Busy Waiting and CPU Scheduling

In this exercise, you get to do two things. You get to empirically measure the quantum used by the EOS linux machines, and you get to appreciate just how bad busy waiting can be as a synchronization technique.

I'm giving you a partial implementation, pingpong.c, to help get you started. It mostly just includes a global variable and a preprocessor constant. You get to write the interesting parts yourself. Your job is to create two new threads, one called ping and one called pong. The ping thread will run the code on the left then exit, and the pong thread will run the code on the right then exit. After creating these threads, the main thread will wait for them to finish. Your program won't need to print anything. We're just interested in how long it takes to run.

ping

```
for ( int i = 0; i < ITERATIONS; i++ ) {
  while ( nextTurn != 0 )
    ;

  nextTurn = 1;
}
```

pong

```
for ( int i = 0; i < ITERATIONS; i++ ) {
  while ( nextTurn != 1 )
    ;

  nextTurn = 0;
}
```

You can see, these two threads are using the global variable, nextTurn to take turns. The ping thread will wait until its value is zero, then set it to one. The pong thread waits until the variable's value is one, then sets it back to zero. They do this for a fixed number of iterations, 500 by default.

It turns out this is a really bad way for these threads to take turns, especially on a uniprocessor. Consider what happens inside ping. Once it sets nextTurn to one, it goes back into its while loop, using busy waiting to wait until pong sets it back. While ping is in this loop, it will continue to run on the CPU, and (on a uniprocessor), it will effectively keep pong from running until it uses up its whole quantum. Then, once ping gets preempted, pong will get a chance to do the same thing. So, on a uniprocessor, the running time of this program depends not on the speed of the CPU but on how long it takes for these two threads to burn through 1000 quanta.

It's getting more and more difficult to find a single-core system for trying out this exercise. The machines at remote.eos.ncsu.edu were single-core for several years, but it looks like that's no longer true (last time I checked). If you're not sure whether you're on a uniprocessor, you can check by running the shell command:

```
cat /proc/cpuinfo
```

This will generate a report on the capabilities of each CPU or CPU core; if it reports on just one core, you're on a uniprocessor. If it reports more, you're on a multiprocessor.

On a multiprocessor, you can restrict the CPU cores to be used with the **taskset** command. It's a mechanism for specifying hard processor affinity. This command lets you specify which CPU cores to use by giving a mask in binary (actually in hexadecimal, but you can think of it as binary). You use it like:

taskset *mask command-to-run arguments-for-that-command*

A mask of 0x1 says to just use core number zero (since the mask has just the lowest-order bit set). A mask of 0x2 says to use just core number one (since the mask just has bit number 1). A mask of 0x3 says to use both cores 0 and 1 (since the mask has both bit number 0 and 1). So, if you run a the pingpong program as follows, it will run it just on core 0:

```
taskset 0x1 ./pingpong
```

We can time the program's execution to get an estimate of the quantum length on a particular machine. Once your implementation is working, try it out on one of the remote EOS linux machines. I don't know if all these machines have identical hardware / configurations, but it took about 10 seconds to run when I tried out my solution using just one core. Time the execution of your program using the time command and the taskset command (here, you're telling **time** to run **taskset** and you're telling **taskset** to run your pingpong program using just the first CPU core):

```
time taskset 0x1 ./pingpong
```

Consider what this tells you about the quantum length for these remote linux machines. Can you estimate the quantum length from this? (You don't have to turn in anything for this, just think about how you would answer.)

If you run pingpong.c on a multiprocessor, you'll see that it runs **much** faster. With multiple cores, one thread can't prevent another from running just by monopolizing one of the CPU cores. Try timing the execution of your pingpong program but letting it use CPU cores 0 and 1 this time. This won't tell you anything about a system's quantum length, but it will let you see how bad a bad synchronization solution can be.

When you're done, turn in your source code for your completed pingpong.c program. Submit the file to the Gradescope assignment named EX05.

.