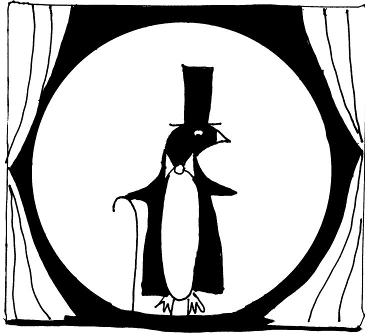

CHAPTER 1

INTRODUCTION TO LINUX SHELLS

1.1 Why Linux?

In 1991, Linus Torvalds, a Finnish college student, developed a UNIX-compatible operating system kernel at the University of Helsinki, Finland. It was designed to be UNIX on a PC. What started as one man's hobby has become a full-fledged 32-bit operating system installed worldwide on an estimated 10 million computers. And the number of users is growing at a phenomenal rate. At first, Linux appealed to hackers who wanted an operating system that allowed them the freedom to get down to the kernel level, to tweak and probe at the code with the same freedom and enthusiasm shown by the UNIX hackers of the early '80s. Now, as with UNIX, Linux is no longer associated solely with college hackers and "geeks" but has exploded in popularity worldwide for both personal and professional use, often serving systems with large networks of computers. For many, Linux is an alternative to Windows, and a large Linux culture has evolved sponsoring consortiums, conventions, expos, newgroups, and publications in a new revolution to rival Window's dominance in the PC world.

With the help of many system programmers and developers, Linux has grown into today's full-fledged UNIX- and POSIX-compatible operating system. In 1992, the Free Software Foundation added its Gnu software to the Linux kernel to make a complete operating system and licensed the Linux source code under its General Public License. Hundreds of Gnu utilities were provided by the Free Software Foundation, including improvements to the standard UNIX Bourne shell. The Bourne Again shell, the Linux default shell, is an enhanced Bourne shell, not only at the programming level, but also when used interactively, allowing the user to tailor his working environment and create shortcuts to improve efficiency. The Gnu tools, such as *grep*, *sed*, and *gawk*, are similar to their UNIX namesakes, but have also been improved and designed for POSIX[1] compliancy. The combination of the kernel and the Gnu tools and the fact that Linux could run on PCs, made

1. The requirements for shell functionality are defined by the POSIX (Portable Operating System Interface) standard, POSIX 1003.2.

1471A-c.fm Page 2 Thursday, April 27, 2000 8:54 AM

Linux a viable alternative to the proprietary UNIX and Microsoft operating systems, not to mention the fact that Linux is free to anyone who wants it, with all its source code, and a number of office suites and software packages. Whether you download Linux from the Internet, or buy a version distributed on a CD, Linux is portable, stable, and secure. It gives your PC the power of a workstation.

1.1.1 What Is POSIX?

In order to provide software standards for different operating systems and their programs, the POSIX (also referred to as the Open Systems Standards) evolved, consisting of participants from the Institute of Electrical and Electronics Engineering (IEEE) and the International Organization for Standardization (ISO). Their goal was to supply standards that would promote application portability across different platforms, to provide a UNIX-like computing environment; i.e., new software written on one machine that would compile and run on another machine with different hardware. For example, a program written for a BSD UNIX machine will run on Solaris, Linux, and HpUX machines. In 1988 the first standard was adopted, called POSIX 1003.1. Its purpose was to provide a C language standard. In 1992, the POSIX group established standards for the shell and utilities to define the terms for developing portable shell scripts, called the IEEE 1003.2 POSIX shell standard and general utility programs. Although there is no strict enforcement of these standards, most UNIX vendors try to comply with the POSIX standard. The term “POSIX compliancy” when discussing shells and their general UNIX utilities, is an attempt to comply to the standards presented by the POSIX committee, when writing new utilities or adding enhancements to the existing ones. For example, the Bourne Again shell is a shell that is almost 100% compliant and *gawk* is a user utility that can operate in strict POSIX mode.

1.2 Definition and Function of a Shell

The shell is a special program used as an interface between the user and the heart of the operating system, a program called the *kernel*, as shown in Figure 1.1. The kernel is loaded into memory at boot time, and manages the system until shutdown. It creates and controls processes, manages memory, file systems, communications, and so forth. All other programs, including shell programs, reside on the disk. The kernel loads programs from the disk 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 Linux, you spend most of your time executing commands from the prompt. You will be using the shell interactively.

If you type the same set of commands on a regular basis, you may want to automate those tasks. To do so, you place each command in an executable file, called a shell script.

1471A-c.fm Page 3 Thursday, April 27, 2000 8:54 AM

A shell script is much like a batch file. 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 also assumes that you have a good understanding of Linux utilities and how they work. There are some utilities, such as *grep, sed*, and *gawk,* that are extremely powerful when 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 will be using the shell as a programming language.

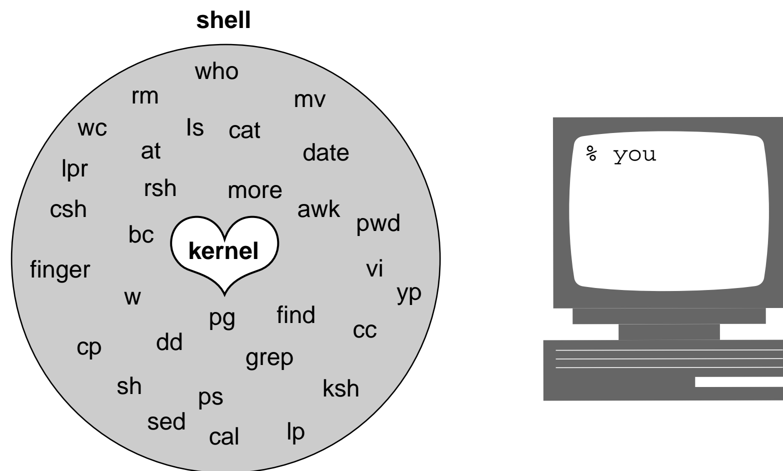

Figure 1.1 The kernel, the shell, and you.

1.2.1 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 shells 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 the shell 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 (superuser). 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, developed at Berkeley, 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 (%).

1471A-c.fm Page 4 Thursday, April 27, 2000 8:54 AM

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 run fine in this shell. The default Korn shell prompt is the dollar sign ($).

1.2.2 The Major Linux Shells

The shells used by Linux do not exclusively belong to the Linux operating system. They are freely available and can be compiled on any UNIX system. But when you install Linux, you will have access to the Gnu shells and tools, 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*.

EXAMPLE 1.1

```
$ cat /etc/shell
/bin/bash
/bin/sh
/bin/ash
/bin/bsh
/bin/tcsh
/bin/csh
/bin/ksh
/bin/zsh
```

EXPLANATION

1 The */etc/shell* file contains a list of all shell programs available on your version of Linux. The most popular versions are *bash* (Bourne Again shell), *tcsh* (TC shell), and *ksh* (Korn 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
```

1.2.3 History of the Shells

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 copyleft 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, 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.

1471A-c.fm Page 6 Thursday, April 27, 2000 8:54 AM

1.2.4 What Shells This Book Covers

Due to the great number of new features provided in the Bourne Again shell and TC shell, this book will concentrate on those two popular shells. The Korn shell, as well as Bourne and C shell, were presented in *UNIX Shells by Example*[2], and will not be covered again here. The Bourne Again shell, as of Release 2.0, is very similar in functionality to the Korn shell as a programming language, with many of the features of the C shell when used interactively. The TC shell, likewise, is almost identical to the C shell when used as a programming language, but has many new features for interactive use.

1.2.5 Uses of a Shell

One of the major uses of a shell is to interpret commands entered at the prompt. The shell parses the command line, breaking it into words (called *tokens*), separated by white space, i.e., 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 Bourne Again and the TC shells also provide further customization with the addition of history and aliases, filename and command completion, spell checking, help features, built-in variables set to protect the user from clobbering files, inadvertently logging out, and to notify the user when a job has completed, etc.

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 online scripting is permitted). They consist of Linux 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. Appendix A of this book contains a list of useful commands and how they work.

1.2.6 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.

2. Quigley, Ellie. *UNIX Shells by Example, 2nd Edition*. Upper Saddle River, NJ: Prentice Hall, 1999.

1471A-c.fm Page 7 Thursday, April 27, 2000 8:54 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.

1.3 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*. Because *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 sets 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 (*Login:*) appears at your console. After you type your login name, you are prompted for a password (you will be given up to 10 chances to enter the correct password). The */bin/login* program then verifies your identity by checking the first field in the /etc/*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 to the shell. The *HOME, SHELL, USER,* and *LOGNAME* variables are assigned values extracted from information in the */etc/passwd* file. The *HOME* variable is assigned your home directory; the *SHELL* variable is assigned the name of the login shell, the last entry in the *passwd* file. The *USER* and/or *LOGNAME* variables are assigned your login name. A PATH variable to help the shell find commonly used utilities is located in specified directories. It is a colon separated list initially set to: */usr/local/bin:/bin:/usr/bin*. When *login* has finished, it will execute the program found in the last entry of the */etc/passwd* file. Normally, this program is a shell. If the last entry in the */etc/passwd* file is */bin/tcsh* or *bin/csh*, the TC shell program is executed. If the last entry in the */etc/passwd* file is */bin/bash, /bin/sh,* or is null, the Bourne Again shell starts up. If the last entry is */bin/pdksh*, the Public Domain Korn shell is executed. This shell is called the *login shell*.

After the shell starts up, it checks for any systemwide initialization files 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 shell prompt appears on your console, unless a windowing program, such as X Windows or Gnome is launched, at which point a number of *xterm* or visual shell windows will appear. When you see the shell prompt, either at the console or in an *xterm* or other desktop window, the shell program is now waiting for your input.

1.3.1 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 or tabs, and the command line is terminated by a newline.[3] The shell then checks to see whether the first word is a built-in command or an executable program stored 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 program 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 (if set).
2. Command line is broken up into tokens (words).
3. History is updated.
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.

1.3.2 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:[4]

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

---

3. The process of breaking the line up into tokens is called *lexical analysis*.
4. Numbers 3 and 4 are reversed for Bourne and Korn(88) shells. Number 3 does not apply for C and TC shells.

1471A-c.fm Page 9 Thursday, April 27, 2000 8:54 AM

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.

1.4 Processes and the Shell

1.4.1 What Is a Process?

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 log in, the process running is normally a shell (*bash*)[5], called the login shell. 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. On Linux systems, the shell will normally start up another process, *xinit*, to launch the X Windowing system. After X Windows starts, a window manager process (*twm, fvwm,* etc.) is executed, providing a virtual desktop.[6] Then from a pop-up menu, you can start up a number of other processes, such as *xterm* (gets a terminal), *xman* (provides manual pages), or *emacs* (starts a text editor). Multiple processes are running and monitored by the Linux kernel, allocating each of the processes a little slice of the CPU in a way that is unnoticeable to the user.

1.4.2 What Is a System Call?

The shell can spawn (create) 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 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.

5. The default Linux shell is *bash*, the Bourne Again shell.
6. A number of desktop environments come with Linux, including Gnome, KDE, X, etc.

1471A-c.fm Page 10 Thursday, April 27, 2000 8:54 AM

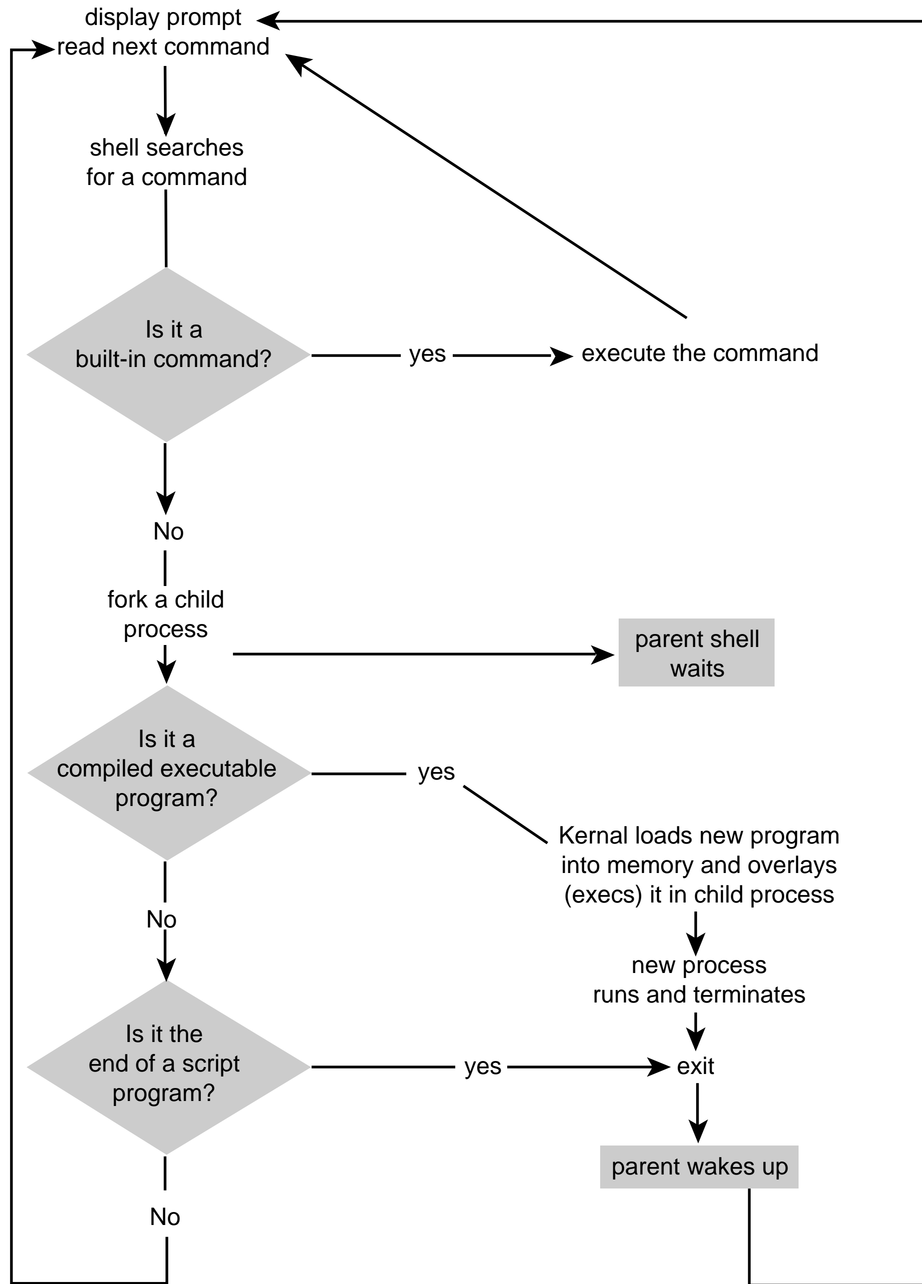

Figure 1.2 The shell and command execution.

1471A-c.fm Page 11 Thursday, April 27, 2000 8:54 AM

1.4.3 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. The following example shows all processes that are running by users on a Linux system. (See Appendix A for *ps* and its options.)

EXAMPLE 1.2

```
$ ps au  (Linux ps)
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
```

The *pstree* Command. Another way to see what processes are running and what processes are child processes is to use the Linux *pstree* command. The *pstree* command displays all processes as a tree with its root being the first process that runs, called *init*. If a user name is specified, then that user's processes are at the root of the tree. If a process spawns more than one process of the same name, *pstree* visually merges the identical

branches by putting them in square brackets and prefixing them with the number of times the processes is repeated. To illustrate, in the following example, the *httpd* server process has started up 10 child processes. (See Appendix A for list of *pstree* options.)

EXAMPLE 1.3

```
pstree
init---4*[getty]
init-+-atd
     |-bash---startx---xinit-+-X
     |                       `-fvwm2-+-FvwmButtons
     |                               |-FvwmPager
     |                               `-FvwmTaskBar
     |-cardmgr
     |-crond
     |-gpm
     |-httpd---10*[httpd]
     |-ifup-ppp---pppd---chat
     |-inetd
     |-kerneld
     |-kflushd
     |-klogd
     |-kswapd
     |-lpd
     |-2*[md_thread]
     |-5*[mingetty]

               |-nmbd
     |-nxterm---bash---tcsh---pstree
     |-portmap
     |-sendmail
     |-smbd
     |-syslogd
     |-update
     |-xclock
     `-xload
```

1.4.4 System Calls for Creating and Terminating Processes

The *fork* System Call. A new process is created 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 or not the first word is a built-in command or an executable program. If the command is built-in, the shell handles it, but if not, the shell invokes the *fork* system call to make a copy of itself (see 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 "The *wait* System Call" below.)

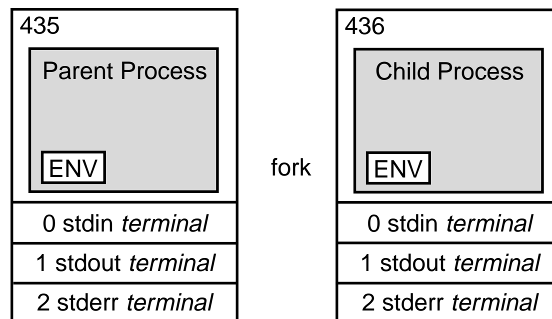

Figure 1.3 The *fork* system call.

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.[7] 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 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.

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

1471A-c.fm Page 14 Thursday, April 27, 2000 8:54 AM

The *exit* System Call. A 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.[8] 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 *ls* program. The *ls* program would run in place of the child, inheriting all the environment variables, open files, user information, and state information. When the process (*ls*) 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 on page 15 for an example of process creation and termination.

EXAMPLE 1.4

```
    (C and TC 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
    (Tcsh, Bourne, Korn, and Bash 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 TC shell command line prompt (>). After the command has made a copy of *filex* called *filey*, the program exits and the prompt appears. The tcsh *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.

---

8. If the program is terminated by a signal, its return value is 128 + *n*, where *n* is the signal number.

1471A-c.fm Page 15 Thursday, April 27, 2000 8:54 AM

EXPLANATION (CONTINUED)

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 tcsh *status* variable. Any number other than zero indicates that the program failed.

3 The Tcsh, Bourne, Bash, and Korn shells process the *cp* command as the TC 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. [a]

a. The TC shell provides both the *status* and *?* variables for holding exit status.

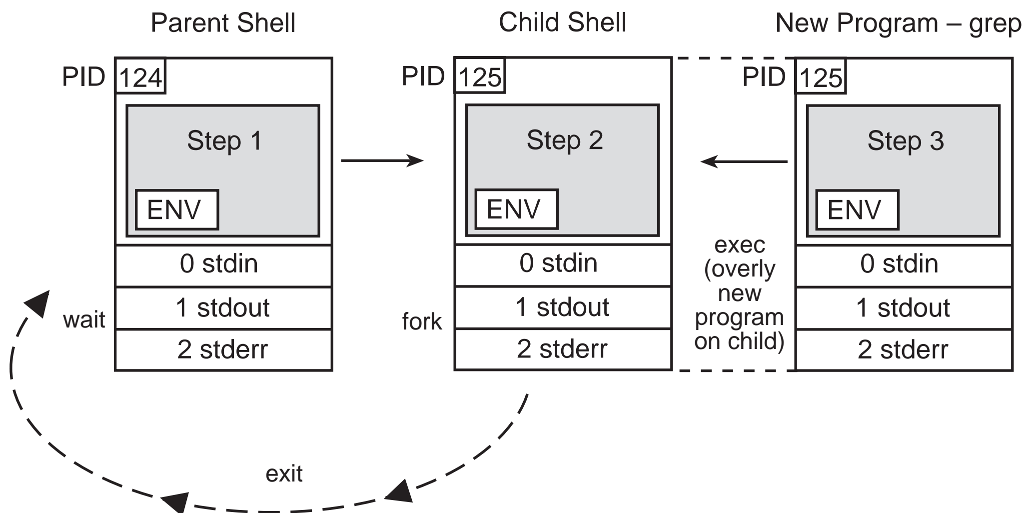

Figure 1.4 The *fork, exec, wait,* and *exit* system calls. See following Explanation.

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.

1.5 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.

1.5.1 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 UID, found in the */etc/passwd* file, is called the real UID. Its value is a positive integer that is 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[9]) 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.

1.5.2 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

9. The *setgid* permission is system-dependent in its use. On some systems, the *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.

1471A-c.fm Page 17 Thursday, April 27, 2000 8:54 AM

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:

```
 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.

1.5.3 Changing Ownership and Permissions

The *chmod* Command. The *chmod* command changes permissions on files and directories. Every Linux file has a set of permissions associated with it to control who can read, write, or execute the file. There is one owner for every Linux file and only the owner or the superuser can change the permissions on a file or directory. 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. To see what permissions a file has, type at the shell prompt:

```
ls -l filename
```

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 for everyone else. The permissions are stored in the mode field of the file's inode. The user must own the files to change permissions on them.[10]

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

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

Table 1.1 Permission Modes

Decimal	***Binary***	***Permissions***
0	000	none
1	001	--x
2	010	-w-
3	011	-wx
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 1.5

```
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 ellie 0 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, *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.

1471A-c.fm Page 19 Thursday, April 27, 2000 8:54 AM

The *chown* Command. The *chown* command changes the owner and group on files and directories. On Linux, only the superuser, *root*, can change ownership. To see the usage and options for *chown*, use the *chown* command with the *--help* option as shown in Example 1.6. Example 1.7 demonstrates how to use *chown*.

EXAMPLE 1.6

```
(The Command Line)
# chown --help
Usage: chown [OPTION]... OWNER[.[GROUP]] FILE...
  or:  chown [OPTION]... .[GROUP] FILE...
Change the owner and/or group of each FILE to OWNER and/or GROUP.

  -c, --changes          be verbose whenever change occurs
  -h, --no-dereference   affect symbolic links instead of any
                         referenced file
                         (available only on systems with lchown
                         system call)
  -f, --silent, --quiet  suppress most error messages
  -R, --recursive        operate on files and directories recursively
  -v, --verbose          explain what is being done
      --help             display this help and exit
      --version          output version information and exit

Owner is unchanged if missing. Group is unchanged if missing, but
changed to login group if implied by a period. A colon may replace
the period.

Report bugs to fileutils-bugs@gnu.ai.mit.edu
```

EXAMPLE 1.7

```
(The Command Line)
1   $ ls -l filetest
    -rw-rw-r--   1 ellie    ellie           0 Jan 10 12:19 filetest
2   $ chown root filetest
    chown: filetest: Operation not permitted
3   $ su root
    Password:
4   # ls -l filetest
   -rw-rw-r--   1 ellie    ellie            0 Jan 10 12:19 filetest
5   # chown root filetest
6   # ls -l filetest
    -rw-rw-r--   1 root     ellie           0 Jan 10 12:19 filetest
7   # chown root:root filetest
8   # ls -l filetest
    -rw-rw-r--   1 root     root            0 Jan 10 12:19 filetest
```

1471A-c.fm Page 20 Thursday, April 27, 2000 8:54 AM

EXPLANATION

1 The user and group ownership of *filetest* is *ellie.*

2 The *chown* command will only work if you are the superuser, i.e., user *root*.

3 The user changes identity to *root* with the *su* command.

4 The listing shows that user and group are *ellie* for *filetest*.

5 Only the superuser can change the ownership of files and directories. Ownership of *filetest* is changed to *root* with the *chown* command. Group ownership is still *ellie*.

6 Output of *ls* shows that *root* now owns *filetest*.

7 The colon (or a dot) is used to indicate that owner *root* will now change the group ownership to *root*. The groupname is listed after the colon. There can be no spaces.

8 The user and group ownership for *filetest* now belongs to *root*.

1.5.4 The Working Directory

When you log on, 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.

1.5.5 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.

EXAMPLE 1.8

```
1   > cd /

2   > pwd
    /

3   > bash

4   $ cd /home

5   $ pwd
    /home

6   $ exit

7   > pwd
    /

    >
```

EXPLANATION

1. The > prompt is a TC 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 Bash shell is started.
4. The *cd* command changes directories to */home*. The dollar sign (*$*) is the Bash prompt.
5. The *pwd* command displays the present working directory, */home*.
6. The Bash shell is exited, returning to the TC shell.
7. In the TC shell, the present working directory is still /. Each shell has its own copy of *cd*.

1.5.6 Redirection and Pipes

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,[11] 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 (see Figure 1.5).When redirecting standard input, the shell closes file descriptor 0 (the terminal) and assigns that descriptor to a file (see Figure 1.6). The Bourne and Korn shells handle errors by assigning a file to file descriptor 2 (see Figure 1.7). The TC shell, on the other hand, goes through a more complicated process to do the same thing (see Figure 1.8).

EXAMPLE 1.9

```
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.)

4 Any errors from the *find* command are redirected to *errors*. Output goes to the terminal. (The Bourne, Bash, 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 and TC shells redirect errors this way. > is the TC shell prompt.)

11. See the built-in commands, *limit* and *ulimit*, discussed in Table 8.32 on page 377.

1471A-c.fm Page 23 Thursday, April 27, 2000 8:54 AM

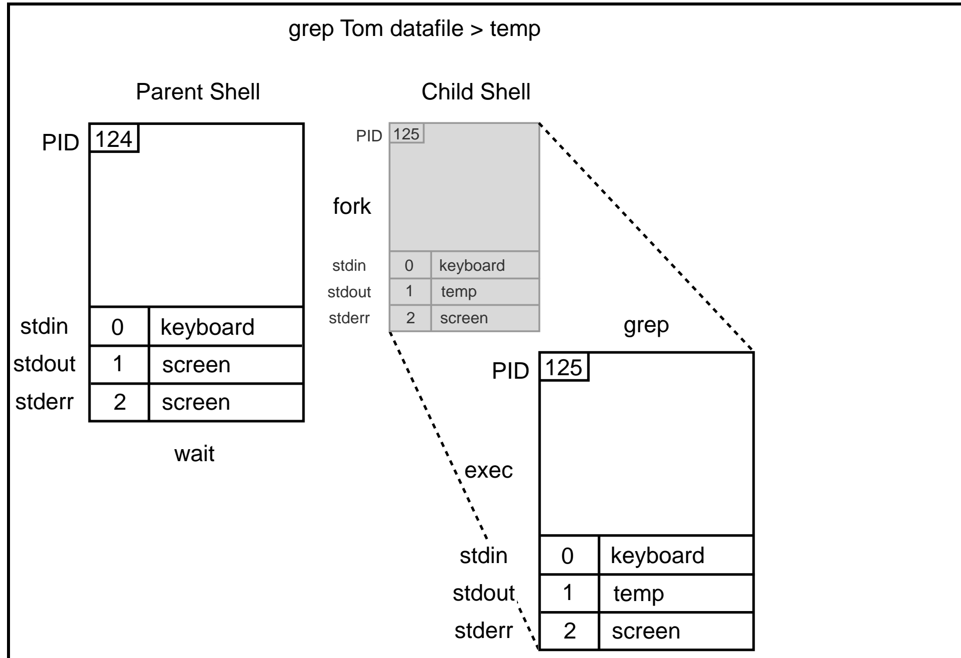

Figure 1.5 Redirection of standard output.

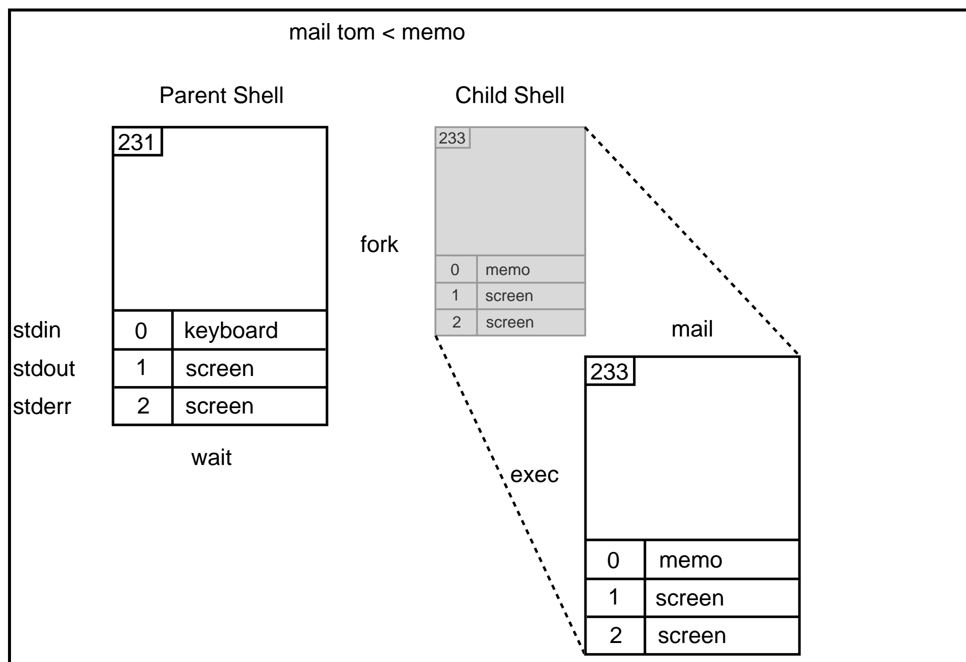

Figure 1.6 Redirection of standard input.

1471A-c.fm Page 24 Thursday, April 27, 2000 8:54 AM

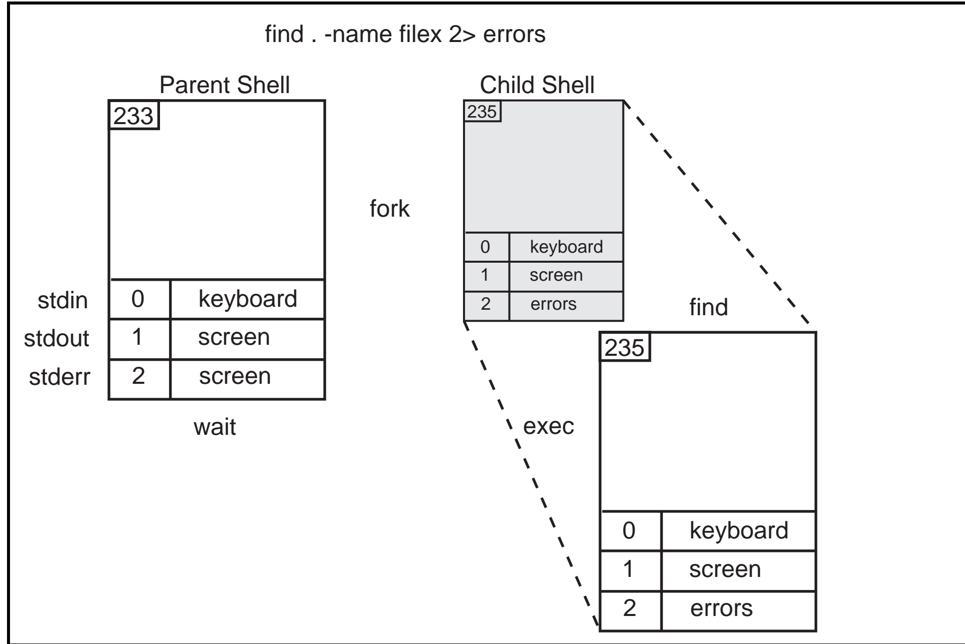

Figure 1.7 Redirection of standard error (Bourne, Bash, and Korn shells).

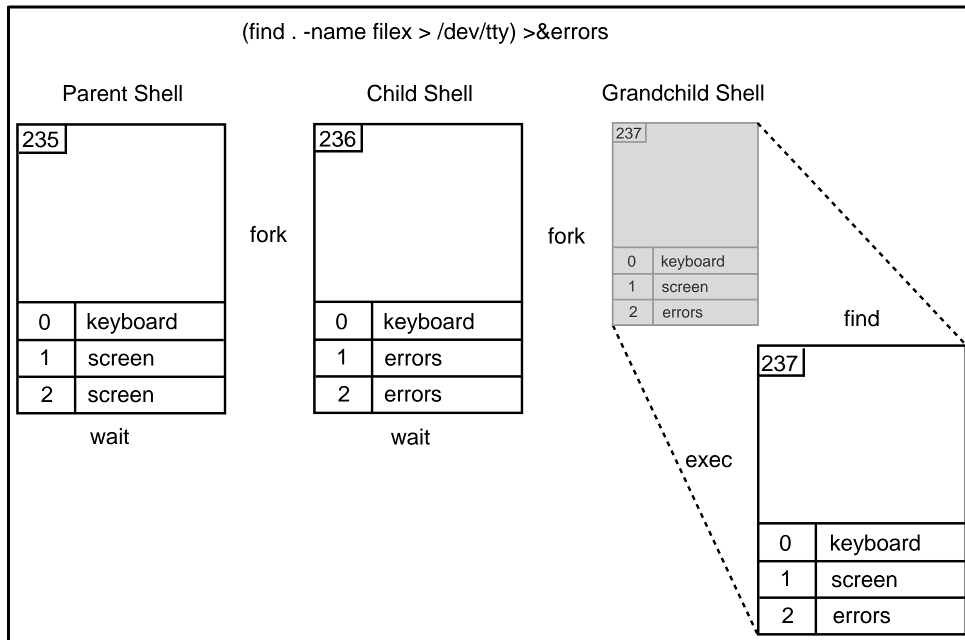

Figure 1.8 Redirection of standard error (C and TC shell).

1471A-c.fm Page 25 Thursday, April 27, 2000 8:54 AM

Pipes. A pipe allows processes to communicate with each other. It is a mechanism whereby the output of one command is sent as input to 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 to the buffer.

The syntax of the *pipe* command is:

```
who | wc
```

In order to accomplish the same thing without a pipe, it would take three steps:

```
who > tempfile
wc tempfil
rm tempfile
```

With the pipe, the shell sends the output of the *who* command as input to the *wc* command; i.e., the command on the left-hand side of the pipe writes to the pipe and the command on the right-hand side of the pipe, reads from it. You can tell if a command is a writer if it normally sends output to the screen when run at the command line. A reader is a command that waits for input from a file or from the keyboard or from a pipe.

Pipes are created 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.

1471A-c.fm Page 26 Thursday, April 27, 2000 8:54 AM

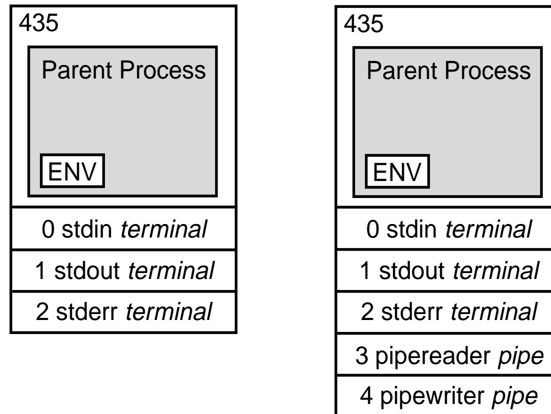

Figure 1.9 The parent calls the *pipe* system call for setting up a pipeline.

(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*.

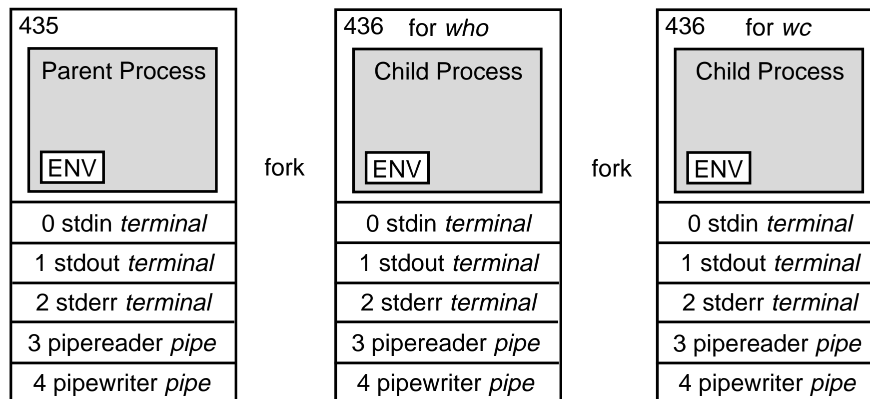

Figure 1.10 The parent forks two child processes, one for each command in the pipeline.

(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.

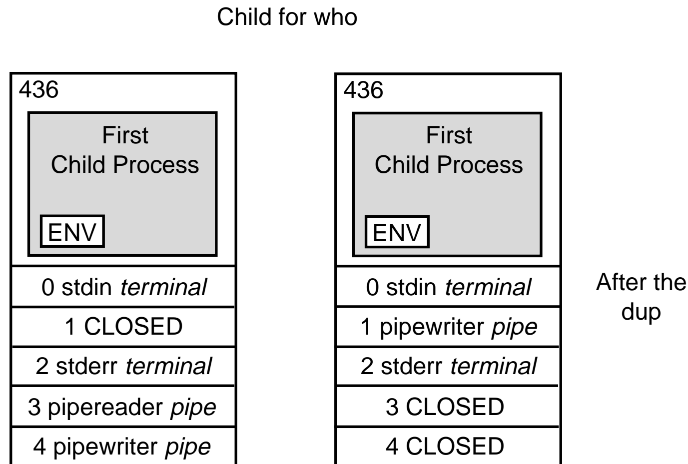

Figure 1.11 The first child is prepared to write to the pipe.

(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.

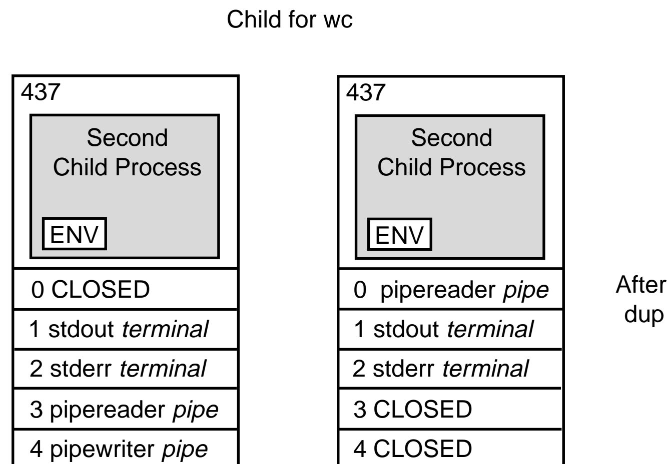

Figure 1.12 The second child is prepared to read input from the pipe.

1471A-c.fm Page 28 Thursday, April 27, 2000 8:54 AM

(4) Child 2 closes its standard input. It then duplicates (*dup*s) the *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. Because *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.

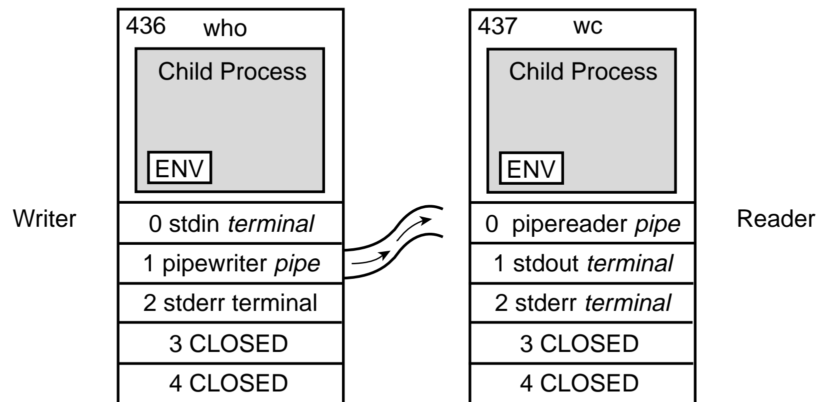

Figure 1.13 The output of *who* is sent to the input of *wc*.

(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. The last command in the pipe (*wc*) sends output to the standard out.

1.5.7 The Shell and Signals

A signal sends a message to a process and normally causes the process to terminate, usually due to some unexpected event such as a hangup, bus error, or power failure, or by a program error such as illegal division by zero or an invalid memory reference. Signals can also be sent to a process by pressing certain key sequences. For example, you can send or deliver 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. Each process can take an action in response to a given signal:

1. The signal can be ignored.
2. The process can be stopped.
3. The process can be continued.
4. The signal can be caught by a function defined in the program.

The Bourne, Bash, and Korn shells allow you to handle signals coming into your program, (see “Trapping Signals” on page 459) either by ignoring the signal, by specifying some action to be taken when a specified signal arrives, or by resetting the signal back to its

1471A-c.fm Page 29 Thursday, April 27, 2000 8:54 AM

default action. The C and TC shells are limited to handling ^C (Control-C), the interrupt character.

Table 1.2 lists the standard signals which a process can use.

Table 1.2 Standard Signals

Number	*Name*	*Description*	*Action upon Process*
0	EXIT	shell exits	termination
1	SIGHUP	terminal has disconnected	termination
2	SIGINT	user presses Control-C	termination
3	SIGQUIT	user presses Control-\	termination
4	SIGILL	illegal hardware instruction	program error
5	SIGTRAP	produced by debugger	program error
8	SIGFPE	arithmetic error; e.g., division by zero	program error
9	SIGKILL	cannot be caught or ignored	termination
10	SIGUSR1	application-defined signal for user	
11	SIGSEGV	invalid memory references	program error
12	SIGUSR2	application-defined signal for user	
13	SIGPIPE	broken pipe connection	operator error
14	SIGALRM	time-out	alarm sent
15	SIGTERM	termination of a program	termination
17	SIGCHLD	child process has stopped or died	ignored
18	SIGCONT	starts a stopped job; can't be handled or ignored	continue if stopped
19	SIGSTOP	stops a job; can't be handled or ignored	stops the process
20	SIGSTP	interactive stop; user presses Control-z	stops the process
21	SIGTTIN	a background job is trying to read from the controlling terminal	stops the process
22	SIGTTOU	a background job is trying to write to the controlling terminal	stops the process

1471A-c.fm Page 30 Thursday, April 27, 2000 8:54 AM

1.6 Executing Commands from Scripts

When the shell is used as a programming language, commands and shell control constructs are typed in an editor into 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 must convert the program into machine language for it to be 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 Linux programs/utilities/commands.

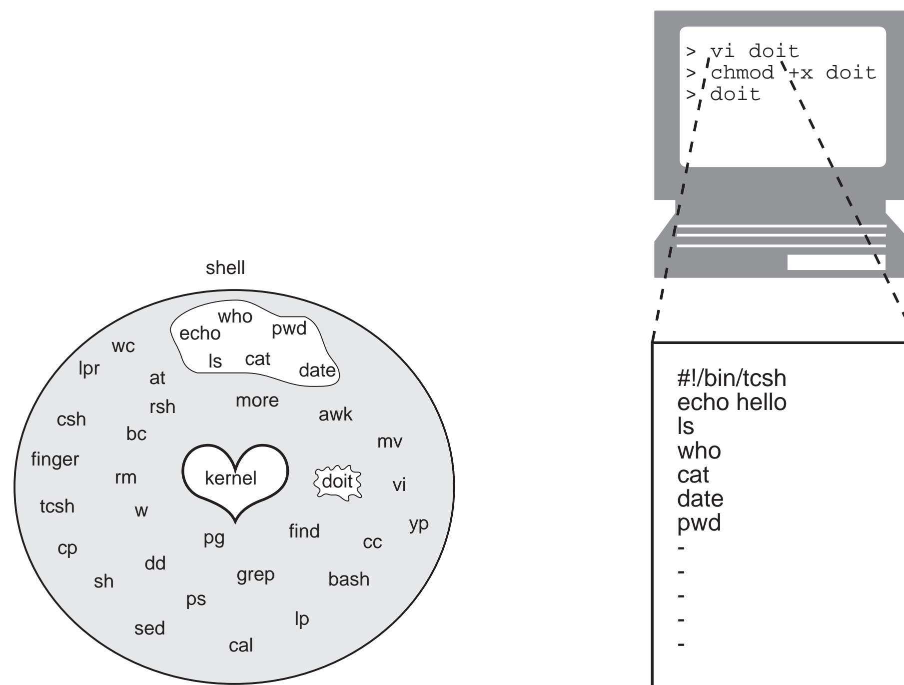

Figure 1.14 Creating a generic shell script.

EXPLANATION

1. Go into your favorite editor and type a set of Linux 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 TC 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 Linux command.

1471A-c.fm Page 31 Thursday, April 27, 2000 8:54 AM

1.6.1 Sample Scripts: Comparing Shells

At first glance, the following programs look very similar. They are. And they all do the same thing. The main difference is the syntax. After you have worked with these 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, TC, Bash, Bourne, and Korn shells is found in Appendix B. Notice that the C and TC shell scripts are identical; and that the Bash and Korn shell scripts are very similar with minor syntax or command changes. The Bourne shell is quite different in syntax but can be run by either the Bash or Korn shells.

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*.

1.6.2 The TC Shell Script

EXAMPLE 1.10

```
1   #!/bin/tcsh -f
    # TC shell
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
         # 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 $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
```

1471A-c.fm Page 32 Thursday, April 27, 2000 8:54 AM

EXAMPLE 1.10 (CONTINUED)

```
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 TC shell script. The *-f* option is a fast startup. It says, "Do not execute the *.tcshrc* file," an initialization file that is automatically executed every time a new *tcsh* 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 path name of a file called *guests*.

4 This line reads: If the file *guests* does not exist, then print to the screen "*guests nonexistent*" 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 TC 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 TC 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*.

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.

1.6.3 The C Shell Script

EXAMPLE 1.11

```
1   #!/bin/csh -f
    # Standard Berkeley C Shell
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
     # 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 $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.

If you look closely, this is the only line that differs from the TC shell script, shown in Example 1.10; therefore, lines 2 through 9 were already explained in the previous example.

1.6.4 The Bourne Again Shell Script

EXAMPLE 1.12

```
1   #!/bin/bash
    # Gnu bash versions 2.x
2   # The Party Program--Invitations to friends from the
    # "guest" file
3   guestfile=~/shell/guests
4   if [[ ! -e "$guestfile" ]]
    then
        printf "${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 $(cat $guestfile)
    do
        if  [[ $person = root ]]
        then
             continue
        else
             # 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
    printf "Bye..."
```

EXPLANATION

1. This line lets the kernel know that you are running a Bash shell (Bourne Again) script. Any versions prior to 2.x will not support all of this syntax. Older versions of Bash are similar to the standard Bourne shell. All versions are backward compatible.

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 path name of a file called *guests*.

4. This line reads: If the file *guests* does not exist, then print to the screen "*guests nonexistent*" 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, ensuring that each person is assigned a food.

9. This marks the end of the looping statements.

1471A-c.fm Page 36 Thursday, April 27, 2000 8:54 AM

1.6.5 The Bourne Shell Script

EXAMPLE 1.13

```
1   #!/bin/sh
    # Standard AT&T Bourne Shell
2   # The Party Program--Invitations to friends from the
    # "guest" file
3   guestfile=/home/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`
    d
        if  [ $person = root ]
        then
             continue
        else
             # 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 [ $# -eq  0 ]
             then
               set cheese crackers shrimp drinks "hot dogs" sandwiches
             fi
         fi
9   done
    echo "Bye..."
```

1471A-c.fm Page 37 Thursday, April 27, 2000 8:54 AM

EXPLANATION

1 This line lets the kernel know that you are running a *sh* shell (Bourne) script. This is the Bourne shell distributed with UNIX systems, such as Solaris and HP-UX. It is the last released version from ATT, SVR4. If running Linux, this script will work fine even if the shell is Bash.

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 path name of a file called *guests*. Tilde expansion is not allowed with Bourne shell. See other shell examples.

4 This line reads: If the file *guests* does not exist, then print to the screen "*guests nonexistent"* 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, ensuring that each person is assigned a food.

9 This marks the end of the looping statements.

1471A-c.fm Page 38 Thursday, April 27, 2000 8:54 AM

1.6.6 The Korn Shell Script

EXAMPLE 1.14

```
1   #!/bin/ksh
    # AT&T Korn Shell
2   # The Party Program--Invitations to friends from the
    # "guest" file
    # AT&T Korn Shell (1988)
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
             # 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 path name of a file called *guests*.
4. This line reads: If the file *guests* does not exist, then print to the screen "*guests nonexistent*" and exit from the script. The *-a* switch to test existence of a file is a Korn shell option.
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, ensuring that each person is assigned a food.
9. This marks the end of the looping statements.