Introductory Shell Scripting

Written by Dave Pawlowski, September 25, 2012

Introduction

You've been introducted to a several Linux commands. While these commands are useful, what if you needed to execute a series of commands over and over again? It would be really annoying if you had to type each one of them everytime you needed to use them, and not much better if you used the history feature of Linux. As it turns out, there is an easy way to handle repetitive tasks. Create a script! A script is basically a small program that executes one or more Unix commands automatically. There are many different languages that are typically used for scripting, such as Perl, but we will be using the shell- creating shell scripts. Being able to effectively create shell scripts quickly can be a really useful tool, and make doing routine tasks that may take a long time if done using only the command line very quick.

A first script

Let's start off by making a script that prints a message to the screen. Open up a file in emacs called first.sh. The .sh suffix doesn't really mean anything, other than it would be an indication that this file is a shell script.

Add the folllowing lines to the file:

```
#!/bin/bash
#This is a comment!
echo "Hello World"
```

The first line of the script tells Linux the shell that should be used to interpret the script. In this case, we are telling Linux to use /bin/bash, or the Bourne Again shell. (I know that you are forced to use tcsh by default when you logon to chuck. But bash really is a better scripting language. It has more features, including several that are really fundamental to good programming.)

The second line of that script begins with a special symbol: #. This marks the line as a comment, and is ignored by the shell. The only exception is when the very first line of the file starts with #!, like ours. Then Linux knows to interpret it as described above.

The third line runs a command: echo, with one argument, the string "Hello World".

Click on your terminal and try to run the program by typing:

```
% ./first.sh
```

The "./" tells Linux to search the current directory for the command.

You should get an error. The reason is that Linux doesn't know that this is an executable file. You need to change the permissions of the file to be executable. This is done with the chmod command. Chmod is used to change the permissions of any file or directory. This is what you would use to make a file readable by everyone, or writeable by people that are only in your Linux group. Chmod takes at least 1 file as an argument, as well as a sequence of letters or numbers that specify what the permissions should be.

The easiest way to handle permissions is to remember that there are three types of users: the owner, the group, and everyone else. In order to change the permissions for the user that owns the file, you would do something like this:

```
% chmod u=rwx file
```

Only the owner of a file, and root can change the permissions of a file. This above example tells Linux to give the user (u) that owns the file read (r), write (w), and execute (x) permissions. Similarly, you can change the group's permissions:

```
% chmod g=rx file
```

gives all members of the Linux group (g) that the file belongs to read and execute permission.

Finially, you can change the permssions for everyone else:

```
% chmod o=r file
```

gives read only permission to everyone else (o for other). There are a few shortcuts- let's say that you wanted to give everyone permission to do anything to a file:

```
% chmod ugo=rwx file
```

Note that we just combined the 3 types of users. Alternatively, we could have done

```
% chmod a=rwx file
```

where a stands for all types of users.

You can also add permissions to the existing ones by

```
% chmod u+x file
```

This says to take the current permissions and add executable permission for the owner.

So back to our example. Give yourself permission to execute the file and then try to run it again. You should see that the shell outputs the words "Hello World" on the next line after the call to the script.

That's a pretty simple script, but it does something. If you haven't created one before, congrats- that's your first program. Let's play around with the script a little. Create a new file called first2.sh:

```
#!/bin/bash
# This is a comment!
echo "Hello      World"       # This is a comment, too!
echo "Hello World"
echo "Hello * World"
echo Hello * World
echo Hello      World
echo "Hello" World
echo Hello "     " World
echo "Hello \"*\" World"
echo ‘hello‘ world
echo ’hello’ world
```

Change the permissions and run it. It's ok if you get an error or two. Try to make some sense of whats going on with each line, and if you can't figure out every line, that's ok, we'll cover all the differences eventually.

Variables

Pretty much every programming language out there uses the concept of variables- a symbolic name for a chunk of memory to which we can assign, read, and manipulate its contents. Scripting with the shell is the same. You may be familiar with a few variables already, such as user.

Lets play with the ability of Linux to be able to handle variables. Open up another file, call it var.sh or something, and enter the following:

```
#!/bin/bash
my_message="Hello World"
echo $my_message
```

This assigns the string "Hello World" to the variable my_message, then echos the value of the variable. The shell does not care about the type of variable used. It can store strings, integers, real numbers, etc. **Note: the shell is picky about spaces. This won't work if you put spaces around the equals sign.**

Note how the shell refers to variables. The third line of the above script includes the $ character. If you remove this, then the echo command won't know that my_message is, in fact, a variable, and will simply write the word "my_message" in the terminal. Requiring a special character to signify that a certain word is a variable is not typical of most programming languages any longer, but it is the case when using the shell.

It is possible to interactively set variable names using the read command:

```
#!/bin/bash
echo What is your name?
read my_name
echo "Hello $my_name!  I hope things are going well."
```

Certain programming languages, like C and Fortran require you to declare variables and their type before you use them. This is not the case in most shells. What this means is that if you try to reference a variable that has not been set, you will get an empty string, and not an error. For example try running this:

```
#!/bin/bash
my_variable=500
echo $my_var
echo $mv_vra
```

You should see that the second echo doesn't actually do anything, since the variable name is misspelled. This can be a tricky source of bugs in your code, as in most programming languages, trying to print a variable that hasn't been set would give you an error.

Scope

If you were to set a variable in the command line and then try to use it in a program it wouldn't work. Try it. First create a simple script, myvar2.sh:

```
#!/bin/bash
echo "my_var is: $my_var"
my_var="hi there!"
echo "my_var is: $my_var"
```

Once you've done that, go back to the command line and run the script. You will see that the first echo gives you nothing, and the second spits out the string "hi there'!"

Now, try setting the variable in the command line:

```
% set my_var="hello"
```

!!!Caution, we are scripting using the Bourne shell, sh, but the shell that you normally use is tcsh!!!! Variables are set using the set command in tcsh and csh.

Now run the code:

```
% ./myvar2.sh
```

The output should be the same! The reason is that when you set a variable as we have been doing, the variable, by default is local to the current shell. When a command is executed, Linux actually spawns a seperate shell to run the command. We want to make the variable available to all sub-processes started by the current shell- we want to change the scope of the variable. This is done in bash using the export command, and using the setenv command in tcsh. So, execute this in the command line:

```
% setenv my_var hello
% ./myvar2.sh
```

Now you should see the variable is set already when we call our script.