Name: Abburi Venkata Sai Mahesh Roll No.: CS18BTECH11001

Quiz1

CS5300: Parallel and Concurrent Programming Autumn 2020

Instructions: Please write the answers in the question paper itself. If required, please take extra sheets.

Q1 **(5 points)**. In the class, we discussed that the proof of composability of Linearizability, i.e. Theorem 3.6.1. Where will this theorem break down while considering Sequential Consistency instead of Linearizability?

A. It violates the second condition that if method call m0 precedes method call m1 in H, then the same is true in S. This is because the reordering of method executed by different threads may happen and leading to the case that m1 may precedes m0 in the secquential execution. To understand this better let us assume that thread A executes m0 and thread B executes m1 with not overlapping intervals. Then in sequential execution we can also consider the either order(m1 → m0 can be true) which cannot be acceptable by the second condition said as a part of the theorem, therefore leading to declare that it is not composable.

Q2 **(8 points)**. Lamport's Bakery mutual exclusion algorithm assumed sequential consistency being provided by the underlying system (i.e., the OS). If the OS does not provide will the algorithm still work correctly? If yes, please give justification. Otherwise, please give a counter-example.

A. No, the bakery algorithm does not ensures mutual exclusion if the OS does not provide sequential consistency. Let us consider the following example.

Consider that the OS in not sequential and now consider the execution format of line 13, 14, 15 in the algorithm given in text book for 2 treads A, B in the following format:

1. Line 14 by thread A → label[A] = 1.
2. Line 15 by thread A → while loop breaks as flag[A] & flag[B] is still false.
3. Line 14 by thread B → label[B] = 2.
4. Line 15 by thread B → while loop breaks as flag[A] & flag[B] is still false.
5. Line 13 by thread A → flag[A] = true → return from the lock() method and enter critical section.
6. Line 13 by thread B → flag[B] = true → return from the lock() method and enter critical section.

This order indicates that both the thread A, B will enter into critical section at the same time violating the mutual exclusion principle. This examples proves that if the OS does not provide sequential consistency, the bakery algorithm doesn't work well.

Q3 **(8 points)**. Consider the following implementation of a simple multi-threaded Queue

Listing 1: Multi-Threaded Queue

```
1
2  // Multi-threaded Queue Object
3  public class SimpQueue<T> {
4      int head, tail;
5      T[] items;
6      ShObj checker;
7      static final int CAPACITY = 1024;
8
9      public Lock Based Queue (int capacity){
10         head = 0; tail = 0;
11         checker = new ShObj ();
12         items = (T[]) new Object[capacity];
13     }
14
15     public void enq (T x) throws FullException{
16         checker.Set();
17
18         try{
19             if (tail-head == items.length)     // Check if the Q is
Full
20                 throw new FullException();
21             items[tail % items.length] = x;
22             tail++;
23            }finally
24             checker.Unset();
25         }
26     }
27
28     public T deq ()
29     checker.Set();
30         try{
31             if (tail == head)                  // Check if the Q is Empty
32                 throw new Empty Exception ();
33             T x = items[head % items.length];
34             head ++;
35             return x;
36            }finally
37             checker.Unset();
38         }
39     }
40 }
41
42   // Shared Object
43 public class ShObj {
44
```

```
45      boolean shared;
46
47      ShObj () {
48          shared = false;     // Initialize
49      }
50
51      void Set() {
52          while (TAS(shared))       // Atomic Test & Set instruction in invoked
53              ;/* do nothing */
54      }
55
56      void Unset () {
57          shared = false;
58      }
59
60      // Atomic Test & Set instruction provided by the harware
61      boolean TAS( boolean target) {
62          boolean rv = target;
63          target = true;    // Set target value to true
64          return rv ;
65      }
66  }
```

It can be seen that this code is a simple modification of concurrent queue based on global lock as discussed in the class. This queue implementation uses an object called ShObj which internally uses an atomic hardware instruction: Test and Set (TAS). The implementation of TAS is also shown. Note that TAS is executed atomically.

Question: Is the above queue implementation

(a) Non-Blocking? If so, wait-free or lock-free

(b) Blocking? If so, starvation-free or deadlock-free.

Please justify your answer.

A. Blocking:

Let us consider two threads A, B where A is performing enq(x) and B is performing deq(x). If thread A acquires the lock() the shared variable value is set to true. If B is trying to acquire the lock before the enq(x) by thread A completes(before A calls unlock()), then B is stuck in the while loop of while(TAS(shared)) and couldn't acquire it and only exits when the thread A calls the unlock() method. This implies that thread B is unable to acquire lock and blocked as long as thread A is delayed which shows that the implementation is blocking.

Not Starvation-free:

Let us consider two threads A, B are trying to perform some enqueue and dequeue operations which finally involves a series of lock() and unlock() methods. Let us consider the following implementation order(assume no exceptions occurred):

1. B calls lock() method as the shared is false it sets shared to true and acquires lock .
2. context switch to thread A calls lock() but stuck in while loop as shared is true.
3. B unlock() sets shared to false.

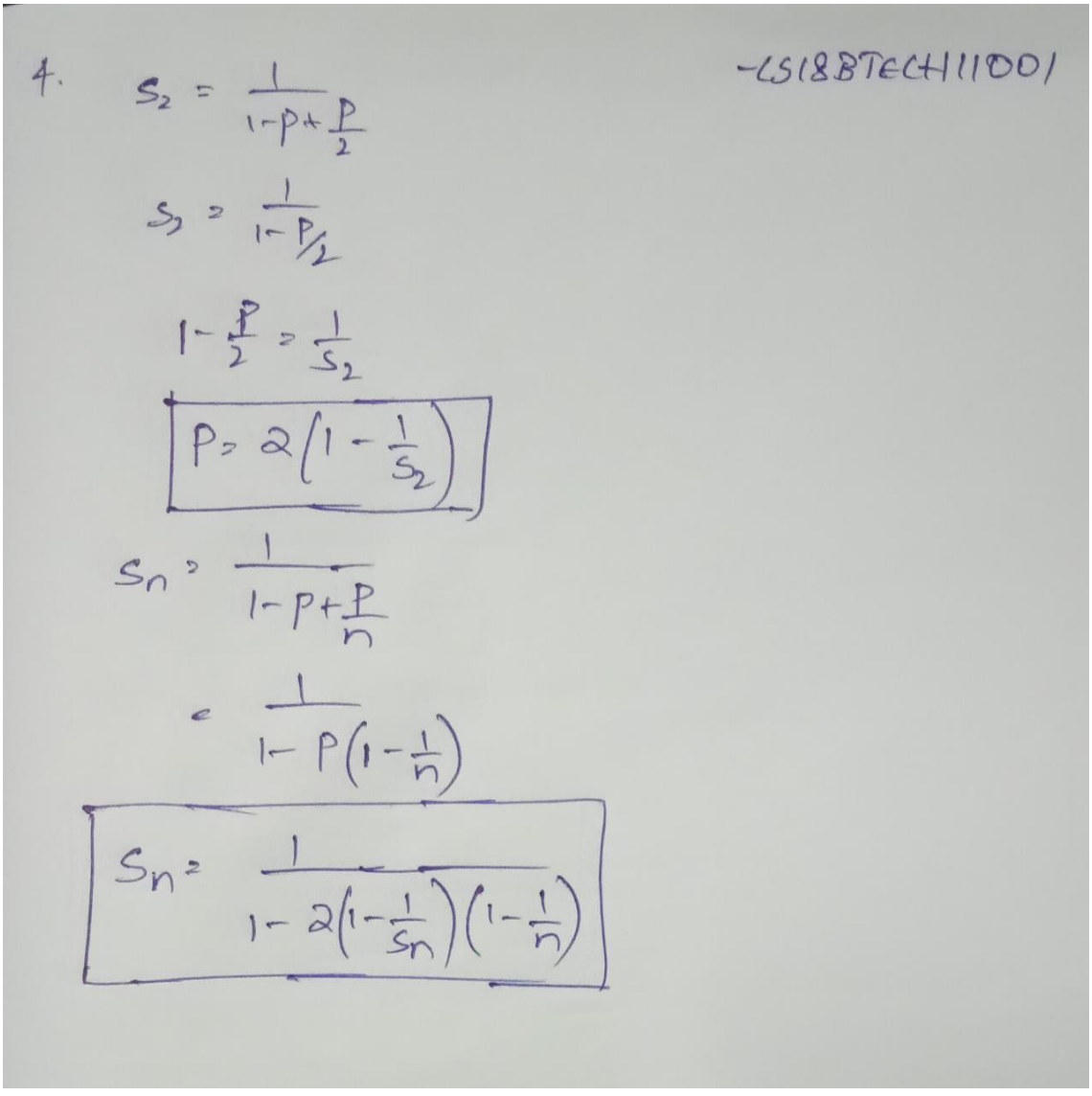
When the scheduling is done recursively in the following order $1 \rightarrow 2 \rightarrow 3 \rightarrow 1 \ldots$ then even though the thread A is given scheduling their isn't any progress(A is not able to acquire the lock for a long time) seen from it as it is shuck in the while loop and is getting staved. So the above implementation is not starvation free.

Deadlock-free:

Let us consider that a deadlock occurs - at any point of time no thread is making progress(all threads are stuck in the while loop). Then, let us consider the first condition that the shared variable is set from false to true(as it is initially false and some thread should eventually set it to true) which implies that this thread breaks the while loop before and acquires the lock. Then it should eventually call the unlock() method as the try finally block is used in the code which ensures that even if there is any exception the unlock method is called. Which states that the shared variable will be set to zero. And then any other thread follows the same scenerio. In the whole case we have seen that atleast one thread is making the progress(not all thread are stuck waiting) which leads to a contradiction to our assumption and therefore ensuring that the implementation is deadlock-free.

Q4 **(5 points)**. Running your application on two processors yields a speedup of $S2$. Use Amdahl's Law to derive a formula for Sn , the speedup on n processors, in terms of n and $S2$.

A.



Q5 **(5 points)**. Show that the Filter lock allows some threads to overtake others an arbitrary number of times.

A. The above condition can happen when a thread is not allocated the time slice for a long time. Let us consider 3 threads A, B, C with none of them haven't yet entered for loop. Now consider the following order of execution:

1. C acquires lock() (surpassed all n-1 levels).
2. context switch to thread B calls lock() and entered level 1 while setting victim to B.
3. context switch to thread A calls lock() and entered level 1, rewriting the victim at level 1 to A.
4. context switch to thread C and calls unlock().
5. thread C calls lock() and entered level 1, rewriting the victim at level 1 to C.
5. context switch to thread B finishes all levels and acquire the lock().
7. thread B and calls unlock().
8. thread B calls lock() and entered level 1, rewriting the victim at level 1 to B.
9. context switch to thread C finishes all levels and acquire the lock().

When the scheduling is done recursively as $4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 4 \rightarrow \ldots$. Then the every time thread B, C will overtake A for many number of times.