Project 1: Measuring the Overhead of Context Switches, Process Creation and Thread Creation

Due date: Friday, January 29 11:59pm on Canvas
Note: *Request for re-grading can only be made within a week from when the grade is posted.*

In this first project, you'll measure the costs of a ***context switch***, ***process creation*** and ***thread creation***. This is an *individual* project.

You must submit a `tar` file named *p1-<netid>.tar* containing the following:

- Code in C that implements the 3 objectives of this project:
    - `context_switch.c` – measures the time it takes to do a context switch.
    - `process.c` – measures the time it takes to create a process.
    - `thread.c` – measures the time it takes to create a thread.
- A `makefile` for easy compilation
- A 1-page PDF report, named report.pdf, that describes the results you obtained (presented as tables or plots, whatever makes best sense). Note that it is not required to include how you implemented the project – that is best seen in your code and the comments you must include in your code.

Example execution:

```
> tar xf p1-palombo.tar
> ls
> context_switch.c    makefile    process.c    report.pdf
thread.c
> make
…
> ls
> context_switch    context_switch.c    makefile    process
process.c    report.pdf    thread    thread.c
> ./context_switch
…
```

Hints:

Measuring the cost of creating a process and creating a thread is relatively easy. For example, you could repeatedly call the corresponding system call (`fork` for process creation, `pthread_create` for thread creation), and time how long it takes; dividing the time by the number of iterations gives you an estimate of the cost of the system call.

One thing you'll have to take into account is the precision and accuracy of your timer. A typical timer that you can use is `gettimeofday()`; read the man page for details. What you'll see there is that `gettimeofday()` returns the time in microseconds since 1970; however, this does not mean that the timer is precise to the microsecond. Measure back-to-back calls to `gettimeofday()` to learn something about how precise the timer really is; this will tell you how many iterations you'll have to run in order to get a good measurement result.

If `gettimeofday()` is not precise enough for you, you might look into using the `rdtsc` instruction available on x86 machines.

Measuring the cost of a context switch is a little trickier. As an example, the *lmbench* benchmark does so by running two processes on a single CPU, and setting up two UNIX pipes between them; a pipe is just one of many ways processes in a UNIX system can communicate with one another. The first process then issues a write to the first pipe, and waits for a read on the second; upon seeing the first process waiting for something to read from the second pipe, the OS puts the first process in the blocked state, and switches to the other process, which reads from the first pipe and then writes to the second. When the second process tries to read from the first pipe again, it blocks, and thus the back-and-forth cycle of communication continues. By measuring the cost of communicating like this repeatedly, *lmbench* can make a good estimate of the cost of a context switch. You can try to re-create something similar here, using pipes, or perhaps some other communication mechanism such as UNIX sockets.

One difficulty in measuring context-switch cost arises in systems with more than one CPU. (Un?)Fortunately, the C4 lab machines are multi-processor: run `lscpu` and `nproc` on one of the machines to see this. What you need to do on such a system is to ensure that your context-switching processes are located on the same processor. Fortunately, most operating systems have calls to bind a process to a particular processor; on Linux, for example, the `sched_setaffinity()` call is what you're looking for. Check it out on C4 lab machines with:

```
man 2 sched_setaffinity
```

By ensuring both processes are on the same processor, you are making sure to measure the cost of the OS stopping one process and restoring another on the same CPU.

Grading rubric:

We will be using automated scripts to grade your project. Therefore, it is very important that your submission meets the requirements exactly as described above.

Percentage(%)	**Rubric comment**
5	General requirements (file names, tar file, directory structure)
10	Compilation and execution (makefile is present, code compiles and runs without errors)

15	process.c
15	thread.c
30	context_switch.c
5	Code is organized and well-commented
20	Report