Session 2: Shell Scripting
Craig A. Struble, Ph.D.
July 14, 2010

1 Introduction

Shell scripts are a way to automate the execution of a series of Linux commands. Essentially, any command you can run from the Linux command line can be run as part of a script. While automation itself is a time saving skill, a more important reason to write shell scripts is that they provide a means for documenting your research steps and workflow. In some cases, scripting can lead to publication as you develop processes, methods, or algorithms that others find useful.

Creating a Shell Script
Creating a shell script consists of three basic steps:

1. Create a text file with a text editor like pico, vi or emacs containing the sequence of commands to execute.
2. Make the file executable with the command

   $ chmod a+x scriptname

   where scriptname is the name of your script.
3. Execute the script from the command line by typing

   $ ./scriptname

   assuming your script is in your working directory.

We will explore some variations on this theme as we work through the session, such as adding command line parameters, and storing your scripts in a location so that you can access them from any directory. In general though, the three basic steps are all you need to be more productive.

In this session, we focus on using the Bourne Again Shell (bash), which is commonly used on Linux systems. Several other shell scripting languages exist including C-Shell (csh), Bourne Shell (sh), and Z-Shell (zsh). Each shell has variations on syntax and functionality, but basically behave in the same way. As your comfort level grows, you can explore other shells to find the one that suits your needs best.

1.1 Assumptions
This session assumes you are familiar with command line basics, are able to navigate the Linux directory structure, and can create and edit files using a Linux text editor (e.g. pico, vi, or emacs). These topics were covered in Session 1, and those notes can be used as a reference.
1.2 Exercise

Exercise

1. Using a text editor, create a file named `hello` with the contents

```bash
#!/bin/bash
read name
echo Hello $name
```

2. Make it executable with the command

```
$ chmod a+x hello
```

3. Run the script

```
$ ./hello
```

which means “run the file named `hello` in my working directory (i.e. the ./)”. The script will wait for you to type in some text and hit the `return` key. Once you hit return, `Hello` followed by what you typed should be displayed and the prompt returned to you.

4. Modify `hello` so that it prompts you to type in your name. Hint: use the `echo` command to print out the prompt before reading the name. Run your modified script.

2 Variables

Variables

- `Variables` are named locations storing values. In the first exercise, `name` is a variable used to store the text you typed.

- General format for assigning variables

```
variable=value
```

where `variable` is the name you want to give the location and `value` is the value you want to store.

- A simple example

```
year=2010
```

Retrieving Values

- Retrieve values preceding name with a dollar sign ($).

- So if we have the script

```
year=2010
echo The year is $year
```
• The output of the script will be

```
The year is 2010
```

Variables are a good way to make your scripts more readable and more maintainable. Consider the following `calendars` script

**Without Variables**

```bash
#!/bin/bash
echo The calendar for 2010 is:
cal 2010
echo Last "year's" calendar was:
cal 2009
```

Using variables, while making `calendars` slightly longer, is much easier to understand and modify later.

**With Variables**

```bash
#!/bin/bash
year=2010
echo The calendar for $year is:
cal $year
echo Last "year's" calendar was:
cal $(( year - 1 ))
```

To change the year, only one modification is needed to the script instead of two in the first version (changing 2010 and 2009). Furthermore, the relationship between the years is much clearer when variables are used. When possible, consider using variables instead of directly including constant values in your scripts.

The `calendars` script introduced a couple other things. First, notice the use of double quotes in the text "year’s". This was necessary because a single quote (’) is a special character used to create strings containing spaces and other characters. Another way to do this is to *escape* the single quote with a backslash like:

```bash
echo Last year\’s calendar was:
```

The other concept introduced is *arithmetic*. Arithmetic is done with

```bash
$(( EXPRESSION ))
```

If the `EXPRESSION` is not a typical arithmetic expression using operators like `+`, `-`, `*` and `/`, an error will occur. `bash` shell scripting only supports integer operations, so it’s not suited for complex mathematics. It is useful for many automation tasks, however.

**Exercises**

1. Modify the `calendars` script so that it has a variable named `p_year` for previous year. Assign it the value of the year previous to the one stored in `year`.

2. Continue to modify the `calendars` script so that it prints out
The calendar for the previous year 2009 was:

instead of

Last year’s calendar was:

using the \texttt{p\_year} variable.

3 Comments

It’s good practice to comment your scripts by writing meaningful notes that can be read to easier understand the meaning and intent of the script. In \texttt{bash} comments start with a hash symbol \# and run until the end of the line. The \texttt{calendars} script we’ve been developing can be documented with a simple header that indicates it’s purpose.

With Comments

\begin{verbatim}
#!/bin/bash
#
# Print out this year's and last year's calendars.
#
year=2010
echo The calendar for $year is:
cal $year
echo Last "year's" calendar was:
cal $(( year - 1 ))
\end{verbatim}

4 Command Line Parameters

Editing scripts to make small modifications is not the desirable approach for creating general tools. More typically in Linux, \texttt{command line parameters or arguments} are used to pass information from the command line to a script.

Consider the command

\texttt{./myscript one two three}

The command has three command line parameters, which are provided to numbered variables \$1, \$2, and \$3 as illustrated below. Note that the command itself is assigned to \$0.

\begin{center}
\begin{tabular}{c}
\texttt{./myscript} \\
\$0 \\
\texttt{one} \\
\$1 \\
\texttt{two} \\
\$2 \\
\texttt{three} \\
\$3 \\
\end{tabular}
\end{center}

With Command Line Parameters

\begin{verbatim}
#!/bin/bash
#
# Print out this year's and last year's calendars.
\end{verbatim}
year=$1
echo The calendar for $year is:
cal $year
echo Last "year’s" calendar was:
cal $(( year - 1 ))

We can then execute the script with

$ ./calendars 2010

or

$ ./calendars 2020

The number of command line parameters provided to a script is stored in the special variable $#.

#!/bin/bash
#
# Script name: nparams
# Print the number of command line parameters.
#
echo You passed $# parameters.

Exercise

1. Use the nparams script to find out how many command line parameters are passed in the following cases:

   $ ./nparams
   $ ./nparams one two three
   $ ./nparams "one two" three
   $ ./nparams "one two three"
   $ ./nparams one 'two three'

2. Write a script named params to print out the first 5 command line parameters. Then, use that script to see what the parameters are for the following runs.

   $ ./params
   $ ./params one two three
   $ ./params "one two" three
   $ ./params "one two three"
   $ ./params one 'two three'

To get started, the script should look like

#!/bin/bash
echo The first parameter is $1
echo The second parameter is $2

to print the first two parameters.
5 Conditional Execution

Making decisions is frequently necessary in scripts. The general form of a conditional statement is

```
if CONDITION
  then
    COMMAND1
    COMMAND2
    ...
fi
```

The commands COMMAND1, COMMAND2 up through the terminating fi keyword will be executed if the CONDITION has an exit status of 0 (which Linux interprets as true). Every program on Linux has an exit status, including scripts that either have the exit status of the last command executed or the status provided to the exit command, which we'll see below.

Note that then and fi must be lines on themselves¹

In most situations, the test command, which is frequently abbreviated by [], is used. The test command has too many options to list here, but running

```
$ man test
```

will provide all the different possible tests that can be tried. We'll see one example below.

Let's modify the calendars script so that it checks to make sure exactly one command line parameter is sent.

With Conditional Statements

```
#!/bin/bash
#
# Print out this year's and last year's calendars.
#
# Check for one parameter. If the number of parameters
# is not equal (-ne) to 1, print usage and exit.
if [ $# -ne 1 ]
  then
    echo "Usage: $0 year"
    exit 1
fi

# Everything is OK
year=$1
echo The calendar for $year is:
cal $year
echo Last "year's" calendar was:
cal $(( year - 1 ))
```

¹Actually, you can put them on lines with the if or the last command if a semicolon (;) is used, but let's keep things simple.
6 Loops

Repetition is frequently needed in scripts. bash provides several different kinds of loop constructs, but only two will be described here, which are the most frequently used loops in scripts.

6.1 For Loops

A for loop in bash repeats instructions over a sequence of items or WORDS. The term WORDS is used because the items are space separated strings that will be assigned to a variable.

The general form of a for loop is

```bash
for VARIABLE in WORDS
    do
        COMMAND1
        COMMAND2
        ...
    done
```

where VARIABLE is assigned each word in WORDS in turn. The commands between do and done will be executed each time in the loop. Like with if the keywords do and done must be on lines of their own.

Countdown

```bash
#!/bin/bash
#
# Countdown
#
# for i in 10 9 8 7 6 5 4 3 2 1
# do
#    echo T minus $i
# done
# echo "Liftoff!"
```

6.2 While Loops

While loops are used to repeat commands as long as a specific condition is satisfied or holds true. This is similar to repeating a sequence of if statements until the condition fails (has a non-zero exit status).

The general form of a while loop is

```bash
while CONDITION
    do
        COMMAND1
        COMMAND2
        ...
    done
```

Like if, CONDITION is true if the exit status of the command is 0. When CONDITION is true, the commands in the loop will be executed. The CONDITION will be executed again, and the loop continues to run until CONDITION has a non-zero exit status.
Another Countdown

```bash
#!/bin/bash
#
# Another Countdown
#
i=10
while [ $i -gt 0 ]
do
    echo T minus $i
    i=$(( i - 1 ))
done
echo "Liftoff!"
```

Beware, it is very easy to create an infinite loop with `while` loops. Always make sure to do something inside the loop to potentially change the outcome of `CONDITION`. Sometimes infinite loops are useful, but think carefully about what you really want to do.

7 File Redirection

One of the most common uses for shell scripting in research computing is to automate the generation of files. This is particularly handy for generating Condor submission files and DAGMAN input files, which will be covered in later sessions.

**Redirect Output**

To make the output of a command go to a file instead of the screen, use the `>` operator. The following will create a file named `hello.out` containing the text `Hello World`.

```
echo Hello World > hello.out
```

If the file already exists, it will be overwritten, so be careful.

**Redirect Output, Appending**

To append instead of overwrite the file, use the `>>` operator. The file will be created if it does not already exist, but will add to the end of an existing file. The following will add the text `Goodbye World` to the file `hello.out`.

```
echo Goodbye World >> hello.out
```

**Redirect Input**

Use the `<` operator to read input from a file instead of typing in on the keyboard. The following will copy the contents of `somefile` and output them to the file `duplicatefile`.

```
cat < somefile > duplicatefile
```

A very powerful tool in scripting for creating files is a “here document”, which is accomplished using the `<<` operator. The contents of the current file become input to the command, up to a line containing the text immediately following the `<<` operator. The real power of this approach is that variable substitution occurs in the text, so it is easy to programmatically create text files in a script. The following will create a file named `afile` containing the lines `Hello World`, `Goodbye World`, and `My name is ...`, where `...` is the first command line argument of the script.
Here Documents

```bash
#!/bin/bash
name=$1
cat <<EOF > afile
Hello World
Goodbye World
My name is $name
EOF
```

A more powerful example shows how to use this to create a simple Condor submission file.

Condor Submission Generation

```bash
#!/bin/bash
#
# Create a simple Condor submission file.
# This doesn't handle complicated quoting
# or executables outside of the working directory.
#
if [ $# -lt 1 ]
then
    echo "Usage: $0 program argument1 argument2 ...
    exit 1
fi

program=$1
shift # Remove the first argument

# Generate the submission file
cat <<EOF > $program.submit
Universe=vanilla
Executable=$program
Arguments=$@
Output=$program.out
Error=$program.err
Log=$program.log
Queue
EOF

echo $program.submit created.
```

8 Accessing Scripts from Anywhere

Accessing Scripts from Anywhere

If a script you write is generally useful, you'll want to access it from any directory, not just your working directory. Almost all shells use a special variable `$PATH` to list the directories where commands are found. On `pere.marquette.edu`, the `bin` directory in `$HOME` (i.e. `$HOME/bin`) is in the path. So to use your script from anywhere, move it with
Case Study 1: Preparing Data

In this case study, we will prepare a script to generate directories and input files for analysis on a Cartesian grid. We specify the maximum $x$ and $y$ coordinates of our study on the command line and generate a simple input file containing $x$ and $y$. Each input file will be stored in a directory named run$N$, where $N$ is defined by

$$N(x, y) = maxx \ast y + x$$

The script for this case study will be included after the conclusion of the bootcamp.

Case Study 2: Make Project Structures

In this case study, we will prepare a script to create a simple project directory structure and a template README.txt file. The directory structure will be

```
$ProjectName
  bin  data  doc  workspace
```

The script for this case study will be included after the conclusion of the bootcamp.