CSCE 313
Quiz 3
Jeffrey Xu
527008162
10/20/20

1. [20 pts] Assume the following processes A, B, C are loaded in memory of a system that uses multiprogramming. These processes have 13, 7 and 11 instructions respectively, Also assume that the dispatcher lives at address 100 in memory and spans 4 instructiors (i.e. 100-103). The following table shows only instruction addresses in the memory with I/O requests labelled, along with the duration of these I/O operations in terms of CPU instructions. Although I/O operations do not take CPU instructions, the duration means that the I/O operations will finish by the time the corresponding number of CPU instructions execute. Please draw a trace of these 3 processes running together in the CPU using Shortest Remaining Time First (SRTF) with preemption and no timer (i.e., we are not running a timer that is used by time sharing or round-robin). Recall that for SRTF, you only need to decide based on the next CPU burst, not the entire remaining time of the process. You can skip the first invocation of the dispatcher to decide the first process to run in the CPU.

Process A	**Process B**	**Process C**
5000	8000	12000
5001	8001	12001 (I/O, takes 3 ins.)
5002	8002	12002
5003	8003 (I/O, takes 7 ins.)	12003
5004 (I/O, takes 6 ins.)	8004	12004
5005 (I/O, takes 4 ins.)	8005	12005
5006	8006	12006 (I/O, takes 1 ins.)
5007		12007
5008 (I/O, takes 5 ins.)		12008 (I/O, takes 2 ins.)
5009		12009
5010		12010
5011		
5012		

Solution:

We must decide which process to run next based on each processes CPU bursts from previous CPU cycles. The shortest CPU remaining time will be the number of instructions till the next I/O instruction. Also, since we are using preemptive STRF, anytime a new process gets put into the ready queue, we must do a context switch. We also assume that A, B and C all are loaded at the beginning. We also are doing a context switch everytime a process goes from the blocked queue to the ready queue.

1. 12000
2. 12001

—-I/O—-

3. 100
4. 101
5. 102
6. 103
7. 8000
8. 8001
9. 8002
10. 8003

—-I/O—-

11. 100
12. 101
13. 102
14. 103
15. 12002
16. 12003
17. 12004

—-New Process in—-

18. 100
19. 101
20. 102
21. 103
22. 12005
23. 12006

—-I/O—-

24. 100
25. 101
26. 102
27. 103
28. 12007
29. 12008

—-I/O—-

```
30. 100
31. 101
32. 102
33. 103
34. 12009
35. 12010
—-I/O—-
36. 100
37. 101
38. 102
39. 103
40. 8004
41. 8005
42. 8006
—-I/O—-
43. 100
44. 101
45. 102
46. 103
47. 5000
48. 5001
49. 5002
50. 5003
51. 5004
—-I/O—-
52. 100
53. 101
54. 102
55. 103
56. No-op
57. No-op
—-New Process in—-
58. 100
59. 101
60. 102
61. 103
62. 5005
—-I/O—-
63. 100
```

64. 101
65. 102
66. 103
—-New Process in—-
67. 100
68. 101
69. 102
70. 103
71. 5006
72. 5007
73. 5008
—-I/O—-
74. 100
75. 101
76. 102
77. 103
78. No-op
—-New process in—-
79. 100
80. 101
81. 102
82. 103
83. 5009
84. 5010
85. 5011
86. 5012

2 [5 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();
}
```

Solution:

The mutex in this function protects the critical section of enqueuing an item onto the queue for the BoundedBuffer. It makes sure that the machine is free to operate on the buffer and makes sure that there are no race conditions. We could do without the mutex only in the case that there is one producer. If there are multiple producers, then you must need a mutex to prevent race conditions.

3 [5 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();
}
```

Solution:

You cannot change the order of the first two lines because you may cause a deadlock. If the producer locks the mutex, then no other thread can access the CPU. Then, if the producer needs for emptySlots to increment, it must depend on a consumer. However a consumer cannot access the queue since the producer locked the mutex so a deadlock would occur between the producer and consumer.

4 [10 pts] If we run 5 instances of ThreadA() and 1 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() {
        muetx.P()
        /* Start Critical Section */
        ......
        /* End Critical Section */
        mutex.V()
}


ThreadB() {
        mutex.V()
```

```
    /* Start Critical Section */
    .....
    /* End Critical Section */
    mutex.P()
}
```

Solution:

The maximum number of threads in the critical section is three. Assume that we run an instance of ThreadA first. This instance will lock the mutex and start running the critical section. Now, assume that we encounter a context switch in the middle of the critical section of this instance of ThreadA and now start running an instance of ThreadB. Now ThreadB will unlock the mutex and start running the critical section making it 2 threads in the critical section. Then, assume that a context switch occurs in ThreadB and we run a new instance of ThreadA. Now this new instance of ThreadA will lock the mutex and start running the critical section. This gives us 3 individual threads running the critical section simultaneously. We see that if we make any other context switches, it'll either switch to one of the three currently running threads and run those or switch to a new instance of ThreadA which has to wait for the mutex to unlock before it can enter the critical section. Therefore, the maximum number of threads that can be in the critical section for this problem is 3.

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 connections with the server and downloading equal sized chunks 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. You cannot assume m=M or n=N. Assume that you can call the function 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 that launches hundreds of producer and consumer threads. 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 instances of the semaphore class given in Semaphore.h. Test your program to make sure that it is correct. Include a file called Q5.cpp in your submission that contains the correct program.

Here is the expected output with m=3 and n=5:

```
Producer [1] left buffer=1
Producer [3] left buffer=2
Producer [2] left buffer=3
>>>>>>>>>>>>>>>>>>>>Consumer [1] got <<<<<<<<<<3
>>>>>>>>>>>>>>>>>>>>Consumer [2] got <<<<<<<<<<3
>>>>>>>>>>>>>>>>>>>>Consumer [5] got <<<<<<<<<<3
>>>>>>>>>>>>>>>>>>>>Consumer [3] got <<<<<<<<<<3
>>>>>>>>>>>>>>>>>>>>Consumer [4] got <<<<<<<<<<3
Producer [5] left buffer=4
Producer [3] left buffer=5
Producer [6] left buffer=6
>>>>>>>>>>>>>>>>>>>>Consumer [6] got <<<<<<<<<<6
>>>>>>>>>>>>>>>>>>>>Consumer [7] got <<<<<<<<<<6
>>>>>>>>>>>>>>>>>>>>Consumer [8] got <<<<<<<<<<6
>>>>>>>>>>>>>>>>>>>>Consumer [10] got <<<<<<<<<<6
>>>>>>>>>>>>>>>>>>>>Consumer [9] got <<<<<<<<<<6
// ETC.
```

Solution:

File is attached.

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 this 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.

Solution:

File is attached.

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.]

Solution:

```
class Mutex {
private:
    const int LOCKED = 1;
    const int UNLOCKED = 0;
    int reg = 1;
    int temp = 0;
public:
    lock() {
        while(true) {
            swap(temp, reg);
            if(temp == UNLOCKED) {
                swap(temp, reg);
                break;
            } else{
                swap(temp, reg);
            }
        }
    }
    unlock() {
        temp = UNLOCKED;
        swap(temp, reg);
    }
```

}