SHAWN POWERS

Hulk Bash!

Not a programmer? No worries, Bash scripting doesn't have to be rocket science.

I'm not a programmer. Anyone who has read my "code" through the years in my columns would agree. That doesn't mean I don't have a constant need to depend on scripts to help automate my job. Let's face it, system administrators don't have enough arms on their bodies or minutes in a day to accomplish all the various things that need to be done. Any sysadmins worth their salt know enough about scripting to make sure they don't do the same task over and over. If you need to do something more than once, you should be using a script. So in this article, I want to give a quick primer, along with a little bit of insight regarding Bash scripting for Linux.

I'm going to assume you're like me and don't have a programming background. If you're a programmer, you're probably writing your own programs. My paltry scripting abilities won't do you much good, and you'll probably e-mail me about how inefficient I'm being. (Many will anyway, and that's fine, but be sure to read the next section.)

What Hack-Job Scripting Isn't

There's a place for efficient, well planned programming. There's even a place for well thought-out scripting (Dave Taylor, for example, will give you far more insight on the proper way to script). If you're in the server room as a system administrator, however, sometimes you just need to write a five-line script to automate a task that would take you an hour on your own. If it takes you six hours to write a "proper" script that is designed to save you five hours... well, you fail 6th-grade math.

Error handling is another very important aspect of any programming. Again, however, if you're just making a script to create a JSON configuration file, thinking too much about potential errors is just silly. Look at the config file when you're done, and if it's wrong, fix the script. In my early years as a system administrator, I was afraid to use a scripting language, because I wasn't a "programmer", and I figured I'd do something wrong. I was right, I *did* do things wrong. I still do. But so do programmers, and there's no better way to learn than by doing. So let's look at some of the basic things a Bash script can do and then experiment. The worst you'll do is mess something up, but then you get to fix it, which is often more fun than breaking it in the first place!

A Few Basic Things I Didn't Know, and a Few I Did

When I first started scripting, I did some really, really dumb things. They worked, so I don't regret them at all, but if I'd known then what I know now, I might have gotten a little more sleep in my 20s. Here's a very partial list of useful things to know when using Bash scripts.

The Backtick On the Bash command line, and therefore in Bash scripts, when you enclose a command in backticks, Bash replaces the stuff inside the backticks with the results. For example, say you have a text file "file.txt" that has a single line of text in it, reading "This is cool." On the command line, if you type:

```
echo "cat file.txt"
```

The command line will return simply:

```
cat file.txt
```

But, if instead you enter:

```
echo "`cat file.txt`"
```

You'll get the following:

```
This is cool.
```

What's happened is that the Bash shell takes the output of `cat file.txt` and uses that in the echo statement, because it was encased in backticks. An embarrassing truth is that for a long, long time that's the only way I knew to pass information to a Bash script. If I wrote a script that needed input, I'd save the input into a text file, and then encase `cat thatfile.txt` in backticks. It worked, but I really wish I'd learned earlier about command-line arguments.

Command-Line Arguments If you need user input, Bash scripts allow for this by taking input from the command line when the script is launched. You can reference those variables inside the script, making things much, much simpler. Here's an example snippet:

```
#!/bin/bash
# This is my script, named coolscript
echo "My name is $1, you killed my father, prepare to $2."
```

There are ways to write to a file directly inside Bash, but it's far more convenient to have the Bash file dump its results to the screen, and once you have the script tweaked, redirect the output to a file.

If you launch the script by typing:

```
chmod +x coolscript (NOTE: This makes the script executable,
it only needs to be done once)
./coolscript "Inago Montoya" "Die"
```

The script will take your two arguments and substitute them in the script. So the output will be:

```
My name is Inago Montoya, you killed my father, prepare to die.
```

You can have any number of arguments, and the $1, $2, $3 pattern will follow. If your arguments are strings, like my example, note that encasing your arguments in quotes will allow for spaces. Without the quotes, the output of the script would be:

```
My name is Inago, you killed my father, prepare to Montoya.
```

You wouldn't get any errors, but the word "die" would be stored in the $3 variable and just not used in the script. Command-line arguments are something I use all the time. It's a great way to get information into the script. If you want an interactive experience, you can use the `read` command, but I usually just use command-line arguments because it saves time.

Redirecting Output Because system administrator scripting is often a quick hack to solve a problem, redirecting output to a file is fairly common. There are ways to write to a file directly inside Bash, but it's far more convenient to have the Bash file dump its results to the screen, and once you have the script tweaked, redirect the output to a file. The process is simple, but the ability is ridiculously useful. The following command (which could be in a script):

```
echo "This is cool stuff"
```

will immediately respond by displaying "This is cool stuff" on your screen. Generally, you'll have a more complex script that will display many things on the screen (like a repetitive JSON config file or something), but it still will just print it to the screen. If you want to save the output to a file, you either can copy it and paste it (which I did at first), or you can redirect the output to a file. To do that, change the command to:

```
echo "This is cool stuff" > coolfile.txt
```

You won't get anything printed on the screen, but you also shouldn't get any errors. The cool part is that you will now have a new file called "coolfile.txt", which contains a single line of text. I'm sure you can guess what text that is! One disadvantage of the > redirector is that it writes over whatever file you specify. So if you repeat the command, you'll end up with a brand-new file, named exactly the same thing, with a single line of text. Thankfully, if you use two greater than signs (>>), it will append to the end of the file as opposed to overwriting it. So if you type:

```
echo "This is line one" > oneliner.txt
echo "This is line two" >> oneliner.txt
```

The file "oneliner.txt" actually will contain two lines of text. Try playing around with redirecting text. What happens if you try to use a double greater than symbol when a file doesn't exist? Will it error out? Will it create a file? Give it a try and see if you can figure out the way redirection works.

Conditionals: Getting Iffy with It

One of the most common uses for a script is to do some "thing" based on whether or not some other "thing" is true. The construct for accomplishing such a thing is to use an IF/THEN conditional statement. The format works like this:

```
#!/bin/bash
# An example of an IF/THEN statement
if [ true ]
then
echo "The condition is true!"
echo "I love true conditions..."
else
echo "Uh oh, the condition is false"
fi
```

A quick walk-through should be clear. If whatever is inside the square brackets evaluates as true, the portion of the script after "then" is executed. If it evaluates as false, the part after the "else" is executed. The else portion of the statement is optional. If the else portion isn't there, the if

Just as useful, if not more useful than a conditional statement in a Bash script, is the loop.

statement just doesn't do anything if the conditional statement is false. It's important to end the entire statement with "fi", which tells Bash that the list of things to do is over. The difficult part is often figuring out what to put inside the [ ] brackets. The "conditional statement" can get fairly complex, but there are a few common examples:

```
if [ -e /tmp/filename.txt ]
```

This translates to "if the file /tmp/filename.txt exists, then this is true", so the if statement would execute whatever is in the then part of the script.

```
if [ -d /tmp/thing ]
```

This translates to "if /tmp/thing exists, *and* it's a directory, then this is true", so if there is a file at /tmp/thing rather than a folder, the statement will evaluate as false. In that case, the else part of the script will execute, or if there's no else part, the script will just move past the fi statement doing nothing at all. There are a bunch of things that can live in the conditional brackets. If you want to see a huge list of possibilities, check out **http://www.tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html**.

Usually, the majority of solutions can be met with creative uses of if/then/else, especially if you nest if/then statements inside other if/then statements. The logic can become fairly complex. There is the `case` command, which is ideal in some scenarios. Rather than having two options (true/false, if/then), case allows for a list of options. case statements are a little more complex, but just as logical. If you'd like to learn more about case, check out the in-depth guide at **http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_03.html**.

Getting Loopy

Just as useful, if not more useful than a conditional statement in a Bash script, is the loop. I find two types of loops particularly useful: the FOR loop and the WHILE loop. Let's start with the WHILE loop, because it works much like the if/then statement above. Here's an example of a WHILE loop, which I'll dissect next:

```
#!/bin/bash
# A simple WHILE loop
COUNT=0
while [  $COUNT -ne 10 ]
do
        echo "The counter is $COUNT"
        let COUNT=COUNT+1
done
```

There's a few new concepts here, but they're fairly straightforward. First, you set a variable named `COUNT` to zero. Then you set up the while conditional statement. In this case, it's a comparison, comparing the value of the COUNT variable to the number 10. The `-ne` means "not equals", so in English, the conditional statement reads, "While the variable named COUNT doesn't equal 10, repeat the following." Everything between the `do` and `done` will loop over and over until the conditional statement evaluates as false. As you can probably guess, it's very easy to make an infinite loop with a WHILE loop.

Once the loop begins (with the `do` statement), the script echos "The counter is 0", then the variable `COUNT` is incremented by 1, and the loop starts over because `COUNT` is still not equal to 10. Eventually, `COUNT` does equal 10, so the loop stops and moves past the `done` statement. In the script above, the last thing to print on the screen will be "The counter is 9", because after that is printed, the `COUNT` variable is incremented to 10, and the loop doesn't run again. What would happen if you changed the incrementor to `let COUNT=COUNT+3`? (Answer: the loop would never end and would count by 3 until you got tired and pressed Ctrl-C to end the script.)

The conditional statement works just like the if conditional statement, and the link above will give you lots of conditional tests to use inside the [ ] brackets. Many are easy to guess, like `-eq` is "equals", `-lt` is "less than" and so on. It's important to know that the conditional statement is evaluated *before* the loop is run, so if the statement starts as false, the stuff between the `do` and `done` never will execute.

The FOR Loop

The last construct I'm going to cover here is the FOR loop. It's the hardest loop to wrap your brain around, but it's also one of the most useful. If it seems too complicated or confusing, I urge you to keep playing around with it until it makes sense. Really, FOR loops are incredibly useful. Here's a couple simple FOR loops that do the same thing:

```
#!/bin/bash
# Simple FOR loop example
for x in 1 2 3 4 5
do
   echo "Loop number $x"
done

#!/bin/bash
# Simple FOR loop example with range
for x in {1..5}
do
   echo "Loop number $x"
done
```

Basically, the FOR loops above will print:

```
Loop number 1
Loop number 2
Loop number 3
Loop number 4
Loop number 5
```

What the loop actually does is take the "set" of items from the second half of the FOR statement (so 1 2 3 4 5, or {1..5}) and runs the loop as many times as there are items. Every time the loop runs, it assigns the particular item in the set to the variable in the first part of the FOR statement (so the variable $x in this case). The examples above make it fairly easy to see what is happening, but it can become really complicated, so understanding the basics is key. I'm finishing this article with another code snippet. See if you can figure out what it's going to do, and I'll go over the results.

First, say you have three text files in a folder /tmp/folder/ by themselves:

- file1.txt: contains the text "This is file 1" on a single line.
- file2.txt: contains "This is file 2" on a single line.
- file3.txt: contains two lines of text, "This is line 1" and "This is line 2".

Next, create the Bash script that will deal with the files in a FOR loop:

```
#!/bin/bash
# A script that manipulates files with a FOR loop
for x in `ls /tmp/folder/`
do
   echo "I am the file named: $x"
   cat /tmp/folder/$x
   echo " "
done
```

This returns:

```
I am the file named: file1.txt
This is file 1

I am the file named: file2.txt
This is file 2
```

```
I am the file named: file3.txt
This is line 1
This is line 2
```

The confusing part of this FOR loop is that it's not dealing with a series of numbers, but rather with a set of "things". At the beginning, you should have noticed the backticks to use the output of the `ls` command to create the set. So, since there were three items in the folder, the FOR loop executed three times. Each time, the value of the "item" was assigned to $x. If you follow the logic of the script along with the output of the results, it should make sense.

Your Mission: Play!

I think I'll wrap up there this month. With the information in this article, you should be able to create some fairly complex scripts. See if you can write scripts and have them give you the results you expect. Next month, I want to build on these skills to give you some real-world use cases for scripting as a system administrator.

I'm not a programmer. Thankfully, I'm a fairly logical person, however, and with scripting, logic is king. There are many more complex things to do with shell scripting, but with the basics, you can do so many amazingly useful things. I'm excited for you to get comfortable with scripting—it can be a lifesaver!■

Shawn Powers is the Associate Editor for *Linux Journal*. He's also the Gadget Guy for LinuxJournal.com, and he has an interesting collection of vintage Garfield coffee mugs. Don't let his silly hairdo fool you, he's a pretty ordinary guy and can be reached via e-mail at shawn@linuxjournal.com. Or, swing by the #linuxjournal IRC channel on Freenode.net.

Send comments or feedback via http://www.linuxjournal.com/contact or to ljeditor@linuxjournal.com.