Understanding Shell Scripts

This section explains how more advanced shell scripts work. The information is also adequate to equip you to write many of your own useful shell scripts. The section begins by showing how to process a script's arguments. Then it shows how to perform conditional and iterative operations.

Processing Arguments

You can easily write scripts that process arguments, because a set of special shell variables holds the values of arguments specified when your script is invoked. **Table 13-10** describes the most popular such shell variables.

For example, here's a simple one-line script that prints the value of its second argument:

```
echo My second argument has the value $2.
```

Suppose you store this script in the file `second`, change its access mode to permit execution, and invoke it as follows:

```
./second a b c
```

The script will print the output:

```
My second argument has the value b.
```

Table 13-10. Special Shell Variables Used in Scripts

Variable	Meaning
`$#`	The number of arguments.
`$0`	The command name.
`$1`, `$2`, ... , `$9`	The individual arguments of the command.
`$*`	The entire list of arguments, treated as a single word.
`$@`	The entire list of arguments, treated as a series of words.
`$?`	The exit status of the previous command. The value 0 denotes successful completion.
`$$`	The process id of the current process.

Notice that the shell provides variables for accessing only nine arguments. Nevertheless, you can access more than nine arguments. The key to doing so is the **shift** command, which discards the value of the first argument and shifts the remaining values down one position. Thus, after executing the **shift** command, the shell variable `$9` contains the value of the tenth argument. To access the eleventh and subsequent arguments, you simply execute the **shift** command the appropriate number of times.

Exit Codes

The shell variable `$?` holds the numeric exit status of the most recently completed command. By convention, an exit status of zero denotes successful completion; other values denote error conditions of various sorts.

You can set the error code in a script by issuing the **exit** command, which terminates the script and posts the specified exit status. The format of the command is:

exit *status*

where *status* is a non-negative integer that specifies the exit status.

Conditional Logic

A shell script can employ conditional logic, which lets the script take different action based on the values of arguments, shell variables, or other conditions. The **test** command lets you specify a condition, which can be either true or false. Conditional commands (including the **if**, **case**, **while**, and **until** commands) use the **test** command to evaluate conditions.

The test command

Table 13-11 describes some commonly used argument forms used with the **test** command. The **test** command evaluates its arguments and sets the exit status to 0, which indicates that the specified condition was true, or a non-zero value, which indicates that the specified condition was false.

Table 13-11. Commonly Used Argument Forms of the test Command

Form	Function
-d *file*	The specified file exists and is a directory.
-e *file*	The specified file exists.
-r *file*	The specified file exists and is readable.
-s *file*	The specified file exists and has non-zero size.
-w *file*	The specified file exists and is writable.
-x *file*	The specified file exists and is executable.
-L *file*	The specified file exists and is a symbolic link.
f1 -nt *f2*	File *f1* is newer than file *f2*.

Form	Function
f1 -ot *f2*	File *f1* is older than file *f2*.
-n *s1*	String *s1* has nonzero length.
-z *s1*	String *s1* has zero length.
s1 = *s2*	String *s1* is the same as string *s2*.
s1 != *s2*	String *s1* is not the same as string *s2*.
n1 -eq *n2*	Integer *n1* is equal to integer *n2*.
n1 -ge *n2*	Integer *n1* is greater than or equal to integer *n2*.
n1 -gt *n2*	Integer *n1* is greater than integer *n2*.
n1 -le *n2*	Integer *n1* is less than integer *n2*.
n1 -lt *n2*	Integer *n1* is less than or equal to integer *n2*.
n1 -ne *n2*	Integer *n1* is not equal to integer *n2*.
!	The `not` operator, which reverses the value of the following condition.

Form	Function
`-a`	The `and` operator, which joins two conditions. Both conditions must be true for the overall result to be true.
`-o`	The `or` operator, which joins two conditions. If either condition is true, the overall result is true.
`\( ... \)`	You can group expressions within the **test** command by enclosing them within `\(` and `\)`.

To see the **test** command in action, consider the following script:

```
test -d $1
echo $?
```

This script tests whether its first argument specifies a directory and displays the resulting exit status, a zero or a non-zero value that reflects the result of the test.

Suppose the script were stored in the file `tester`, which permitted read access. Executing the script might yield results similar to the following:

```
$ ./tester /
0
$ ./tester /missing
1
```

These results indicate that the `/` directory exists and that the `/missing` directory does not exist.

The if command

The **test** command is not of much use by itself, but combined with commands such as the **if** command, it is useful indeed. The **if** command has the following form:

```
if command
then
  commands
```

```
else
 commands
fi
```

Usually the command that immediately follows the word **if** is a **test** command. However, this need not be so. The **if** command merely executes the specified command and tests its exit status. If the exit status is 0, the first set of commands is executed; otherwise the second set of commands is executed. An abbreviated form of the **if** command does nothing if the specified condition is false:

```
if command
then
  commands
fi
```

When you type an **if** command, it occupies several lines; nevertheless it's considered a single command. To underscore this, the shell provides a special prompt (called the *secondary prompt*) after you enter each line. Often, scripts are entered by using a text editor; when you enter a script using a text editor you don't see the secondary prompt, or any other shell prompt for that matter.

As an example, suppose you want to delete a file `file1` if it's older than another file `file2`. The following command would accomplish the desired result:

```
if test file1 -ot file2
then
  rm file1
fi
```

You could incorporate this command in a script that accepts arguments specifying the filenames:

```
if test $1 -ot $2
then
  rm $1
  echo Deleted the old file.
fi
```

If you name the script *riddance* and invoke it as follows:

```
riddance thursday wednesday
```

the script will delete the file `thursday` if that file is older than the file `wednesday`.

The case command

The **case** command provides a more sophisticated form of conditional processing:

```
case value in
  pattern1) commands;;
  pattern2) commands ;;
  ...
esac
```

The **case** command attempts to match the specified value against a series of patterns. The commands associated with the first matching pattern, if any, are executed. Patterns are built using characters and metacharacters, such as those used to specify command arguments. As an example, here's a **case** command that interprets the value of the first argument of its script:

```
case $1 in
  -r) echo Force deletion without confirmation ;;
  -i) echo Confirm before deleting ;;
   *) echo Unknown argument ;;
esac
```

The command echoes a different line of text, depending on the value of the script's first argument. As done here, it's good practice to include a final pattern that matches any value.

The while command

The **while** command lets you execute a series of commands iteratively (that is, repeatedly) so long as a condition tests true:

```
while command
do
 commands
done
```

Here's a script that uses a **while** command to print its arguments on successive lines:

```
echo $1
while shift 2> /dev/null
do
  echo $1
done
```

The commands that comprise the **do** part of a **while** (or another loop command) can include **if** commands, **case** commands, and even other **while** commands. However, scripts rapidly become difficult to understand when this occurs often. You should include conditional commands within other conditional commands only with due consideration for the clarity of the result. Include a comment command (#) to clarify difficult constructs.

The until command

The **until** command lets you execute a series of commands iteratively (that is, repeatedly) so long as a condition tests false:

```
until command
do
 commands
done
```

Here's a script that uses an **until** command to print its arguments on successive lines, until it encounters an argument that has the value *red*:

```
until test $1 = red
do
  echo $1
  shift
done
```

For example, if the script were named *stopandgo* and stored in the current working directory, the command:

```
./stopandgo green yellow red blue
```

would print the lines:

```
green
yellow
```

The for command

The **for** command iterates over the elements of a specified list:

```
for variable in list
do
  commands
done
```

Within the commands, you can reference the current element of the list by means of the shell variable `$ variable`, where `variable` is the name specified following the **for**. The list typically takes the form of a series of arguments, which can incorporate metacharacters. For example, the following **for** command:

```
for i in 2 4 6 8
do
  echo $i
done
```

prints the numbers 2, 4, 6, and 8 on successive lines.

A special form of the **for** command iterates over the arguments of a script:

```
for variable
do
  commands
done
```

For example, the following script prints its arguments on successive lines:

```
for i
do
  echo $i
done
```

The break and continue commands

The **break** and **continue** commands are simple commands that take no arguments. When the shell encounters a **break** command, it immediately exits the body of the enclosing loop (**while**, **until**, or **for**) command. When the shell encounters a **continue** command, it immediately discontinues the current iteration of the loop. If the loop condition permits, other iterations may occur; otherwise the loop is exited.

Periscope: A Useful Networking Script

Suppose you have a free email account such as that provided by Yahoo! You're traveling and find yourself in a remote location with Web access. However, you're unable to access files on your home machine or check email that has arrived there. This is a common circumstance, especially if your business requires that you travel.

If your home computer runs Microsoft Windows, you're pretty much out of luck. You'll find it extraordinarily difficult to access your home computer from afar. However, if your home computer runs Linux, gaining access is practically a piece of cake.

In order to show the power of shell scripts, this subsection explains a more complex shell script, **periscope**. At an appointed time each day, **periscope** causes your computer (which you must leave powered on) to establish a PPP connection to your ISP, which is maintained for about one hour. This provides you enough time to connect to an ISP from your hotel room or other remote location and then connect via the Internet with your home Linux system, avoiding long distance charges. Once connected, you have about an hour to view or download mail and perform other work. Then, **periscope** breaks its PPP connection, which it will re-establish at the appointed time the next day.

Example 13-1 shows the **periscope** script file, which is considerably larger than any script you've so far encountered in this chapter. Therefore, we'll disassemble the

script, explaining it line by line. As you'll see, each line is fairly simple in itself and the lines work together in a straightforward fashion.

Example 13-1. Periscope

```
PATH=${PATH}:/usr/local/bin
route del default
wvdial &
sleep 1m
ifconfig | mail userid@mail.com
sleep 1h
killall wvdial
sleep 2s
killall -9 wvdial
killall pppd
sleep 2s
killall -9 pppd
echo "/root/periscope" | at 10:00
```

The first line of the script augments the search path for the script to include `/usr/local/bin`, the directory that contains the **wvdial** external command. Some versions of the startup scripts may not include this path in the search path, so explicitly placing it there avoids a possible problem.

```
PATH=${PATH}:/usr/local/bin
```

The next line is perhaps the most complex line of the entire script:

```
route del default
```

The **route** command is normally issued by the system administrator. You've probably never issued the command yourself, because a network configuration program has issued it on your behalf. The effect of the command is to delete the default network route, if any. The default route is the one along which TCP/IP sends packets when it knows no specific route to their specified destination. It's necessary to delete the default route because the **wvdial** program, which the script uses to establish its PPP connection, will not override an existing default route.

```
wvdial &
```

The next line launches the **wvdial** program. As specified by the ampersand (&), the program runs in the background, so the script continues executing while **wvdial** starts up and runs. The next line pauses the script for one minute, giving **wvdial** time to establish the PPP connection:

```
sleep 1m
```

The next line runs the **ifconfig** command and mails its output to the specified user (you must replace *userid@mail.com* with your own email address, which you can access remotely):

```
ifconfig | mail userid@mail.com
```

The **ifconfig** command produces output that looks something like this:

```
ppp0   Link encap:Point-Point Protocol
       inet addr:10.144.153.105  P-t-P:10.144.153.52 Mask:255.255.255.0
       UP POINTOPOINT RUNNING  MTU:552  Metric:1
       RX packets:0 errors:0 dropped:0 overruns:0
       TX packets:0 errors:0 dropped:0 overruns:0
```

You'll probably see other sections describing your Ethernet interface (*eth0*) and a loopback device (*lo*). The `inet addr` given in the command output (10.144.153.105) is the IP address of your computer. By mailing the output to yourself, you provide a simple way to discover your computer's IP address, which is likely to be different each time it connects to your ISP.

The next line causes the script to pause for an interval of one hour:

```
sleep 1h
```

You can easily change this interval to something more appropriate to your own needs.

The connection interval now having elapsed, the next line terminates all executing instances of the `wvdial` program:

```
killall wvdial
```

TIP

Appendix E, briefly describes the **killall** command and other possibly unfamiliar commands employed in this script.

The script then pauses for two seconds, to ensure that **wvdial** has completely terminated:

```
sleep 2s
```

Under some circumstances, a program will ignore a termination request. The next line deals with this possibility by sending a special code that compels a reluctant program to terminate without further delay:

```
killall -9 wvdial
```

Behind the scenes, **wvdial** launches a program known as **pppd**, which actually establishes and manages the PPP connection. Another **killall** command is designed to terminate **pppd** if **wvdial** has failed to do so:

```
killall pppd
```

Again, the script pauses for a few seconds:

```
sleep 2s
```

And, again the script uses the *-9* option to specify that any remaining instances of **pppd** should terminate immediately:

```
killall -9 pppd
```

Finally, the script uses the **at** command to schedule itself for execution at 10:00 tomorrow:

```
echo "/root/periscope" | at 10:00
```

The **at** command reads one or more commands from its standard input and executes them at the time specified as an argument.

To try the script for yourself, you must have installed the **wvdial** program, as explained in **Chapter 11**. Place the script in the file `/root/periscope`. Of course, you'll probably want to customize the script to specify an appointment time and duration of your own choosing. To start **periscope**, log in as `root` and issue the command:

```
(echo "/root/periscope" | at 10:00)&
```

When 10:00 (or such other time as you specified) comes around, your Linux system should obediently dial your ISP and maintain the connection for the specified interval of time.

Using periscope

At the appointed time, fire up your computer and access your email account. You should find a mail message that contains the **ifconfig** output giving your computer's current IP address. Now you can use **telnet** or an **ssh** client—your choice corresponds to the server you're running on your Linux system—to contact your computer and work for the remainder of the specified connection time. At the end of the connection time, your Linux system will sever its PPP connection and begin counting down until it's again time to connect.

Continuing Onward

Because it's quite a simple script, **periscope** doesn't do full justice to the capabilities of Linux. For example, suppose you want to establish connections at varying times or on varying days of the week. Or, suppose you want to schedule the next connection each time you log in.

Linux is able to answer such challenges in a variety of ways. For example, the **cron** program, though more complicated to use than the **at** command, provides the ability to specify program launch times very flexibly. For example, **cron** can let you establish a connection at 10:00 in the morning of the third Friday of each month.

You can learn more about Linux from **Appendix E**, which summarizes many useful Linux commands that you can use and include in shell scripts. A good way to continue learning about Linux is to peruse **Appendix E** and try each of the commands described there. Read their man pages and learn more about them. Ask yourself how the commands might be used in scripts that would facilitate your use of Linux.

If you truly catch the Linux bug, as many have, you'll want to peruse other Linux works, such as:

- *Running Linux*, 3rd edition, by Matt Welsh, Kalle Dalheimer, and Lar Kaufman (Sebastopol, CA: O'Reilly & Associates, 1999).
- *Learning the bash Shell*, 2nd edition, by Cameron Newham and BIll Rosenblatt (Sebastopol, CA: O'Reilly & Associates, 1998).
- *Linux Network Administrator's Guide*, by Olaf Kirch (Sebastopol, CA: O'Reilly & Associates, 1995).
- *Learning the vi Editor*, 6th edition, by Linda Lamb and Arnold Robbins (Sebastopol, CA: O'Reilly & Associates, 1998).

You'll also find a wealth of useful information on the Web sites described in **Chapter 1** and in periodicals such as *Linux Journal* and *Linux Magazine*.

However, don't merely read about Linux; work with it. Write, test, and debug your own scripts. Share scripts you've written with others and study scripts written by others. Above all, read, communicate, and share what you've learned and what you want to learn. These activities are the foundation of the Linux culture and they are means whereby Linux users—and Linux itself—grow and develop.

Previous chapter

< Using the Shell

Next chapter

A. Linux Directory Tree >