CSCE 313

QUIZ 3

Fall 2019

Name:…………………………………………………………… UIN:………………………………..

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 is there to make sure that race condition will not happen inside the critical section. We have to have the mutex, unless the program is single threaded.

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();
}
```

No, It can not be switched. Suppose there is no slot in the queue, and the mutex is locked so the consumer cannot consume as well, thus it will result in a dead lock.

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();
}
```

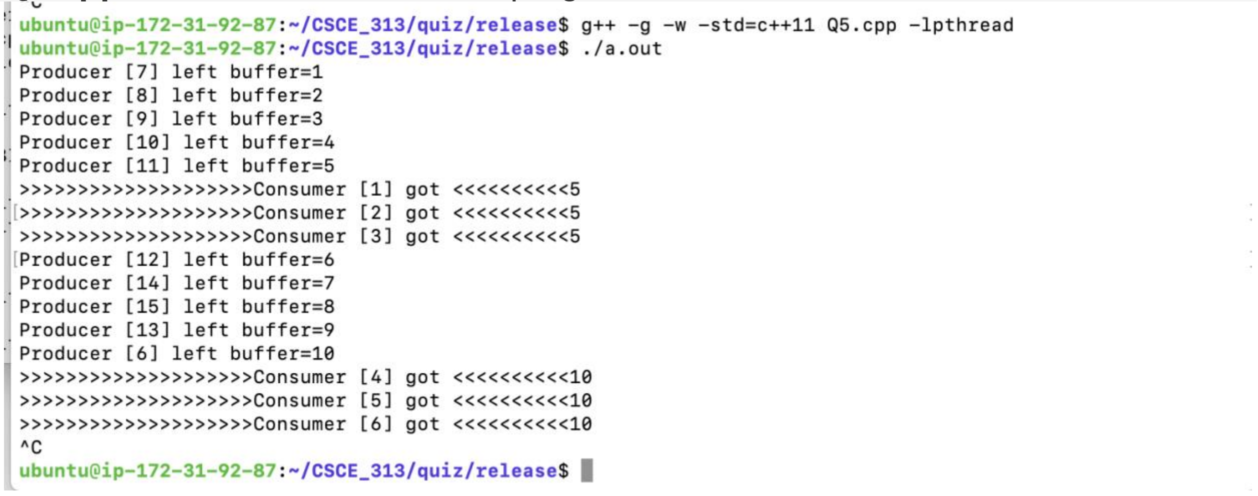
```
ThreadB(){
   mutex.V()
   /* Start Critical Section */
       …….
   /* End Critical Section */
   mutex.P();
}
```

It will have maximum of 3 thread running at the same time. Suppose one Thread A ran and locked mutex, then thread b immediately followed and it unlock the mutex, so there can be another threadA coming in and lock the mutex again. Thus, maximum of 3 thread can be running at the same time.

Question 5 [25 pts]: Consider a multithreaded web crawling and indexing program, which needs to first download a web page and then parse the HTML of that page to extract links and other useful information from it. The problem is both downloading a page and parsing it can be very slow depending on the content. Your goal is to make both these components as fast as possible. First, to speed up downloading, you delegate the task to **m** downloader threads, each with only a portion of the page to download. (Note that this is quite common in real life and a typical web browser does this all the time as long as the server supports this feature. Usually it is done through opening multiple TCP connection with the server and downloading equal sized chunk through each connection). The M chunks are downloaded to a single page buffer. Once all the chunks are downloaded into the buffer, you can then start parsing it. However, since you want to speed up parsing as well, you now use **n** parsing threads who again can parse the page independently, and together they take much less time.

By now, you probably see that M download threads are acting as Producers and N parser threads as Consumers. Additionally, note that the both downloader and parser threads come from a pool of M Producer threads and N Consumer threads where M>m and N>n. Out of many of these, you have to let exactly **m** Producer threads carry out the download and then exactly **n** consumer threads parse, and then the whole cycle will repeat. IOW, in each cycle, you are employing **m** out of **M** Producer worker threads (who are all eagerly waiting) to download the page simultaneously, and then **n** out of **N** Consumers are concurrently parsing the downloaded page. You cannot assume m=M or n=N. Assume that you can a function call download(URL) to download the page and parse (chunk) to parse a chunk of the page. No need to be any more specific/concrete than that. The main thing of interest is the Producer-Consumer relation.

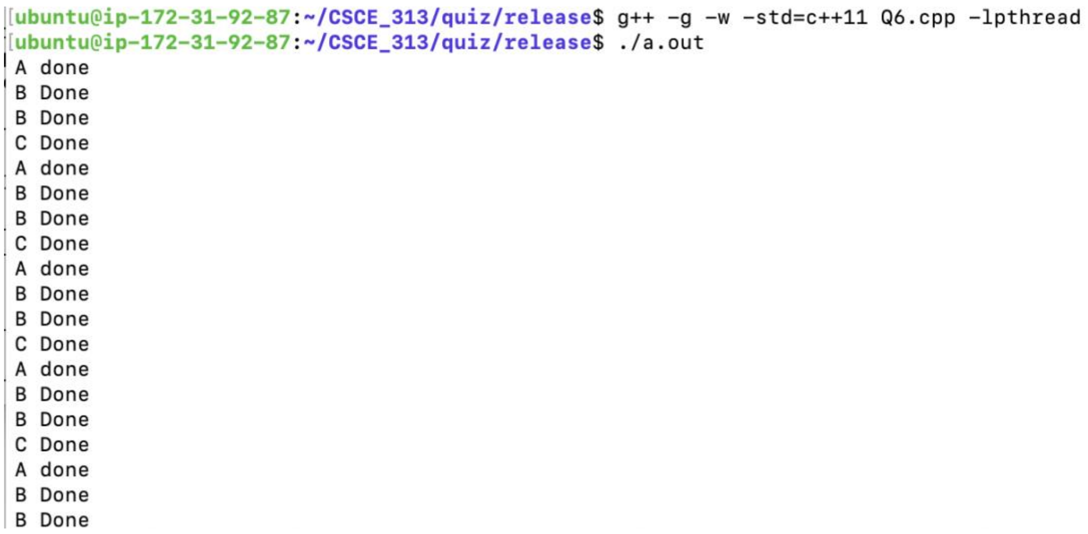
Look at the given program **1PNC.cpp** that works for 1 Producer and n Consumer threads. Be sure to run the program first to see how it behaves. You need to extend the program such that it works for m producers instead of just 1. Add necessary semaphores to the program. However, you will lose points 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.



```
ubuntu@ip-172-31-92-87:~/CSCE_313/quiz/release$ g++ -g -w -std=c++11 Q5.cpp -lpthread
ubuntu@ip-172-31-92-87:~/CSCE_313/quiz/release$ ./a.out
Producer [7] left buffer=1
Producer [8] left buffer=2
Producer [9] left buffer=3
Producer [10] left buffer=4
Producer [11] left buffer=5
>>>>>>>>>>>>>>>>>>>>Consumer [1] got <<<<<<<<<<5
>>>>>>>>>>>>>>>>>>>>Consumer [2] got <<<<<<<<<<5
>>>>>>>>>>>>>>>>>>>>Consumer [3] got <<<<<<<<<<5
Producer [12] left buffer=6
Producer [14] left buffer=7
Producer [15] left buffer=8
Producer [13] left buffer=9
Producer [6] left buffer=10
>>>>>>>>>>>>>>>>>>>>Consumer [4] got <<<<<<<<<<10
>>>>>>>>>>>>>>>>>>>>Consumer [5] got <<<<<<<<<<10
>>>>>>>>>>>>>>>>>>>>Consumer [6] got <<<<<<<<<<10
^C
ubuntu@ip-172-31-92-87:~/CSCE_313/quiz/release$
```

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.

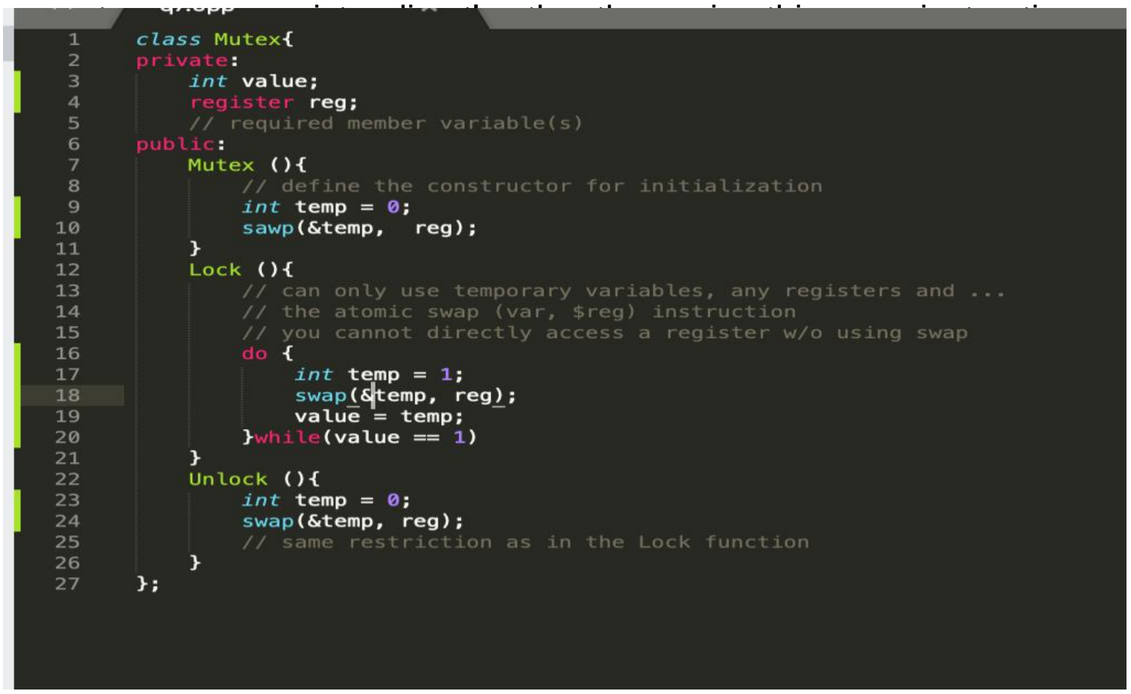
Assumptions and Instructions: There are 100s of A, B, C threads trying to run. Write only the thread functions with proper wait and signal operation in terms of semaphores. You can use the necessary number of semaphores as long as you declare them in global and initialize them properly with correct values. 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.



```
ubuntu@ip-172-31-92-87:~/CSCE_313/quiz/release$ g++ -g -w -std=c++11 Q6.cpp -lpthread
ubuntu@ip-172-31-92-87:~/CSCE_313/quiz/release$ ./a.out
A done
B Done
B Done
C Done
A done
B Done
B Done
C Done
A done
B Done
B Done
C Done
A done
B Done
B Done
C Done
A done
B Done
B Done
```

CSCE 313 # QUIZ 3 Fall 2019

<u>Question 7 (20 pts):</u> Implement a Mutex using the atomic **`swap(variable, register)`** instruction in x86. Your mutex can use busy-spin. Note that you [illegible]



```
class Mutex{
private:
    int value;
    register reg;
    // required member variable(s)
public:
    Mutex (){
        // define the constructor for initialization
        int temp = 0;
        sawp(&temp,  reg);
    }
    Lock (){
        // can only use temporary variables, any registers and ...
        // the atomic swap (var, $reg) instruction
        // you cannot directly access a register w/o using swap
        do {
            int temp = 1;
            swap(&temp, reg);
            value = temp;
        }while(value == 1)
    }
    Unlock (){
        int temp = 0;
        swap(&temp, reg);
        // same restriction as in the Lock function
    }
};
```