Introduction to the LINUX operating system

Vasileios Paschalidis[1], Erik Wessel[2]
[1] *Departments of Astronomy & Physics, University of Arizona, Tucson* [2] *Department of Physics, University of Arizona, Tucson*

This note provides a continuation to the basic introduction to the LINUX operating system by going over basic bash scripting, and makefiles.

PACS numbers:

I. BASH SCRIPTING

A Bash script is a plain text file which contains a series of commands. These commands we would normally type ouselves in the linux terminal (such as ls or cp for example). However, when we want to automate doing large batch of jobs a bash script comes in very handy. In addition, a bash script can execute other bash scripts, so you can have a hierarchy of bash scripts that accomplish specific tasks.

It is convention to give files that are Bash scripts an extension of ".sh" (myscript.sh for example), but since linux is an extensionless operating system, this convention does not have to be respected.

Let's create our first bash script, by creating a file myscript.sh which simply prints "Hello world" on the screen. In emacs create a file myscript.sh with the following content

```
#!/bin/bash
# My first bash script
echo "Hello World!"
```

and save this file.

Anything that can be executed in linux must have executable permissions. To make the myscript.sh executable type

```
chmod u+x myscript.sh
```

To execute the bash script run

```
./myscript.sh
```

In the script the hash exclamation mark (#!) character sequence followed by a path, tells the rest computer what interpreter (or program) should be used to run (or interpret) the rest of the lines in the text file. (For Bash scripts it will be the path to Bash, but there are many other types of scripts and they each have their own interpreter.) This must be the first line of the script (not second or third even if the first lines are blank).

Also, the hash must be followed by the exclamation mark. Otherwise anything after the hash is treated as a comment. This is why the second line in the script "# My first bash script" did not print anything on screen. Like when writing code, comments are extremely useful in a script to tell the developer and possible future developers what specific parts of the script are doing.

A. Command line arguments

Bash scripts can get arguments. The following script is a simple copy script, it copies file $1 (the first argument in the script) to something file $2 (second argument in the script), then prints "Details for file" using the name of file $2, and finaly lists the details

```
#!/bin/bash
# A simple copy script
cp $1 $2
# Let's verify the copy worked
echo Details for file $2
ls -lh $2
```

Take the above script and put it in a file that you call "copy.sh", and save it. Make the script executable, i.e.,

```
chmod u+x copy.sh
```

Create an empty file file1.txt, by running

```
touch file1.txt
```

and then execute

```
./copy.sh file1.txt file2.txt
```

The script should copy file1.txt to file2.txt, and then print on screen the details of file2.txt.

You can add more arguments to a script which inside the script can be accessed through the variables \$1, \$2, \$3, up to \$9. Variable \$0 is the name of the script. In general shell variables are treated as strings, but we can also do numerical calculations with them.

Put the following line in a shell script

```
backupfolder=~/Documents
wheretobackup=/media/vpaschal/Seagate\ Backup\ Plus\ Drive/My_laptop_back/
rsync -aP $backupfolder $wheretobackup
```

B. Variables

In the previous example \$1, \$2 actually represent variables whose values are assigned by the arguments passed to the script at execution time. Variables are extremely important in doing magical operations with linux.

We can also define variables that are local to the script, and which can take on the values from the arguments of the script. For example, the previous script can also be written as

```
#!/bin/bash
# A simple copy script
file1=$1
file2=$2
cp $file1 $file2
# Let's verify the copy worked
echo Details for file $file2
ls -lh $2
```

The above script does the same operation as the previous script. The difference is that we introduced the local variables "file1" and "file2" here, which take on the values of variables \$1, and \$2, respectively. Put the previous lines in a script called, e.g., "copy2.sh", make the script executable, create a file file1.txt, and execute the script it in the same

```
./copy2.sh file1.txt file2.txt
```

The result should be identical as in the case of the "copy.sh" script.

We can also do arithmetic computations with variables. Let's look at the following script.

```
#!/bin/bash
# Basic arithmetic using double parentheses
a=$(( 4 + 5 ))
echo $a # 9

var1=4
var2=5
a=$(($var1+$var2))
echo $a # 9

b=$(( $a + 3 ))
echo $b # 12
```

```
b=$(( $a + 4 ))
echo $b # 13
(( b++ ))
echo $b #   14

(( b += 3 ))
echo $b # 17

a=$(( var1 * var2 ))
echo $a # 20
```

One other extremely useful utility are "for loops". For loops have the general sysntax "for (variable) in (list); do (operations); done". There are many ways to set a list of values that a variable can obtain. Let's look at the following simple script demonstrating a for loop, where the variable "i" takes on values 0 to 5 (with unit increment) and prints it on screen.

```
#!/bin/bash
# Basic for loop
for i in ‘seq 0 5‘
do
  echo $i
done
```

Put the above in a script called "loop.sh", make it executable and run it. You should see on your screen

```
0
1
2
3
4
5
```

In the above, the "‘seq 0 5‘" operator creates the list of numbers 0 1 2 3 4 5

Now, let's use the above tools we learned to sum the numbers from 0 to 1000 with unit increment.

```
#!/bin/bash
# Basic for loop for summing
sum=0
for i in ‘seq 0 1000‘
do
  sum=$(($i+$sum))
done
echo "The sum from 0 to 1000 is:" $sum
```

Put the above in a script called "sum.sh", make it executable and run it. You should see on your screen

```
The sum from 0 to 1000 is: 500500
```

Now, let's look at a very useful application of loops. Let's first create 1000 files called file1.txt, file2.txt ....file1000.txt using the following script

```
#!/bin/bash

for i in ‘seq 1 1000‘
do
  touch file$i.txt # the strings file $i and .txt are concatenated here
done
ls file* # let’s list the files after we have created them
```

Add the above lines in a "generate_files.sh" script and execute it. This may take a couple of seconds to finish, but you should see all the files you generated. Now, let's a loop to rename the files from file1.txt file2.txt ... to file_1.txt file_2.txt ..., using a cool linux utility called sed, and the following script

```
#!/bin/bash

for i in `ls file*` # here we create a list using ls
do
  j=`echo $i | sed s/file/"file_"/` # echo prints the value of i and the output is piped with the pipe oper
done
ls file* # let's list the files after we have changed their names
```

Note that above the apostrophe is the left apostrophe, i.e., the key above your left tab key.

With the stroke of one key and the above script we were able to rename 1000 files. Similarly we can manipulate files, even change their content and parse them. We can create loops with conditionals and exit clauses. Your imagination is the limit, and the internet is your friend! The linux community is extremely supportive, and chances are that what you want to do, somebody else has already a bash script for it. Linux is an extremely powerful operating system.

C. Exercises

Do the exercises in `https://ecs-network.serv.pacific.edu/past-courses/fall-2018-ecpe-170/lab/bash-scripting-exercise`

When you get to problem 5, replace the program

```
./amplify Lenna_org_1024.pgm
```

with a bash script that takes three arguments

```
./three_args.sh [arg1] [arg2] [arg3]
```

and outputs the result of the following operation

```
(arg1 + arg2)*arg3
```

II. MAKEFILES

The Linux utility `make` provides a convenient way for combining jobs/files that depend on other files. To drive `make` we need a `Makefile`. Most often, the `Makefile` tells `make` how to compile and link a program.

A makefile consists of "rules" which have the following structure

```
target ... : prerequisites ...
        recipe
        ...
```

A `target` can be the name of a file. Examples of targets are executable or object files.

The `prerequisites` are files used as input to generate the `target`. A `target` can depend on several files.

A `recipe` is an action that `make` executes. **Note: you need to put a tab character at the beginning of every recipe!**

Often a `recipe` is in a `rule` with `prerequisites` and will generate a `target` file if any of the `prerequisites` change. This is extremely useful when compiling a very large code that contains multiple files, because using `make` will recompile only those files that have changed.

However, a `rule` with a `recipe` for a `target` need not have `prerequisites`. For example, a `rule` containing the `rm` command associated with a `target` called "clean" does not have prerequisites.

A simple example `Makefile` would contain the following

```
myprogram : main.o utils.o
            g++ -o myprogram main.o utils.o
```

```
main.o : main.C header.h
        g++ -c main.C

utils.o : utils.C
        g++ -c utils.C

clean :
        rm myprogram main.o utils.o
```

If we have the files main.C, utils.C and header.h, and run `make` in the same directory as the `makefile`, the above makefile will then build the executable `myprogram`. The flag "-c" is to just compile files, but not link them. If instead you type `make clean`, make will then remove the files myprogram, main.o, utils.o.

In the above makefile, we have 4 targets that include the executable file `myprogram`, and the object main.o, utils.o and the target `clean`. The prerequisites for myprogram are main.o and utils.o which have their own prerequisites which are main.C, header.h and utils.C. A `recipe` follows each line that contains a `target` and `prerequisites`, that simply compiles the relevant files.

The target `clean` is not a file, and all makefiles typically contain it for convenience.

If we have a code that contains many files, it is far more convenient to use Variables to make our Makefiles simpler. Using variables the previous makefile can be simplified as follows

```
OBJS =  main.o utils.o
cpp=g++
cppflags= -c

myprogram : ${OBJS}
            ${cpp} -o myprogram ${OBJS}

main.o : main.C myheader.h
         ${cpp} ${cppflags} main.C

utils.o : utils.C
        ${cpp} ${cppflags} utils.C

clean :
        rm myprogram ${OBJS}
```

The variable `OBJS` above contains the object files our program depends on. The variable `cpp` is the compiler, and `cppflags` is the compiler flags we want to use. Everything else is basically the same and to use a variable we always put the variable name within curly brackets that follow a dollar sign: `${}`. Using variables way we can simply change the compiler program, the flags, add more flags, such as compiler optimization flags, as well as add/remove more object files.

A. Compiling multiple programs with one Makefile

We can use one makefile to compile multiple programs. For example lets assume that we have 3 programs. One program is in files program1.C header.h c utils.C, the second program is contained in file program2.C and the third in file program3.C. The following Makefile can compile these programs

A simple example `Makefile` would contain the following

```
# Target to compile all
all: program1 program2 program3

# Target for program in program1.C, utils.C and header.C.
program1 : program1.o utils.o
            g++ -o myprogram1 program1.o utils.o
```

```
program1.o : program1.C header.h
        g++ -c program1.C

utils.o : utils.C
        g++ -c utils.C

# Target for program in program2.C
program2 : program2.o
            g++ -o program2 program2.o

program2.o : program2.C
        g++ -c program2.C

# Target for program in program3.C
program3 : program3.o
            g++ -o program3 program3.o

program3.o : program3.C
        g++ -c program3.C

clean :
        rm program1 program2 program3 *.o
```

The above Makefile compiles program1 if you type `make program1`, program2 if you type `make program2`, program3 if you type `make program3`. It will compile all programs if you type `make all`. Finally, `make clean` will remove the executables and all object files.

III. BASIC PLOTTING WITH GNUPLOT

Gnuplot is linux utility for plotting data files and functions. Gnuplot has many intrinsic functions one can plot. To start plotting data or functions first we need to launch gnuplot. To do so type

```
gnuplot
```

Now you will be in the gnuplot command line. To plot the $\sin(x)$ simply type

```
p sin(x)
```

and hit "enter". To specify a range in the x axis type

```
p [-3.1415:3.1415] sin(x)
```

and hit enter. The last command plots $\sin(x)$ from $-\pi$ to π. You can also set log scales if you desire. For example type

```
p [0,10] x**2
```

and hit enter. The operator "**" is the "to the power of" operator. So, this last command plots the function $f(x) = x^2$. To set the y-axis to be in log scale (i.e., a log-linear plot) just type

```
set log y; p [0.1:10] x**2
```

Here you learned that the semicolon separates different commands. To also set the x-axis in log scale type

```
set log y; set log x; p [0.1:10] x**2
```

and hit enter. You should see a straight line.

We can also label the axes

```
set log y; set log x; set xlabel "x"; set ylabel "f(x)=x*x"; p [0.1:10] x**2
```

Gnuplot has multiple intrinsic functions `log` (natural log) `log10` (base 10 log), `sin(x)`, , `cos(x)`, `tan(x)`, `exp(x)` (the exponential function), ...

Gnuplot can make plots not only from analytic functions, but also from data. Put the following data into a file and save it as "data.txt".

```
0.1     0.00316096 0.00537051
0.05    0.00101441 0.00185594
0.025   0.00025341 0.00048454
0.0125  6.3341e-05 0.00012385
0.00625 1.5834e-05 3.1312e-05
```

As you can see the above data set has 3 columns. To make a plot of column 2 vs column 1, while in the gnuplot command line simply type

```
p "data.txt" u 1:2 w l
```

and hit enter. Here in double quotation marks we place the name of the file "u" stands for "using", "1:2" stands for column 2 vs column 1, so the first column that appears goes to the x-axis of the plot. The "w l" stands for "with lines", i.e., it connects the data points. If you want to show the points alone use "w p" (with points), and if you want both points and lines use "w lp" (with line-points).

We can also create a single plot that shows more than one curves, by separating with commas the other curves. We can use, e.g. the third column, as follows

```
p "data.txt" u 1:2 w l, "data.txt" u 1:3 w l
```

The data here are very close to each other, so we need to take log scales, i.e.,

```
set log x; set log y; p "data.txt" u 1:2 w l, "data.txt" u 1:3 w l
```

Let's also add labels to the axes

```
set log x; set log y; set xlabel "{/Symbol D} x"; set ylabel "Error";
p "data.txt" u 1:2 w l, "data.txt" u 1:3 w l
```

where you can run the first line (i.e., hit enter), and then run the second line (hitting enter again). Here the construct `{/Symbol D}` allows you to write Greek letters, the particular one is the upper case delta, i.e., Δ. We can also add titles to the key (or legend) of the plot, by using the `t` (title) operator as follows

```
set log x; set log y; set xlabel "{/Symbol D} x"; set ylabel "Error";
p "data.txt" u 1:2 w l t "Approximation 1", "data.txt" u 1:3 w l t "Approximation 2"
```

We can also add analytic functions to our plot, e.g.,

```
set log x; set log y; set xlabel "{/Symbol D} x"; set ylabel "Error";
p "data.txt" u 1:2 w l t "Approximation 1", "data.txt" u 1:3 w l t "Approximation 2", 0.01*sin(x) w l
```

Notice that the legend is automatically set to the analytic function name.

Once we are happy with how our plot looks, we can output it to a file. To do this we first decide the file type and set the filename. For example to output a pdf file type

```
set output "myplot.pdf"
```

where in double quotes you enter your filename with the corresponding extension. Then you need to set the terminal to be the corresponding one by typing

```
set term pdf
```

and then finally either type

```
replot
```

if you have already seen the plot in the gnuplot x-terminal, or to be safe just type the whole command again, i.e.,

```
set log x; set log y; set xlabel "{/Symbol D} x"; set ylabel "Error";
p "data.txt" u 1:2 w l t "Approximation 1", "data.txt" u 1:3 w l t "Approximation 2", 0.01*sin(x) w l
```

To exit gnuplot, either hit `ctrl+D` or type *exit* and hit enter. If the Greek letter is not rendered properly on the machine you first have to generate an eps file, and then convert it to pdf. This can be done by using the postscript terminal of gnuplot. To do this at the step where you set the output file name you type

```
set output "myplot.eps"
```

where in double quotes you enter your filename with the corresponding extension. Then you need to set the terminal to be the corresponding one by typing

```
set term post eps enhanced color
```

and finally replot

```
set log x; set log y; set xlabel "{/Symbol D} x"; set ylabel "Error";
p "data.txt" u 1:2 w l t "Approximation 1", "data.txt" u 1:3 w l t "Approximation 2", 0.01*sin(x) w l
```

where you can hit enter after the first line, and then continue onto a second line. All different commands can be in separate lines if you want. Hitting enter after the second line will give the file you want, and you can exit gnuplot. After generating the `myplot.eps` file, you can convert it to pdf by using the linux `ps2pdf` utility, i.e., typing

```
ps2pdf -dEPSCrop myplot.eps
```

To view the plot you can use the `evince` utility, by typing `evince myplot.eps` or `evince myplot.pdf`. You can further convert the pdf plot to png or jpg by using the linux convert (imagemagic) utility. To convert to png run

```
convert -density 300 myplot.pdf myplot.png
```

and to convert to jpg run

```
convert -density 300 myplot.pdf myplot.jpg
```