CSI3131 – Operating Systems
Lab1

Objectives:
Experiment with spawning processes using fork. Examining the process status as it progresses.

Lab Setup
You will need a Unix/Linus OS shell with gcc/g++ c-compiler.

- If you're using MAC you're probably already setup for this Lab, as MAC OS is a UNIX based OS.
- If you're using Windows OS, you have many options:
    1. Windows 10 provides aSubsystem for linux, here is a link from Microsoft on how to set it up for windows 10:https://docs.microsoft.com/en-us/windows/wsl/install-win10
    2. Install Cygwin. Go tohttp://preshing.com/20141108/how-to-install-thelatest-gcc-on-windows/and follow the instructions to install Cygwin
    3. Use Linux server provided in the lab

Part1
Systemverilog is hardware description language that is extensively used in ASIC/FPGA design and verification. The language supports the following three forms of the fork constructs for threads:

fork
  task1;
  task2;
  task3;
join
	fork
	  task1;
	  task2;
	  task3;
	join_none
		fork
		  task1;
		  task2;
		  task3;
		join_any

In the above three fork scenarios, the process spawns 3 different threads of processing preforming tasks in addition to the original thread that is also performing its own task.
Depending on the join construct used the behavior of the threads may differ as follows:

- The join construct makes the parent wait for all child threads to finish before resuming the parent process.
- The join_none construct makes the parent not to wait for any child threads to finish before resuming the parent process, i.e. it runs concurrently with the child threads.

- The join_any construct makes the parent wait for at least one child process to finish before resuming the parent process.

Your task is to mimic the above behavior for processes rather than threads. Assuming you have 3 applications in the path named task1, task2, and task3.

The solution of the first mode is provided together with a task application. The attached Lab1.zip file includes a directory part1 that contains 2 files:

1. task.c is a simple console application that sleeps for n seconds, marking the start/end of the process. The time period (n) is passed to the task as a command line argument
2. fork_join.c is the solution of the fork-join scenario as explained above. The fork_join application takes up to 4 arguments as command line arguments, namely task1, task2, task3, and parent task time periods, respectively. Checkout how the task is launched inside the child process. https://linux.die.net/man/3/execlp
3. Copy the fork_join.c file to fork_join_none.c and make the necessary modifications to make it behave as a fork-join-none case.
4. Copy the fork_join.c file to fork_join_any.c and make the necessary modifications to make it behave as a fork-join-any case.

NOTE: Keep the logging messages intact as it will be used to automatically mark your assignment. Failure to do so will result in automatic deductions!

Also attached is the expected results for few of the test cases that will be used in marking, other cases will also be used to verify the correctness of your code.

5. Your submission should include only the 4 c-files named above. Don't include executables not log files.

Part2

Linux presents information on processes via the file system. Statistics on running processes can be examined by opening regular files. Note that these files do not exist on the hard drive, but in main memory. In fact they are not files at all, but simply system data presented as files.

The directory /proc contains a set of directories and files that provides access to general OS information about each process running on the system. Each process has a unique identifier, called the PID (Process Identifier). In /proc, the PID is presented as a directory. The contents of the PID directory provides information on the corresponding process. You can obtain documentation on the proc directory using "man proc" command.

In this part of the lab, you shall examine how a process executes and changes Some utilities (shell programs) and C programs have been provided to examine the changes in processes (by reading the contents of the /proc directory) as they execute different programs.

You will gain insight into the state of a process and the difference between running in the two modes of execution (kernel and user).

Enter command: ls /proc. The content of the /proc directory will be listed. You will see many number entries (representing directories containing information about processes, for example directory 23406 contains information about process with PID 23406), as well as some more meaningful entries (for example 'version' and 'cpuinfo').

Enter command cat /proc/version | more. This should display the content of the file '/proc/version' that contains information about the OS version.

Enter command cat /proc/cpuinfo | more. You might want to explore the content of other files as well.

Enter command ps. This will list the processes you have launched. At least two processes will be listed: 'ps' (the one you just launched and which is producing this and a shell process (probably 'bash', but could be different, depending on your settings). Write down the PID of the shell process.

Enter command cat /proc/NNN/stat | more, where NNN is the PID of your shell process from the previous step. You will see a bunch of numbers containing lots of information about this process. You can learn the meaning of the numbers by entering man proc | more command, which would show you the manual for proc. You can see most of this information in human readable form in the file '/proc/NNN/status'. Try cat /proc/NNN/status

1. Compile and run calcloop.c (g++ calcloop.c –o calcloop.exe)
2. Compile and run cploop.c
3. Compile procmon.c and preview its code – no need to modify this file
4. Run calcloop.exe in the background and run procmon.exe passing it calcloop's PID – checkout the process progress, it should look somewhat like the following:

```
$ ./calcloop &
[1] 6740

$ ./procmon 6740


        Monitoring /proc/6740/stat:

Time         State            SysTm    UsrTm
  0     Sleeping(memory)        0      4062
  1     Running                 0      4906
  2     Sleeping(memory)        0      5078
  3     Sleeping(memory)        0      5078
  4     Sleeping(memory)        0      5078
  5     Running                 0      5937
```

```
  6     Sleeping(memory)      0     6062
  7     Sleeping(memory)      0     6062
  8     Sleeping(memory)      0     6062
  9     Running               0     6953
 10     Sleeping(memory)      0     7015
 11     Sleeping(memory)      0     7015
 12     Sleeping(memory)      0     7015
 13     Running               0     7984
 14     Sleeping(memory)      0     8062
 15     Sleeping(memory)      0     8062
 16     Sleeping(memory)      0     8062
 17     Running               0     9000
 18     Sleeping(memory)      0     9031
 19     Sleeping(memory)      0     9031
 20     Sleeping(memory)      0     9031
procmon: Cannot open /proc/6740/stat, the monitored process is not running any more.
[1]+  Done                    ./calcloop
```

Notice how the user-time is incremented every time the process is running and remains constant every time the process is sleeping (idle)

5. Run cploop.exe in the background and run procmon.exe passing it cploop's PID – checkout the process progress, it should look somewhat like the following:

```
$ ./cploop.exe &
[1] 6884

$ ./procmon 6884


        Monitoring /proc/6884/stat:

Time        State            SysTm    UsrTm
  0     Running              1859      265
  1     Sleeping(memory)     2093      343
  2     Sleeping(memory)     2093      343
  3     Sleeping(memory)     2093      343
  4     Running              2656      484
  5     Sleeping(memory)     2687      484
  6     Sleeping(memory)     2687      484
  7     Sleeping(memory)     2687      484
  8     Sleeping(memory)     3312      625
  9     Sleeping(memory)     3312      625
 10     Sleeping(memory)     3312      625
 11     Running              3515      656
 12     Sleeping(memory)     3921      734
 13     Sleeping(memory)     3921      734
 14     Sleeping(memory)     3921      734
 15     Running              4390      812
 16     Sleeping(memory)     4546      843
 17     Sleeping(memory)     4546      843
 18     Sleeping(memory)     4546      843
 19     Sleeping(memory)     5203      921
 20     Sleeping(memory)     5203      921
 21     Sleeping(memory)     5203      921
 22     Running              5312      937
 23     Sleeping(memory)     5890     1156
 24     Sleeping(memory)     5890     1156
 25     Sleeping(memory)     5890     1156
 26     Running              6078     1203
 27     Sleeping(memory)     6515     1281
 28     Sleeping(memory)     6515     1281
 29     Sleeping(memory)     6515     1281
 30     Running              6968     1328
procmon: Cannot open /proc/6884/stat, the monitored process is not running any more.
[1]+  Done                    ./cploop.exe
```

Notice how both system and user times are incremented every time the process is running. Reflect on why this is the case.

6. Your task is to complete the code of mon.c, which is programmatically performing steps 1-5. Follow the instructions commented in the code:

```
/* Here comes your code.*/
/* It should do the following:
    1. fork/launch the program ’calcloop’ and get its pid
    2. fork/launch ’procmon pid’ where pid is the pid of the process
launched in step 1
    3. wait till calcloop process ends
    4. wait till procmon process ends
    5. fork/launch the program ’cploop’ and get its pid
    6. fork/launch ’procmon pid’ where pid is the pid of the process
launched in step 5
    7. wait till cploop process ends
    8. wait till procmon process ends
*/
```

NOTE: if you cannot launch procmon process twice you may split the job into two files: ‘mon_calcloop.c’ & ‘mon_cploop.c’

7. Include only the source code for the files in part2 directory
8. Zip part1 & part2 directory in one zip file lab1.zip and submit on bright space