Session 1-3: What is the Shell and its Responsibilities?
Chapter 3

In this chapter you'll learn what the Shell is and what it does.

- The Kernel and the Utilities
- The Login Shell
- Typing Commands to the Shell
- The Shell's Responsibilities

The Kernel and the Utilities

The Kernel and the Utilities

The Unix system is itself logically divided into two pieces: the *kernel* and the *utilities* (see Figure 3.1).

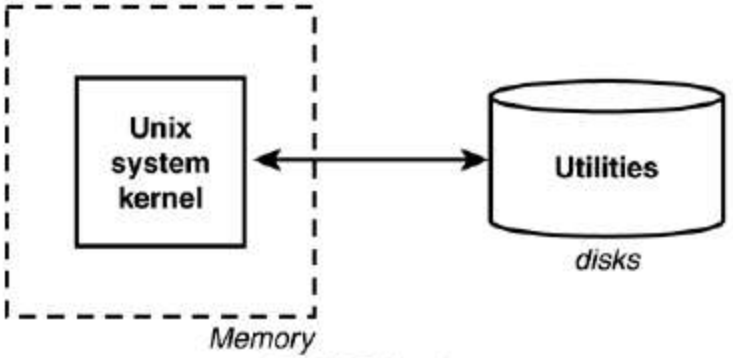

Figure 3.1. The Unix system.

The kernel is the heart of the Unix system and resides in the computer's memory from the time the computer is turned on and *booted* until the time it is shut down.

The utilities, on the other hand, reside on the computer's disk and are only brought into memory as requested. Virtually every command you know under the Unix system is classified as a utility; therefore, the program resides on the disk and is brought into memory only when you request that the command be executed. So, for example, when you execute the `date` command, the Unix system loads the program called `date` from the computer's disk into memory and initiates its execution.

The shell, too, is a utility program. It is loaded into memory for execution whenever you log in to the system.

The Login Shell

The Login Shell

A terminal is connected to a Unix system through a direct wire, modem, or network. In the first case, as soon as you turn on the terminal (and press the Enter key a couple of times if necessary), you should get a `login:` message on your screen. In the second case, you must first dial the computer's number and get connected before the `login:` message appears. In the last case, you may connect over the network via a program such as `ssh`, `telnet`, or `rlogin`, or you may use some kind of networked windowing system (for example, X Window System) to start up a terminal emulation program (for example, xterm).

For each physical terminal port on a system, a program called `getty` will be active. This is depicted in Figure 3.2.

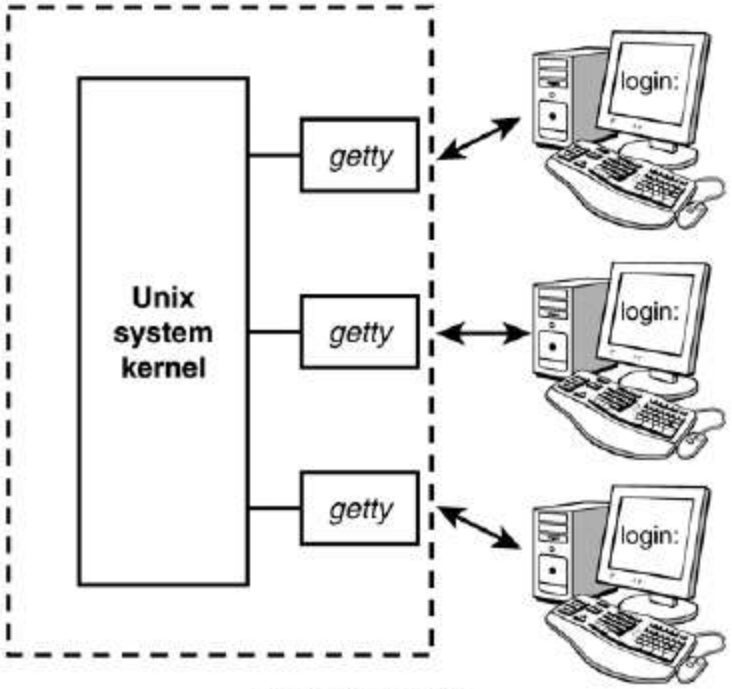

Figure 3.2. The `getty` process.

init – getty - login – the shell /etc/passwd and /etc/shadow

The Unix system—more precisely a program called `init`—automatically starts up a `getty` program on each terminal port whenever the system is allowing users to log in. `getty` determines the baud rate, displays the message `login:` at its assigned terminal, and then just waits for someone to type in something. As soon as someone types in some characters followed by Enter, the `getty` program disappears; but before it goes away, it starts up a program called `login` to finish the process of logging in (see Figure 3.3). It also gives `login` the characters you typed in at the terminal—characters that presumably represent your login name.

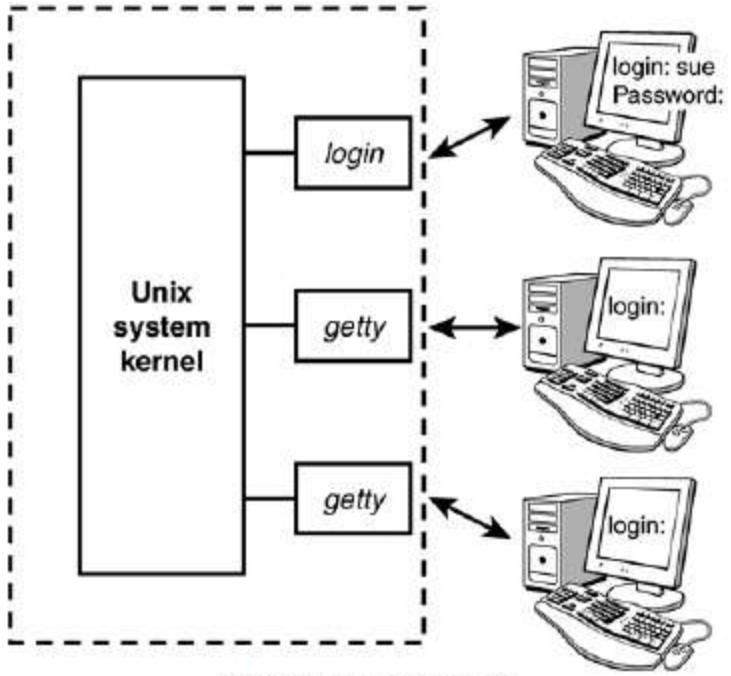

Figure 3.3. `login` started on `sue`'s terminal.

When `login` begins execution, it displays the string `Password:` at the terminal and then waits for you to type your password. After you have typed it, `login` then proceeds to verify your login name and password against the corresponding entry in the file `/etc/passwd`. This file contains one line for each user of the system. That line specifies, among other things, the login name, home directory, and program to start up when that user logs in.[1] The last bit of information (the program to start up) is stored after the *last* colon of each line. If nothing follows the last colon, the *standard* shell `/usr/bin/sh` is assumed by default. The following three lines show typical lines from `/etc/passwd` for three users of the system: `sue`, `pat`, and `bob`:

```
sue:*:15:47::/users/sue:
pat:*:99:7::/users/pat:/usr/bin/ksh
bob:*:13:100::/users/data:/users/data/bin/data_entry
```

[1] *The file's name (`passwd`) derives from a time when encrypted versions of the users' passwords were stored in this file along with other user information. The encrypted passwords are no longer stored in `/etc/passwd` but for security reasons are now kept in the `/etc/shadow` file, which is not readable by normal users.*

After `login` checks the password you typed in against the one stored in `/etc/shadow`, it then checks for the name of a program to execute. In most cases, this will be `/usr/bin/sh`, `/usr/bin/ksh`, or `/bin/bash`. In other cases, it may be a special custom-designed program. The main point here is that you can set up a login account to automatically run any program whatsoever whenever someone logs in to it. The shell just happens to be the program most often selected.

Three users login

So `login` initiates execution of the standard shell on `sue`'s terminal after validating her password (see Figure 3.4).

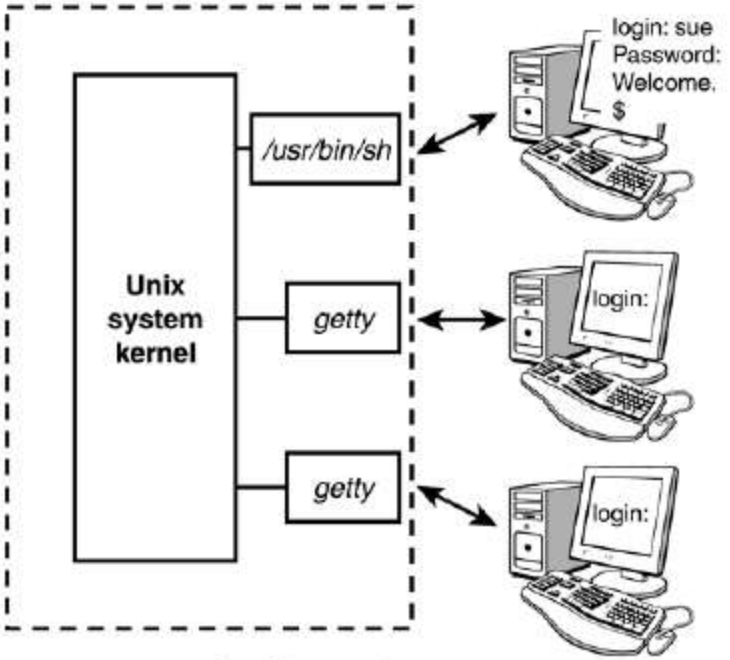

Figure 3.4. `login` executes `/usr/bin/sh`.

According to the other entries from `/etc/passwd` shown previously, `pat` gets the program `ksh` stored in `/usr/bin` (this is the Korn shell), and `bob` gets the program `data_entry` (see Figure 3.5).

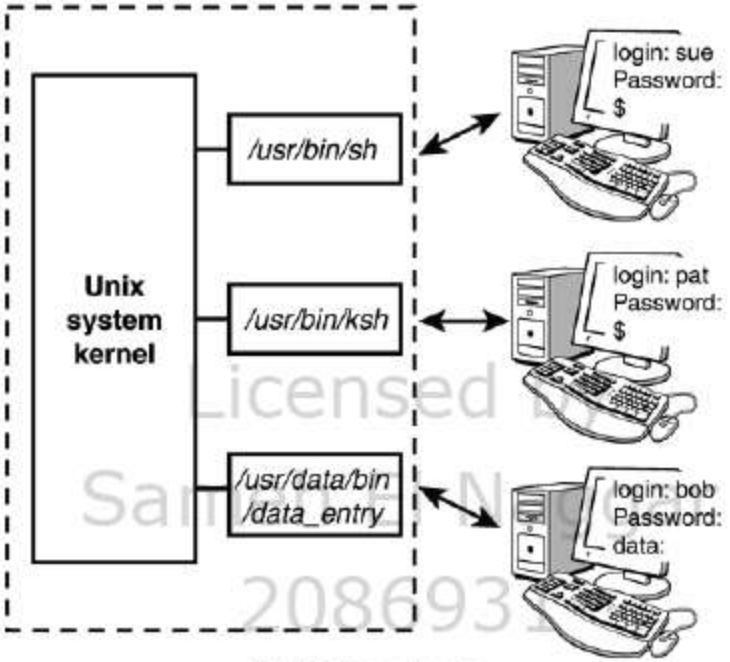

Figure 3.5. Three users logged in.

The `init` program starts up other programs similar to `getty` for networked connections. For example, `sshd`, `telnetd`, and `rlogind` are started to service logins via `ssh`, `telnet`, and `rlogin`, respectively. Instead of being tied directly to a specific, physical terminal or modem line, these programs connect users' shells to *pseudo ttys*. These are devices that emulate terminals over network connections. You can see this whether you're logged in to your system over a network or on an X Windows screen:

```
$ who
phw      pts/0    Jul 20 17:37        Logged in with rlogin
$
```

Typing commands to the Shell

Typing Commands to the Shell

When the shell starts up, it displays a command prompt—typically a dollar sign $—at your terminal and then waits for you to type in a command (see Figure 3.6, Steps 1 and 2). Each time you type in a command and press the Enter key (Step 3), the shell analyzes the line you typed and then proceeds to carry out your request (Step 4). If you ask it to execute a particular program, the shell searches the disk until it finds the named program. When found, the shell asks the kernel to initiate the program's execution and then the shell "goes to sleep" until the program has finished (Step 5). The kernel copies the specified program into memory and begins its execution. This copied program is called a *process*; in this way, the distinction is made between a program that is kept in a file on the disk and a process that is in memory doing things.

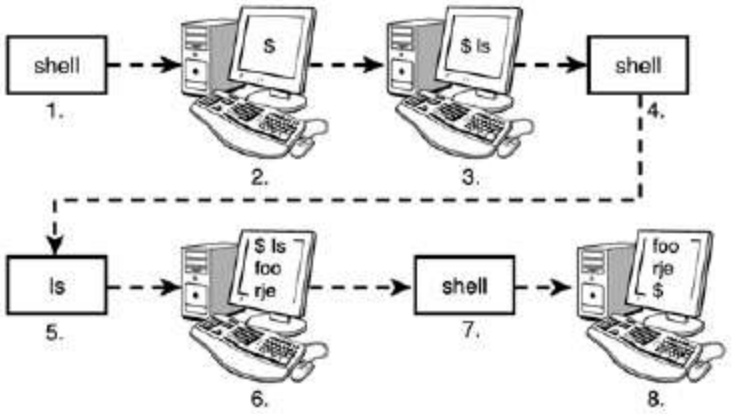

Figure 3.6. Command cycle.

If the program writes output to standard output, it will appear at your terminal unless redirected or piped into another command. Similarly, if the program reads input from standard input, it will wait for you to type in input unless redirected from a file or piped from another command (Step 6).

When the command finishes execution, control once again returns to the shell, which awaits your next command (Steps 7 and 8).

It's important for you to recognize that the shell is just a program. It has no special privileges on the system, meaning that anyone with the capability and devotion can create his own shell program. This is in fact the reason why various flavors of the shell exist today, including the older Bourne shell, developed by Stephen Bourne; the Korn shell, developed by David Korn; the "Bourne again shell," mainly used on Linux systems; and the C shell, developed by Bill Joy.

Note that this cycle continues as long as you're logged in. When you log off the system, execution of the shell then terminates and the Unix system starts up a new `getty` (or `rlogind`, and so on) at the terminal and waits for someone else to log in. This cycle is illustrated in Figure 3.7.

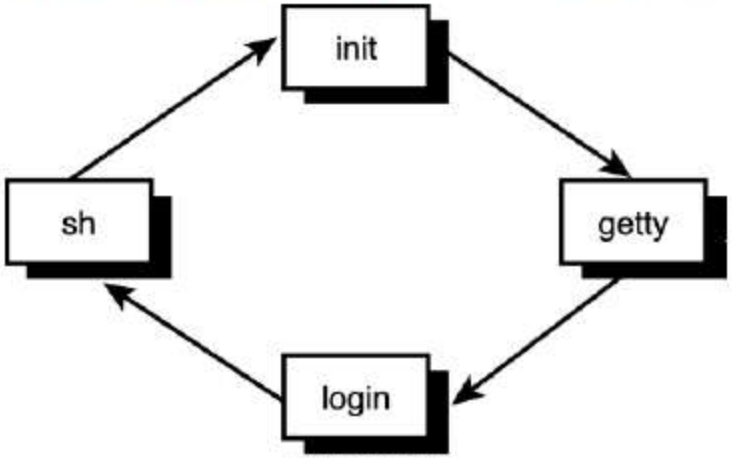

Figure 3.7. Login cycle.

The Shell's responsibilities

The Shell's Responsibilities

Now you know that the shell analyzes each line you type in and initiates execution of the selected program. But the shell also has other responsibilities, as outlined in Figure 3.8.

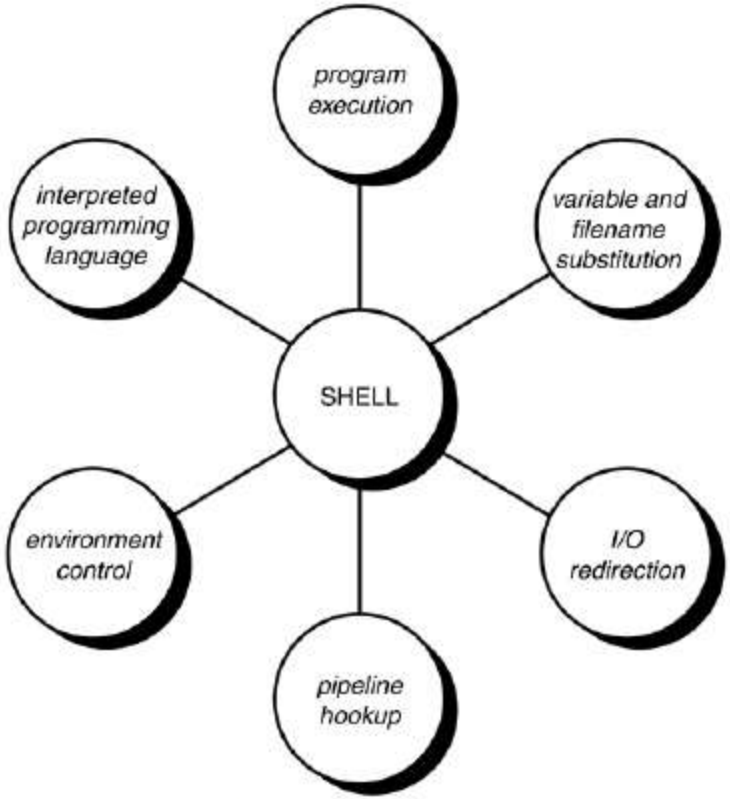

Figure 3.8. The shell's responsibilities.

Program Execution

Program Execution

The shell is responsible for the execution of all programs that you request from your terminal. Each time you type in a line to the shell, the shell analyzes the line and then determines what to do. As far as the shell is concerned, each line follows the same basic format:

program-name arguments

The line that is typed to the shell is known more formally as the command line. The shell scans this command line and determines the name of the program to be executed and what arguments to pass to the program.

The shell uses special characters to determine where the program name starts and ends, and where each argument starts and ends. These characters are collectively called **whitespace characters**, and are the space character, the horizontal tab character, and the end-of-line character, known more formally as the newline character. Multiple occurrences of whitespace characters are simply ignored by the shell. When you type the command

mv tmp/mazewars games

the shell scans the command line and takes everything from the start of the line to the first whitespace character as the name of the program to execute: mv. The set of characters up to the next whitespace character is the first argument to mv: tmp/mazewars. The set of characters up to the next whitespace character (known as a word to the shell)—in this case, the newline—is the second argument to mv: games. After analyzing the command line, the shell then proceeds to execute the mv command, giving it the two arguments tmp/mazewars and games (see Figure 3.9).

As mentioned, multiple occurrences of whitespace characters are ignored by the shell. This means that when the shell processes this command line:

echo when do we eat?

it passes four arguments to the echo program: when, do, we, and eat? (see Figure 3.10).

Passing arguments

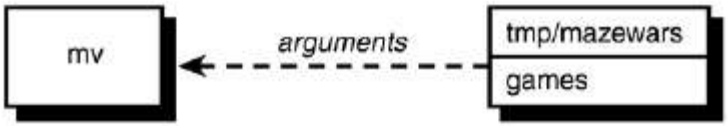

Figure 3.9. Execution of `mv` with two arguments.

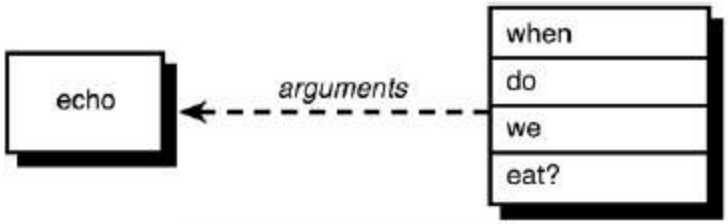

Figure 3.10. Execution of `echo` with four arguments.

As mentioned, multiple occurrences of whitespace characters are ignored by the shell. This means that when the shell processes this command line:

```
echo      when      do   we          eat?
```

it passes four arguments to the echo program: when, do, we, and eat? (see Figure 3.10).Because echo takes its arguments and simply displays them at the terminal, separating each by a space character, the output from the following becomes easy to understand:

```
$ echo when do we eat?
when do we eat?
$
```

The fact is that the echo command never sees those blank spaces; they have been “gobbled up” by the shell. When we discuss quotes in Chapter 6, “Can I Quote You on That?,” you'll see how you can include blank spaces in arguments to programs.

Variable and filename substitution

Variable and Filename Substitution

Like any other programming language, the shell lets you assign values to variables. Whenever you specify one of these variables on the command line, preceded by a dollar sign, the shell substitutes the value assigned to the variable at that point. This topic is covered in complete detail in Chapter 5, “And Away We Go.”

The shell also performs filename substitution on the command line. In fact, the shell scans the command line looking for filename substitution characters *, ?, or [...] before determining the name of the program to execute and its arguments. Suppose that your current directory contains the files as shown:

$ **ls**
mrs.todd
prog1
shortcut
sweeney
$

Now let's use filename substitution for the echo command:

$ **echo *** *List all files*
mrs.todd prog1 shortcut sweeney
$

How many arguments do you think were passed to the echo program, one or four? Because we said that the shell is the one that performs the filename substitution, the answer is four. When the shell analyzes the line

echo *

it recognizes the special character * and substitutes on the command line the names of all files in the current directory (it even alphabetizes them for you):

echo mrs.todd prog1 shortcut sweeney

Then the shell determines the arguments to be passed to the command. So echo never sees the asterisk. As far as it's concerned, four arguments were typed on the command line (see Figure 3.11).

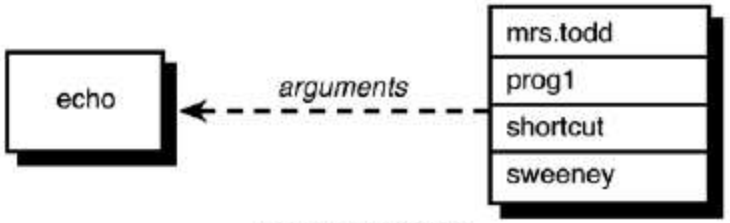

Figure 3.11. Execution of echo.

I/O Redirection

I/O Redirection
It is the shell's responsibility to take care of input and output redirection on the command line. It scans the command line for the occurrence of the special redirection characters <, >, or >> (also << as you'll learn in Chapter 13, “Loose Ends”).
When you type the command
echo Remember to tape Law and Order > reminder
the shell recognizes the special output redirection character > and takes the next word on the command line as the name of the file that the output is to be redirected to. In this case, the file is reminder. If reminder already exists and you have write access to it, the previous contents are lost (if you don't have write access to it, the shell gives you an error message).
Before the shell starts execution of the desired program, it redirects the standard output of the program to the indicated file. As far as the program is concerned, it never knows that its output is being redirected. It just goes about its merry way writing to standard output (which is normally your terminal, you'll recall), unaware that the shell has redirected it to a file.
Let's take another look at two nearly identical commands:
$ **wc -l users**
5 users
$ **wc -l < users**
5
$
In the first case, the shell analyzes the command line and determines that the name of the program to execute is wc and it is to be passed two arguments: -l and users (see Figure 3.12).

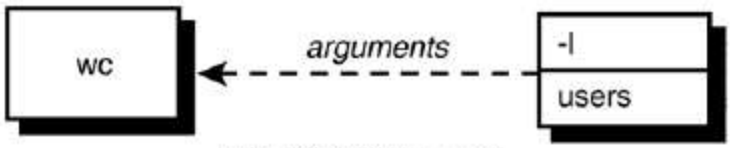

Figure 3.12 Execution of wc -l users.

I/O Redirection

When wc begins execution, it sees that it was passed two arguments.

- The first argument, -l, tells it to count the number of lines.
- The second argument specifies the name of the file whose lines are to be counted. So wc opens the file users, counts its lines, and then prints the count together with the filename at the terminal.

Operation of wc in the second case is slightly different.

- The shell spots the input redirection character < when it scans the command line.
- The word that follows on the command line is the name of the file input is to be redirected from.
- Having “gobbled up” the < users from the command line, the shell then starts execution of the wc program, redirecting its standard input from the file users and passing it the single argument -l (see Figure 3.13).

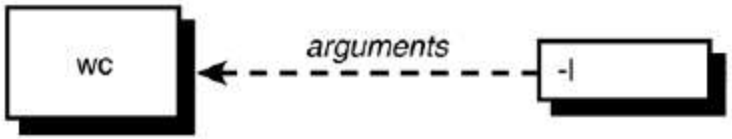

Figure 3.13. Execution of wc -l < users.

When wc begins execution this time, it sees that it was passed the single argument -l. Because no filename was specified, wc takes this as an indication that the number of lines appearing on standard input is to be counted. So wc counts the number of lines on standard input, unaware that it's actually counting the number of lines in the file users. The final tally is displayed at the terminal —without the name of a file because wc wasn't given one.

The difference in execution of the two commands is important for you to understand. If you're still unclear on this point, review the preceding section

Pipelines and environment Control

Pipeline Hookup

Just as the shell scans the command line looking for redirection characters, it also looks for the pipe character |. For each such character that it finds, it connects the standard output from the command preceding the | to the standard input of the one following the |. It then initiates execution of both programs.
So when you type
who | wc -l
the shell finds the pipe symbol separating the commands who and wc. It connects the standard output of the former command to the standard input of the latter, and then initiates execution of both commands. When the who command executes, it makes a list of who's logged in and writes the results to standard output, unaware that this is not going to the terminal but to another command instead.
When the wc command executes, it recognizes that no filename was specified and counts the lines on standard input, unaware that standard input is not coming from the terminal but from the output of the who command.

Environment Control

The shell provides certain commands that let you customize your environment. Your environment includes your home directory, the characters that the shell displays to prompt you to type in a command, and a list of the directories to be searched whenever you request that a program be executed.
You'll learn more about this in Chapter 11, “Your Environment.”

Interpreted Programming Language

Interpreted Programming Language

The shell has its own built-in programming language. This language is interpreted, meaning that the shell analyzes each statement in the language one line at a time and then executes it.

This differs from programming languages such as C and FORTRAN, in which the programming statements are typically compiled into a machine-executable form before they are executed.

Programs developed in interpreted programming languages are typically easier to debug and modify than compiled ones. However, they usually take much longer to execute than their compiled equivalents.

The shell programming language provides features you'd find in most other programming languages. It has looping constructs, decision-making statements, variables, and functions, and is procedure-oriented. Modern shells based on the IEEE POSIX standard have many other features including arrays, data typing, and built-in arithmetic operations