Meet Semaphores

In this exercise, you get to redo the pingpong.c program from our earlier exercise, but, instead of using a busy-waiting synchronization solution, you will use POSIX (anonymous) semaphores. This will let us see how using the right synchronization mechanisms can give us a much more efficient solution than a naively-coded solution.

I'm giving you a partial implementation, pingpong.c, to help get you started. It's mostly just contains some header includes and an empty main() method. As before, you'll need to add code to create two new threads, have them alternate execution for a fixed number of ITERATIONS and then have them main thread join with them before exiting. This time, instead of using our own busy-waiting solution to have the threads take turns, we'll use semaphores. The code below illustrates what each thread should do.

ping

```
for ( int i = 0; i < ITERATIONS; i++ ) {
  acquire( pingNext );
  release( pongNext );
}
```

pong

```
for ( int i = 0; i < ITERATIONS; i++ ) {
  acquire( pongNext );
  release( pingNext );
}
```

Above, I'm using our semaphore pseudocode. You'll need to use the POSIX semaphore API instead. You'll need to declare your two semaphores as global variables, initialize them in main and destroy them when you're done.

Once you have your program working, you can time its execution using the time shell command. You should see much faster execution than on the previous version of this program, and you shouldn't see much variation between single-core and multi-core systems. Although the pingpong.c program doesn't do anything useful, it's a good illustration of how we'd like our multi-threaded programs to work. By properly using OS-provided synchronization mechanisms, we can often get a better-performing program while leaving more system resources available for other programs.

When your pingpong.c program is working, submit it to the Gradescope assignment named EX06.