Name:..Asa Hayes……………………………………………… UIN:…525003952……………..

Score: out of Total 100

**Question 1 [10 pts]:** The following is the Producer(.) function in a BoundedBuffer implementation. What is the purpose of the mutex in the following? Can we do without the mutex? In what circumstances?

Producer(item) {

     emptySlots.P();

     mutex.P();

     Enqueue(item);

     mutex.V();

     fullSlots.V();

}

The mutex in this case regulates which items are able to reach the Enqueue statement, limiting it to one at a time. While the emptySlots and fullSLots objects are used for managing available space in the buffer (keeping to the bounds), the mutex specifically serves as a lock. The reason for this is that, due to the producer being called many times by many different items attempting to enqueue at the same time, there needs to be a limiter so that only 1 is able to enter at a time, also giving the semaphores time to process the individual items. If this were not in place, multiple items would attempt to attempt to enqueue at the same time, destabilizing the bounded buffer and moving towards a race condition which would eventually crash the program. The mutex could only safely be removed if there were no multithreading used and this was just a single, in-line running bounded buffer.

**Question 2 [10 pts]:** The following is the Producer(.) function in a BoundedBuffer implementation. Can we change the order of the first 2 lines? Why or why not?

Producer(item) {

     emptySlots.P();

     mutex.P();

     Enqueue(item);

     mutex.V();

     fullSlots.V();

}

We cannot safely switch the first two lines. “emptySlots.P();” is required to ensure the integrity of the buffer. If the mutex was before the emptySlots check, this would unlock the Enqueue statement for \*any\* thread that passes emptytSlots.P(), as the mutex can only be closed in the thread it was started in. This could lead to a deadlock where a thread is waiting for a return from the emptySlots semaphore while it is locked behind another thread’s mutex.

**Question 3 [20 pts]:** If we run 5 instances of ThreadA() and 1 instance of ThreadB(), what can be the maximum number of threads active simultaneously in the Critical Section? The mutex is initially unlocked. Note that ThreadB() is buggy and mistakenly unlocks the mutex first instead of locking first. Explain your answer.

|  |  |
| --- | --- |
| **ThreadA(){        mutex.P()        /\* Start Critical Section \*/        …….        /\* End Critical Section \*/        mutex.V(); }** | **ThreadB(){    mutex.V()    /\* Start Critical Section \*/        …….    /\* End Critical Section \*/    mutex.P(); }** |

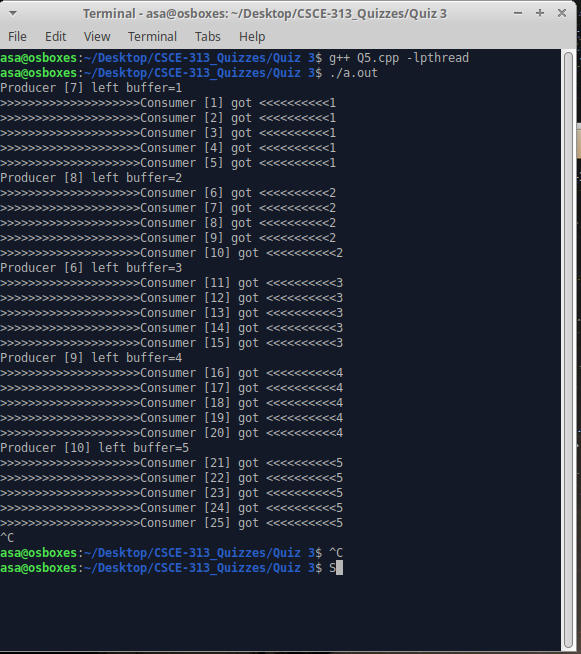
In thread A, since the mutex is properly implemented, only 1 thread can be in the critical section at a time. This is by design, as even though threads are intended to work at the same time, accessing critical memory and registers in multiple threads concurrently would destabilize things for all threads. If both A and B were used properly, they would each be able to enter their critical sections at the same time, with the hope being that they aren’t working on any of the same registers. However, with the introduction of the reversed implementation in ThreadB, the ordering is ruined and there could be many scenarios depending on timing. Unlocking an already unlocked mutex, as is done in ThreadB, causes undefined behavior. It could cause multiple concurrent critical access by ThreadA for a race condition, it could create a deadlock to where no threads can receive the signal to enter their critical section, or (at a very low chance) it might work normally, or any number of other outcomes. I would say that the worst case would likely be 3 threads in their critical section at the same time, 1 for ThreadB and two for ThreadA in the case that the undefined behavior went that way. If ThreadB was bugged in a way that did not cause undefined behavior, ThreadA would likely be unaffected by its issues as a mutex is normally only unlockable by the thread it was locked it, but said undefined behavior destabilizes such controls.

**Question 5 [25 pts]:** [Introductory text removed to make room for images]

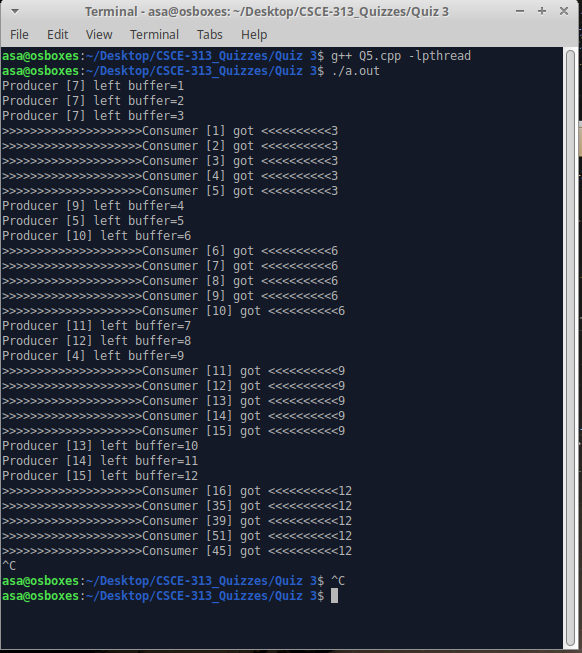
Look at the given program **1PNC.cpp** that works for 1 Producer and n Consumer threads. Be sure to run the program first to see how it behaves. You need to extend the program such that it works for m producers instead of just 1. Add necessary semaphores to the program. However, you will lose points if you add unnecessary Semaphores or Mutexes. To keep things simple, declare the mutexes as semaphores as well. You are given a fully implemented Semaphore.h class that you can use for Semaphores. Test your program to make sure that it is correct. In your submission directory, include a file called **Q5.cpp** that contains the correct program.

Code Completed to specifications. Attached in submission folder, filenames not altered.

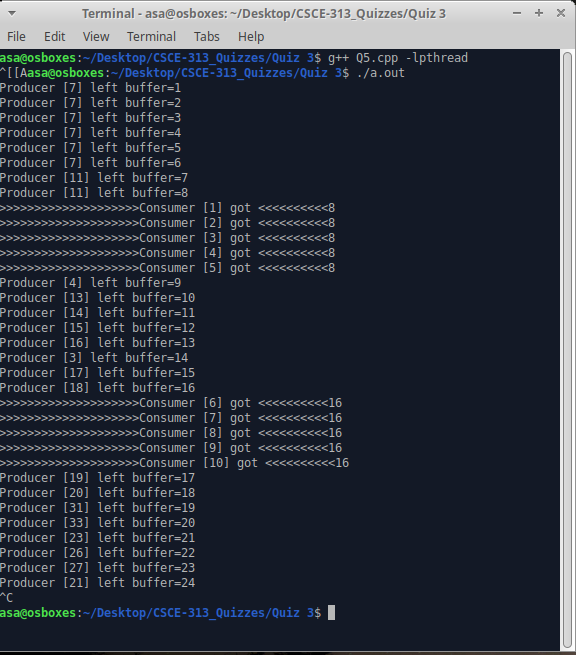
Result of running with NP=1, NC=5



Result of running with NP=3, NC=5



Result of running with NP=8, NC=5



**Question 6 [15 pts]:** There are 3 sets of threads A, B, C. First 1 instance of A has to run, then 2 instances of B and then 1 instance of C, then the cycle repeats. This emulates a chain of producer-consumer relationship that we learned in class, but between multiple pairs of threads. Write code to run these set of threads.

Assumptions and Instructions: There are 100s of A, B, C threads trying to run. Write only the thread functions with proper wait and signal operation in terms of semaphores. You can use the necessary number of semaphores as long as you declare them in global and initialize them properly with correct values. The actual operations done by A, B and C does not really matter. Submit a separate C++ file called **Q6.cpp** that includes the solution.

Q6.cpp included in submission folder.

**Question 7 (20 pts):** Implement a Mutex using the atomic **swap(variable, register)** instruction in x86. Your mutex can use busy-spin. Note that you cannot access a register directly other than using this swap instruction. [Note that there is no direct swap() instruction available in C/C++. So, we are expecting pseudocode instead of a fully functional code.]

Pseudocode included in submission folder as “Q7.txt”.