Project 1: System Calls and Context Switch Measurements

Due date: Friday, February 1st, 11:59pm on Canvas

In this first project, you'll measure the costs of a *system call* and the cost of a *context switch*. You may work in teams of 2. The output of this project includes:

- A report (in PDF) that describes:
    - The methods you tried, challenges you encountered, what you've learned in the process and which method(s) eventually worked.
    - The results you obtained (presented as tables or plots, whatever makes best sense).
    - A listing of the output of the execution of your project.
    - An estimation of the time you spent working on the project.

  There is no requirement on the length of your report: you'll be graded based on how well your report shows your understanding of the problem. You may do that in 2 pages or 5 pages. We certainly hope you do not need 10 pages, but if you do, we'll read them all!
- Code in C that implements the objectives of this project, submitted as a tar file:
    - Include a `makefile` for easy compilation + a `README` file that describes how to run your project (e.g., arguments, etc).
    - Submit as a `tar` file

Hints:
Measuring the cost of a system call is relatively easy. For example, you could repeatedly call a simple system call (e.g., performing a 0-byte read), and time how long it takes; dividing the time by the number of iterations gives you an estimate of the cost of a 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 of your null system-call test 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. 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 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.