successfully, and a nonzero exit status means that the program failed in some way.

For example, if the command *ls* had been typed at the command line, the parent shell would *fork* a child process and go to sleep. The child shell would then *exec* (overlay) the

3. To remove zombie processes, the system must be rebooted.

quigley.book Page 10 Friday, September 28, 2001 10:00 AM

ls program in its place. The *ls* program would run in place of the child, inheriting all the environment variables, open files, user information, and state information. When the new process finished execution, it would exit and the parent shell would wake up. A prompt would appear on the screen, and the shell would wait for another command. If you are interested in knowing how a command exited, each shell has a special built-in variable that contains the exit status of the last command that terminated. (All of this will be explained in detail in the individual shell chapters.) See Figure 1.4 for an example of process creation and termination.

EXAMPLE 1.2

```
(C Shell)
1   % cp filex filey
    % echo $status
    0
2   % cp xyz
    Usage: cp [-ip] f1 f2; or: cp [-ipr] f1 ... fn d2
    % echo $status
    1

(Bourne and Korn Shells)
3   $ cp filex filey
    $ echo $?
    0
    $ cp xyz
    Usage: cp [-ip] f1 f2; or: cp [-ipr] f1 ... fn d2
    $ echo $?
    1
```

EXPLANATION

1. The *cp* (copy) command is entered at the C shell command line prompt. After the command has made a copy of *filex* called *filey*, the program exits and the prompt appears. The csh *status* variable contains the exit status of the last command that was executed. If the status is zero, the *cp* program exited with success. If the exit status is nonzero, the *cp* program failed in some way.
2. When entering the *cp* command, the user failed to provide two filenames: the source and destination files. The *cp* program sent an error message to the screen and exited with a status of one. That number is stored in the csh *status* variable. Any number other than zero indicates that the program failed.
3. The Bourne and Korn shells process the *cp* command as the C shell did in the first two examples. The only difference is that the Bourne and Korn shells store the exit status in the ? variable, rather than the *status* variable.

quigley.book Page 11 Friday, September 28, 2001 10:00 AM

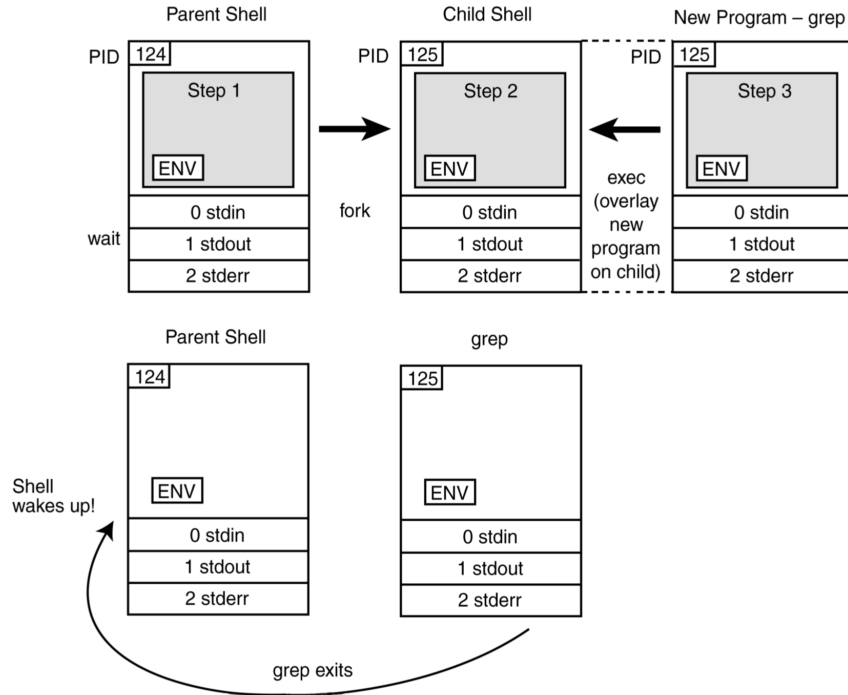


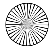
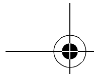
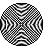
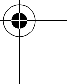
Figure 1.4 The *fork, exec, wait,* and *exit* system calls.

EXPLANATION

1. The parent shell creates a copy of itself with the *fork* system call. The copy is called the child shell.
2. The child shell has a new PID and is a copy of its parent. It will share the CPU with the parent.
3. The kernel loads the *grep* program into memory and executes (*exec*) it in place of the child shell. The *grep* program inherits the open files and environment from the child.
4. The *grep* program exits, the kernel cleans up, and the parent is awakened.

1.4 The Environment and Inheritance

When you log on, the shell starts up and inherits a number of variables, I/O streams, and process characteristics from the */bin/login* program that started it. In turn, if another shell is spawned (forked) from the login or parent shell, that child shell (*subshell*) will inherit certain characteristics from its parent. A subshell may be started for a number of reasons: for handling background processing, for handling groups of commands, or for

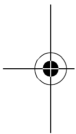
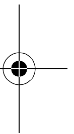
 

executing scripts. The child shell inherits an environment from its parent. The environment consists of process permissions (who owns the process), the working directory, the file creation mask, special variables, open files, and signals.

1.4.1 Ownership

When you log on, the shell is given an identity. It has a real user identification (*UID*), one or more real group identifications (*GID*), and an effective user identification and effective group identification (*EUID* and *EGID*). The EUID and EGID are initially the same as the real UID and GID. These ID numbers are found in the *passwd* file and are used by the system to identify users and groups. The EUID and EGID determine what permissions a process has access to when reading, writing, or executing files. If the EUID of a process and the real UID of the owner of the file are the same, the process has the owner's access permissions for the file. If the EGID and real GID of a process are the same, the process has the owner's group privileges.

The real UID, from the */etc/passwd* file, is a positive integer associated with your login name. The real UID is the third field in the password file. When you log on, the login shell is assigned the real UID and all processes spawned from the login shell inherit its permissions. Any process running with a UID of zero belongs to root (the superuser) and has root privileges. The real group identification, the GID, associates a group with your login name. It is found in the fourth field of the password file.



The EUID and EGID can be changed to numbers assigned to a different owner. By changing the EUID (or EGID[4]) to another owner, you can become the owner of a process that belongs to someone else. Programs that change the EUID or EGID to another owner are called *setuid* or *setgid* programs. The */bin/passwd* program is an example of a *setuid* program that gives the user root privileges. *Setuid* programs are often sources for security holes. The shell allows you to create *setuid* scripts, and the shell itself may be a *setuid* program.

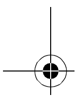
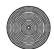
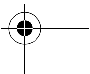
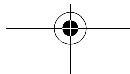
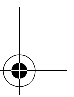
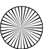
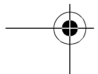
1.4.2 The File Creation Mask

When a file is created, it is given a set of default permissions. These permissions are determined by the program creating the file. Child processes inherit a default mask from their parents. The user can change the mask for the shell by issuing the *umask* command at the prompt or by setting it in the shell's initialization files. The *umask* command is used to remove permissions from the existing mask.

Initially, the *umask* is 000, giving a directory 777 (*rwxrwxrwx*) permissions and a file 666 (*rw–rw–rw–*) permissions as the default. On most systems, the *umask* is assigned a value of *022* by the */bin/login* program or the */etc/profile* initialization file.

The *umask* value is subtracted from the default settings for both the directory and file permissions as follows:

4. The *setgid* permission is system-dependent in its use. On some systems, a *setgid* on a directory may cause files created in that directory to belong to the same group that is owned by the directory. On others, the EGID of the process determines the group that can use the file.



```
 777 (Directory)       666 (File)
-022 (umask value)    -022 (umask value)
-------               ---------
 755                   644

Result: drwxr-xr-x    -rw-r--r--
```

After the *umask* is set, all directories and files created by this process are assigned the new default permissions. In this example, directories will be given read, write, and execute for the owner; read and execute for the group; and read and execute for the rest of the world (others). Any files created will be assigned read and write for the owner, and read for the group and others. To change permissions on individual directories and permissions, the *chmod* command is used.

1.4.3 Changing Permissions with *chmod*

There is one owner for every UNIX file. Only the owner or the superuser can change the permissions on a file or directory by issuing the *chmod* command. The following example illustrates the permissions modes. A group may have a number of members, and the owner of the file may change the group permissions on a file so that the group can enjoy special privileges.

The *chown* command changes the owner and group on files and directories. Only the owner or superuser can invoke it. On BSD versions of UNIX, only the superuser, *root*, can change ownership.

Every UNIX file has a set of permissions associated with it to control who can read, write, or execute the file. A total of nine bits constitutes the permissions on a file. The first set of three bits controls the permissions of the owner of the file, the second set controls the permissions of the group, and the last set controls the permissions of everyone else. The permissions are stored in the *mode* field of the file's inode.

The *chmod* command changes permissions on files and directories. The user must own the files to change permissions on them.[5]

Table 1.1 illustrates the eight possible combinations of numbers used for changing permissions.

Table 1.1 Permission Modes

Decimal	*Octal*	*Permissions*
0	000	none
1	001	– –x
2	010	–w–
3	011	–wx

5. The caller's EUID must match the owner's UID of the file, or the owner must be superuser.

quigley.book Page 14 Friday, September 28, 2001 10:00 AM

Table 1.1 Permission Modes (continued)

4	100	r– –
5	101	r–x
6	110	rw–
7	111	rwx

The symbolic notation for *chmod* is as follows:
r = read; *w* = write; *x* = execute; *u* = user; *g* = group; *o* = others; *a* = all.

EXAMPLE 1.3

```
1   $ chmod 755 file
    $ ls -l file
    -rwxr-xr-x 1 ellie 0 Mar  7 12:52 file
2   $ chmod g+w file
    $ ls -l file
    -rwxrwxr-x  1 ellie 0 Mar 7 12:54 file
3   $ chmod go-rx file
    $ ls -l file
    -rwx-w---- 1 ellie0 Mar 7 12:56 file
4   $ chmod a=r file
    $ ls -l file
    -r--r--r-- 1 ellie 0 Mar 7 12:59 file
```

EXPLANATION

1. The first argument is the octal value *755*. It turns on *rwx* for the user, and *r* and *x* for the group and others for file.
2. In the symbolic form of *chmod*, write permission is added to the group.
3. In the symbolic form of *chmod*, read and execute permission are subtracted from the group and others.
4. In the symbolic form of *chmod*, all are given only read permission. The = sign causes all permissions to be reset to the new value.

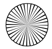
EXAMPLE 1.4

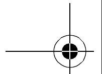
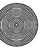
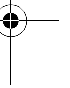
```
(The Command Line)
1   $ chown steve filex
2   $ ls -l

(The Output)
-rwxrwxr-x 1 steve groupa 170 Jul 28:20 filex
```

EXPLANATION

1. The ownership of *filex* is changed to *steve*.
2. The *ls –l* command displays the owner *steve* in column 3.

quigley.book Page 15 Friday, September 28, 2001 10:00 AM



1.4.4 Changing Ownership with the *chown* Command

The Working Directory. When you log in, you are given a working directory within the file system, called the *home directory*. The working directory is inherited by processes spawned from this shell. Any child process of this shell can change its own working directory, but the change will have no effect on the parent shell.

The *cd* command, used to change the working directory, is a shell built-in command. Each shell has its own copy of *cd*. A built-in command is executed directly by the shell as part of the shell's code; the shell does not perform the *fork* and *exec* system calls when executing built-in commands. If another shell (script) is forked from the parent shell, and the *cd* command is issued in the child shell, the directory will be changed in the child shell. When the child exits, the parent shell will be in the same directory it was in before the child started.

EXAMPLE 1.5

```
1   % cd /

2   % pwd
    /

3   % sh

4   $ cd /home

5   $ pwd
    /home

6   $ exit

7   % pwd
    /

    %
```

EXPLANATION

1. The prompt is a C shell prompt. The *cd* command changes directory to /. The *cd* command is built into the shell's internal code.
2. The *pwd* command displays the present working directory, /.
3. The Bourne shell is started.
4. The *cd* command changes directories to */home*.
5. The *pwd* command displays the present working directory, */home*.
6. The Bourne shell is exited, returning back to the C shell.
7. In the C shell, the present working directory is still /. Each shell has its own copy of *cd*.

Variables. The shell can define two types of variables: local and environment. The variables contain information used for customizing the shell, and information required by other processes so that they will function properly. Local variables are private to the shell in which they are created and not passed on to any processes spawned from that shell. Environment variables, on the other hand, are passed from parent to child process, from child to grandchild, and so on. Some of the environment variables are inherited by the login shell from the */bin/login* program. Others are created in the user initialization files, in scripts, or at the command line. If an environment variable is set in the child shell, it is not passed back to the parent.

File Descriptors. All I/O, including files, pipes, and sockets, are handled by the kernel via a mechanism called the *file descriptor*. A file descriptor is a small unsigned integer, an index into a file-descriptor table maintained by the kernel and used by the kernel to reference open files and I/O streams. Each process inherits its own file-descriptor table from its parent. The first three file descriptors, 0, 1, and 2, are assigned to your terminal. File descriptor 0 is standard input (*stdin*), 1 is standard output (*stdout*), and 2 is standard error (*stderr*). When you open a file, the next available descriptor is 3, and it will be assigned to the new file. If all the available file descriptors are in use,[6] a new file cannot be opened.

Redirection. When a file descriptor is assigned to something other than a terminal, it is called *I/O redirection*. The shell performs redirection of output to a file by closing the standard output file descriptor, 1 (the terminal), and then assigning that descriptor to the file (Figure 1.5).When redirecting standard input, the shell closes file descriptor 0 (the terminal) and assigns that descriptor to a file (Figure 1.6). The Bourne and Korn shells handle errors by assigning a file to file descriptor 2 (Figure 1.7). The C shell, on the other hand, goes through a more complicated process to do the same thing (Figure 1.8).

EXAMPLE 1.6

```
1   % who > file
2   % cat file1 file2 >> file3
3   % mail tom < file
4   % find / -name file -print 2> errors
5   % ( find / -name file -print > /dev/tty) >& errors
```

EXPLANATION

1. The output of the *who* command is redirected from the terminal to *file*. (All shells redirect output in this way.)
2. The output from the *cat* command (concatenate *file1* and *file2*) is appended to *file3*. (All shells redirect and append output in this way.)
3. The input of *file* is redirected to the *mail* program; that is, user *tom* will be sent the contents of *file*. (All shells redirect input in this way.)

---

6. See built-in commands, *limit* and *ulimit*.

quigley.book Page 17 Friday, September 28, 2001 10:00 AM

EXPLANATION

4. Any errors from the *find* command are redirected to *errors*. Output goes to the terminal. (The Bourne and Korn shells redirect errors this way.)
5. Any errors from the *find* command are redirected to *errors*. Output is sent to the terminal. (The C shell redirects errors this way.)

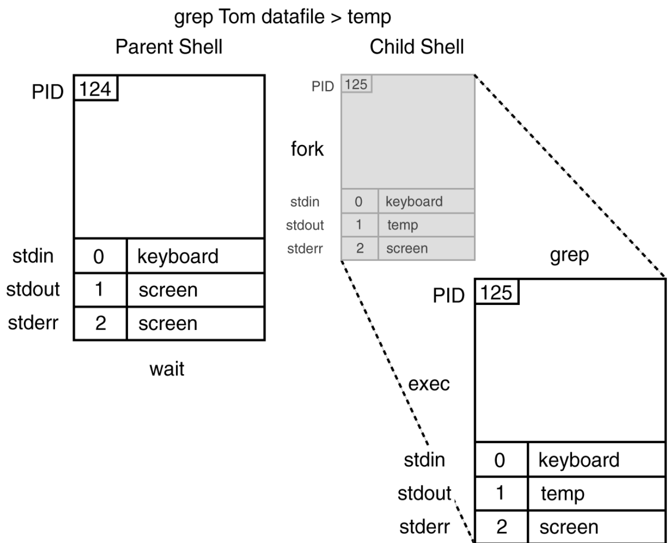


Figure 1.5 Redirection of standard output.

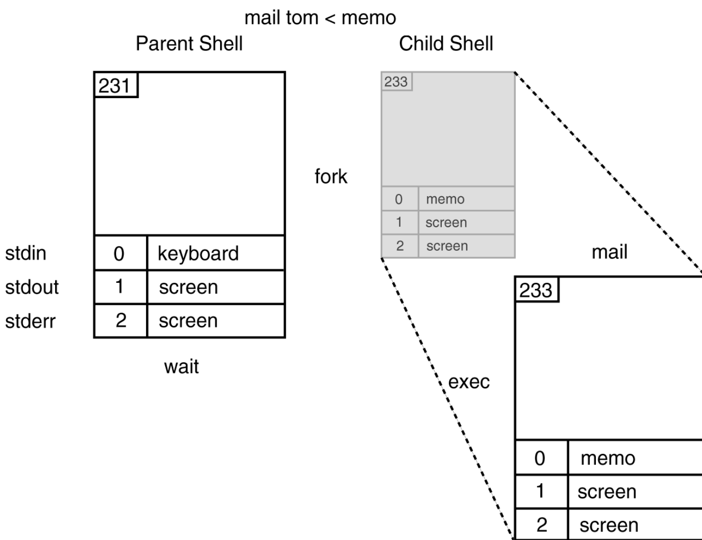


Figure 1.6 Redirection of standard input.

quigley.book Page 18 Friday, September 28, 2001 10:00 AM

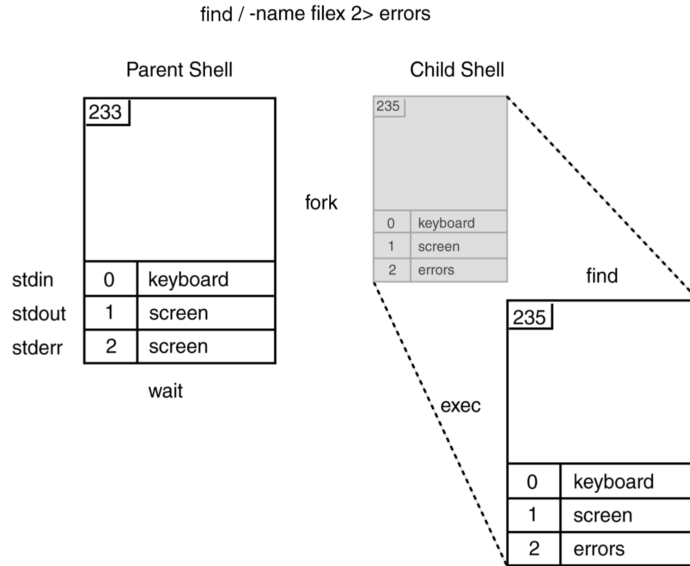


Figure 1.7 Redirection of standard error (Bourne and Korn shells).

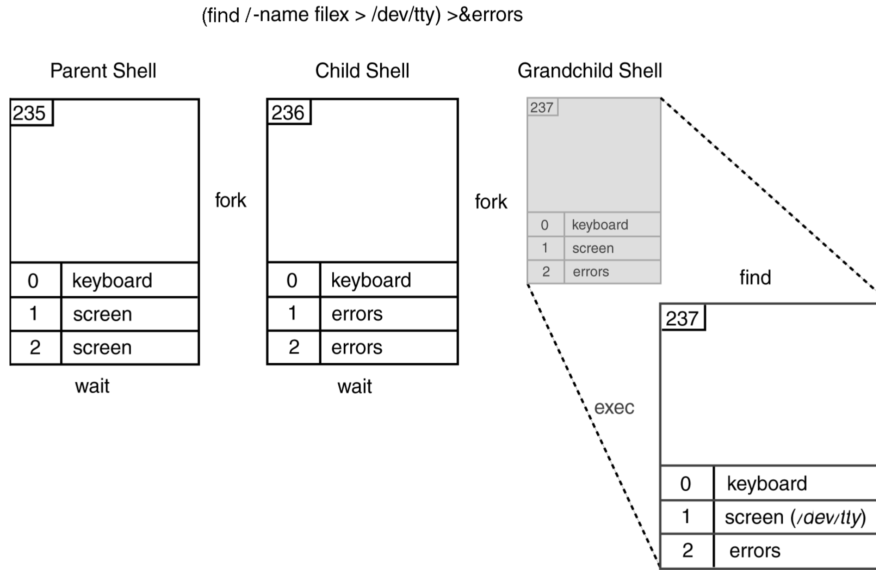


Figure 1.8 Redirection of standard error (C shell).

Pipes. Pipes allow the output of one command to be sent to the input of another command. The shell implements pipes by closing and opening file descriptors; however, instead of assigning the descriptors to a file, it assigns them to a pipe descriptor created with the *pipe* system call. After the parent creates the pipe file descriptors, it forks a child process for each command in the pipeline. By having each process manipulate the pipe descriptors, one will write to the pipe and the other will read from it. The pipe is merely a kernel buffer from which both processes can share data, thus eliminating the need for intermediate temporary files. After the descriptors are set up, the commands are *exec*'ed concurrently. The output of one command is sent to the buffer, and when the buffer is full or the command has terminated, the command on the right-hand side of the pipe reads from the buffer. The kernel synchronizes the activities so that one process waits while the other reads from or writes from the buffer.

The syntax of the *pipe* command is

```
who | wc
```

The shell sends the output of the *who* command as input to the *wc* command. This is accomplished with the *pipe* system call. The parent shell calls the *pipe* system call, which creates two pipe descriptors, one for reading from the pipe and one for writing to it. The files associated with the pipe descriptors are kernel-managed I/O buffers used to temporarily store data, thus saving you the trouble of creating temporary files. Figures 1.9 through 1.13 illustrate the steps for implementing the pipe.

(1) The parent shell calls the *pipe* system call. Two file descriptors are returned: one for reading from the pipe and one for writing to the pipe. The file descriptors assigned are the next available descriptors in the file-descriptor (fd) table, *fd 3* and *fd 4*. See Figure 1.9.

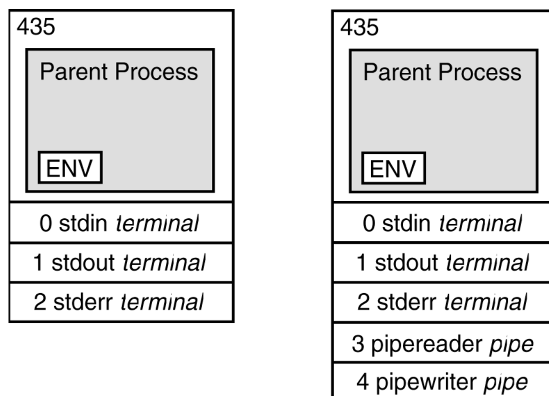


Figure 1.9 The parent calls the *pipe* system call for setting up a pipeline.

(2) For each command, *who* and *wc*, the parent forks a child process. Both child processes get a copy of the parent's open file descriptors. See Figure 1.10.

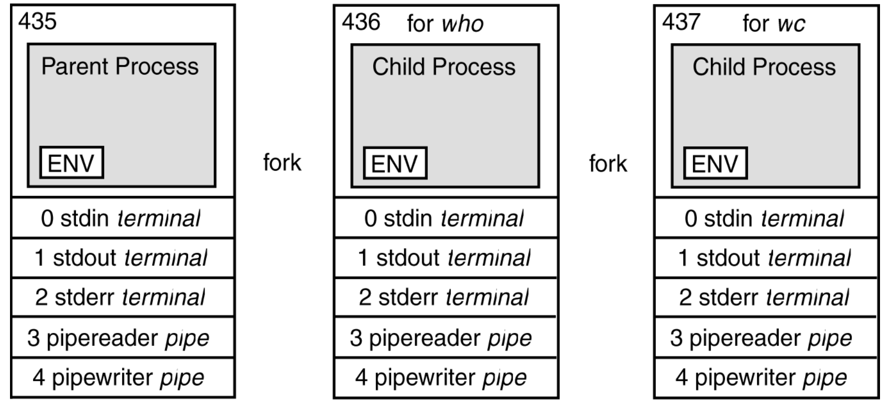


Figure 1.10 The parent forks two child processes, one for each command in the pipeline.

(3) The first child closes its standard output. It then duplicates (the *dup* system call) file descriptor *4*, the one associated with writing to the pipe. The *dup* system call copies *fd 4* and assigns the copy to the lowest available descriptor in the table, *fd 1*. After it makes the copy, the *dup* call closes *fd 4*. The child will now close *fd 3* because it does not need it. This child wants its standard *output* to go to the pipe. See Figure 1.11.

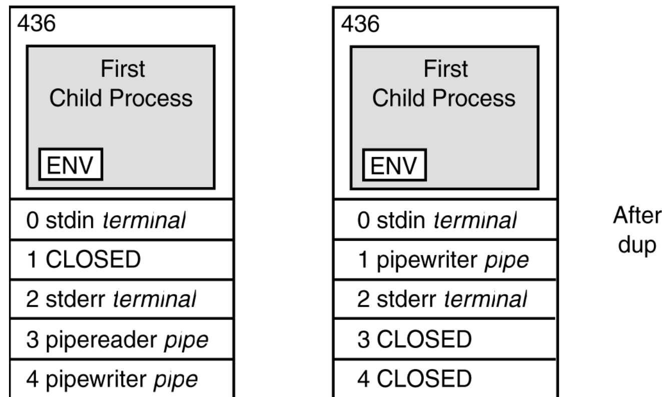


Figure 1.11 The first child is prepared to write to the pipe.

quigley.book Page 21 Friday, September 28, 2001 10:00 AM

(4) Child 2 closes its standard input. It then duplicates (*dups*) *fd 3*, which is associated with reading from the pipe. By using *dup*, a copy of *fd 3* is created and assigned to the lowest available descriptor. Since *fd 0* was closed, it is the lowest available descriptor. *Dup* closes *fd 3*. The child closes *fd 4*. Its standard *input* will come from the pipe. See Figure 1.12.

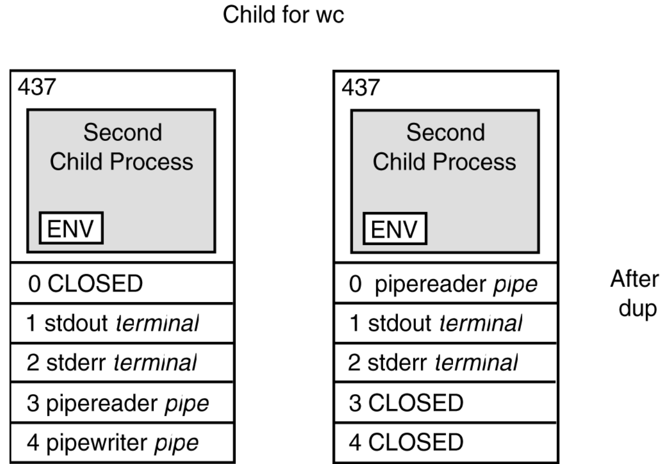


Figure 1.12 The second child is prepared to read input from the pipe.

(5) The *who* command is executed in place of Child 1 and the *wc* command is executed to replace Child 2. The output of the *who* command goes into the pipe and is read by the *wc* command from the other end of the pipe. See Figure 1.13.

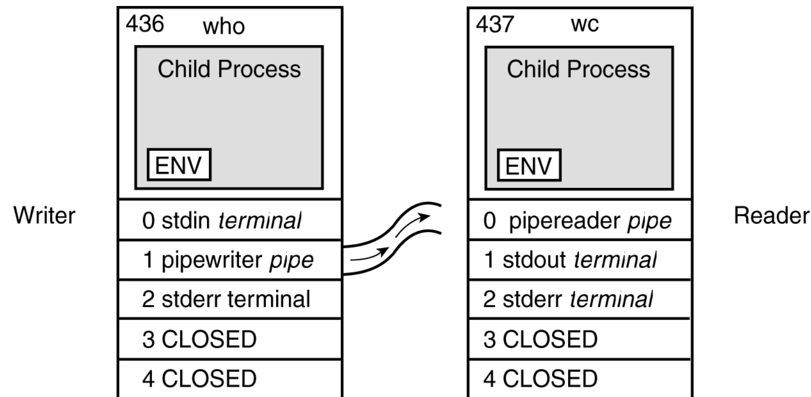


Figure 1.13 The output of *who* is sent to the input of *wc*.

1.4.5 The Shell and Signals

A *signal* sends a message to a process and normally causes the process to terminate, usually owing to some unexpected event such as a segmentation violation, bus error, or power failure. You can send signals to a process by pressing the Break, Delete, Quit, or Stop keys, and all processes sharing the terminal are affected by the signal sent. You can kill a process with the *kill* command. By default, most signals terminate the program. The shells allow you to handle signals coming into your program, either by ignoring them or by specifying some action to be taken when a specified signal arrives. The C shell is limited to handling ^C (Control-C).

1.5 Executing Commands from Scripts

When the shell is used as a programming language, commands and shell control constructs are typed in an editor and saved to a file, called a script. The lines from the file are read and executed one at a time by the shell. These programs are interpreted, not compiled. Compiled programs are converted into machine language before they are executed. Therefore, shell programs are usually slower than binary executables, but they are easier to write and are used mainly for automating simple tasks. Shell programs can also be written interactively at the command line, and for very simple tasks, this is the quickest way. However, for more complex scripting, it is easier to write scripts in an editor (unless you are a really great typist). The following script can be executed by any shell to output the same results. Figure 1.14 illustrates the creation of a script called *doit* and how it fits in with already existing UNIX programs/utilities/commands.

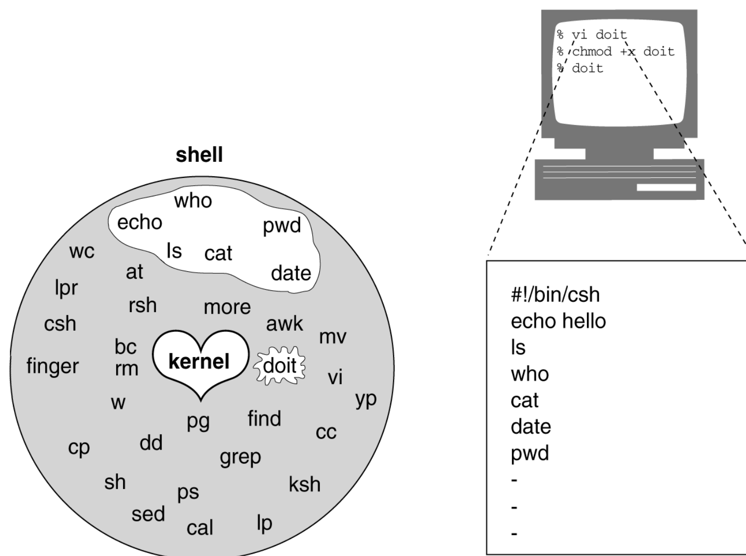


Figure 1.14 Creating a generic shell script.

quigley.book Page 23 Friday, September 28, 2001 10:00 AM

EXPLANATION

1. Go into your favorite editor and type in a set of UNIX commands, one per line. Indicate what shell you want by placing the pathname of the shell after the *#!* on the first line. This program is being executed by the C shell and it is named *doit*.
2. Save your file and turn on the execute permissions so that you can run it.
3. Execute your program just as you would any other UNIX command.

1.5.1 Sample Scripts: Comparing Three Shells

At first glance, the following three programs look very similar. They are. And they all do the same thing. The main difference is the syntax. After you have worked with all three shells for some time, you will quickly adapt to the differences and start formulating your own opinions about which shell is your favorite. A detailed comparison of differences among the C, Bourne, and Korn shells is found in Appendix B.

The following scripts send a mail message to a list of users, inviting each of them to a party. The place and time of the party are set in variables. The people to be invited are selected from a file called *guests*. A list of foods is stored in a word list, and each person is asked to bring one of the foods from the list. If there are more users than food items, the list is reset so that each user is asked to bring a different food. The only user who is not invited is the user *root*.

1.5.2 The C Shell Script

EXAMPLE 1.7

```
1   #!/bin/csh -f
2   # The Party Program--Invitations to friends from the "guest" file
3   set guestfile = ~/shell/guests
4   if ( ! -e "$guestfile" ) then
        echo "$guestfile:t non-existent"
        exit 1
    endif
5   setenv PLACE "Sarotini's"
    @ Time = `date +%H` + 1
    set food = ( cheese crackers shrimp drinks "hot dogs" sandwiches )
6   foreach person ( `cat $guestfile` )
        if ( $person =~ root ) continue
```

EXAMPLE 1.7 (CONTINUED)

```
7       mail -v -s "Party" $person << FINIS   # Start of here document
        Hi ${person}! Please join me at $PLACE for a party!
        Meet me at $Time o'clock.
        I'll bring the ice cream. Would you please bring $food[1] and
        anything else you would like to eat? Let me know if you can't
        make it. Hope to see you soon.
            Your pal,
            ellie@`hostname`        # or `uname -n`
    FINIS
8       shift food
        if ( $#food ==  0 ) then
            set food = ( cheese crackers shrimp drinks "hot dogs"
                         sandwiches )
        endif
9   end

    echo "Bye..."
```

EXPLANATION

1. This line lets the kernel know that you are running a C shell script. The *–f* option is a fast startup. It says, "Do not execute the *.cshrc* file," an initialization file that is automatically executed every time a new csh program is started.
2. This is a comment. It is ignored by the shell, but important for anyone trying to understand what the script is doing.
3. The variable *guestfile* is set to the full pathname of a file called *guests*.
4. This line reads: If the file *guests* does not exist, then print to the screen "*guests non-existent*" and exit from the script with an exit status of 1 to indicate that something went wrong in the program.
5. Set variables are assigned the values for the place, time, and list of foods to bring. The *PLACE* variable is an environment variable. The *Time* variable is a local variable. The @ symbol tells the C shell to perform its built-in arithmetic; that is, add 1 to the *Time* variable after extracting the hour from the *date* command. The *Time* variable is spelled with an uppercase *T* to prevent the C shell from confusing it with one of its reserved words, *time*.
6. For each person on the guest list, except the user *root*, a mail message will be created inviting the person to a party at a given place and time, and asking him or her to bring one of the foods on the list.
7. The mail message is created in what is called a *here document*. All text from the user-defined word *FINIS* to the final *FINIS* will be sent to the *mail* program. The *foreach* loop shifts through the list of names, performing all of the instructions from the *foreach* to the keyword *end*.

EXPLANATION (CONTINUED)

8 After a message has been sent, the food list is shifted so that the next person will get the next food item on the list. If there are more people than food items, the food list will be reset to ensure that each person is instructed to bring a food item.

9 This marks the end of the looping statements.

1.5.3 The Bourne Shell Script

EXAMPLE 1.8

```
1   #!/bin/sh
2   # The Party Program--Invitations to friends from the "guest" file
3   guestfile=/home/jody/ellie/shell/guests
4   if [ ! -f "$guestfile" ]
    then
        echo "`basename $guestfile` non-existent"
        exit 1
    fi
5   PLACE="Sarotini's"
    export PLACE
    Time=`date +%H`
    Time=`expr $Time + 1`
    set cheese crackers shrimp drinks "hot dogs" sandwiches
6   for person in `cat $guestfile`
    do
        if [ $person =~ root ]
        then
             continue
        else
7            mail -v -s "Party" $person <<- FINIS
             Hi ${person}! Please join me at $PLACE for a party!
             Meet me at $Time o'clock.
             I'll bring the ice cream. Would you please bring $1 and
             anything else you would like to eat? Let me know if you
             can't make it. Hope to see you soon.
                 Your pal,
                 ellie@`hostname`
             FINIS
8            shift
             if [ $# -eq 0 ]
             then
                set cheese crackers shrimp drinks "hot dogs" sandwiches
             fi
        fi
9   done
    echo "Bye..."
```

EXPLANATION

1. This line lets the kernel know that you are running a Bourne shell script.
2. This is a comment. It is ignored by the shell, but important for anyone trying to understand what the script is doing.
3. The variable *guestfile* is set to the full pathname of a file called *guests*.
4. This line reads: If the file *guests* does not exist, then print to the screen "*guests non-existent*" and exit from the script.
5. Variables are assigned the values for the place and time. The list of foods to bring is assigned to special variables (positional parameters) with the *set* command.
6. For each person on the guest list, except the user *root*, a mail message will be created inviting each person to a party at a given place and time, and asking each to bring a food from the list.
7. The mail message is sent when this line is uncommented. It is not a good idea to uncomment this line until the program has been thoroughly debugged, otherwise the e-mail will be sent to the same people every time the script is tested. The next statement, using the *cat* command with the *here document*, allows the script to be tested by sending output to the screen that would normally be sent through the mail when line 7 is uncommented.
8. After a message has been sent, the food list is shifted so that the next person will get the next food on the list. If there are more people than foods, the food list will be reset, insuring that each person is assigned a food.
9. This marks the end of the looping statements.

1.5.4 The Korn Shell Script

EXAMPLE 1.9

```
1   #!/bin/ksh
2   # The Party Program--Invitations to friends from the "guest" file
3   guestfile=~/shell/guests
4   if [[ ! -a "$guestfile" ]]
    then
        print "${guestfile##*/} non-existent"
        exit 1
    fi
5   export PLACE="Sarotini's"
    (( Time=$(date +%H) + 1 ))
    set cheese crackers shrimp drinks "hot dogs" sandwiches
6   for person in $(< $guestfile)
    do
        if  [[ $person = root ]]
        then
             continue
        else
```

EXAMPLE 1.9 (CONTINUED)

```
                # Start of here document
7               mail -v -s "Party" $person <<- FINIS
                Hi ${person}! Please join me at $PLACE for a party!
                Meet me at $Time o'clock.
                I'll bring the ice cream. Would you please bring $1
                and anything else you would like to eat? Let me know
                if you can't make it.
                       Hope to see you soon.
                               Your pal,
                               ellie@`hostname`
                FINIS
8               shift
                if (( $# ==  0 ))
                then
                  set cheese crackers shrimp drinks "hot dogs" sandwiches
                fi
            fi
9   done
    print "Bye..."
```

EXPLANATION

1. This line lets the kernel know that you are running a Korn shell script.
2. This is a comment. It is ignored by the shell, but important for anyone trying to understand what the script is doing.
3. The variable *guestfile* is set to the full pathname of a file called *guests*.
4. This line reads: If the file *guests* does not exist, then print to the screen "*guests non-existent*" and exit from the script.
5. Variables are assigned the values for the place and time. The list of foods to bring is assigned to special variables (positional parameters) with the *set* command.
6. For each person on the guest list, except the user *root*, a mail message will be created inviting the person to a party at a given place and time, and assigning a food from the list to bring.
7. The mail message is sent. The body of the message is contained in a *here document*.
8. After a message has been sent, the food list is shifted so that the next person will get the next food on the list. If there are more people than foods, the food list will be reset, insuring that each person is assigned a food.
9. This marks the end of the looping statements.