24 – Writing Your First Script

In the preceding chapters, we have assembled an arsenal of command line tools. While these tools can solve many kinds of computing problems, we are still limited to manually using them one by one on the command line. Wouldn't it be great if we could get the shell to do more of the work? We can. By joining our tools together into programs of our own design, the shell can carry out complex sequences of tasks all by itself. We can enable it to do this by writing *shell scripts*.

What are Shell Scripts?

In the simplest terms, a shell script is a file containing a series of commands. The shell reads this file and carries out the commands as though they have been entered directly on the command line.

The shell is somewhat unique, in that it is both a powerful command line interface to the system and a scripting language interpreter. As we will see, most of the things that can be done on the command line can be done in scripts, and most of the things that can be done in scripts can be done on the command line.

We have covered many shell features, but we have focused on those features most often used directly on the command line. The shell also provides a set of features usually (but not always) used when writing programs.

How to Write a Shell Script

To successfully create and run a shell script, we need to do three things.

1. **Write a script.** Shell scripts are ordinary text files. So, we need a text editor to write them. The best text editors will provide *syntax highlighting*, allowing us to see a color-coded view of the elements of the script. Syntax highlighting will help us spot certain kinds of common errors. `vim`, `gedit`, `kate`, and many other editors are good candidates for writing scripts.
2. **Make the script executable.** The system is rather fussy about not letting any old text file be treated as a program, and for good reason! We need to set the script file's permissions to allow execution.

3. **Put the script somewhere the shell can find it.** The shell automatically searches certain directories for executable files when no explicit pathname is specified. For maximum convenience, we will place our scripts in these directories.

Script File Format

In keeping with programming tradition, we'll create a "Hello World" program to demonstrate an extremely simple script. Let's fire up our text editors and enter the following script:

```
#!/bin/bash

# This is our first script.

echo 'Hello World!'
```

The last line of our script is pretty familiar; it's just an `echo` command with a string argument. The second line is also familiar. It looks like a comment that we have seen used in many of the configuration files we have examined and edited. One thing about comments in shell scripts is that they may also appear at the ends of lines, provided they are preceded with at least one whitespace character, like so:

```
echo 'Hello World!' # This is a comment too
```

Everything from the # symbol onward on the line is ignored.

Like many things, this works on the command line, too:

```
[me@linuxbox ~]$ echo 'Hello World!' # This is a comment too
Hello World!
```

Though comments are of little use on the command line, they will work.

The first line of our script is a little mysterious. It looks as if it should be a comment since it starts with #, but it looks too purposeful to be just that. The `#!` character sequence is, in fact, a special construct called a *shebang*. The shebang is used to tell the kernel the name of the interpreter that should be used to execute the script that follows. Every shell script should include this as its first line.

Let's save our script file as `hello_world`.

Executable Permissions

The next thing we have to do is make our script executable. This is easily done using `chmod`.

```
[me@linuxbox ~]$ ls -l hello_world
-rw-r--r-- 1 me      me      63 2009-03-07 10:10 hello_world
[me@linuxbox ~]$ chmod 755 hello_world
[me@linuxbox ~]$ ls -l hello_world
-rwxr-xr-x 1 me      me      63 2009-03-07 10:10 hello_world
```

There are two common permission settings for scripts: 755 for scripts that everyone can execute, and 700 for scripts that only the owner can execute. Note that scripts must be readable to be executed.

Script File Location

With the permissions set, we can now execute our script:

```
[me@linuxbox ~]$ ./hello_world
Hello World!
```

For the script to run, we must precede the script name with an explicit path. If we don't, we get this:

```
[me@linuxbox ~]$ hello_world
bash: hello_world: command not found
```

Why is this? What makes our script different from other programs? As it turns out, nothing. Our script is fine. Its location is the problem. In Chapter 11, we discussed the `PATH` environment variable and its effect on how the system searches for executable programs. To recap, the system searches a list of directories each time it needs to find an executable program, if no explicit path is specified. This is how the system knows to execute `/bin/ls` when we type `ls` at the command line. The `/bin` directory is one of the directories that the system automatically searches. The list of directories is held within an environment variable named `PATH`. The `PATH` variable contains a colon-separated list of directories to be searched. We can view the contents of `PATH`.

```
[me@linuxbox ~]$ echo $PATH
/home/me/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:
/bin:/usr/games
```

Here we see our list of directories. If our script were located in any of the directories in the list, our problem would be solved. Notice the first directory in the list, `/home/me/bin`. Most Linux distributions configure the `PATH` variable to contain a `bin` directory in the user's home directory to allow users to execute their own programs. So, if we create the `bin` directory and place our script within it, it should start to work like other programs.

```
[me@linuxbox ~]$ mkdir bin
[me@linuxbox ~]$ mv hello_world bin
[me@linuxbox ~]$ hello_world
Hello World!
```

And so it does.

If the `PATH` variable does not contain the directory, we can easily add it by including this line in our `.bashrc` file:

```
export PATH=~/bin:"$PATH"
```

After this change is made, it will take effect in each new terminal session. To apply the change to the current terminal session, we must have the shell re-read the `.bashrc` file. This can be done by "sourcing" it.

```
[me@linuxbox ~]$ . .bashrc
```

The dot (`.`) command is a synonym for the `source` command, a shell builtin that reads a specified file of shell commands and treats it like input from the keyboard.

Note: Ubuntu (and most other Debian-based distributions) automatically adds the `~/bin` directory to the `PATH` variable if the `~/bin` directory exists when the user's `.bashrc` file is executed. So, on Ubuntu systems, if we create the `~/bin` directory and then log out and log in again, everything works.

Good Locations for Scripts

The `~/bin` directory is a good place to put scripts intended for personal use. If we write a script that everyone on a system is allowed to use, the traditional location is `/usr/local/bin`. Scripts intended for use by the system administrator are often located in `/usr/local/sbin`. In most cases, locally supplied software, whether scripts or compiled programs, should be placed in the `/usr/local` hierarchy and not in `/bin` or `/usr/bin`. These directories are specified by the Linux Filesystem Hierarchy Standard to contain only files supplied and maintained by the Linux distributor.

More Formatting Tricks

One of the key goals of serious script writing is ease of *maintenance*, that is, the ease with which a script may be modified by its author or others to adapt it to changing needs. Making a script easy to read and understand is one way to facilitate easy maintenance.

Long Option Names

Many of the commands we have studied feature both short and long option names. For instance, the `ls` command has many options that can be expressed in either short or long form. For example, the following:

```
[me@linuxbox ~]$ ls -ad
```

is equivalent to this:

```
[me@linuxbox ~]$ ls --all --directory
```

In the interests of reduced typing, short options are preferred when entering options on the command line, but when writing scripts, long options can provide improved readability.

Indentation and Line-Continuation

When employing long commands, readability can be enhanced by spreading the command over several lines. In Chapter 17, we looked at a particularly long example of the `find` command.

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec
```

```
chmod 0600 ‘{}’ ‘;’ \) -or \( -type d -not -perm 0700 -exec chmod
0700 ‘{}’ ‘;’ \)
```

Obviously, this command is a little hard to figure out at first glance. In a script, this command might be easier to understand if written this way:

```
find playground \
    \( \
         -type f \
         -not -perm 0600 \
         -exec chmod 0600 ‘{}’ ‘;’ \
    \) \
    -or \
    \( \
         -type d \
         -not -perm 0700 \
         -exec chmod 0700 ‘{}’ ‘;’ \
    \)
```

By using line continuations (backslash-linefeed sequences) and indentation, the logic of this complex command is more clearly described to the reader. This technique works on the command line, too, though it is seldom used, as it is awkward to type and edit. One difference between a script and a command line is that the script may employ tab characters to achieve indentation, whereas the command line cannot since tabs are used to activate completion.

Configuring `vim` For Script Writing

The `vim` text editor has many, many configuration settings. There are several common options that can facilitate script writing.

The following turns on syntax highlighting:

`:syntax on`

With this setting, different elements of shell syntax will be displayed in different colors when viewing a script. This is helpful for identifying certain kinds of programming errors. It looks cool, too. Note that for this feature to work, you must have a complete version of `vim` installed, and the file you are editing must have a

shebang indicating the file is a shell script. If you have difficulty with the previous command, try **`:set syntax=sh`** instead.

The following turns on the option to highlight search results.

```
:set hlsearch
```

Say we search for the word `echo`. With this option on, each instance of the word will be highlighted.

The following sets the number of columns occupied by a tab character.:

```
:set tabstop=4
```

The default is eight columns. Setting the value to 4 (which is a common practice) allows long lines to fit more easily on the screen.

The following turns on the "auto indent" feature:

```
:set autoindent
```

This causes `vim` to indent a new line the same amount as the line just typed. This speeds up typing on many kinds of programming constructs. To stop indentation, press `Ctrl-d`.

These changes can be made permanent by adding these commands (without the leading colon characters) to your `~/.vimrc` file.

Summing Up

In this first chapter of scripting, we looked at how scripts are written and made to easily execute on our system. We also saw how we can use various formatting techniques to improve the readability (and thus the maintainability) of our scripts. In future chapters, ease of maintenance will come up again and again as a central principle in good script writing.

Further Reading

- For "Hello World" programs and examples in various programming languages, see:
  http://en.wikipedia.org/wiki/Hello_world
- This Wikipedia article talks more about the shebang mechanism:
  http://en.wikipedia.org/wiki/Shebang_(Unix)