COMP150 Lab 5: Processes

This lab will have two parts. One a problem from the text that is a thought exercise. The other is a hands on exercise.

Part 1. Text p139, E5.2.

Part 2: Hands on lab below.

In this lab, you will work with processes. As always, be sure to take screenshots where asked that include the prompt with your username, the command(s) you ran, and the output. For shell scripts, you should copy and paste your script file into your Word document submission and include a screenshot with a sample run of the script.

As usual, start your CentOS VM and log in. We're going to start with a few simple `ps` commands. Open two terminals up and do the following:

1. In one terminal, run: `ps -ef`
2. In the other terminal, run: `ps aux`

Take a moment to compare the output of the two versions of the command. Both show much of the same information, although each form does show some unique information. Don't forget that the first line of output in both cases includes the column names. So, to make it easier to compare the two versions, let's pipe the output through `head` to print only the first two lines in each case:

1. In one terminal, run: `ps -ef | head -n 2`
2. In the other terminal, run: `ps aux | head -n 2`

As you can see, the first form includes the parent PID (`PPID`), but the second form includes more memory related information as well as the process status (`STAT`). Now, often when you use `ps`, you are looking for some specific process or set of processes. To do that, you pipe the output of `ps` to `grep`. For example, let's say you want to find all instances of `bash` that are currently running. Do:

1. In one terminal, run: `ps -ef | grep bash`
2. You should see 3 or 4 processes listed in the output, including two instances of `bash`.
3. If you only see 3 processes, run the above command again a few times, until you get 4 in the list.
4. Notice that one of the processes is the "`grep bash`" process that we used to find `bash`!
5. To get rid of that you have to pipe the output to another instance of `grep` that filters out all lines containing "`grep`". To do that, recall the `-v` option to `grep` is used to return all lines not containing the string specified.
6. So, run: `ps -ef | grep bash | grep -v grep`

7. **TAKE A SCREENSHOT** of the output that should include 3 processes (two instances of `bash` and one of another process called `ssh-agent`).

Searching for specific processes is a very common task, so next we will write a script to do exactly that. Indeed, whenever you find yourself repeating a task with lots of typing over and over again then that's probably a good time to slap together a script.

Write a script named `find_proc.sh` that is used to search through the process list for a particular process (or group of processes) based on the above commands. The script should take one required command line argument, which is the string to search for with `grep`. Of course, you should do normal error checking and print a message to the user if necessary. In addition, you should print the `ps` column headings, but only if there is at least one line of actual output (i.e., one process found in the list). Here are some sample runs:

```
$ ./find_proc.sh
usage: find_proc.sh STRING
$ ./find_proc.sh something
$ ./find_proc.sh sshd
UID        PID  PPID  C STIME TTY          TIME CMD
root      2143     1  0 18:33 ?        00:00:00 /usr/sbin/sshd
```

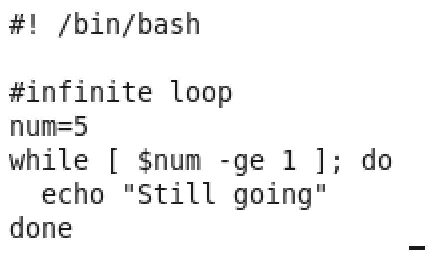
Note that you will need to adapt the primary command in two ways. First, you clearly need to change the search string in the first `grep` to the argument supplied by the user. Second, you will notice that the `find_proc.sh` process also shows up in the output of the script, so you will have to add something else to the command to filter it out. To print the ps column headings correctly, you will have to test the number of lines of output from the `ps`/`grep` pipeline and if there is at least one line of output you will need to run an additional `ps` command with the help of `head` to get just the first line of `ps` output.

Once you've got it tested and working, copy and paste the script file into your submission document and **TAKE A SCREENSHOT** of the output of the script using some example arguments.

Next, let's look at `top`. In one of your terminals:

1. Run: `top`
2. Spend a few moments looking over all of the information available.
3. By default, processes are sorted by current CPU usage.
4. To change the sort order to go by current memory usage, enter: `M`
5. **TAKE A SCREENSHOT** of the output of `top` while the processes are sorted by memory usage.
6. Now, change the sort order back to CPU usage, by entering: `P`

Of course, the list isn't terribly interesting at this point because nothing much is actually going on in your virtual machine. So, we are going to add another script that will actually utilize the CPU. You can use a simple infinite loop script such as



```
#! /bin/bash

#infinite loop
num=5
while [ $num -ge 1 ]; do
  echo "Still going"
done
```

Ok, now let's monitor the system with `top` while something is actually doing something:

1. In one terminal, run: `top`
2. In your other terminal, run: `./infinite.sh &`
3. That will run the script in the background.
4. You should see `infinite.sh` shoot to the top of the process list in `top`, where it is using 90-100% of the CPU.
5. Now, let's see what happens with two competing CPU-intensive processes. Run: `./infinite.sh &`
6. That starts a second instance of the infinite loop script.
7. You should see a second instance of `infinite.sh` at the top of the list now, with both instances sharing more or less equally, at 40-50% of the CPU each.
8. **TAKE A SCREENSHOT** of the `top` window showing both `infinite.sh` processes.

It's time to play with process priorities a bit. You can find the priority of your current shell by simply running `nice` with no arguments. If you want to change the priority of an existing process, you have to use `renice`. So, leave `top` running in one terminal, and in the other:

1. Run: `nice`
2. You should see output of "`0`" (zero) meaning that your `bash` shell has the default priority value of zero.
3. Look in the top window and find the PID of both `infinite.sh` processes. Pick one of the two.
4. Lower the priority of one of the two processes like this: `renice 5 -p PID`
    a. Clearly, replace `PID` in the above command with the `PID` you chose.
5. This makes the chosen `infinite.sh` process "nicer" by 5 and results in it receiving a smaller share of the CPU. You should see the CPU usage for one `infinite.sh` drop while the other increases.

6. Now, lower the priority again: `renice 10 -p PID`
7. You should see the CPU usage drop to roughly 33% for the lower priority `infinite.sh` process and the usage increase to roughly 66% for the original priority instance.
8. One more time: `renice 19 -p PID`
9. The lower priority `infinite.sh` should now be getting roughly 5% of the CPU while the other receives roughly 95%.
10. **TAKE A SCREENSHOT** of the `top` window showing both `infinite.sh` processes with wildly different CPU utilizations.
11. Now, try to reset the nice value back to the original value of 0: `renice 0 -p PID`
12. You should get an error saying "`setpriority: Permission denied`". This is because as a normal user, you are not allowed to raise the priority of any process, even if you were the one who lowered it.

Ok, let's stop those CPU hogs with the `kill` command. Leave `top` running in the one terminal, and in the other:

1. Start by killing the `infinite.sh` process with the original priority of zero (the one using more than 90% of the CPU): `kill -TERM PID`
   a. This time, replace `PID` with the process ID of the `infinite.sh` process using most of the CPU, not the one we used above.
2. You should see that process disappear from the top window. What's more, the lowest priority `infinite.sh` is now using most of the CPU because there is no real contention for it any longer.
3. Ok, now go ahead and kill the other `infinite.sh` process: `kill -TERM PID`
4. We're done with `top` for now, so in the `top` window, to quit enter: `q`

Thankfully the `infinite.sh` script responds nicely to the `TERM` signal from `kill` and is removed from the system. Sometimes, programs might not respond to the `TERM` signal. In these cases you will need to use *kill -9 PID*

Now, the `kill` command can be used to send signals to any process, so you always have to be careful that you are sending your kill signals to the correct PID. As a normal user, you can't send kill signals to processes owned by anyone but yourself. For example:

1. Try running (as yourself only, not root!): `kill -9 1`
2. That would kill the `init` process, but you will get an "`Operation not permitted`" error.

Still, you can accidently kill processes that are owned by you. To demonstrate that, we are going to kill your login session. MAKE SURE THAT HAVE TAKEN SCREENSHOTS AND SAVED YOUR WORK BEFORE PROCEEDING. Ready?

1. Run your find_proc.sh script: `./find_proc.sh gnome-session`
2. That will return the process that controls your entire login session. Get the PID.
3. Run: `kill -TERM PID`
4. You will be logged out immediately and shortly brought back to the login prompt for the system.

So, the lesson here is to be careful what you're doing when sending kill signals around.

As the final task of this lab, you will update your `login.sh` script from a previous lab to include some information from the `/proc` information files. Specifically, you need to retrieve the processor type and total amount of memory in the system and include that in the output of the `login.sh` script. `/proc/cpuinfo` contains the processor information, and you need to look at the "`model name`" line. `/proc/meminfo` contains the memory information, and you need to look at the `"MemTotal"` line. You'll need to get rid of the stuff on the left of the ":" and replace it with more user friendly output in your script. Use `cat`, `grep`, and `cut` like before to get the values out of the file. Here is a sample run:

```
  $ ./login.sh
  Welcome to localhost.localdomain, Marianne Lepp!
  You are logged in as maverick and your current directory is /home/lepp/
scripts.
  The time is 2:32pm.
  System Processor:  Intel(R) Core(TM)2 Duo CPU     T9600  @ 2.80GHz
  System Memory:        515312 kB
```

Note that you don't need to worry about the extra/uneven whitespaces around the cpu type and memory amount. Once you've got it tested and working, copy and paste the script file into your submission document and **TAKE A SCREENSHOT** of the output of the script.