COMP 4736 **Lab 01: Introduction to Linux Tools**

Notes:

- You will need to be comfortable working in Linux environment, especially using command lines. For example, you will need perform basic operations such as creating, editing, copying, moving and viewing files using Linux commands, running commands from the terminal, and using command-line techniques like redirection, using pipes and searching for a string in the output. You will find many helpful tutorials online. Here is one from Ubuntu: https://ubuntu.com/tutorials/command-line-for-beginners#1-overview

- For each question, provide commands entered and output screen captures to support your answers.

Questions:

1. In this question, we will understand the hardware configuration of your working machine using the `/proc` filesystem.
   (a) Run command `more /proc/cpuinfo` and explain the following terms: `processor` and `cores`. Use the command `lscpu` to verify your definitions. You may want to understand the concept of CPU hyperthreading at a high level before attempting this question.
   (b) How many cores does your machine have?
   (c) How many processors does your machine have?
   (d) What is the frequency of each processor?
   (e) What is the architecture of your CPU?
   (f) How much physical memory does your system have?
   (g) How much of this memory is free?
   (h) What is total number of number of forks since the system booted up?
   (i) What is the number of context switches made by `init`?

2. In this question, we will understand how to monitor the status of a running process using the `top` command. Compile the program `cpu.c` given to you and execute it in the `bash` (or any other shell of your choice) as follows.

   ```
   $ gcc cpu.c -o cpu
   $ ./cpu
   ```

   This program runs in an infinite loop without terminating. Now open another terminal, run the `top` command and answer the following questions about the `cpu` process.

   (a) What is the PID of the process running the `cpu` command?
   (b) How much CPU and memory does this process consume?
   (c) What is the current state of the process? For example, is it running or in a blocked state or a zombie state?

3. In this question, we will understand how the Linux shell (e.g., the `bash` shell) runs user commands by spawning new child processes to execute the various commands.

   (a) Compile the program `cpu-print.c` given to you and execute it in the `bash` or any other shell of your choice as follows.

```
$ gcc cpu-print.c -o cpu-print
$ ./cpu-print
```

   This program runs in an infinite loop printing output to the screen. Now, open another terminal and use the `ps` command with suitable options to find out the `pid` of the process spawned by the shell to run the `cpu-print` executable. You may want to explore the `ps` command thoroughly to understand the various output fields it shows.

   (b) Find the PID of the parent of the `cpu-print` process, i.e., the shell process. Next, find the PIDs of all the ancestors, going back at least 5 generations (or until you reach the `init` process).

   (c) We will now understand how the shell performs output redirection. Run the following command.

```
./cpu-print > /tmp/tmp.txt &
```

   Look at the proc file system information of the newly spawned process. Pay particular attention to where its file descriptors 0, 1, and 2 (standard input, output, and error) are pointing to. Using this information, can you describe how I/O redirection is being implemented by the shell?

   (d) Next, we will understand how the shell implements pipes. Run the following command.

```
/cpu-print | grep hello &
```

   Once again, identify the newly spawned processes, and find out where their standard input/output/error file descriptors are pointing to. Use this information to explain how pipes are implemented by the shell.

   (e) When you type in a command into the shell, the shell does one of two things. For some commands, executables that perform that functionality already come with your Linux kernel installation. For such commands, the shell simply invokes the executable like it runs the executables of your own programs. For other commands where the executable does not exist, the shell implements the command itself within its code. Consider the following commands that you can type in the bash shell: `cd`, `ls`, `history`, `ps`. Which of these commands already exist as executables in the Linux kernel directory tree that are then simply executed by the bash shell, and which are implemented by the bash code itself?

— End of Lab 01 —