CLI Advanced

https://www.jmunixusers.org @JMUnixUsers

This lab was created to familiarize you with some more advanced techniques of using command line and the Unix/Linux terminal. Processes, user permissions, as well as Bash scripting will be covered in this lab.

Part I. Getting Setup

In order to follow along with this lab, you will need access to a Bash terminal. Typically, Unix/Linux machines will use Bash as the shell by default. If you plan to use Windows, Window Subsystem for Linux (WSL) will suffice.

The lab can be found on either stu or GitHub. On stu, make a copy of "/cs/students/uug/CLIAdvanced" in your stu home directory or locally. Use 'cp -rp' to recursively copy all directories and files, perserving permissions and properties. To download from GitHub, navigate to https://github.com/bronstrom/CLIAdvanced and grab the files. *NOTE: I would highly recommend using git and clone the repo, rather than just downloading a zipped version.*

Part II. Processes

Processes are specific instances of an executed program or command. In our terminal, we're able to view processes running simultaneously at the current instant.

Let's check out what processes are currently running.

Type and run the command 'ps'.

You will now see a table with a number of running commands. If you're only running a single terminal session and this command, the output will look something like this:

```
 PID   TTY     TIME     CMD
2373  pts/0  00:00:00   bash
2742  pts/0  00:00:00    ps
```

Using the flags -e (or -A) and -f we can see a full list of all processes currently running and a more in depth status, respectively. Before we were only seeing the extent of active commands running in the terminal.

Run 'ps -ef' to see all process and a full description.

Rather than using 'ps', with 'top' we can see a full list of processes in real time. Use 'q' to leave 'top'.

Try the command 'top'.

The PID (process ID) is the unique number that the system assigns to a process. If we want to individually lookup what PID us assigned to a currently running command, run 'pidof [command]'.

Try the command 'pidof bash'.

Attempt running the command 'pidof ls'. (Nothing will show because it is currently not being executed.)

Next, navigate to the contents of the lab folder.

Execute the script, by running './infinite_odds.sh'.
NOTE: If that doesn't work, type 'chmod + x infinite_odds.sh' then './infinite_odds.sh'

There will now be an infinite loop of even numbers increasing going up. Whoever created this file made an unforgivable rookie mistake! Let's open a new terminal session and leave the script infinitely running. Let's see what our current process status. We can use 'ps -a' to see a simple listing of the active processes between both terminal sessions:

Either use 'ls -a' or 'top' to see our current processes between both sessions.

This command may or may not be consuming a ton of resources on the system, but it's definitely an annoyance. To end this process we can use the 'kill' command. Sounds terrifying right?! We can use the PID following the kill command to stop the process in it's tracks.

Locate the PID of the command and run 'kill [PID]'.

Check the other terminal session to confirm the script has been terminated. Alternatively, we could have just used ‘Ctrl+C’, the interrupt/kill signal, to end the command which would have been so much quicker!

Part III. File Permissions

Limited file permissions is an important security concept. You might not know it yet but let me tell you now: it also comes into play with Linux machines. Say you didn’t want an untrustworthy person to snoop around through some sensitive files on your computer. Simply having a user password should help prevent this from happening when your away from your machine, but file permissions add another layer of security.

To control whether someone has full editing, read, and write access, the ‘chmod’ command can be utilized. You may have heard of the command: ‘chmod +x’, which simply means your escalating your user’s rights to be able to execute the file.

Locate the file in the lab directory and attempt to execute ‘./nice_message.sh’. Notice you’re unable to run it.

Now try running ‘chmod +x nice_message.sh’ and then execute the file.

Wow, isn’t that a nice message? Permission types include read (r), write (w), and execute (x).

Type and run ‘chmod -wx nice_message.sh’ in the terminal.

Open the file in a text editor, and try to make some edits to it.

Attempt to execute the file.

In this case, we are unable to write to the file or execute it, but we still have the ability to read it. The minus (-) removes permissions and the plus (+) adds permissions depending on the following type(s).

Furthermore, we can restrict the permissions to certain groups of people. The following include: the owner (u), members assigned to the file’s group (g), and everyone else (o). Although, we may not have the best ability to observe these permissions on a personal machine check out the following listed bellow:

chmod u=rwx,g=rw,o=r nice_message.sh

chmod 764 nice_message.sh

These commands have the same output when executed, giving the owner full read, write, and execute access, group just read and write, and everyone else only read access. The second approach makes use of binary references.

Lastly, to view privileges we can use ‘ls -l’. If you executed any of the commands above you will see something like this permission scheme for ‘./nice_message.sh’:

```
-rwxrw-r–.   1   jmunixuser   jmunixuser   91   Feb   23   08:11   nice_message.sh
```

Part IV. Bash Scripting

So far we’ve been writing one-line commands for Bash to interpret. Now we want to start forming more complex series of commands which feel closer to programs. These are called scripts.
Let’s begin as we typically do with programming-esque languages and write a ”Hello World” script. It would be wrong to start with something other than that, right?

Create a script file named ”hello_world.sh” and open it in a text editor.

In order to signify this as a script file, on the first line write ‘#!/bin/sh’. To print a couple words, all we need to do is write echo and the phrase. This is what the contents of you’re complete ./hello_world.sh should look like:

```
#!/bin/sh

echo "Hello, world!"
```

Fairly strait forward. This seems like something we could have just wrote one line in the terminal and executed much more quickly. It’s really when we start to automate logic and execute multiple lines of commands, when scripting seems significant and quite efficient. Let’s try running this file.

Attempt to run ‘./hello_world.sh’ in the terminal.

It seems as though we are unable to execute this file on creation. In this case, we'll need to use our new found knowledge and raise our privileges.

Run 'chmod +x hello_world.sh' and then execute the file.

Perfect, runs just as expected. Let's move on to something a little more our style.

Create a new file with any name you like using 'touch [filename]'.

Raise our executable permissions with 'chmod +x [filename]'. Now open the file in a text editor.

Like most programming languages, we can create variables and perform arithmetic. In Bash script, variables happen to be typeless. Basically, they all variables are treated like strings until they need to be used as numbers for logic. Try out the example bellow or a variation of it. In this example, the script multiplies a variable named NUM (whose value is two) by the value five.

```
#!/bin/sh

NUM=2
echo $((5 * $NUM))
```

Notice that a $ was placed before the variable and logo. This means we utilize 'NUM' as a number as well as treat everything inside the quotes as numbers and perform a calculation.

An interesting feature is the ease of supplying the script with a command line argument. $1 references the first argument. $0 happens to be the variable that references the script path.

Try swapping out the value '5' in the previous example for '$1'.

To execute the file with the argument '3', use the terminal like so: "./[filename] 3"

Next, lets try to add some conditionals that change how our program outputs. In the case that we receive an even command line argument and it's greater than NUM, we should divide the result, otherwise we'll stick to multiplying. Below is the syntax for solving this condition.

```
#!/bin/sh

NUM=2
if [ $1 -gt $NUM ] && [ $(($1 % $NUM)) -eq 0 ]; then
    echo $(($1 / $NUM))
else
    echo $(($1 * $NUM))
fi
```

Notice how we close the condition with "fi" which is just the reverse of "if". Also, a flag-like syntax is commonly used for binary operators in bash conditions (-eq stands for equal and -gt is greater than), instead of using symbols.

Lastly, let's cover a loop example. Below is a short form for loop example using brace expansion.

```
#!/bin/sh

for x in {5..15}; do
    echo: "$x is quite the number"
done
```

Brace expansion allows us to specify a range of commands. In this for loop example, 'x' begins at '5' and increments by one each iteration of the loop. This will continue to the point where 'x' satisfies the condition and reaches '15'. So, everything in the loop is performed ten times in total.

For more in-depth scripting examples, see the samples directory provided in CLIAdvanced. The sample directory contains examples of case statements, loops, functions, and arrays.