[CPEN 331. Fall 2019](https://sites.google.com/view/cpen331-fall2019/home)

**Assignment 2**

In this assignment you finally get to write your own code in OS161! By the end of this assignment you will

* Have a good understanding of the implementation of spinlocks and semaphores in OS161
* Implement locks
* Implement condition variables.

**Step 1. Prepare**

Make sure you don't have any uncommitted updates in your repo. Now, tag your repository as shown here:

git tag asst2-start

Now push that new tag:

git push --tags

Make a directory submit/asst2 in your os161 tree. You will put your file with the answers to code reading questions in that directory.

**Command line arguments to OS161**

Your solutions to ASST2 will be tested by running OS161 with command line arguments that correspond to the menu options in the OS161 boot menu. Please **DO NOT** change any existing menu option strings, since we will be using them to automate testing.

**Physical memory and CPUs**

In order to execute the tests in this assignment, you may need more than the 512 KB of memory configured into System/161 by default. We suggest that you allocate at least 2MB of RAM to System/161. This configuration option is passed to the mainboard device with the ramsize parameter in your sys161.conf file. Furthermore, having many CPUs will enhance the rigour of your testing. Make sure the mainboard device line looks like the following:

31 mainboard ramsize=2097152 cpus=4

**Concurrent programming with OS161**

If your code is properly synchronized, the timing of context switches and the order in which threads run should not change the behavior of your solution. Of course, your threads may print messages in different orders, but you should be able to easily verify that they follow all of the constraints applied to them and that they do not deadlock.

**Built-in thread tests**

When you booted OS161 in ASST0, you may have seen the options to run the thread tests (type ? at the menu for a list of commands). The thread test code uses the semaphore synchronization primitive. You should trace the execution of one of these thread tests in GDB to see how the scheduler acts, how threads are created, and what exactly happens in a context switch. You should be able to step through a call to thread\_switch() and see exactly where the current thread changes.

Thread test 1 ("tt1" at the prompt or tt1 on the kernel command line) prints the numbers 0 through 7 each time each thread loops. Thread test 2 ("tt2") prints only when each thread starts and exits. The latter is intended to show that the scheduler doesn't cause starvation—the threads should all start together, spin for awhile, and then end together.

**Debugging concurrent programs**

thread\_yield() is automatically called for you at intervals that vary randomly. While this randomness is fairly close to reality, it complicates the process of debugging your concurrent programs.

The random number generator used to vary the time between these thread\_yield() calls uses the same seed as the random device in System/161. This means that you can reproduce a specific execution sequence by using a fixed seed for the random number generator. You can pass an explicit seed into random device by editing the "random" line in your sys161.conf file. For example, to set the seed to 1, you would edit the line to look like:

28 random seed=1

We recommend that while you are writing and debugging your solutions you pick a seed and use it consistently. Once you are done with initial debugging/testing, remember to set the random device back to autoseed. This should allow you to test your solutions under varying conditions and may expose scenarios that you had not anticipated, which is central to effective testing.

**Step 2. Code reading exercises**

Please answer the following questions and submit them with your assignment. Place the answers in a file called asst2-answers.txt in your submit directory.

1. What happens to a thread when it exits (i.e., calls thread\_exit())? What about when it sleeps?
2. What function(s) handle(s) a context switch?
3. What does it mean for a thread to be in each of the possible thread states?
4. What does it mean to turn interrupts off? How is this accomplished? Why is it important to turn off interrupts in the thread subsystem code?
5. What happens when a thread wakes up another thread? How does a sleeping thread get to run again?
6. What function(s) choose(s) the next thread to run?
7. How does it (do they) pick the next thread?
8. What role does the hardware timer play in scheduling? What hardware independent function is called on a timer interrupt?
9. Describe how wchan\_sleep() and wchan\_wakeone() are used to implement semaphores.
10. How does the implementation of wchan ensure that a thread never misses a wakeup signal: that another thread cannot attempt to awaken the first thread ju