CSC375: Operating Systems
Lab. 1
Due: June 20, 2022

Experiment with the following Linux Commands:

1. ps, top, ..
2. Memory Address Space
3. Fprk(), exec() system calls

The following files will be needed in this assignment:

- `shell.cpp`
- `fibonacci.cpp`
- `fork.cpp`

A. System information

1. Command **uname -a** tells you the OS that the system is running
2. Command **cat /proc/cpuinfo** to view the CPUs in the system, note that recall many system information are presented as virtual files under the directory **/proc**, for example, meminfo (for memory information) and interrupts.
3. Command **top** lists the top process (in terms of CPU usage) in terms of CPU usage.

B. Processes related commands

Unix command **ps** lists processes in the system. There are many options that control what processes will be listed, and what information about processes will be listed. Here we practice/show ten useful usages of this command:

1. List currently running processes: **ps -ef** or **ps -aux**

    -e: to display all processes -f: to display full format listing

2. List processes based upn the UID and commands: **ps -u**, **ps -C**
3. List all threads for a particular process: **ps -L** (more on threads later).

Practice

1. List all processes that you are running using command **ps -f -U** followed by your user name. Read the manual for **ps** to find out the meaning of each output column. Draw the tree that represents the parent/child relation among these processes of yours.
2. The Unix command **kill** can be used to send a **signal** to a process. A **signal** is a mechanism provided by OS for processes to communicate with each other. For now, you can just remember the signal number 9 is for terminating a process:

    ```
    $kill -9 30453
    ```

    The above command will send a TERMINATING signal to process 30453, which will generally kill the process.

C. Intro. to the Unix Shell

In the above practice of **ps** command, you should notice that there is a running process called **bash** (or whatever shell you are running). **bash** stands for Bourn Again SHell, one of many variants of shell program that is Unix systems' **command line interpreter** (similar to Windows's command console). This is the program that reads the **command lines** you type into terminal window, parses them and executes them (by creating a new process to run that program.

Practice

1. Can you verify that programs that you start in a terminal window are run in a child process of the corresponding shell? How?
2. Use online manual (man) to learn about the following commands: **head**, **tail**, **grep**, **sort**. Note that the last two commands have many options, so you might want to use google to search for some example usages.
3. try the following examples:
   - Input/output/error output redirection

     ```
     $ ls -l > output_of_ls   ## this saves the output of ls -l to the file
     $ echo "10" > input_fibo ## instead of display the message in terminal, send it
                              ## to the file
     $./fibo < input_fibo # run a.out, but reads input from user_input file
     $g++ mylab.cpp > compile_result ##
     $g++ mylab.cpp >& compile_error ## this will redirect standard error
     output to the file
     $g++ mylab.cpp > compile_result 2>&1
     # redirect output to compile_result, and error output to same place as output
     ```

     Note that when g++ displays the compilation error and warning messages to standard error output. So If you want to store them to a file, you need to redirect the error output.

     Question: Does the program **fibo** know the standard input has been redirected to the file?

   - Command pipeline (|): redirect the first command's output as standard input to the second command:

     ```
      $ ps -U zhang -f | grep bash # search for processes running bash
      $ g++ mylab.cpp 2>&1 | head -10 ## To see the first line of
     (standard or error) output of g++
     ```

   - Scripting mode: reading command lines from a file (other than from user input)...

     ```
     $bash testScript ## read commands from testScript, not standard input
     (keyboard anymore).
     ## Note that if testScript is made executable, and has the following
     ## as first line:
     #!/bin/bash
     ## Then we can simply say ./testScript, as the system will know to
     use bash to interpret it...
     ```

4. Write command lines for each of the following tasks:
   - To show the first 20 lines of the shell.cpp file.
   - Compile your program, and show only the first 10 lines of the output/error output.
   - Compile your program, and show only error messages (you can grep for error)
   - Find out where variable **result** are used in all .cpp file. (Note that you can use wildcard to match with any string, i.e., *.cpp will select all files with .cpp as suffix.)
5. Try compile the simple shell program, and execute it.

6. Learn about system call **fork()**
7. Learn about the system call **execvp()**

D. Process's Address Space

We know that the address space of a process refers to the set of memory addresses used by a process is called the process's address space. In this part of the lab, we will again use the **fibo** program and **gdb** to learn more about the address space of a process...

1. Can you estimate how large is the stack frame for **Fibo** function?

   Hint: you can find out the address of **n** (the parameter of the function, which locates in the frame). Also it's save to assume that the stack frame for **Fibo(4)** is the same as **Fibo(5)**(and so on), and that the parameter **b** locates at the same place in the stack frame.

2. Can you find out in which direction the stack grows (to the higher address, or to the lower address)?
3. Are address of your global (local) variables same as others (check with someone sit next to you)?

E. System calls tracing, timing

Perform system call tracing on command **tree**, write down 4 different system calls that are called during this command's execution. Use **man** command to find out what those system calls do.

```
strace tree
```

Use command **time** to measure the **real, user, system** taken to execute the above **tree** command.

```
time tree
```

Note that

- real time: is the wall clock time elapsed between the evocation of the program to the end of the program
- user time: the amount of CPU time used to execute this program in user mode
- sys time: the amount of CPU time spent in kernel mode to for this program (i.e., when carrying out system calls made by the program).

Submission
Please upload your program to GitHub. Share with me your file.