2 Using the Command Line

The command line is the primary interface to Unix. While there are many graphical interfaces for Unix systems, the command-line interface gives you the greatest control over the system. Furthermore, the command-line interface is virtually identical on every Unix system you are likely to use, from Mac OS X to Linux, to FreeBSD to Solaris. Of course there are differences, but there are far more similarities. Once you learn how to use the command line on Mac OS X, you will be comfortable using it on any Unix system.

A reminder before we go further: Whenever a task in this book asks you to type something, always press *r* at the end of the line unless the task description specifically tells you not to.

Getting to the Command Line

The primary way to get to the command line in Mac OS X is with the Terminal application.

Terminal is an Aqua application that allows you to open multiple windows, each of which provides a place to enter commands and see output from those commands.

Because Terminal is running in the Aqua layer of Mac OS X, you can do anything you'd expect from a Mac graphical application—you can print, copy text and paste into other windows, and adjust preferences such as color and font size.

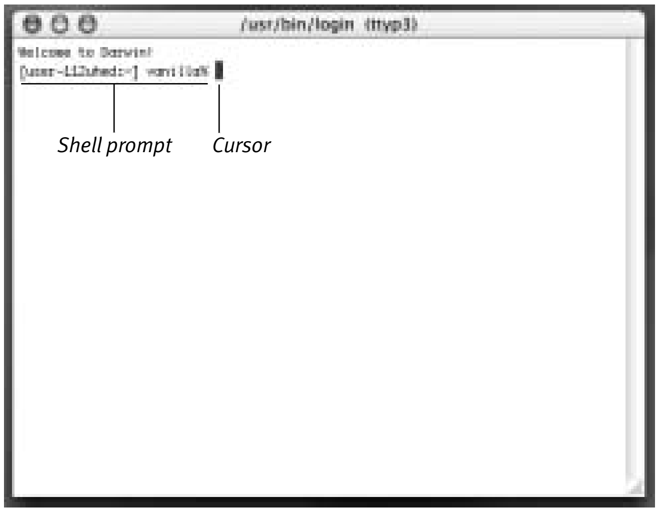
Practically all of your command-line work in Mac OS X will be done using Terminal.

To open Terminal:

- ◆ Locate the Terminal application in the Finder by going to the Applications folder and opening the Utilities folder.
- ◆ Double-click the Terminal application icon.
  A Terminal window containing a shell prompt opens (**Figure 2.1**).

✔ Tips

- ■ Put the Terminal icon in the Dock. You will be using it often.
- ■ When adjusting your Terminal preferences (under Window settings in the Terminal menu), always stick with a mono-spaced font like Monaco (the default) or Courier. Command-line software assumes you are using a mono-spaced (also called fixed-width) font, and proper text layout in Terminal depends on this.
- ■ Experiment with different colors and font sizes for the text and background in Terminal. For example, we prefer 12-point bright green text on a black background because it looks like the screen on an "old-fashioned" computer terminal.
- ■ Open more than one Terminal window (by clicking on New Shell under the File menu), and give each one a different color scheme as a way to differentiate them. You can have as many Terminal windows open as you like.



```
Welcome to Darwin!
[user-112uhedz:~] vanilla%
```

Figure 2.1 This is a screen shot of a window opened in the Terminal application.

Other ways to get to the command line

Using Terminal is by far the most common way to get to the Mac OS X command line, but there are other ways that are useful after you have become proficient in using Unix.

One way is to log in directly to the command line instead of going through the Terminal application in Aqua. **(Note: If you do this, the Aqua interface will not be available until you log out of the command line.)**

To log in to the command line:

1. Enter `>console` as your user name in the log-in screen. Leave the Password field empty (**Figure 2.2**).
2. Click Login or press r .

   This switches you directly to the Darwin layer of Mac OS X. A command-line log-in prompt appears, in white text on a black background (**Figure 2.3**).

   To go back to the Aqua log-in screen, press C D. Otherwise, proceed to log in to the Darwin layer.
3. Type your short user name, and press r (remember to press r after typing each task item). Note: On every other Unix system in the world, this would be your user name, but Mac OS X uses the concept of *short user name* to distinguish it from the regular Mac user name.

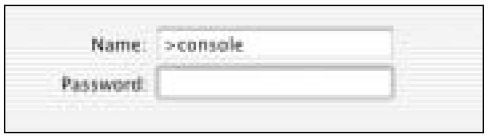
   A command-line password prompt appears.
4. Type your password at the Password prompt.

   Nothing appears on the screen as you type. If you get it wrong, you get another Login prompt, and you are back at step 4.

   If you get it right, the shell prompt appears on your screen.
5. Type `logout` to return to the Aqua log-in screen.

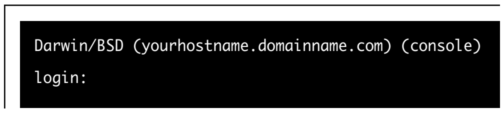
   There is a long pause before Aqua starts up—as much as a minute. Be patient.

Another way to get to the command line is to start up the machine into *single-user mode*. This boots the machine directly into the Darwin layer so that the command line comes up instead of Aqua. You cannot start Aqua from this mode without rebooting. You should only boot to single-user mode if you are extremely comfortable using Unix. See Chapter 11, "Introduction to System Administration," to learn how to boot into single-user mode.



Name:	>console
Password:	

Figure 2.2 You can use the name ">console" to log in to the command line instead of Aqua.



```
Darwin/BSD (yourhostname.domainname.com) (console)
login:
```

Figure 2.3 When you click the Login button, you are switched directly to the Darwin layer of Mac OS X and see the Darwin log-in prompt.

Understanding the Shell Prompt

The first thing you see in the Terminal window is the shell prompt (as we saw in Figure 2.1). The *shell* is a program that sits between you, the user, and the actual operating system. You type commands to the shell, and the shell reads the input, interprets its meaning, and executes the appropriate commands. This is similar to the way the Finder accepts your mouse clicks, interprets their meaning (single click? double click? drag?), and then performs an appropriate action (select item, open item, move item). The shell prompt is a string of text telling you that your shell is waiting for a command line.

The Shell window (in the Terminal folder under Preferences) allows you to specify which shell the Terminal application will use —but don't change the default until you have mastered the material at least through the end of Chapter 5, "Using Files and Directories." Throughout this book we assume you are using the default shell (`tcsh`) unless noted otherwise.

Now that you know what the shell is, let's start using it.

A Variety of Shells

There are many different shell programs available. The default shell on Mac OS X is called `tcsh`. Other shells available on Mac OS X are `sh`, `csh`, and `zsh`.

The `sh` shell is the oldest commonly used shell—`sh` just means "shell." It is also called the Bourne shell after its principal author, Steve Bourne of Bell Labs. Many important system files are actually small programs (scripts) written using `sh` commands (see Chapter 9, "Creating and Using Scripts").

The `csh` shell borrows some of its command syntax from the C programming language (hence the c) and was designed to be an improvement over the `sh` shell for interactive use. The `tcsh` shell is a more advanced form of the `csh` shell (the *t* comes from two old DEC operating systems). Many Unix experts consider the `csh` shell a poor tool for creating scripts. A classic essay making that case is at www.faqs.org/faqs/unix-faq/shell/csh-whynot/.

You can learn more about the `tcsh` shell at www.tcsh.org.

The `zsh` shell was designed as an improvement on another shell, `ksh` (which doesn't come with Mac OS X). It has a command syntax very different from `csh` and `tcsh`. You can learn more about `zsh` at www.zsh.org. If you find out why it is called `zsh`, let me know.

Another common shell worth mentioning is `bash` (for "Bourne Again Shell"—one of those Unix puns we warned you about), an improved version of the old standby `sh`. Although `bash` doesn't come with Mac OS X, it is easily installed. See Chapter 15, "More Open-Source Software" (at www.peachpit.com/vqp/umox), to learn how to install bash.

Using a Command

To use commands, you type them into the shell at the prompt. The shell executes the command line and displays output (if any), and then gives you another shell prompt. When the shell prompt comes up again, even if there's no other output, your shell is ready to accept another command.

Many command lines (but not all) produce output before returning a new shell prompt. It is quite common in Unix for a command to produce no visible output if it is successful (if it fails, a command should always produce output). In Unix, silence implies success.

To run a command:

◆ `ls /Developer/Tools`

This is the ls command, which lists the names of files and directories. The output of the command—a list of the tools installed in the Developer/Tools directory—appears and then a new shell prompt follows (**Figure 2.4**).

The command line you just used consists of two parts: the command (ls) and an argument (/Developer/Tools)

```
[localhost:~] vanilla% ls /Developer/Tools
BuildStrings              RezWack                   cvswrappers
CpMac                     SetFile                   lnresolve
DeRez                     SplitForks                pbhelpindexer
GetFileInfo               UnRezWack                 pbprojectdump
MergePef                  WSMakeStubs               pbxcp
MvMac                     agvtool                   pbxhmapdump
ResMerger                 cvs-unwrap                sdp
Rez                       cvs-wrap                  uninstall-devtools.pl
[localhost:~] vanilla%
```

Figure 2.4 When you type the command line `ls /Developer/Tools`, this is what you see.

The parts of a command line

The parts of a command line are separated by spaces. Basic command lines have up to four kinds of components:

- The command (required)
- *Options* (or *switches* or *flags*) (optional)
- *Arguments* (optional)
- *Operators* and special characters (optional)

Figure 2.5 shows the different parts of a typical command line (you'll recognize this as the command from Chapter 1 that searched for all instances of the word success in a particular directory).

Each command may have multiple options and multiple arguments.

About the "command" part of the command line

When you enter a command line, the shell assumes that the first item on the line is a command.

There are two types of commands: those that are built into the shell you are using and those that are separate files somewhere on your disk.

The overwhelming majority of Unix commands are of the latter type—that is, Unix commands are usually individual files that are actually small (or not-so-small) programs that perform a specific function, such as listing the contents of a directory.

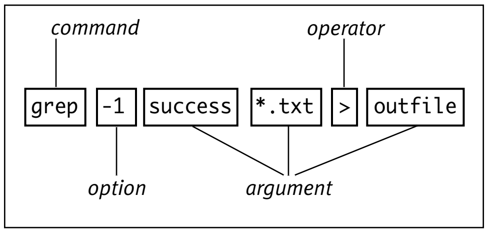


Figure 2.5 The parts of a command line are separated by spaces, and basic command lines have up to four kinds of items in them. This shows the separate parts of the command line.

Unix Commands vs. Mac Applications

Traditional Macintosh applications tend to have a great many features that allow you to accomplish complete projects all from within one application. For example, you can create and manipulate complex documents in a page-layout program. Unix takes a different approach.

In Unix, commands tend to be focused on specific steps you use in a variety of different tasks. For example, where the Mac has a single application (the Finder) for performing many tasks involving files, Unix uses a collection of separate "applications": the `ls` command lists the contents of a directory, the `cd` command switches from one directory to another, the `cp` command copies files, the `mv` command renames files, and so on.

This difference in approach shows a key difference in philosophy between the traditional Mac and Unix ways of thinking. In Unix, you are expected to combine commands in various ways to accomplish your work; in traditional Mac applications, the program's author is expected to anticipate every kind of task you might want to accomplish and provide a way of doing that.

Unix provides a collection of smaller, "sharper" tools and expects you to decide how to put them together to accomplish your goals.

Your PATH—how the shell finds commands

When the shell sees a command, it evaluates whether it is a built-in command—that is, one that is part of the shell itself (for example, the `cd` command is built into the tcsh shell). If the command is *not* built in, then the shell assumes the command is an actual file on the disk and looks for it.

If the command does not contain any `/` (slash) characters, then the shell searches in a list of places known as your `PATH` for a file whose name matches the command name (see the description of the `PATH` environment variable in Chapter 7, "Configuring Your Unix Environment"). If the command contains any `/` characters, then the shell assumes you are telling it not to search your `PATH` but instead to interpret the command as a *relative* or *absolute path* to the command file. Relative and absolute paths are two ways of specifying Unix filenames on the command line, and we explain relative and absolute paths in Chapter 5, "Using Files and Directories."

About command options

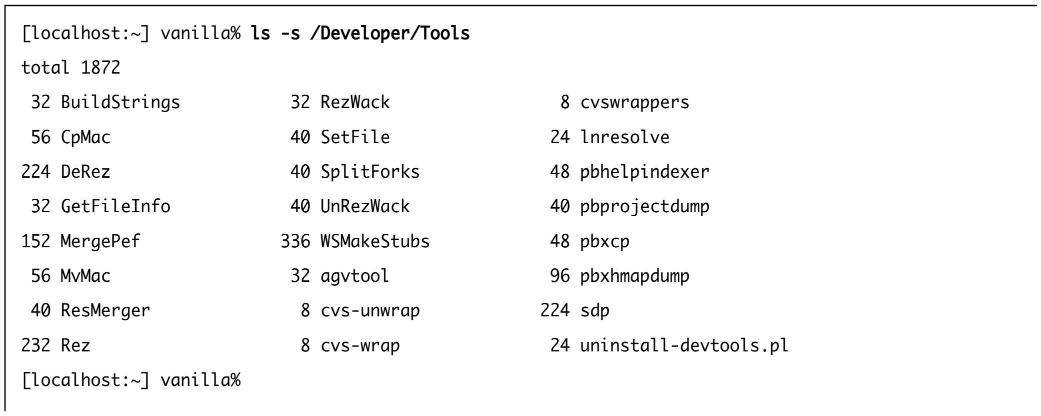
Options (also called switches or flags) modify the way a command behaves. Most commands have at least a few options available, and many commands have a large number of options. As we noted when we talked about Unix's flexibility in Chapter 1, options frequently can be combined.

See Chapter 3, "Getting Help and Using the Unix Manuals," to learn how to ascertain the available options for each command.

To use one option with a command:

◆ `ls -s /Developer/Tools`

  The `-s` option modifies the output of the `ls` command, asking for the size of each file. That number (in *blocks*, which correspond to disk space) is now displayed alongside each filename (**Figure 2.6**).



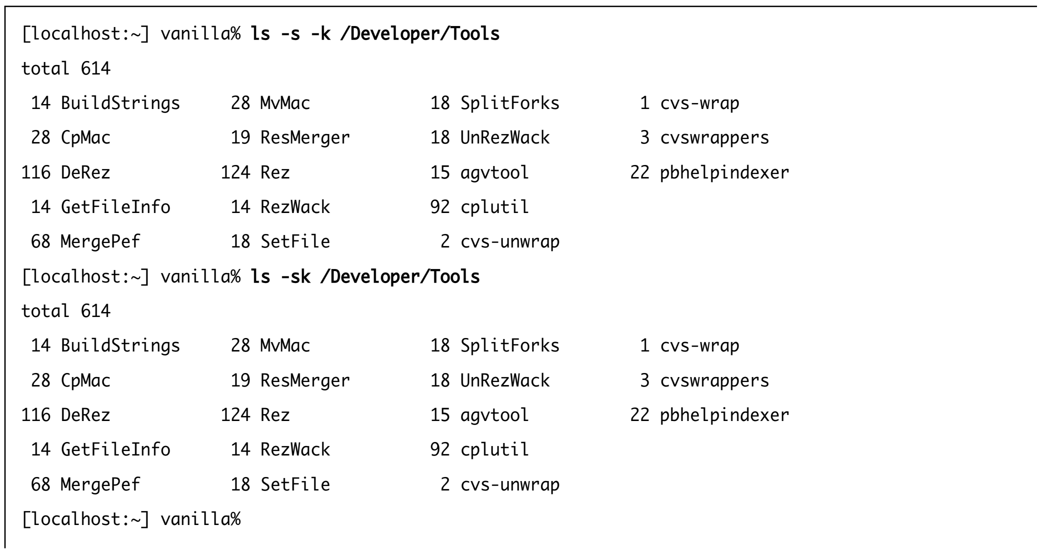
```
[localhost:~] vanilla% ls -s /Developer/Tools
total 1872
 32 BuildStrings          32 RezWack               8 cvswrappers
 56 CpMac                 40 SetFile              24 lnresolve
224 DeRez                 40 SplitForks           48 pbhelpindexer
 32 GetFileInfo           40 UnRezWack            40 pbprojectdump
152 MergePef             336 WSMakeStubs          48 pbxcp
 56 MvMac                 32 agvtool              96 pbxhmapdump
 40 ResMerger              8 cvs-unwrap          224 sdp
232 Rez                    8 cvs-wrap             24 uninstall-devtools.pl
[localhost:~] vanilla%
```

Figure 2.6 When you type this command line, `ls -s /Developer/Tools`, which has one option, the output lists the size of each file in blocks.

Here's a case where we want to combine two options in a command. The `-s` option gave us file sizes, but using a unit of measurement (blocks) that varies depending on how our disk was formatted. If we add the `-k` option, the sizes are shown in kilobytes, regardless of the block size on our disk.

To use multiple options with a command:

- `ls -s -k /Developer/Tools`
  Simply supply both of the options you want.
- You can combine two or more options:
  `ls -sk /Developer/Tools`
  This produces the same output as in the first case and saves typing two characters. **Figure 2.7** shows the output of both command lines. (Unix commands are so short and cryptic because the programmers who invented them wanted to avoid typing.)



```
[localhost:~] vanilla% ls -s -k /Developer/Tools
total 614
 14 BuildStrings     28 MvMac           18 SplitForks        1 cvs-wrap
 28 CpMac            19 ResMerger       18 UnRezWack         3 cvswrappers
116 DeRez           124 Rez             15 agvtool          22 pbhelpindexer
 14 GetFileInfo      14 RezWack         92 cplutil
 68 MergePef         18 SetFile          2 cvs-unwrap
[localhost:~] vanilla% ls -sk /Developer/Tools
total 614
 14 BuildStrings     28 MvMac           18 SplitForks        1 cvs-wrap
 28 CpMac            19 ResMerger       18 UnRezWack         3 cvswrappers
116 DeRez           124 Rez             15 agvtool          22 pbhelpindexer
 14 GetFileInfo      14 RezWack         92 cplutil
 68 MergePef         18 SetFile          2 cvs-unwrap
[localhost:~] vanilla%
```

Figure 2.7 This shows there's no difference between what you get when you use the `-s and -k` options separately and when you combine them in the `-sk` option.

About command arguments

Most commands accept one or more arguments. An argument is a piece of information the command acts upon, such as the name of a file to display. It's similar to the object of a sentence.

You used a command with a single argument in the tasks above. The single argument was /Developer/Tools, the folder whose contents you wanted to list. A command line can contain multiple arguments.

To use multiple arguments with a command:

◆ `ls /Developer /Developer/Tools`

  You simply add as many arguments as needed on the command line, separated by spaces. In this example, the `ls` command gets two arguments and lists the contents of both directories (**Figure 2.8**).

✔ Tips

- You can combine multiple options with multiple arguments—for example, `ls -sk /Developer /Developer/Tools`
- Remember that the shell expects the parts of a command line to be separated by spaces. If an argument has spaces in it, then you need to protect the embedded space(s) from being interpreted as separators. See "About Spaces in the Command Line" below.

Operators and special characters in the command line

A number of special characters often appear in command lines, most frequently the > and & characters.

These special characters are used for a variety of powerful features that manipulate the

```
[localhost:~] vanilla% ls /Developer /Developer/Tools
/Developer:
Applications             Headers                  Palettes
Documentation            Java                     ProjectBuilder Extras
Examples                 Makefiles                Tools

/Developer/Tools:
BuildStrings             RezWack                  cvswrappers
CpMac                    SetFile                  lnresolve
DeRez                    SplitForks               pbhelpindexer
GetFileInfo              UnRezWack                pbprojectdump
MergePef                 WSMakeStubs              pbxcp
MvMac                    agvtool                  pbxhmapdump
ResMerger                cvs-unwrap               sdp
Rez                      cvs-wrap                 uninstall-devtools.pl
[localhost:~] vanilla%
```

Figure 2.8 The `ls` command gets two arguments, `/Developer and /Developer/Tools`, and lists the contents of both directories.

output of commands. The most common of these operators make it easy to save the output of a command to a file, feed the output of one command into another command, use the output of one command as an argument to another command, and run a command line "in the background" (that is, letting you get a shell prompt back even if the command takes an hour to run).

The use of these powerful features is covered later in this chapter (see "Creating Pipelines of Commands").

Table 2.1 summarizes the most frequently used command-line operators and special characters, with examples of their use.

Stopping commands

Some commands run for a long time, and sometimes they can get "stuck" (perhaps because a command is waiting for some other process to finish, or because of a network problem, or for any number of other reasons) and neither give output nor return you to a shell prompt. In those cases, you need a way to stop a command once you have started it. Here are two ways to stop a command.

If you are waiting for the shell prompt to appear, then you use C C to stop the command.

To stop a command with Control-C:

◆ Press C (usually at the lower left of your keyboard) and simultaneously press C. This sends what is called an "interrupt" signal to the command, which should stop running and bring up a shell prompt.

✔ Tip

■ If using C C doesn't work, as a last resort you can close the Terminal window, overriding the warning that appears. The stuck command will be stopped. It doesn't hurt Unix for you to close the window; it's just annoying for you.

To stop a command using the kill command:

◆ `kill pid`

You use the `kill` command to stop other commands if you already have a shell prompt. You need to know the *process ID* of the command you want to stop. For details on obtaining process ID numbers, see "About Commands, Processes, and Jobs," later in this chapter.

The `kill` command doesn't always kill a process. It actually sends a signal asking it to stop. The default signal is hangup.

Sometimes that isn't strong enough. In those cases you can use signal 9, the kill signal that cannot be ignored:

`kill -9 pid`

Using signal 9 terminates the target with extreme prejudice—the stopped command has no chance to clean up before exiting and may leave temporary files around.

Table 2.1

Operators and Special Characters

Symbol	Example and Meaning
>	`command > file` Redirect output to `file`.
>>	`command >> file` Redirect output, appending to `file`.
<	`command < file` `command` gets input from `file`.
|	`cmdA | cmdB` (sometimes called the pipe character). Pipe output of `cmdA` into `cmdB`.
&	`command &` Run `command` in background, returning to shell prompt at once.
`` ` ` ``	``cmdA `cmdB` `` Execute `cmdB` first, then use output as an argument to `cmdA` (often called backtick characters).

Getting help for a command

Most commands have associated documentation in the Unix help system. Unfortunately, Unix help is almost always written for the experienced programmer, not for the novice user, so we have devoted all of Chapter 3 to clarifying it.

You can skip ahead to Chapter 3 and come back if you like, but here is the bare minimum you need to at least begin to explore the help available for commands.

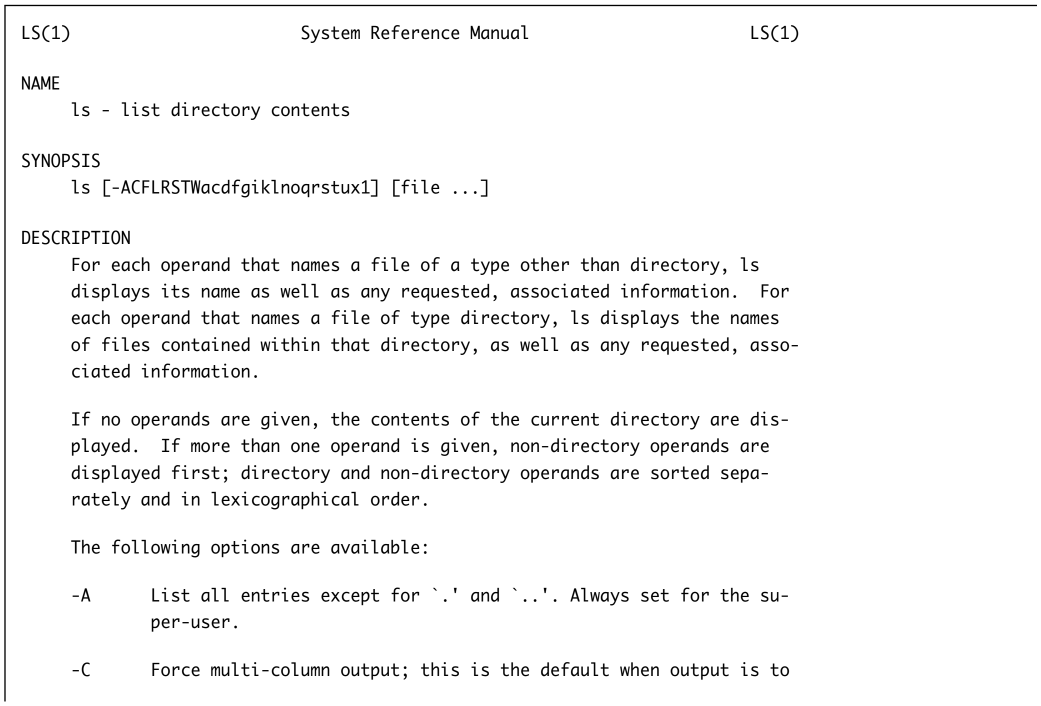
To read the Unix manual for a command:

1. `man` *command*

   This displays the Unix manual for any given command. **Figure 2.9** shows the first screen of the manual for the `ls` command, displayed by typing `man ls`.

   The `man` command displays the Unix manual entry for the named command one screen at a time.
2. To move forward one screen, press the z once.
3. To move backward one screen, press B once.
4. To quit from the `man` command and return to a shell prompt: Q

You should be back at the shell prompt.



```
LS(1)                    System Reference Manual                    LS(1)

NAME
     ls - list directory contents

SYNOPSIS
     ls [-ACFLRSTWacdfgiklnoqrstux1] [file ...]

DESCRIPTION
     For each operand that names a file of a type other than directory, ls
     displays its name as well as any requested, associated information.  For
     each operand that names a file of type directory, ls displays the names
     of files contained within that directory, as well as any requested, asso-
     ciated information.

     If no operands are given, the contents of the current directory are dis-
     played.  If more than one operand is given, non-directory operands are
     displayed first; directory and non-directory operands are sorted sepa-
     rately and in lexicographical order.

     The following options are available:

     -A      List all entries except for `.' and `..'. Always set for the su-
             per-user.

     -C      Force multi-column output; this is the default when output is to
```

Figure 2.9 When you request help from the manuals, such as `man ls`, you get an explanation (this only shows a partial amount of output).

Using Common Commands

You've already learned how to perform some basic Unix commands, but now let's run through a series of commands you'll use on a regular basis (we'll go into detail on several of these in later chapters).

To perform some basic commands:

1. `cd`

   The cd command ("change directory") produces no output. Used with no arguments, it tells your shell to set your "working directory" as your home directory.

2. `pwd`

   This command shows your present working directory —where you "are" in the Unix file system. **Figure 2.10** shows typical output from the pwd command.

3. `ls`

   Figure 2.11 shows typical output from `ls`, which lists the names of files and directories. The actual output depends on what you have in your home directory.

4. `echo "Hello there."`

   The output from the echo command consists of its arguments (in this case, the words `"Hello there"`) (**Figure 2.12**). It also automatically adds a new line (try it with the `-n` option to not add the new line).

pwd—Compare with Aqua

In Aqua, the Finder tells you where you are using the names and positions of windows. One window is always the active window, and the title bar of that window tells you the name of the folder. If the window is the Finder window, then the directory name in the title bar is the equivalent of what the Unix pwd command shows.

```
[localhost:~] vanilla% pwd
/Users/vanilla
[localhost:~] vanilla%
```

Figure 2.10 The pwd command shows your *present working directory*—where you "are" in the Unix file system.

```
[localhost:~] vanilla% ls
Desktop   Documents Library   Movies    Music     Pictures  Public    Sites
[localhost:~] vanilla%
```

Figure 2.11 The `ls` command lists the names of files and directories. The actual output will depend on what you have in your home directory.

```
[localhost:~] vanilla% echo "Hello there."
Hello there.
[localhost:~] vanilla%
```

Figure 2.12 The output from the echo command consists of two arguments (in this case, `"Hello there"`).

5. `echo "Hello $USER, welcome to Unix."`

Figure 2.13 shows output from echo, using the $USER *environment variable* in an argument. $USER is replaced by your short user name (the $ usually indicates that the following term is a variable, and the shell substitutes the value of the variable before executing the command; we'll go into more detail on *environment variables* in Chapter 7).

6. `echo "$USER created this" > file.txt`

In this case, the output from the echo commands doesn't go to your screen, but rather it is *redirected* into the file named `file.txt`, either creating the file with this specific content or copying over anything within it. For more on redirection and output, see "About Standard Input and Output," later in this chapter.

7. `ls`

Figure 2.14 shows the output from the `ls` command. The files listed now include `file.txt`, created in the previous step.

8. `cat file.txt`

The cat command, derived from the word *concatenate*, displays the contents of the file (**Figure 2.15**), again based on the command in step 6 (*concatenate* actually means "combine"; if you read the Unix manual section on cat with `man cat`, you will see how it can be used to combine several files).

echo—Compare with Aqua

The Aqua interface doesn't really have any equivalent of the echo command. Echo exemplifies a tool that is unique to command-line interfaces.

```
[localhost:~] vanilla% echo "Hello $USER, welcome to Unix."
Hello vanilla, welcome to Unix.
[localhost:~] vanilla%
```

Figure 2.13 This shows the output from echo, using the $USER environment variable in an argument. $USER will be replaced by your short username.

```
[localhost:~] vanilla% ls
Desktop   Library   Music     Public    file.txt
Documents Movies    Pictures  Sites
[localhost:~] vanilla%
```

Figure 2.14 This output from the ls command now includes `file.txt`.

```
[localhost:~] vanilla% cat file.txt
vanilla created this
[localhost:~] vanilla%
```

Figure 2.15 The cat command, derived from the word *concatenate*, displays the contents of a file.

cp—Compare with Aqua

In the Finder, you copy files by O -dragging them, or by selecting them and choosing File > Duplicate. After copying them, you can rename them in a separate operation.

At the command line, you select files to be copied by naming them, and then enter their new names at the same time.

9. `cp file.txt filecopy.txt`

   `cp` stands for *copy*. You have made a copy of `file.txt` called `filecopy.txt.` Run the `ls` command again to see it (**Figure 2.16**).

10. `rm filecopy.txt`

    The `rm` command *removes* the file. Run the `ls` command again to confirm that it is gone.

11. `mkdir testdir`

    The `mkdir` command creates (or *makes*) a new directory (that's what Unix calls folders), in this instance named `testdir`.

12. Go back out to the Finder (under Aqua) and open your home directory. You should see the file `file.txt` and the directory `testdir` (**Figure 2.17**).

```
[localhost:~] vanilla% ls
Desktop      Library      Music        Public       file.txt
Documents    Movies       Pictures     Sites        filecopy.txt
[localhost:~] vanilla%
```

Figure 2.16 Running the `ls` command again shows you have made a copy of `file.txt` called `filecopy.txt`.

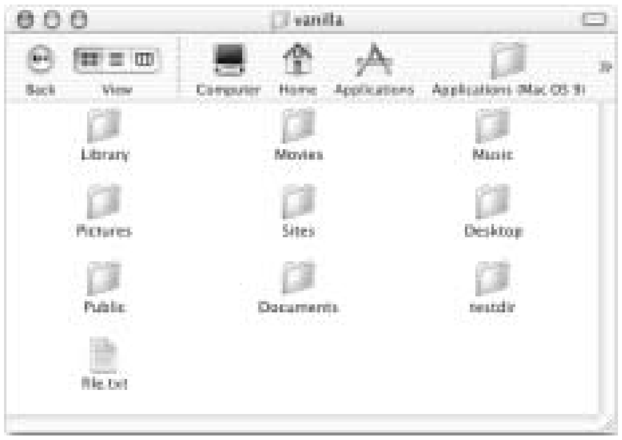


Figure 2.17 The Finder window now shows the new file and directory.

13. `cd testdir`

You have told your shell to change from the current directory to the directory named `testdir`. Notice that your shell prompt has changed to reflect your new directory (**Figure 2.18**).

14. `pwd`

This confirms that you are indeed in the new directory (**Figure 2.19**).

15. `date`

The `date` command displays the current date and time (**Figure 2.20**); unless you've perfected time travel, your output will be different.

16. `date > dates.txt`

This redirects the output of the `date` command into a file named dates.txt. It is often useful to save the output of a command.

17. `date >> dates.txt`

This time we redirect the output using the >> operator (this redirects the output and appends it to the file instead of replacing the contents).

18. `cat dates.txt`

The file contains the results of both redirects from steps 16 and 17 (**Figure 2.21**).

19. `mv dates.txt newname.txt`

The `mv` command renames (or *moves*) a file. In Unix, a file's name is actually the name of its location, so the same command is used to rename and to move files. Run the `ls` command to see that the file `dates.txt` has been renamed to `newname.txt` (**Figure 2.22**).

```
[localhost:~] vanilla% cd testdir
[localhost:testdir] vanilla%
```

Figure 2.18 After you use the `cd` command, the shell prompt changes.

```
[localhost:testdir] vanilla% pwd
/Users/vanilla/testdir
[localhost:testdir] vanilla%
```

Figure 2.19 Running the `pwd` command again shows your new working directory.

```
[localhost:testdir] vanilla% date
Thu Apr  4 14:56:35 PST 2002
[localhost:testdir] vanilla%
```

Figure 2.20 The `date` command displays the current date and time.

```
[localhost:testdir] vanilla% date > dates.txt
[localhost:testdir] vanilla% date >> dates.txt
[localhost:testdir] vanilla% cat dates.txt
Thu Apr  4 15:00:41 PST 2002
Thu Apr  4 15:00:47 PST 2002
[localhost:testdir] vanilla%
```

Figure 2.21 Using redirection, you get a file that contains the results of both redirects.

```
[localhost:testdir] vanilla% ls
newname.txt
[localhost:testdir] vanilla%
```

Figure 2.22 Running the `ls` command again shows the renamed file.

```
[localhost:testdir] vanilla% ls
dates.txt newname.txt
[localhost:testdir] vanilla%
```

Figure 2.23 Running the ls command again shows the file created with the >> operator.

```
[localhost:testdir] vanilla% ls
[localhost:testdir] vanilla%
```

Figure 2.24 Note that when there is nothing to list, the ls command gives no output.

```
[localhost:testdir] vanilla% cd
[localhost:~] vanilla%
```

Figure 2.25 Your shell prompt changes when you use the cd command.

20. `date >> dates.txt`

The >> operator creates a file if it doesn't already exist (**Figure 2.23**).

21. `rm *.txt`

The `*` operator is used in the command line as a *wildcard* to match all the files ending in `.txt`. As a result, the `rm` command actually receives two arguments—`dates.txt` and `newname.txt`—and acts on both of them. For more on wildcard operators, see "Wildcards," later in this chapter. Run the `ls` command to confirm that there are now no files in the current directory; you simply get a shell prompt back (**Figure 2.24**).

22. `cd`

This takes you back to your home directory. Notice that your shell prompt changes (**Figure 2.25**).

cat—Compare with Aqua

The `cat` command not only displays a single file (as shown in step 8 above), but it can be given multiple filenames as arguments in order to display them all, in one long output (hence the name *concatenate*).

The closest thing Aqua has to the `cat` command is the ability to open multiple files with one application by selecting several files and dragging them all onto an application icon, but there isn't really a direct equivalent.

mv—Compare with Aqua

In the Finder, you move files by dragging them to their new location. Renaming them is a separate operation.

At the command line, you can move and rename files in the same operation.

< and >—Compare with Aqua

The Aqua interface has no equivalent to the command lines' ability to redirect output. This is a good example of the difference between the Unix command-line interface and a graphical interface such as Aqua.

The command line is text oriented: Everything is assumed to be text output and can be fed into anything else. (See especially the | operator later in this chapter, in "Creating Pipelines of Commands").

In Aqua, each application is assumed to produce output of a different kind, and applications cannot normally feed their output directly into each other without saving to a file first.

About Commands, Processes, and Jobs

Commands fall into two categories: some commands are built into the shell you are using (for example, the `cd` command), while most are actually separate programs.

To see a list of basic Unix commands:

◆ `ls /bin`

The `/bin` directory contains all of the commands we used in the examples earlier in this chapter. Each command appears as a separate file (**Figure 2.26**). Notice that your shell program (`tcsh`) is included. It, too, is essentially a command, albeit a larger, interactive one.

✔ Tip

■ Other places to see lists of Unix commands are `/usr/bin`, `/sbin`, and `/usr/sbin`. (The *bin* is short for "binary," as most Unix commands are binary files. Not all commands are binary files; some are executable text files or *scripts*.)

```
[localhost:~] vanilla% ls /bin
[         csh        echo      ln        ps        sh        test
bash      date       ed        ls        pwd       sleep     zsh
cat       dd         expr      mkdir     rcp       stty
chmod     df         hostname  mv        rm        sync
cp        domainname kill      pax       rmdir     tcsh
[localhost:~] vanilla%
```

Figure 2.26 The `/bin` directory contains all of the commands we used in the examples earlier in this chapter.

Every time you issue a command that is not already built into a shell, you are starting what Unix calls a *process* or a *job*. You will encounter both terms in Unix literature.

Every process is assigned an identification number when it starts up, called the *PID* (*P*rocess *ID*), as well as its own slice of memory space (this is one of the reasons why Unix is so stable—each process has its own inviolable memory space). At any given moment, there are dozens of processes running on your computer.

To see all the processes you own:

- `ps -U` *`username`*

  Fill in your short user name for `username`. **Figure 2.27** shows typical output with a variety of programs running. Notice how even Aqua programs like iTunes are listed —underneath it all, they are all running on Unix.

  The first column of output lists the PID of each process. (Review "Stopping Commands" earlier in this chapter for an example of using a PID number.)

```
[localhost:~] vanilla% ps -U vanilla
  PID  TT  STAT      TIME COMMAND
   68  ??  Ss     0:19.58 /System/Library/Frameworks/ApplicationServices.framew
  366  ??  Rs    99:34.05 /System/Library/Frameworks/Kerberos.framework/Servers
 1065  ??  Ss    14:29.69 /System/Library/CoreServices/WindowServer console
 1066  ??  Ss     0:02.51 /System/Library/CoreServices/loginwindow.app/loginwin
 1071  ??  Ss     0:03.85 /System/Library/CoreServices/pbs
 1075  ??  S      2:05.73 /System/Library/CoreServices/Finder.app/Contents/MacO
 1076  ??  S      0:21.82 /System/Library/CoreServices/Dock.app/Contents/MacOS/
 1077  ??  S      0:28.22 /System/Library/CoreServices/SystemUIServer.app/Conte
 1078  ??  S      0:00.23 /Applications/iTunes.app/Contents/Resources/iTunesHel
 1080  ??  R    493:55.67 /System/Library/CoreServices/Classic Startup.app/Cont
 1081  ??  S      1:56.28 /Applications/Utilities/Terminal.app/Contents/MacOS/T
 1106  ??  S      7:38.94 /Applications/Acrobat Reader 5.0/Contents/MacOS/Acrob
 1142  ??  S    332:09.07 /Applications/Mozilla/Mozilla.app/Contents/MacOS/Mozi
 1357  ??  Ss     0:00.45 /usr/bin/hdid -f /Users/vanilla/Desktop/Eudora 51b21.
 1363  ??  S      0:00.90 /Applications/Preview.app/Contents/MacOS/Preview -psn
 1365  ??  S      0:01.13 /Applications/TextEdit.app/Contents/MacOS/TextEdit -p
 1436  ??  S      1:25.06 /Applications/Eudora/Eudora 5.1 (OS X) /Applications/
 1082 std  Ss     0:00.64 -tcsh (tcsh)
[localhost:~] vanilla%
```

Figure 2.27 When you type `ps -U username`, you see the variety of programs running, even Aqua programs like iTunes.

To see all the processes on the system:

1. `ps -aux`

   Figure 2.28 shows typical output from using the `-aux` options to `ps` (for *processes*).

2. `ps -auxw`

   Figure 2.29 shows output when using the `-auxw` options. The `w` makes the output wider. **Table 2.2** shows the common options for the `ps` command.

✔ Tips

- You can use two w's to make the output even wider—for example, `ps -auxww`.
- Combine the `-U` option with the `-aux` options to show a particular user's processes: `ps -aux -U` *`username`*.

```
[localhost:~] vanilla% ps -aux
USER      PID %CPU %MEM      VSZ    RSS  TT  STAT      TIME COMMAND
matisse  1142   6.9 12.2   160180  80116  ??  S    333:52.71 /Applications/Mozil
matisse  1080   6.0 18.3  1107988 119684  ??  R    507:33.80 /System/Library/Cor
matisse   366   3.4  0.1     5356    944  ??  Ss    99:54.01 /System/Library/Fra
matisse  1065   0.6  3.9    54776  25692  ??  Ss    14:43.58 /System/Library/Cor
root       72   0.0  0.0     1276    100  ??  Ss     1:49.77 update
root       75   0.0  0.0     1296    104  ??  Ss     0:00.01 dynamic_pager -H 40
root      100   0.0  0.1     2332    372  ??  Ss     0:01.09 /sbin/autodiskmount
root      125   0.0  0.2     3836   1516  ??  Ss     0:03.41 configd
root      163   0.0  0.0     1288    156  ??  Ss     0:03.17 syslogd
root      184   0.0  0.4    20736   2416  ??  Ss     0:00.24 /usr/libexec/CrashR
root      206   0.0  0.1     1580    404  ??  Ss     0:02.03 netinfod -s local
root      213   0.0  0.1     2448    520  ??  Rs     0:08.26 lookupd
root      223   0.0  0.0     1528    304  ??  S<s    1:03.80 ntpd -f /var/run/nt
root      231   0.0  0.3     8964   2040  ??  S      1:15.81 AppleFileServer
root      236   0.0  0.2     3104   1124  ??  Ss     0:08.35 /System/Library/Cor
root      243   0.0  0.0     1288    116  ??  Ss     0:00.00 inetd
root      254   0.0  0.0     1276     84  ??  S      0:00.00 nfsiod -n 4
root      255   0.0  0.0     1276     84  ??  S      0:00.00 nfsiod -n 4
root      263   0.0  0.0     2192    316  ??  Ss     0:00.04 automount -m /Netwo
root      266   0.0  0.2     3740   1148  ??  S      0:00.27 DirectoryService
root      273   0.0  0.1     2432    932  ??  Ss     0:40.56 /usr/sbin/httpd
www       277   0.0  0.1     2432    600  ??  S      0:00.25 /usr/sbin/httpd
root      282   0.0  0.1     2392    960  ??  Ss     0:01.60 /System/Library/Cor
 [localhost:~] vanilla%
```

Figure 2.28 Using the `-aux` options to `ps` (for *processes*) gives you this typical output.

```
[localhost:~] vanilla% ps -auxw
USER      PID %CPU %MEM      VSZ    RSS  TT  STAT      TIME COMMAND
matisse   366   3.2  0.1     5356    944  ??  Rs    99:55.39
/System/Library/Frameworks/Kerberos.framework/Servers/CCacheServer.app/
matisse  1080   3.1 18.3  1107988 119692  ??  S    508:31.05 /System/Library/CoreServices/Classic
Startup.app/Contents/Resources/Tru
matisse  1142   2.1 12.2   160180  80116  ??  S    334:01.62
/Applications/Mozilla/Mozilla.app/Contents/MacOS/Mozilla /Applications/
matisse  1081   1.3  1.6    76212  10168  ??  S      2:04.99
/Applications/Utilities/Terminal.app/Contents/MacOS/Terminal -psn_0_222
matisse  1065   0.3  3.9    54756  25672  ??  Ss    14:46.19 /System/Library/CoreServices/WindowServer
console
root       75   0.0  0.0     1296    104  ??  Ss     0:00.01 dynamic_pager -H 40000000 -L 160000000 -S
80000000 -F /private/var/vm/s
root      100   0.0  0.1     2332    372  ??  Ss     0:01.09 /sbin/autodiskmount -va
root      125   0.0  0.2     3836   1516  ??  Ss     0:03.41 configd
root      163   0.0  0.0     1288    156  ??  Ss     0:03.17 syslogd
root      184   0.0  0.4    20736   2416  ??  Ss     0:00.24 /usr/libexec/CrashReporter
root      206   0.0  0.1     1580    404  ??  Ss     0:02.03 netinfod -s local
root      213   0.0  0.1     2448    520  ??  Ss     0:08.26 lookupd
root      223   0.0  0.0     1528    304  ??  S<s    1:03.81 ntpd -f /var/run/ntp.drift -p
/var/run/ntpd.pid
root      231   0.0  0.3     8964   2040  ??  S      1:15.84 AppleFileServer
root      236   0.0  0.2     3104   1124  ??  Ss     0:08.35
/System/Library/CoreServices/coreservicesd
root      243   0.0  0.0     1288    116  ??  Ss     0:00.00 inetd
[localhost:~] vanilla%
```

Figure 2.29 Using the -auxw options to ps gives you this partial output; adding the w gives you a wider output.

Table 2.2

Common Options for ps

Option	Meaning
-a	Display processes owned by all users.
-u	Display more information, including CPU usage, process ownership, and memory usage.
-x	Include any process not started from a Terminal window.
-w	Wide listing—display the full command name of each process up to 132 characters per line. If more than w is used, adding ps will ignore the width of your Terminal window.
-U *username*	Show process for specified user.

To see a constantly updated list of the top processes:

1. `top`

   The `top` command displays a frequently updated list of processes, sorted by how much processing power each one is using—that is, which one is at the *top* of the list of resource usage (**Figure 2.30**). (The reason they're at 0% is that most processes, at any given time, aren't using that much processor time.)

   Top runs until you stop it by typing:

2. `q`

   This stops the `top` command and returns you to a shell prompt.

✔ Tip

- If you want to save the output of `top` to a file (such as using the > redirect operator), then use the `-l` switch and specify how many samples you want. For example, to get three samples use:

  `top -l3 > toplog.`

```
[localhost:~] vanilla% top
Processes:  52 total, 2 running, 50 sleeping... 151 threads           17:44:12
Load Avg:  0.36, 0.51, 0.59     CPU usage:  9.1% user, 90.9% sys, 0.0% idle
SharedLibs: num =  119, resident = 26.0M code, 1.80M data, 6.98M LinkEdit
MemRegions: num = 4441, resident =  203M + 7.88M private,  106M shared
PhysMem:  61.6M wired, 89.8M active,  367M inactive,  518M used,  122M free
VM: 2.49G + 50.8M   17504(17504) pageins, 156(156) pageouts
  PID COMMAND      %CPU   TIME   #TH #PRTS #MREGS RPRVT  RSHRD  RSIZE  VSIZE
 1539 top          0.0%  0:00.24   1    14    15   288K   328K   524K  1.45M
 1436 Eudora 5.1   0.0%  1:25.76   7   113   220  8.14M  14.1M  10.9M   102M
 1365 TextEdit     0.0%  0:01.13   1    66    81  2.20M  7.46M  3.97M  64.8M
 1363 Preview      0.0%  0:00.90   1    62    85  2.38M  7.50M  4.09M  64.9M
 1357 hdid         0.0%  0:00.45   1    11    37   808K   340K   676K  2.05M
 1142 Mozilla      0.0%  5:35:11   7   102   879  59.0M  33.7M  78.2M   156M
 1106 Acrobat Re   0.0%  7:43.54   1    54   183  6.68M  24.6M  18.6M  88.8M
 1082 tcsh         0.0%  0:00.75   1    24    15   480K   656K   960K  5.72M
 1081 Terminal     0.0%  2:07.07   6   120   138  4.20M  11.8M  9.93M  74.4M
 1080 TruBlueEnv   0.0%  8:38:24  18   187   489  90.8M  28.6M   117M  1.06G
 1078 iTunesHelp   0.0%  0:00.23   1    45    39   528K  2.90M  1.00M  38.1M
 1077 SystemUISe   0.0%  0:28.49   2   107   104  1.46M  6.96M  2.60M  61.0M
 1076 Dock         0.0%  0:22.69   3   108   105  2.07M  8.41M  4.60M  63.1M
 1075 Finder       0.0%  2:05.74   3   112   263  13.6M  14.1M  18.9M  83.1M
 1071 pbs          0.0%  0:03.85   1    29    29  1.98M   812K  2.86M  19.4M
 1066 loginwindo   0.0%  0:02.51   7   133   122  2.71M  6.93M  4.50M  51.5M
 1065 Window Man   0.0% 14:52.00   3   182   219  2.13M  23.1M  25.1M  53.5M
  948 httpd        0.0%  0:00.22   1     9    66   140K  1.23M   604K  2.38M
  947 httpd        0.0%  0:00.17   1     9    66   140K  1.23M   604K  2.38M
[localhost:~] vanilla%
```

Figure 2.30 The `top` command displays a frequently updated list of processes, sorted by how much processing power each one is using.

The Danger of a Space Misplaced

A bug in the installation software for an early version of iTunes could cause the erasure of an entire hard drive if the first character in the drive's name was a space.

The installation script did not allow for that possibility and neglected to use quotes where it should have. Even professional programmers occasionally have trouble dealing with spaces in filenames on Unix systems.

About Spaces in the Command Line

As we have seen, the shell uses spaces to separate the parts of the command line. Having two or more spaces separate a command from its options or its arguments doesn't change anything. When your shell acts on your command line, it breaks it into pieces by looking at where the spaces are. The following command lines both do the same thing:

```
ls -l
```

```
ls      -l
```

But this one is very different:

```
ls -  l
```

In the first two cases the shell sees two items: `ls` and `-l` In the third case the shell sees three items: `ls`, `-`, and `l`.

But there are times when you have to include a space inside an argument—such as in a filename that itself contains spaces. Consider what would happen if you tried the command line

```
ls /Applications (Mac OS 9)
```

If you don't do something special to handle spaces in command-line arguments, you will have problems. The shell treats the spaces as separators, and you will get unexpected and probably undesired results. (**Figure 2.31**)

Here are two ways to handle spaces safely in command-line arguments.

```
[localhost:~] vanilla% ls /Applications (Mac OS 9)
Badly placed ()'s.
[localhost:~] vanilla%
```

Figure 2.31 When you use unprotected spaces in a command-line argument, you get an error message about the misplaced parentheses.

To protect spaces using quotes:

1. `ls "/Applications (Mac OS 9)"`

   When you enclose the argument in quotes, the shell treats everything within the quotes as a single entity.

2. You may also use single quotes:

   `ls '/Applications (Mac OS 9)'`

✔ Tip

- Using single quotes around a string of characters eliminates the effect of any special character, including the $ we saw earlier for environment variables. Compare

  `echo 'hello $USER'`

  with

  `echo "hello $USER"`

  The first one echoes the exact characters, while the second identifies the user.

To protect spaces using the backslash:

◆ `ls /Applications\ (Mac\ OS\ 9)`

The backslash character (`\`) is often used in Unix to *escape* a character. This means “make the next character not special.” In this case it removes the special meaning of “separator” from the space character. This is called *escaping a character*.

✔ Tip

- Many Unix shells (including the default shell on Mac OS X) provide a feature called *filename completion*. When typing a part of a command line that is an already-existing file, you can type just part of it and then press *t* ; the shell tries to fill in the rest for you.

Wildcards

Arguments to commands are frequently filenames. These might be the names of files the command should read, copy, or move. If you want to act on a number of files, you don't want to have to type every filename, especially when all the filenames have some pattern in common.

That's where wildcards come in. Wildcards (often called glob-patterns) are special characters you can type in a command line to make a command apply to a group of files whose names match some pattern—for example, all files ending in .jpg.

When the shell reads a command line, it expands any glob-patterns by replacing them with all the filenames that match. The shell then executes the command line, using the new list of arguments with the command.

To use a glob-pattern to match all filenames starting with Hello:

◆ `ls Hello*`

The asterisk (`*`) is the glob character that matches any number (zero or more) of characters.

The shell finds all the filenames that begin with `Hello` and substitutes that list for the `Hello*` on the command line.

So if the directory contains files with the names `Hello`, `HelloTest`, and `HelloGoodbye`, then the shell changes the command line with the wildcard into

```
ls -l Hello HelloGoodbye HelloTest
```

✔ Tip

■ You can use more than one glob-pattern in a command line, such as

```
rm *.jpg *.gif
```

This removes all the `.jpg` and `.gif` files from the current directory.

To use a glob-pattern to match only one single character:

◆ `ls "File?"`

The `?` character matches any single character. So the example above would match files with names such as `FileA`, `File3`, `Files`, and so on. It would not match `File23` because the pattern only matches one character.

✔ Tip

■ You can combine the `?` and `*` glob characters together. For example,

```
ls ??.*
```

This would list files whose names begin with exactly two characters, followed by a period, followed by anything. (The period is matched literally.)

More specialized glob-patterns

Sometimes you want to use a list of files that matches a more specific pattern. For this you might use a more complex kind of pattern.

To match a range of characters:

1. `ls  /var/log/system.log.[0-3].gz` would result in output similar to that shown in **Figure 2.32**.

   The `[ ]` characters are used to create a glob-pattern called a *character class*. The resulting pattern matches any single character in the class. A range of characters can be indicated by using the hyphen, so that `[0-3]` is the same as `[0123]`.
2. Ranges may be alphabetical as well as numeric:

   `ls Alpha-[A-D]`
3. Unix files names are case-sensitive. You can match either by including both in the character class:

   `ls Alpha-[A-Da-d]`

✔ Tip

- You can create a character class that is quite arbitrary—for example, the glob-pattern

  `Photo-[AD]`

  matches only `Photo-A` and `Photo-D`.

To negate a character class:

- Use the ^ character as the first character in the character class.

  When you do this, the glob-pattern `*[^3-8]` matches anything that does not end in `3`, `4`, `5`, `6`, `7`, or `8` (the `*` matches anything, the `[^3-8]` means "do not match `3-8`").

Patterns and rules similar to those described here are used in many different Unix tools, especially in a set of tools called regular expressions. See Chapter 4, "Useful Unix Utilities," for more on regular expressions.

```
[localhost:~] vanilla% ls  /var/log/system.log.[0-3].gz
/var/log/system.log.0.gz  /var/log/system.log.2.gz
/var/log/system.log.1.gz  /var/log/system.log.3.gz
[localhost:~] vanilla%
```

Figure 2.32 This shows the output when you use a glob-pattern — in this case [0-3] — for a range of characters.

About Standard Input and Output

Normally, the output from command lines shows up on your screen. You type in a command and press r, and the resulting output shows up on your screen. This is actually a special case of the more general-purpose way that Unix handles both input and output.

All Unix commands come with two input/output devices, called `stdin` (for *standard input*) and `stdout` (for *standard output*). You can't see the `stdin` and `stdout` devices the same way you see a printer, but they are always there. You might like to think of `stdin` and `stdout` as valves or hose-connectors stuck on the outside of every command. Think of the `stdout` connector as being fed to your screen, and your keyboard feeding the `stdin` connector.

You have seen redirection of `stdout` with the > and >> operators earlier in this chapter. In this section, you will learn more about redirecting `stdout`, and also how to redirect `stdin`—that is, to have a command get input from someplace other than the keyboard—and how to connect the `stdout` of one command to the `stdin` of another, creating what is called a *pipeline*. This ability to connect several commands together is one of the most important features of Unix's flexibility.

Besides `stdin` and `stdout`, there is one more virtual connection on each command, `stderr` (for *standard error*), which is output connection for warning and error messages. If a command issues an error message, it comes out of the `stderr` connector, which is normally connected to your screen (same as `stdout`). If you redirect `stdout` to a file (with >), `stderr` still goes to your screen. You can redirect `stderr` as well, though.

`stdin`, `stdout`, and `stderr` are often capitalized in Unix manuals and literature. Both upper- and lowercase usage is correct.

Figure 2.33 illustrates the concept of the `stdin`, `stdout`, and `stderr` connectors.

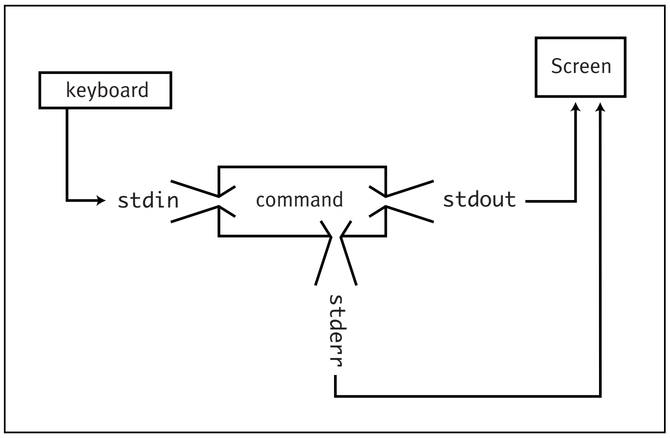


Figure 2.33 Here's how the normal connections of `stdin`, `stdout`, and `stderr` work.

Redirecting stdout

The most common form of redirection is to redirect the output of a command into a file. You will redirect `stdout` to a file when you want to save the output for later use such as editing or viewing.

To save output in a file:

◆ Add `>` *`filename`* to the command line.

For example,

```
ls /Users > users
```

redirects the output into the file named users. If the file does not already exist, it is created. If the file does exist, the contents are overwritten.

✔ Tip

- Sometimes you want to simply throw away the output of a command without ever seeing it. To do this, redirect the output into the special file called `/dev/null`. For example,

```
noisy_command > /dev/null
```

- Anything written to /dev/null is simply discarded. It's a good place to send insults and complaints.

Sometimes you will want to add the redirected output to a file instead of overwriting the contents:

To append output to a file:

◆ Use `>>` *`filename`* at the end of the command—for example,

```
ls /Users >> users
```

redirects the output to the file called users, or creates the file if it did not already exist.

Redirecting stderr

`Stderr` can be redirected as well. One reason to do this is to save errors into a log file. Another reason is to not have warning and error messages clutter up your screen.

To redirect stderr to a file:

◆ Use `>&` *`filename`*

In the `tcsh` shell you cannot redirect `stdout` and `stderr` separately, but you can make `stderr` go to the same file as `stdout`. For example,

```
ls /bin >& outfile
```

puts the `stdout` and `stderr` output into the file named outfile. You can use `>>&` to append instead of overwriting an existing file.

In the `bash` shell you can redirect `stdout` and `stderr` separately.

Redirecting stdin

Sometimes you'll want a command to get its input from a file you have prepared. For example, you might have prepared the text of an email message and want to feed it into the mail command, or you have a list of file names you want to feed into a program.

Most Unix commands allow you to redirect standard input and have it come from a file instead the keyboard.

To take standard input from a file:

◆ Add `<` *`filename`* to the end of the command.

For example, if you have a file that contains a list of 30 file names, you might type

```
ls -l < list_of_files
```

instead of

```
ls -l file1 file2 file3
```

. . . and so on.

Creating Pipelines of Commands

Another way to manipulate `stdin` and `stdout` is to have one command take its input directly from the output of another command. To do this, Unix uses the `|`, or *pipe*, character to connect commands together to form *pipelines*, with the `stdout` of each command being piped into the `stdin` of the next one. The enables you to create an almost infinite variety of command lines processing input and output to what are nothing more than miniature custom applications.

To pipe the output of one command into another:

1. Type the first command that produces output, but don't press *r* until the last step below.

   This can be any command (along with options and arguments) that produces output on `stdout`. For example,

   ```
   ls -l /bin
   ```

   lists the contents of /bin, where the executable files for many commands are stored (**Figure 2.34**). But let's say we want to know the modification date and the filename. Noticing that this information starts 38 characters into each line, we might pipe the output of the `ls` command into the `cut` command, telling `cut` to show us only character 38 and up from each line.

```
[localhost:~] vanilla% ls -l /bin
total 4208
-r-xr-xr-x  1 root  wheel      13728 Dec 20 22:24 [
lrwxr-xr-x  1 root  wheel         19 Dec 20 22:25 bash -> /usr/local/bin/bash
-r-xr-xr-x  1 root  wheel      13980 Dec 20 22:25 cat
-r-xr-xr-x  1 root  wheel      13852 Dec 20 22:22 chmod
-r-xr-xr-x  1 root  wheel      19012 Dec 20 22:25 cp
-r-xr-xr-x  1 root  wheel     318716 Dec 20 22:25 csh
-r-xr-xr-x  1 root  wheel      14716 Dec 20 22:22 date
-r-xr-xr-x  1 root  wheel      22580 Dec 20 22:23 dd
-r-xr-sr-x  1 root  operator   18660 Dec 20 22:23 df
-r-xr-xr-x  1 root  wheel       9856 Dec 20 22:24 domainname
-r-xr-xr-x  1 root  wheel       9216 Dec 20 22:24 echo
-r-xr-xr-x  1 root  wheel      56304 Dec 20 22:23 ed

(...output truncated for brevity...)
```

Figure 2.34 Running `ls -l /bin` lists the contents of /bin, where the executable files for many commands are stored.

2. Add the | character at the end of the command.

   The | character catches the output from what is on its left and passes it to the `stdin` of what is on its right.

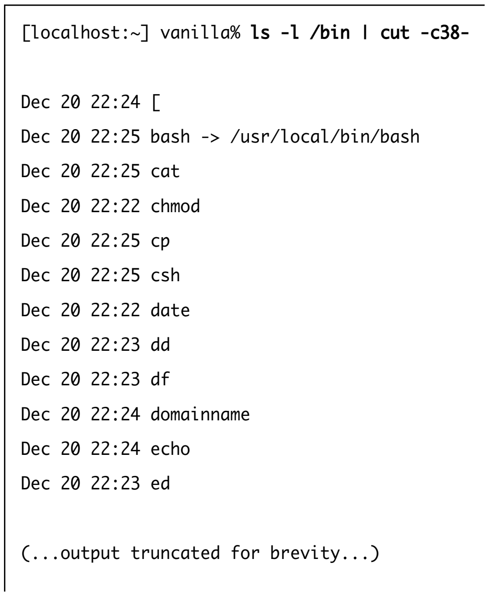
3. Add the command that will receive its input from the pipe. Continuing our example, the command line would now look like this:

   `ls -l /bin | cut -c38-`

   and the result would look like **Figure 2.35**.

   Now you can press *r* . Or if you wanted to redirect the output of the pipeline to a file, the final command line would be:

   `ls -l /bin | cut -c38- > outfile`



```
[localhost:~] vanilla% ls -l /bin | cut -c38-

Dec 20 22:24 [
Dec 20 22:25 bash -> /usr/local/bin/bash
Dec 20 22:25 cat
Dec 20 22:22 chmod
Dec 20 22:25 cp
Dec 20 22:25 csh
Dec 20 22:22 date
Dec 20 22:23 dd
Dec 20 22:23 df
Dec 20 22:24 domainname
Dec 20 22:24 echo
Dec 20 22:23 ed

(...output truncated for brevity...)
```

Figure 2.35 Running `ls -l /bin | cut -c38-` gives this partial output.

Before this next task will work on your Mac OS X system, you need to make a small fix; see Chapter 14, "Installing and Configuring Servers," the section "Fixing the Permissions on the Root Directory so Sendmail Will Work." Once you have made the fix, and assuming your computer is connected to the Internet, you can pipe the output of any command into an email message for another user on the Internet.

To pipe the output of a command into email:

- `*command* | mail -s "*subject*" address`

  `command` can be any command line that produces output on `stdout`. *`Subject`* is the subject of the email message, and *`address`* is a valid email address.

Running a Command in the Background

Some commands take a while to run. For example, a command that searches through a large number of files or a command that must read a large amount of input may take more time than you're willing to twiddle your thumbs.

It is a simple matter to have a command run in the background and get a shell prompt right away so you can keep on working. The command keeps running—you can stop it or bring it to the foreground if you like—but you can let the operating system worry about the command while you do other things.

To run a command line in the background:

◆ Add the & character at the end of the command line.

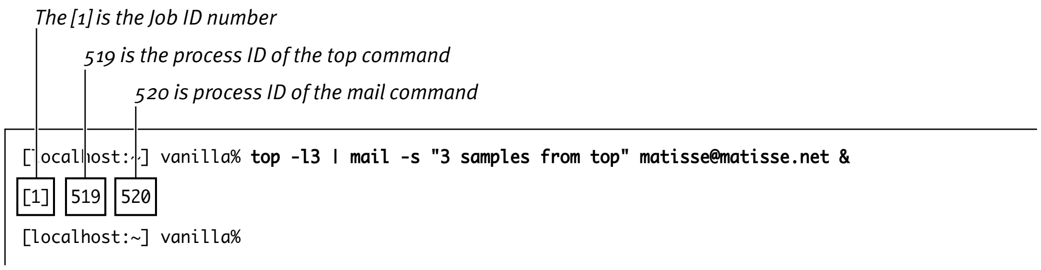
  At the end of any command line, you can add the & character so that the entire command line runs in the background.

The shell shows you the background job number and the process ID numbers for each command on the command line, and gives you a shell prompt right away. **Figure 2.36** shows an example of getting three samples from the `top` command and sending them in email.

Once the job is finished running, the shell displays a notice the next time it gives you a new shell prompt. That is, the shell does not spontaneously notify you, but it waits and displays the notice along with the next shell prompt (**Figure 2.37**).

✔ Tip

■ If a background job needs input from you, it simply sits and waits patiently—possibly forever. Before putting a job in the background, you should have a good idea of how it behaves normally.


The [1] is the Job ID number
519 is the process ID of the top command
520 is process ID of the mail command

```
[localhost:~] vanilla% top -l3 | mail -s "3 samples from top" matisse@matisse.net &
[1] 519 520
[localhost:~] vanilla%
```

Figure 2.36 Running a command in the background, here you've gotten three samples from the top command and sent them in email.

```
[localhost:~] vanilla%
    [1]  - Done          top -l3 | mail -s 3 samples from top matisse@matisse.net
[localhost:~] vanilla%
```

Figure 2.37 The shell notifies you when the job is completed.

You can have several jobs running in the background, and you can bring any of them back to the foreground, or stop any of them by using the kill command, discussed earlier in this chapter.

To see a list of jobs running in the background:

◆ `jobs`

The `jobs` command displays a list of background jobs started from the current shell (**Figure 2.38**). If you have multiple Terminal windows open, each has its own list of background jobs.

Even if a job consists of multiple commands (processes), it has a single job number.

To bring a job back to the foreground:

1. `jobs`

   Run the `jobs` command to get a list of job numbers. Pick the one you want to bring back to the foreground.

2. `fg %n`

   Use the `fg` command (meaning *foreground*) to bring the job from the background. You identify the job with `%` and its job number: `%1` for job 1, `%2` for job 2, and so on.

   The job is now running in the foreground.

✔ Tip

■ Once a job is in the foreground, you can stop it with C C, or with the method shown below.

To stop a background job:

1. `jobs`

   This shows you the list of all jobs running.

2. `kill %n`

   where *n* represents any number, such as

   `kill %2`

   This is the same `kill` command we saw earlier in this chapter, only this time instead of the process ID we are using the job ID. The same options apply here. `Kill` by itself sends a hangup signal to each process in the job, requesting that it quit; `kill -9` sends `kill` signals that can't be ignored, stopping the job in its tracks.

Sometimes you might not realize that a command is going to take a while to finish until after you start it. You might have pressed r and find yourself waiting for the job to finish. Or maybe you want to temporarily stop a job, get a shell prompt, do something else, and then return to the job that was stopped. You can *suspend* the job and get a shell prompt back right away.

A suspended job will be in the background, but it won't keep running; that is, its memory remains active, but it consumes no processor time. You can bring it back to the foreground, or tell it to keep running in the background, just as if you had started it initially with an `&` at the end of the command line.

```
[localhost:~] vanilla% jobs
   [1]  + Suspended                vm_stat 5
   [2]  + Running                  top -l3 | mail -s 3 samples from top matisse
[localhost:~] vanilla%
```

Figure 2.38 Running the `jobs` command gives you a list of jobs running in the background.

Compare with Aqua

The Unix concept of putting a command in the background is very much like the traditional Mac or Aqua situation where you have an application running in one window, and you open a window for a different application. The first application continues to run, and you can get on with other things.

To suspend a job:

◆ C Z

Figure 2.39 shows what happens when you use C Z to suspend the `top` command while it is running. The shell shows you the job ID and process ID of the suspended job, and returns you to a shell prompt.

✔ Tip

■ You can bring the job back to the foreground as described above using the `fg` command. If the job you want to bring back to the foreground is the one you just suspended, you can use `fg` by itself with no arguments.

```
SharedLibs: num =   93, resident = 22.5M code, 1.57M data, 5.52M LinkEdit
MemRegions: num = 3234, resident =  142M + 7.07M private, 84.3M shared
PhysMem:  60.1M wired, 71.8M active,  245M inactive,  377M used,  263M free
VM: 2.34G + 45.8M   9564(0) pageins, 0(0) pageouts

  PID COMMAND      %CPU   TIME   #TH #PRTS #MREGS RPRVT  RSHRD  RSIZE  VSIZE
  439 Mozilla      1.7% 36:35.72   6    87    437  26.6M  25.4M  43.2M   109M
  396 AOL Instan   1.7% 31:01.89  10   121    170  10.7M  11.0M  16.2M  79.3M
  379 Eudora 5.1   0.0%  7:26.90   7   112    162  6.49M  12.5M  11.2M   100M
  356 httpd        0.0%  0:00.13   1     9     65   140K  1.25M   612K  2.38M
  329 tcsh         0.0%  0:00.61   1    24     17   508K   676K   996K  5.99M
  328 ssh-agent    0.0%  0:01.23   1     9     14    80K   352K   172K  1.29M
  327 Terminal     1.7%  0:51.57   5   114    103  2.89M  7.12M  6.08M  68.5M
  325 sh           0.0%  0:00.01   1    16     13   164K   640K   556K  1.69M
^Z
[1]  +   736 Suspended                    top
[localhost:~] vanilla%
```

Figure 2.39 Using C Z suspends the `top` command.

If you have suspended a job and decide you want the job to keep running, but in the background, you can do that, too.

To have a suspended job continue running in the background:

◆ `bg %n`

For example,

`bg %2`

tells job number 2 to start running again, but to do so in the *background*. **Figure 2.40** shows an example of starting a job, suspending it, running another command, and then starting up the suspended job again in the background.

✔ Tip

- If the suspended job is the one you most recently suspended, you can use `bg` with no arguments.

```
[localhost:~] vanilla% find /Developer/Documentation -name "*.htm" > found_files
^Z
[1]  +   743 Suspended   find /Developer/Documentation -name *.htm > found_files
[localhost:~] vanilla% uptime
 9:18AM  up 1 day, 15 mins, 2 users, load averages: 0.08, 0.15, 0.14
[localhost:~] vanilla% bg %1
[1]    find /Developer/Documentation -name *.htm > found_files &
```

Figure 2.40 You can suspend a job, and then restart it in the background using `bg %n`.

Opening Files from the Command Line

One of the great things about using the Unix command line in Mac OS X is that you are also using a Macintosh. So how do you access graphical Mac applications or AppleScripts from the command line? Apple provides a set of command-line tools to do exactly that.

To "double-click" a file from the command line:

◆ `open` *`filename`*

The `open` command performs the equivalent of double-clicking each of its arguments. For example,

```
open *.doc FunReport
```

is the same as double-clicking all the `.doc` files in the current directory, along with the file `FunReport`. The default application for each file is used just as if you had double-clicked the icons in the Finder.

✔ Tip

■ You can specify which application to use with the `-a` switch (or option)—for example,

```
open -a "BBEdit 6.5" found_files
```

would open the file called `found_files` using the BBEdit 6.5 application.

To run an AppleScript from the command line:

◆ `osascript` *`scriptname`*

The `osascript` command executes the script named by its argument.

✔ Tip

■ If you are an experienced AppleScript programmer, read the Unix manual for `osascript` by typing:

```
man osascript
```

You will also be interested in learning about the `osacompile` and `osalang` commands.

Importance of Editing Text in Unix

Editing text from the command line is a crucial part of using Unix.

Unix system-configuration files, system-startup files, source code for software, and much documentation are all contained in text files, which you will have occasion to edit when using the command line.

While you can certainly use your favorite Aqua text editor or word processor to edit text in Mac OS X, you will need to be able to edit files directly from the command line if you do any serious command-line work.

Also, if you want to be able to use other Unix systems besides Mac OS X, you will need to learn how to edit files using one of the command-line tools.

Chapter 6, "Editing and Printing Files," teaches you the basics of using the most common command-line text editor, the `vi` editor. In this chapter, you will use the simpler `pico` editor, which is adequate for the example of creating a shell script but is not appropriate for more complex Unix work such as editing system-startup files.

Creating a Simple Unix Shell Script

A shell script is a text file that contains a series of shell commands. Shell scripts are used for a wide range of tasks in Unix; Chapter 9, "Creating and Using Scripts," covers complex shell scripts that contain loops, functions, and other features associated with computer programming. But here we'll talk about simpler shell scripts, which are often just a series of command lines intended to be executed one after the other.

When you create a script you'll be using frequently, you should save it in a place where your shell normally looks for commands. This way, you can run the script by simply typing its name, as you would for any other command. The list of places where your shell looks for commands is called your `PATH`, and we teach you how to change your `PATH` in Chapter 7, "Configuring Your Unix Environment."

The standard place to store scripts for your personal use (as opposed to scripts intended for use by all users) is the `bin` directory inside your home directory. Mac OS X (as of version 10.2) ships without this directory's having been created for each user and without its being on the list of places where your shell looks for commands (your `PATH`), so before we have you actually create a script, we show you how to create this directory and add it to your `PATH`. (See Chapter 7 for more on your `PATH`.)

To create your personal bin directory:

1. `cd`

   This ensures that you are in your home directory.

2. `mkdir bin`

   This creates a new directory called `bin` (a standard Unix name for directories that contains commands, scripts, or applications).

✔ Tip

- See Chapter 5, "Using Files and Directories," for more on the `mkdir` command.

To add your bin directory to your PATH:

1. `cd`

   This ensures that you are in your home directory. (You can skip this step if you have just done the task above, but it doesn't hurt to do it again.)

2. `set path = ( $path ~/bin )`

   This adds the `bin` directory inside your home directory to the list of places your shell searches for commands. The next step takes care of doing this for all future Terminal windows you open.

3.
   ```
   echo 'set path = ( $path ~/bin )' >>
   ➝ .tcshrc
   ```

   That command line adds a line of code to a configuration file used by your shell. Every new Terminal window you open will have the new configuration.

   Be careful to type it exactly as shown—the placement of spaces and the use of single quotes must be replicated exactly. Be sure to type both greater-than signs.

   The text contained inside the single quotes is added as a new line to the end of the file `.tcshrc` that is inside your home directory. You may check that this was successful by displaying that file with

   ```
   cat .tcshrc
   ```

   and the last line of the output should be

   ```
   set path = ( $path ~/bin )
   ```

Unix shell scripts can be written for any of the available shells, but the standard practice is to write shell scripts for the sh shell. The sh shell can be expected to behave in the same way on any Unix system.

Here is an example of creating a simple shell script that shows you a variety of status information about your computer.

To create a shell script to show system status:

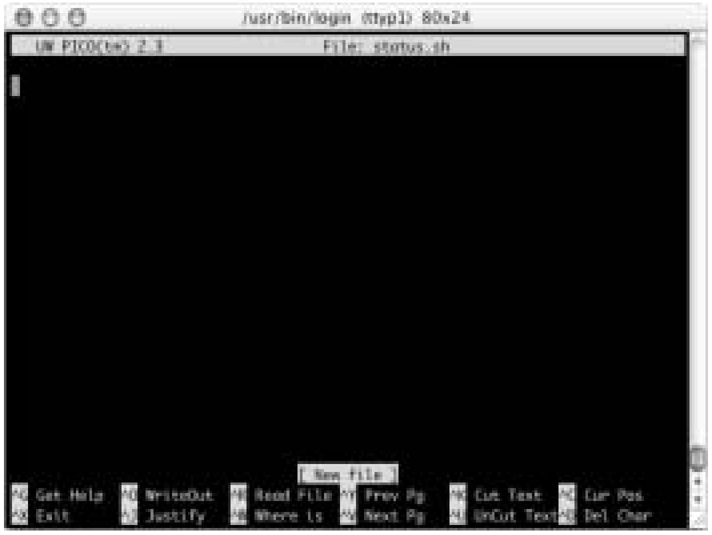
1. `cd`

   This makes sure you are in your home directory.
2. `cd bin`

   This changes your current directory to the bin directory.
3. `pico status.sh`

   This starts up the `pico` editor, telling it to edit (and create) the file `status.sh` (**Figure 2.41**).

   We name the new script with a `.sh` extension as a reminder that it is written using the `sh` scripting language.



```
/usr/bin/login (ttyp1) 80x24
  UW PICO(tm) 2.3                    File: status.sh

                                [ New file ]
^G Get Help   ^O WriteOut   ^R Read File ^Y Prev Pg    ^K Cut Text   ^C Cur Pos
^X Exit       ^J Justify    ^W Where is  ^V Next Pg    ^U UnCut Text ^T Del Char
```

Figure 2.41 This is what you see when you start the `pico` editor.

4. Type in the script from **Figure 2.42**. Note that the highlighted line uses the *back-quote* character—this is the ` character, which is usually in the upper left of your keyboard (to the left of 1).

5. C X

Typing C X causes the `pico` editor to quit. `Pico` asks you if you want to save the changes you have made (**Figure 2.43**).

```
#!/bin/sh
# This is a comment. Comments are good.
# This is my first shell script.
echo "System Status Report"
date
echo -n "System uptime and load:" ;  uptime
echo -n "Operating System: " ; sysctl -n kern.ostype
echo -n "OS Version: " ; sysctl -n kern.osrelease
echo -n "OS Revision number: " ; sysctl -n kern.osrevision
echo -n "Hostname: " ; sysctl -n kern.hostname

bytes=`sysctl -n hw.physmem`
megabytes=`expr $bytes / 1024 / 1024`
echo "Physical memory installed (megabytes): $megabytes"
```

Figure 2.42 This is the code listing of a system-status script.

```
[localhost:bin] vanilla% status.sh
System Status Report
Sat Apr  6 11:36:43 PST 2002
System uptime and load:11:36AM  up 1 day,  2:33, 4 users, load averages: 0.26, 0.42, 0.55
Operating System: Darwin
OS Version: 5.3
OS Revision number: 199506
Hostname: localhost.localdomain
Physical memory installed (megabytes): 640
[localhost:bin] vanilla%
```

Figure 2.43 The `pico` editor asks if you want to save changes.

6. y

Type a y to tell pico that yes, you want to save the changes you have made.

Pico then asks you to confirm the filename to save to (**Figure 2.44**).

7. Press r .

Pico exits, and you are back at a shell prompt.

8. chmod 755 status.sh

The chmod command (*change mode*) sets the file status.sh to be an executable file. See Chapter 8, "Working with Permissions and Ownership," for more on the chmod command.

Now all you have to do is tell tcsh (your shell) to rebuild its database of available commands to include status.sh.

9. rehash

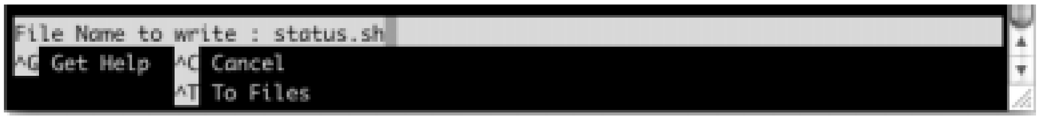
The rehash command makes tcsh rebuild its list of where to find commands. Tcsh now knows where to find the new command, status.sh.

You can use the new command as you would any other—just type its name at a shell prompt and press r .

10. status.sh

Figure 2.45 shows typical output from the command. You have just created your first new command.

Welcome to Unix.



```
File Name to write : status.sh
^G Get Help   ^C Cancel
              ^T To Files
```

Figure 2.44 The pico editor asks you to confirm the filename you're using.

```
[localhost:bin] vanilla% status.sh
System Status Report
Sat Apr  6 11:36:43 PST 2002
System uptime and load:11:36AM  up 1 day,  2:33, 4 users, load averages: 0.26, 0.42, 0.55
Operating System: Darwin
OS Version: 5.3
OS Revision number: 199506
Hostname: localhost.localdomain
Physical memory installed (megabytes): 640
[localhost:bin] vanilla%
```

Figure 2.45 Typing *status.sh* shows you output from the command you have just created.