**CSC 338 Parallel and Distributed Computing**

**Exercise No. 10, April 24, 2017**

**Pthreads and Critical Sections with Semaphores**

**Goal**

Learn how to handle critical sections with pthreads and semaphores

**Background**

Busy-waiting has the advantage that we can control the order in which threads access a critical section. Mutexes are more efficient but threads access critical sections in random order. Semaphores are another way to control access to critical sections.

**Procedure**

1. The exercise folder has a program, *pth\_msg.c,* in which each thread sends a message to the next higher-ranked thread, with the highest ranked thread sending to thread 0. Study the code, noting the use of shared variables, In particular, note how the messages are passed. The variable *messages* is an array of strings (remember, a string in C is just an array of characters so an array of strings is an array of character arrays—in other words, a two dimensional character array).

At any rate, the way in which the threads communicate is by putting a message in the destination's string. For example, thread 0 puts its message into messages[1], thread 1 puts its message into messages[2], and so forth. Then each string prints its own string: thread 0 prints messages[0], thread 1 prints messages[1], etc.

Compile and execute the program with 5 threads.

You probably found that the program has a problem – some of the threads received no message. This is because those threads printed their strings before they had been filled with characters. This isn't really a critical section problem. Rather, it's a classic producer-consumer synchronization problem: one thread consumes something that another thread produces and the consumer can't proceed until the producer finishes its task.

1. Semaphores are a good solution to this problem. A semaphore is an unsigned int. In general, a semaphore can be initialized to any value but we will use a binary semaphore, which only takes on values of 0 and 1. Roughly speaking, a value of 0 corresponds to a locked mutex, while a value of 1 corresponds to an unlocked mutex.

Now turn your attention to the second program in the exercise folder: *pth\_msg\_sem.c.* Compile and execute the program with several different numbers of threads. You should find that all the messages print as expected.

1. Answer the following questions regarding *pth\_msg\_sem.c.*
   1. Semaphores are not automatically built into pthreads. What do you have to include to get access to semaphores?
   2. What global variables, if any, do you have to add in order to use semaphores? If you do have to add any global variables what is their type?
   3. One reason semaphores are a good way to solve this problem is that there is no concept of ownership for semaphores—any thread can lock or unlock a semaphore. In this program, we use an array of semaphores—each thread has its own semaphore. What is the statement that allocates memory for the array of semaphores?
   4. How are the semaphores initialized? Are they initially locked or unlocked?
   5. The semaphores are used in Send\_msg() and here we see why it's an advantage that any thread can lock or unlock a semaphore. Each thread puts its message in the appropriate buffer then unlocks the destination's semaphore. Then it waits for its own semaphore to be unlocked before printing its received message. What is command that unlocks a semaphore? What is the command that waits on a semaphore?
   6. What commands are used in main() to clean up after using the semaphores?
2. To solidify your understanding of semaphores, go back to *pth\_msg.c* and modify it so that each thread sends its message to the rank below it (thread n-1 sends to thread n-2, etc, thread 0 sends to thread n-1). Add semaphores, referring to *pthr\_msg\_sem.c* where necessary, so the threads are correctly synchronized.