PHYSICS 2T C Programming Under Linux

Linux Lab 2 Advanced command line usage

INTRODUCTION

In this lab you will learn how to use the command line in an effective manner and make use of some of the more advanced features available in BASH.

GETTING STARTED

Start by opening the Terminal application. Verify that you are in your home directory, you can do this by typing **pwd** (Print Working Directory) into the terminal. It should be similar to **/home/0800890**.

To obtain all the files required for this lab, please enter the following into your BASH shell:

git clone https://bitbucket.org/glaphysp2t/linux-lab02.git

Confirm you have the necessary files by entering the command: **ls linux-lab02**

MANAGING PROCESSES

When developing scripts and programs is it essential you know how to properly manage running processes. It is not uncommon to accidentally create a script or program that contains an infinite loop and have to stop it manually.

Fortunately, managing processes is a relatively simple thing to do. On Linux **top** is similar to the Task Manager found in Microsoft Windows. Run it, take a look at some of the information then use q to quit. Top is very useful for a quick overview of the system and for continued monitoring. In general, however, it is overkill for what you need.

In the **linux-lab02** directory there is a script called **loop.sh** which you will need to make it executable before you will be able to run it.

When you start the script, it will repeatedly print the message “Hello World”. The quickest, and often most desirable way of stopping a misbehaving script is to use the **ctrl + c** combination. This will terminate the program and return you to the command prompt.

Sometimes you may not want to terminate a program, but pause it while you do something else. The **ctrl + z** combination allows you to do this.

1. Open **xclock** using **xclock -update 1** and pause it using **ctrl + z**. It should no longer be ticking. (Make sure you have the terminal window selected for this to work)

2. Use the **jobs** command to get a list of the currently running jobs and their status, xclock should be listed.
3. Start copies of **top** and **emacs**, pause them both using the **ctrl + z** combination.
4. Run the **jobs** command again to see a list of all the jobs currently running.

5. james@x100e:~/Physics-2T/labs/Lab02$ jobs
   [1] Stopped xclock -update 1
   [2]- Stopped top
   [3]+ Stopped emacs
6. The number inside the square brackets is used to identify the job. **fg 2**, for example, would bring **top** into the foreground and restart it. Use the **fg** command to restart **emacs**.
7. Insert some text into a new document then use **ctrl + z** to pause **emacs** again. Try inserting more text, what happens?
8. You can send a process to the background using the **bg** command. Sending a process to the background allows it to run without interfering with the bash prompt.
9. Use the **bg** command to send **emacs** to the background. jobs will now show the process as Running.

It's also possible to start a process in the background by adding an ampersand (&) after the command like so: **emacs &**. This is very useful for quickly launching a GUI application from the command line and leaves the terminal free to be used. Just like using the **bg** command, anything started this way will be listed by jobs.

Another command which you should be aware of is **ps**, which displays a list of the current active processes. The **ps** command provides you with a PID (process id) which you can then use in with the kill command to kill that process.

Note: The PID of a process is NOT the same as the job number provided by the jobs command.

1. Start a copy of **xclock** in the background (remember to add **&**).
2. Use the **ps** command to check that it is running and note its PID.
3. Use the **kill** command to terminate the **xclock** process.

For more information about the **ps** and **kill** commands, read their man pages. The **ps** command in particular has many options for filtering which processes it lists.

There are also a few other commands working looking at. Read the man pages for **pkill**, **killall** and **pidof** before continuing.

QUESTIONS

1. Start top on your computer.
    a) What is your computer's current load average?
    b) How many processes are currently running?
    c) How would you show only processes started by your user?
2. What effect does an ampersand (**&**) have when put after a command?
3. Using only **ps** and **wc** how could you count the number of running processes.

VARIABLES AND THE PATH VARIABLE

In bash, variables are created when they are set. They do not have types, every bash variable is considered and stored as a string.

Note: Variables are case sensitive, so **Foo** and **foo** are two different variables.

Setting a variable is as simple as **LAB="Lab02"** (be careful not to leave a space at either side of the equals sign).

To read a variable you prefix its name with a dollar **($)** sign like so: **$LAB**. This is most often used in conjunction with the echo command: **echo $LAB**.

Variables tend not to be used very often from the command line however when they are it often involves editing the **PATH** variable.

The **PATH** variable is an example of an environmental variable. Environmental variables are used by bash to configure and store information about the current environment. You can see a list of your path variables, along with their current value, using the **env** command.

The **PATH** variable itself contains a list of colon separated directory paths which are search while looking for a command to execute. Any command in a directory listed here can be run by you from any directory without needing to enter its path.

1. Use the **echo** command to print the contents of your **PATH** variable.
2. Create a new directory in your home folder named **bin**. This is where you will store any scripts or binary files you want to be able to use anywhere on the system.
3. Add the new directory to your **PATH** variable using **PATH="$PATH:~/bin"**.
4. Check that the new directory has been added to your **PATH**.
5. Copy the script called hello from the **linux-lab02** directory into **~/bin** and make sure that it is executable.
6. Navigate to several different directories and run the **hello** command, it should work anywhere you run it.
7. Close the terminal window and re-open it.
8. Try running the **hello** command again.

Variables are not persistent. That is, they only exist in the session they were created. You can, however, add them to your **.bashrc** file so that they are always set.

QUESTIONS

4. What security issues could arise from adding a directory to your PATH variable?

5. Consider the following variables:

   A=Apple

   B=Ball

   C=Cat

   What would be the output of following:

   (Use echo if you're not sure)

   a) $A b) AB c) $AB d) $Cat e) ${C}at f) “$A $B $C”

   g) '$A $B $C' h) \$A

REDIRECTION AND PIPES

REDIRECTION

Bash allows the redirection of a command's output using the the greater-than symbol: **>**

For example, **ls -l > ls.txt** will redirect the output of **ls -l** and place it into **ls.txt**. This can be quite useful if you want to save the output of a command for processing later.

It's can also be very useful in conjunction with the cat command. You can quickly concatenate 2 files together using **cat data1.txt data2.txt > data3.txt**. The contents of both files will be added together and saves as **data3.txt**. Try this using the 2 data files in the **linux-lab02** directory.

PIPES

Pipes are a similar concept to redirections, however the output is sent (piped) to another program instead of a file.

To pipe a command's output you use the pipe **|** symbol.

Pipes can be extremely useful for chaining commands together. For example, you may want to find the 5 most commonly occurring values spread across several files. To do this you would need to concatenate all the files together, sort them, count how many times each line occurs, sort the file again to find the highest and then use tail command to get the last 5 lines.

It would look something like this:

```
$ cat data1.txt data2.txt > all_data.txt
$ sort all_data.txt > sorted_data.txt
$ uniq -c sorted_data.txt > uniq_data.txt
$ sort uniq_data.txt > sorted_uniq_data.txt
$ tail -5 sorted_uniq_data.txt
```

Alternatively, you could pipe the output of each command, allowing you to do all of the above in a single line and without creating several files to store the intermittent results.

```
$ cat data1.txt data2.txt | sort | uniq -c | sort -n | tail -5 > sorted.txt
```

QUESTIONS

6. In the example using pipes, why was the data sorted **before** being sent to the **uniq** command?
   Hint: Read the description of the uniq command in its man pages
7. Read the man pages for the **grep** command.
   a) In a single sentence, what does the grep command do?
   b) Which command would search for lines containing the word “Chapter” in the ./linux-lab02/extras/Bash-Begginers-Guide.txt file?
8. Explain what the following command does and it works:

   sort ~/.bash_history | uniq -c | sort -n | tail > commands.txt

9. What is the difference between a single > and double >> when used for redirection? (Hint: Try directing output to a file several times using them)