Writing Linux scripts for bash

The default in some distros for the script language is **bash**, and that is what will be used here. However, there are a number of script languages available, and almost all of them will allow use of the standard shell script language; each having some number of extensions peculiar to them. In practice, this is not a problem, as use of these extra features is rare, certainly among the casual use.

There is a book that describes the scripting language, and I would recommend reference [1] as the one to start with.

This paper will mention several things that you may encounter when copying text from web pages to perform various operations, but which you may not use yourself. It is, however, useful to be aware of them so that you can understand what you are copying.

Basic syntax structure

The basic unit of a script is the line. Each line of text is treated independently, except for special cases that will be introduced later.

Every line that begins with a **#** sign is treated as a comment and will not perform any action.

The very first line must begin with the two characters **#!** for Unix system reasons. This first line is what defines the program that will be used to interpret the script. Thus, typically, the first line of a bash script will be exactly this:

```
#!/bin/bash
```

Various options can be added to this first line, although they are not generally useful for our purposes.

The bash program will read each line in turn and then split it into commands separated by semicolons. These commands will then be executed in turn.

NOTE: In Unix, the case of a name of anything is important, upper and lower cases are different.

Execution of a command

Each command that is executed, in standard Unix style, is intended to receive input on its *standard input*, and produce output on its *standard output*. It may also produce error messages on another output, but that will not be considered here at the moment. What a command does, is modified by the options provided to it in the command syntax when you write the command.

Many commands will take one or more filenames as options to define the files on which it will operate. However, if no filenames are provided, then often a command will assume that it is to operate on the *standard input*, where that makes sense.

If you are using *bash*, or another shell, from a terminal, or terminal window, then the *standard input* is what you type, and the *standard output* is displayed on the screen.

If you are executing a command and it is taking too long, or is not what you wanted to do, then using the key combination CTRL+C will interrupt and stop the command.

If a command is reading its input from the terminal, or you terminal window, then the key combination CTRL+D will indicate the end of the input stream.

Each command executes in an environment. One of the most important parts of the environment is the notion of the *working directory*. This is the directory in which a filename will be found, if no other indication is used to say where it is.

Various commands

Commands come in various types. The lists here are examples of some of the more useful ones, although there are many others that you might wish to use. The only practical way to discover many of them is simply to ask other people whether a command exists to do a job. It is more important to know that a way exists, than to remember exactly what it is, as that can be looked up.

Listing information

Here are some ways to list information:

man will give you information about a command name given as a parameter to the command. The output will fill a screen and wait for you to hit the space bar before showing the next screen. Using the carriage return key will just allow it to move down one line. You can stop the display and return to the command line by hitting the **q** (for quit) key.

ls (note that is is lower case L) will list the names of files in a directory, and optionally some information about each file.

hostname tells you the name of the machine you are on.

uname will tell you more about the system you are on, including, optionally, the kernel release number, machine hardware, and other very simple information.

date will tell you the date and time in various formats.

whoami will give you the name of the person logged on and executing the command.

file will tell you something about the contents of a file. For example, if the output of the file command contains the word text then it can be treated as a text file for editing purposes, etc.

which will help you find where a command resides. This can be useful if you ever accidentally re-use the name of a system command for one of your own.

ps can tell you what processes are presently executing, and under which user names.

mount will provide information of what file systems are accessible, and under what names.

df will tell you how much space is used by file systems or directories.

pwd says what the current working directory is.

find will discover all files with certain characteristics within a directory structure. It is useful when wanting to do the same thing to all pictures of type *.jpg*, say.

Simple operations

These commands will provide elementary means of manipulating directories and files as a whole.

cp, **mv**, and **rm** will copy, move and remove a file, or optionally a directory.

mkdir makes a new directory.

ln (small L) makes a new link, giving a different path to an existing file or directory.

Rudimentary text editing commands

Traditionally, Unix has always operated by means of humanly readable text files. So it is no surprise that there are a lot of small commands that will process a text file in particular ways. Among these are some of the most useful commands of all when doing simple processing tasks. Many of them can be used to manipulate single textual strings as well as whole files.

echo will take its parameters from the command line and put them out onto the *standard output* separating them with a blank.

cat will copy each of its parameters in turn to the same *standard output* stream, so it can be used to concatenate (hence the abbreviated name) several files into one.

more, or **less**, will copy a file to the terminal, one screenful at a time. Using the **space bar**, **carriage return**, the **b key** (back a page) or the **q key** (quit) will enable you to control how it continues.

printf will format the parameters onto standard output. Note the name of this command means print formatted. It must not be confused with the *print* command that does something very different and will almost certainly not be what you want.

wc produces counts of lines, words and letters in a file.

head and **tail** will limit their output to the first, or last respectively, few lines of a file.

sort will sort the contents of a file and put it out onto *standard output*. It can be made to sort alphabetically, numerically, ascending, descending, and use any number of fields within each line.

tr can be made to translate some characters into others in a simple and regular way. For example, it can be made to reduce all text to lower case by translating the upper case letters to lower case ones.

cut can remove parts of each line from a file. For example, from a file containing names, addresses and phone numbers, it could remove just the names. But for this to work, the file must have a simple, regular structure.

grep will isolate lines from a file that contain a certain pattern, and drop the rest.

basename and **dirname** can be used to divide the name of a full path to a file into its constituent parts. This is very useful when manipulating files in one directory to make equivalents elsewhere. The first extract the name of the file within its directory, and the latter extracts the name of the directory. basename also has some useful options to derive suffixes of file names as well.

Simple editing

There are two programs you may come across that allow editing of text files in scripts.

sed is a simple editor that operates line by line and do a wide range of things if you really want to. It is commonly used do replacement of one string by another when upgrading files, etc.

awk is a more complex editor that has its small language. It will allow you to extract parts of each line according to several different criteria and perform various operations on them. I mention it here more to tell you it might be used for in case you come across it in any script you take from the web.

Reference [1] contains some basic information on these editors, but much more can be found in reference [2].

Pipes

The normal execution of a script goes from one line to the next sequentially. However, one fundamental method of combining processes is to connect the output from one command to the input of the next through what is called a *pipe*. A pipe is represented by a vertical bar, and is a very common structure.

For example, if you have a number of text files, and you want to find the total number of words in all of them added together, then this line would do the job:

```
cat *.txt | wc -w
```

Command line substitution

The above example also shows the use of what is known as *globbing*. Essentially, you do not have to list all the file names which end in with the characters *txt* but you can use the bash processor to identify them all and replace the string **.txt* with a list of the names that have zero or more characters (that is what the asterisk means) followed by the string *.txt* and when these are put into the cat command, all the files will be copied out to the *standard output*.

There two ways to avoid the expansion of an asterisk. The first is to place a backslash before the asterisk. A backslash will tell bash to treat the next character literally and not expand it in any way. This process is given the name of *escaping* a character. To *escape* a whole string when necessary, then the string can be enclosed in single quotes.

Compare the output of these commands:

```
echo **
echo '**'
echo \*
echo '\*'
echo \\\*
```

Any character can be escaped, which includes blanks, semicolons, even the backslash character. However, you cannot escape a single quote with single quotes, but must always precede it with a backslash instead.

A special case of a backslash is when it is the very last character on a line. Because it escapes the following newline character, it allows the command string to flow past it onto the next line. In other words it is the way to continue a command string from one line to the next.

Variables

You can define a variable, and give it a value, and then use it later in the script. One use of these is so that you can make it easier to maintain a script. If you set the value of a variable to, sy, the directory containing all your documents, then if the directory should be moved or change its name, then you need only change the beginning of the script to make the change apply to all uses of the variable.

To define and set a value, you use an equals sign, and no blanks surrounding it:

```
docs=/home/andy/Documents
```

To use the variable, then you put a dollar sign in front of it:

```
cd $docs
```

which will make the value of the variable *docs* the current working directory.

If you have written a script which takes parameters, the you can reference the parameters using the special variable names **$1**, **$2**, **$3**, etc. **$0** will give you the name of the script as it was executed.

Capturing the output of a command in a variable

A common thing to want to do is to capture the output from some command and put it into a variable for use later. For example, the name of host on which you are running is found by the output from the command **hostname**. To retain that name for later use in a script, there are two recognised methods. The first is traditional and will be available on all useful shells, but has the disadvantage of being difficult to read sometimes, and cannot be used to insert the output from one command directly into the parameters to another. However, the second one can be used so and is to be generally recommended.

```
h=`hostname`
h=$(hostname)
```

This technique is very useful in conjunction with pipes, so that some processing can be done on the output from a command before being saved for later use. For example, this will capture the number of uppercase characters in a text file passed in as the first parameter to a command:

```
s=$(cat $1 | tr -c -d '[:upper:]' | wc -c)
```

Loops

Sometimes you wish to do the same thing for every element in a list, or for a certain number of times, and so. This is accomplished by means of a loop construct, and there are several means of doing it. The simplest is by just enumerating the list:

```
for f in *; do
  echo size of "$f" is $(ls -ld $f | awk '{print $5}')
done
```

Other types of loops need the ability to test for some condition, and that's the next topic.

Tests

The loop in the previous section will give some number when it does not really make any sense, as, for example, when the file is really a link or a directory.

All commands return an exit code, and this can be tested for. There is a special variable **$?** which contains the exit code from the previous command. An exit code of zero is considered to be true, and all others false. There is a particularly useful command **test** which has the ability to perform various tests and return a true or false indicator in its return code. In all Unix systems, the **test** command has an alternative name, **[**, which helps write conditions as it looks like a set of brackets. So if we want to restrict the listing of sizes to only genuine files, then we can write the central part of the loop like this:

```
if [ -f $f ]; then
  echo size of "$f" is $(ls -ld $f | awk '{print $5}')
fi
```

The **if** construct can be extended to add other clauses. For example, we can test for file, or link, or else it is not interesting:

```
if [ -L $f ]; then
  echo link "$f" points to $(ls -ld $f | awk -F'>' '{print $2}')
elif [ -f $f ]; then
  echo size of "$f" is $(ls -ld $f | awk '{print $5}')
else
  echo $f is not interesting
fi
```

Other types of tests

The same tests can be conducted in a **while** loop to continue doing something until a condition is no longer met. For example, processing all the parameters, one by one, can be done this way:

```
while [ $# -gt 0 ]; do
  p="$1"
  echo Parameter found: "$p"
  shift
done
```

The **shift** command removes the first parameter, and leaves the rest for later processing. So this loop extracts each parameter in turn and echoes the value of it to the terminal.

Arithmetic

Simple integer arithmetic can also be performed directly by the shell. To indicate that arithmetic values are involved and are to be manipulated as such, double brackets are used, and then within those brackets, the leading $ symbols are not needed before variable names, although it is not an error if they are used when requesting a value rather than assigning one. One way of listing each parameter with its location on the command line would be:

```
i = 1
while [ $# -gt 0 ]; do
  p="$1"
  echo Parameter $i is: "$p"
  shift
  ((i = i+1))
done
```

If you want to operate on enormous numbers, or real values rather than integers, then the command **bc** is invaluable. Here is the way to get the value of π into a variable in the shell:

```
pi=$(echo 'scale = 15; 4*a(1)' | bc -l)
```

References

[1] *Classic Shell Scripting*, Arnold Robbins and Nelson Beebe, O'Reilly Media, Sebastopol, CA, USA, 2005, ISBN 0-596-00595-4

[2] *sed & awk*, Dale Dougherty, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1990, ISBN 0-937175-59-5