Linux scripting basics

What is a script?

Andy Pepperdine

22 January 2009

This document is the adjunct to a talk I gave on 22 Jan 2009 to the FOSS group of the U3A in Bath. I had noticed there was some misunderstanding of what a script was, and how it relates to the general operation of the system and the user command line interface (CLI). I assume as little knowledge as I can sensibly do so, and only attempt to give the reader enough information to understand what is going on, where to look for help and how to ask questions to find out more. The subject is much too large for a full treatment in one hour, but I hope that this is useful.

The examples are given using Ubuntu 8.10 (Intrepid Ibex).

A little knowledge is a dangerous thing – only use what you understand. When in doubt ask someone for advice, the shell is a powerful weapon.

1. Introduction and concepts

In Linux, as in all Unix-based systems, every operation that the computer undertakes is done by a ***process***. These processes have certain contexts and abilities which will be explained as we go through the descriptions. Each process executes a ***program***.

One important process is started whenever the user starts a ***terminal***. In Ubuntu, this is found under the *Applications → Accessories → Terminal* menu item, and opens a window showing a ***prompt***.

In this document, I shall use the convention that `this monospace font` is used for all terminal displays, and it will be **`in bold`** for text that the user types, and `regular` for what the system shows. So the initial window has its first line like this:

```
andrew@notebook:~$
```

Your text will be different, but the prompt will always be shown to indicate that the terminal is waiting for input from you.

The program running in the terminal process will then receive your typed ***commands*** and execute them for you.

2. Commands – Options – Parameters – Input – Output

The line that you type in a terminal window is interpreted as a command, consisting of a ***command name***, ***options***, and ***parameters*** (or ***arguments***) The command name identifies what the program is that is to be executed. The options can modify its behaviour, and the parameters say what the data is that it will operate on. Examples will make this clear as time goes on.

In addition, there are three special files that all processes will see, and that you can manipulate. These are one input file, and two output files.

The input file is known as ***standard input*** (or stdin). Most commands will read stdin if no other file is specified. The usefulness of this will become apparent as examples of how to combine commands to do useful combinations of jobs become clear.

The principle output file is known as ***standard output*** (or stdout), and most commands will send their output to this file if no other option is given.

The other output is known as ***standard error*** (or stderr) and contains any error messages that are produced, so they can be separated when necessary from the normal output.

When writing commands in the terminal window, all these files are set to operate from the terminal window display. So stdin will read whatever you subsequently type, and stdout and stderr will both be displayed after the command line.

When the program has finished its work, the prompt will appear once more to ask for another command.

Finally, each command sends back a ***return code*** which is either zero (0) or non-zero (anything else). A zero code indicates the command terminated "normally", otherwise, it may return any non-zero value, typically one (1). This is invaluable when testing for certain conditions in a shell script.

3. Shells and logins

For Unix-like systems, each user is given access to a ***shell***. This is the program which is executing behind your terminal window, receiving your typed commands, and ***invoking*** them to perform some requested operation. In Ubuntu the shell program is the one named **`bash`**.

> Historical note: A very popular shell in the early days of Unix was one written by Stephen Bourne of AT&T to replace the original one by Ken Thompson. It has become the standard shell for the root user, and it is most unlikely not to be present on any Unix-like system today. It's name is ***sh***, a typical short cryptic name for a Unix command. When the Free Software movement got under way, another shell was written to the same specification so it was compatible with all the existing scripts that were around by then. Since it was a re-write, it was known as the Bourne-again shell, hence bash.

By default, when a process executes, there is a directory known as the ***working directory*** and is set when the shell starts to your home directory.

To discover your current working directory, there is a command **`pwd`** which will print the working directory name in full. Try it and see.

```
andrew@notebook:~$ pwd
/home/andrew
andrew@notebook:~$
```

As you see, the name is shown with a full path, starting with a **`/`** . It can also be referred to by the symbol **`~`**, which is where the **`~`** in the prompt comes from.

The working directory is where all file names are relative to. So a reference in the above example to `example.txt` would be to the file found at `/home/andrew/example.txt`, or `~/example.txt`.

When any command is executed from the terminal, its working directory is set to the one that is current in the terminal window.

4. Sample commands – syntax

The command line starts with the name of the command to be executed. Normally, this is then followed by the options, and finally the parameters (or arguments), but some commands can mix them for their special uses. Between the command name and the options is ***white space***, which is a space, tab, or other non-printing formatting character, except the end of line. Between each option,

and argument is also white space. Command names can be any non-space combination of characters, and will correspond to the name of a file, which the operating system will then find and invoke, passing to it all the options and arguments. This file must be given the executable permission before the operating system will attempt to open it.

To find the file, the system has a list of ***search path***s which are the pathnames of directories in which it will look, in turn one by one, until it finds the command name. However, you can override this by giving the fully-qualified path name (one containing the `/` character) in which case it will not search for it but go directly to it. How you can discover what the search path is, I'll show later.

A useful command from the terminal is **`ls`** which lists the names of files and subdirectories in a directory. In its most basic form it is just:

```
andrew@notebook:~$ ls
Accounts  Desktop    Examples  Pictures  Templates
bin       Dev        Import    Public    Videos
Data      Documents  Music     Settings  WebSite
andrew@notebook:~$
```

Without any options or parameters it lists all the entries in the current working directory.

Options in Linux always begin with the **`-`** (hyphen, or minus character). Most options to commands are just single characters following the minus, in which case, they can be (but do not have to be) all placed without spaces after a single minus. Look at the following:

```
andrew@notebook:~$ ls -l
total 60
drwxr-xr-x  2 andrew andrew 4096 2008-12-25 19:25 Accounts
drwxr-xr-x  3 andrew andrew 4096 2008-12-25 19:29 bin
drwxrwxr-x  5 andrew andrew 4096 2008-12-13 19:45 Data
drwxrwxr-x  6 andrew andrew 4096 2008-05-19 13:55 WebSite
andrew@notebook:~$
```

[To save space, I've removed some of the output; the principles are unaffected.]

The `l` (ell, for *long*) option makes `ls` output a longer format giving information about the permissions, sizes, ownership, and time last modified.

To list them in time order (the most recent first), give it the `t` option as well:

```
andrew@notebook:~$ ls -lt
total 60
drwxr-xr-x  3 andrew andrew 4096 2008-12-25 19:29 bin
drwxr-xr-x  2 andrew andrew 4096 2008-12-25 19:25 Accounts
drwxrwxr-x  5 andrew andrew 4096 2008-12-13 19:45 Data
drwxrwxr-x  6 andrew andrew 4096 2008-05-19 13:55 WebSite
andrew@notebook:~$
```

But this could equally well have been written:

```
andrew@notebook:~$ ls -l -t
```

and it would have the same effect. Note that if more than one option is given with a leading minus sign, then they must be separated by white space.

The command can also take one or more parameters, which are path names, either files or

directories. Names of files just list the file name in the format the options define. But names of directories are used to list the names of all entries in the directory.

So, we can write:

```
andrew@notebook:~$ ls WebSite
collect  index.shtml  src     upload.in
Collect  Mirror       upload  XCollect
andrew@notebook:~$
```

and both options and parameters can be given:

```
andrew@notebook:~$ ls -r WebSite
XCollect   upload  Mirror       Collect
upload.in  src     index.shtml  collect
andrew@notebook:~$
```

where the `r` option reverses the order of the output.

The `ls` command has a large number of options, so many in fact, that it has been said that a true Linux guru is someone who can tell you what they all are. But for mere mortals, a very useful command is the `man` command (for manual, as in technical manual). This will tell you what all the options and parameters are. To see what the `man` command can do, then `man man` will do the job.

Some other simple commands that can do no damage while you are experimenting are:

`ps` which lists currently running processes, and can say who owns them

`uname` which gives the name of the machine and level of operating system that is running

`wc` gives a word count in a file

Try them and see.

5. Combining commands

One of the most useful things you can do in the bash shell is to combine strings of commands together. Each command has these three standard input and output files available. The shell gives a way of hooking the output from one command into the I put of another, giving the simple Unix commands surprising usefulness and power.

Suppose I want to know how many files I have in my `bin` directory. Then we know how to list the file names, and we know how to count words, but we need to count the number of filenames in the output from the `ls` command by using `wc`. This is achieved by means of a ***pipe*** represented by the vertical bar **`|`** character. We have to know one other thing, though. The `ls` command will try to reduce the number of lines in its output and put the names in several columns when the output is going to a terminal. It assumes that terminals are looked at by humans and that they do not like scrolling back and forth. However, it knows when this is the case, and so when the output goes elsewhere, then it lists the names one per line. In particular, when the output is piped into the input of another command, it puts them one per line. `wc` can just count lines.

So I can write:

```
andrew@notebook:~$ ls bin | wc -l
51
andrew@notebook:~$
```

There are 51 entries in the `bin` directory in this example.

When a command is started, the standard input is taken from the terminal, and both standard outputs will display on the terminal. However, it is possible to point these standard files elsewhere.
Standard output can be “re-directed” by using the character **>** followed by the file to take the output.

So to save the output from the `ls` command in the file `ls.out`, the line would be

```
andrew@notebook:~$ ls bin > ls.out
```

And the way of picking it up by the `wc` command, is to use the **<** character:

```
andrew@notebook:~$ wc -l < ls.out
```

6. Scripts – execution of commands

When the operating system finds a command to execute, it tries to identify the type of execution it must perform. In the case of a normal command (like ls) it will create a process, load the binary file, and pass control to it.

However, if it finds a *text* file, then it will look at the first line of the text, and see whether it is a script,which are identified as such if the **first two characters are #!**. If they are, then the first line will be treated like a command line we have just been discussing, where the first group of characters are the name of the command, and the remainder are options and parameters. This command though names the program to be called and to be passed the options and parameter in the usual way, but in this case, it is also asked to interpret the file named by the user on the command line.

A ***script*** is such a text file which has been given execute permissions as well as being editable by any simple text editor. Whenever you look at a script file, then the first two characters will be this magic incantation.

So, when the program requested by the first line is the same as that used to process terminal input, then the commands will be the same as you would type at the command line. For almost all Linux distributions, the shell used for the command line is **`bash`** and the first line of scripts will be:

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

If you look at scripts written in other languages, like Perl, or Python, you will see that the first line will contain the name of the Perl or Python interpreter respectively.

Here is a simple command sequence that will count the number of files in the current directory:

```
#!/bin/bash
ls | wc -l
```

If you type those lines into a text editor, and save them in a file called *countfiles*, in the current directory, and try to execute it you might get:

```
andrew@notebook:~/Temp$ countfiles
bash: countfiles: command not found
andrew@notebook:~/Temp$
```

The reason being that the file is not in the search path for commands (see later). So let's try giving it a name that avoids the automatic search:

```
andrew@notebook:~/Temp$ ./countfiles
bash: ./countfiles: Permission denied
```

```
andrew@notebook:~/Temp$
```

Still something wrong, but this time, we can see that we have not given it permissions to allow it to be executed. So, let's try again:

```
andrew@notebook:~/Temp$ chmod +x countfiles
andrew@notebook:~/Temp$ ./countfiles
3
andrew@notebook:~/Temp$
```

That's better. It can now be executed.

Environment variables

When a command starts, it also has access to the values of certain pre-defined ***environment variables***, containing various things that tell it what type of environment it is in. This includes things like native language, time zone, which shell we are running under, who is logged on, where the home directory is, etc. The complete list can be obtained by issuing the **env** command with no parameters. It is possible, but not usually needed, to change these values for the execution of a single command. The names of these variables, and their values, are dependent on the system under which he command runs. Names should not be assumed to exist everywhere.

Command search path

When the system looks for a command, it has the notion of a ***search path***, which is a list of directories. It tries each of these directories in turn until it finds a command of the name, or fails to find one at all. This path is searched whenever a command name is given that does not contain a slash character (`/`). But if it does contain a slash, then it assumes the name is a pathname that is relative to the current working directory.

The path is held in an environment variable, and can be seen by the command:

```
andrew@notebook:~$ echo $PATH
/home/andrew/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
andrew@notebook:~$
```

Each directory in the list is separated by colons from the others. In Ubuntu, the first in the list is a directory local to the current user, providing it exists. So if you create a directory called `bin` and place your `countfiles` script into it, then you can execute it as though it were a simple command:

```
andrew@notebook:~$ countfiles
19
andrew@notebook:~$
```

7. Essential Syntax

Computer scientists use the term Syntax to refer to the way in which a command can be analysed and how it is to be understood when broken down into its constituent parts. We have already seen how a line is interpreted as a command name, options and parameters. We have also seen the special use of the pipe (`|`), and file re-direction (`< >`).

A command line is divided by white space (blanks, tabs, etc.) into separate pieces. The first is the command name, and the others are options and parameters in sequence. However, it may happen that you wish to give to a command a parameter containing a blank, how do you do that? The

answer is by some form of *quoting*. This can take a number of forms, the easiest ones to understand are single quotes (' '). Single quotes take everything between the first and next quotation mark and treat is literally as a single string. Another method is to *escape* a single character by prefixing it with a backslash character (\ ). So both the following lines do the same thing:

```
andrew@notebook:~$ echo ':  :'
:  :
andrew@notebook:~$ echo :\ \ :
:  :
andrew@notebook:~$
```

Each command displays a pair of colons separate by *two* blanks.

Variables

Shell variables can contain values – usually textual strings. Their names are referenced in a script by a dollar symbol ($) followed by a string of alphanumeric characters. However, when they are given a value, the dollar is missing usually, as will be seen in the following examples.

There are also a number of very special variables which usually have names consisting of a dollar symbol and one other special character. Some of these we will also meet.

As mentioned earlier, commands can return a value which is non-zero if an error occurs. This can be seen by use of the variable name $?, like this:

```
andrew@notebook:~$ ls junk
ls: cannot access junk: No such file or directory
andrew@notebook:~$ echo $?
2
andrew@notebook:~$
```

Now let's look at a more practical example. Suppose that we want to count the number of files in the Pictures directory:

```
#!/bin/bash
cd ~/Pictures
ls | wc -l
```

would be a suitable script. It is simple, but the directory is fixed in name. Suppose we want to use it on any directory – then this would work.

```
#!/bin/bash
dir=~/Pictures
cd $dir
ls | wc -l
```

where the name is specified by a variable name.

```
#!/bin/bash
cd $1
ls | wc -l
```

which counts entries in the first parameter.

But what if we want to default to Pictures, if no parameter is given. This can be done with special syntax, like this:

```
#!/bin/bash
parm=$1
dir=${parm:=~/Pictures}
cd $dir
ls | wc -l
```

Another method will shown when testing is introduced later.

File globbing

File globbing is the name given to a very special, but extremely useful feature in the shell. It enables one command to be passed a list of all the files that match a given pattern. The only one I'll show here is the use of the asterisk (`*`). When a command line contains an asterisk as a separate parameter, it is replaced by a list of the names of all the files in the current directory. If there are other characters around the asterisk, then the list contains all names of files with any number of characters where the asterisk is in the name.

Comments

On a line, all text from a # character to the end of line is ignored. This is the way to write comments to remind you of what each line and section is doing.

8. Testing

Tests can be written using the `test` command, for which an alias is the left square brace (`[`), which makes it easier to read in many cases. It can be used to test for equality of strings, or equality of numbers, or presence of a file name or whether a name is a directory, etc. There are several examples in the scripts in the following sections.

The most common use of a test, is in an if statement, the full form of which is:

```
if [ expression ]
then
        command-if-true
elif [ expression2 ]
then
        command-if-true
else
        command-if-all-false
fi
```

[Note: the above is not completely correct. Strictly speaking, the `if` statement tests the return value of whatever the command it is that follows the `if` keyword.]

The other common place to find a test command is in the `while` command, as shown below.

Parameters and options

The parameters written on the command line are known by the names `$1`, `$2`, `$3`, etc. But there are several ways of getting to them. The following two scripts will each write out the name of the command, and then list in sequence all the parameters passed to them.

These examples can be used as templates for writing values out, performing arithmetic, and some types of loops. This paper is too short for a detailed description, but the reference is recommended for further information.

```
#!/bin/bash
# Shell script that will list all the options and parameters passed to the script
# This version uses echo and shifts the parameters once per loop
echo Number of parameters = $#
echo "0: <$0>"
p=1
while [ $# -ne 0 ]
do
        echo "$p: <$1>"
        shift
        : $((p = p + 1))
done
```

A common way of printing to the standard output is to use the echo command, which just takes the parameters and writes them out separated by single blanks, and ending with a newline.

This script shows a couple of other things. To get the number of parameters passed in, there is a special variable named `$#`. the name of the command is in `$0`, and this is not included in the count.

To test for an arithmetic value, the `test` command uses the “operators” `-ne`, `-eq`, `-lt` etc. To test for equality or other wise of strings, then use the “operators” `=` and `!=`. There are many other operators which are listed in the reference.

The way in which arithmetic can be performed is in a little odd if you want to stick to the POSIX standard. Essentially, the only defined arithmetic operations occur in substitutions, and so to make a change in a value it can be done in the “do nothing command”, :.

Finally, the shift command is very useful in “consuming” a parameter, reducing the count by one, and shifting all of them along one. So each time round the loop, one parameter is removed, making the next one `$1`.

An alternative method is as follows:

```
#!/bin/bash
# Shell script that will list all the options and parameters passed to the script
# This version accesses parameters by listing them all for a for loop
printf 'Number of parameters = %3d\n' $#
p=0
printf '%3d: <%s>\n' $p "$0"
for t in "$@"
do
        : $((p = p + 1))
        printf '%3d: <%s>\n' $p "$t"
done
```

The second technique uses the `printf` command to write to the output, which allows you to define how the output is to look.

It also demonstrates use of a special variable named `$@` which expands to a list of all the parameters one after the other. Another way of writing a loop, is the `for` statement which assigns the given variable (`t` in this case) to each value in the list. The double quotes around the list ensures that each entry in the list is treated as a single value; if it was not there, then the list would expand to a string and be split again by looking where the white space is.

It is always a good idea to surround parameters by double quotes whenever there is any possibility that it may contain whitespace.

Real example

There is a package of over 200 command line applications for manipulating picture files, and these will be used to shrink pictures. To use this script under Ubuntu, the package `netpbm` must be installed.

We ask for a script to run through a directory of picture files (as jpegs) and generate scaled versions of each picture and place it in another directory, but with the same name. The script can be invoked, for instance, like this:

```
scalephotos ~/Pictures/Nov2008 ~/Smaller/Nov2008 0.1
```

which will reduce each picture to one tenth the size on each side and place the result in the Smaller directory.

```
#!/bin/bash
# reduce the size of a directory of photos to quarter size in each direction
# Usage: scalephotos src-dir tgt-dir [scale-factor]
# check validity of parameters
if [ $# -lt 2 ] || [ $# -gt 3 ] || [ -z "$1" ] || [ -z "$2" ]
then
   echo Usage: $(basename "$0") 'source-dir target-dir [scale-factor]'
   exit 1
fi
# default scale factor to one quarter
scale=0.25
if [ $# -eq 3 ] && [ ! -z "$3" ]
then
   scale="$3"
fi
 # if target directory does not exist, create it
if [ ! -d "$2" ]
then
   mkdir -p "$2"
fi
# but if it could not be created, stop
if [ ! -d "$2" ]
then
   echo "Target is not a directory"
   exit 2
```

```
fi
# for each jpg in the source directory, scale and put into target
for pic in $(cd "$1"; ls *.jpg)
do
   jpegtopnm "$1/$pic" | pnmscale $scale | pnmtojpeg > "$2/$pic"
done
# finished
exit 0
```

9. References

Arnold Robbins, Nelson H.F. Beebe, *Classic Shell Scripting*, O'Reilly, 2005, 0-596-00595-4; covers several topics and applications, not just the shell.