Name: Abburi Venkata Sai Mahesh Roll No.: CS18BTECH11001

Mid Semester Exam
CS5300: Parallel and Concurrent Programming
Autumn 2020

Instructions: (1) You must submit your final answer copy as pdf. (2) You should avoid submitting scan copies of hand-written notes. Only if you wish to attach any figure, you can attach the scans of the figures in your pdf.

Q1 (5 points). Consider a solution to the consensus protocol described in Listing 1. The proposed array consists of all integers and is initialized to 0. Each thread proposes an integer value. To decide on a value, each thread computes the product of all the values proposed. Then the thread computes the mod of the product with the total number of processes N. Let the result of mod operation be retIndex. The thread returns the value proposed by thread retIndex. The following code illustrates this

Listing 1: Product Consensus

```
1  public class ProdConsensus<T<int>> extends ConsensusProtocol<T<int>> {
2
3      public T<int> decide(T<int> Value) {
4
5          // Assume that the proposed array initially had value 0
6          propose(value);
7
8          int prod = 1;
9                                      {
10         for (int i=0; i<N; i++)
11             prod = proposed[i];          // Compute the product of all the values proposed
12         }
13
14         int retIndex= prod%N;
15
16         return proposed[retIndex];
17     }
18 }
```

Does this algorithm satisfy (a) consistency (b) validity. Please justify your answers.

A.

Consistency: False

When the decide() is called by threads, as the initial value of proposed is 0 all threads will return 0 until the first call of last thread modifies its proposed value which then return a non zero value and may be different for different threads. So the the decide returns different values for different threads and thus not satisfying considtency.

Validity: False

As all the threads may not agree upon a common value as proved in consistency, it is irrelavent to talk about Validity. So it is better to declare the algorithm as invalid.

Q2 (9 points). Consider the executions shown in Figure 1, Figure 2, Figure 3. Histories, H1; H2 operate on a variable (register) x. The sequential behavior of a register has already been discussed in the class.

While, H3 operates on a hash table h1. Its consists of add and lookup methods on different keys. A lookup on a key k returns the value stored for key k. An add on k with value v over-writes the previous value with v.

Which of these histories are linearizable? If the history is linearizable, please give the equivalent sequential history. If not, please modify the history to make it linearizable and show the modifications.

A.

The Histories H1, H2 are linearizable.

Sequencial history of H1:

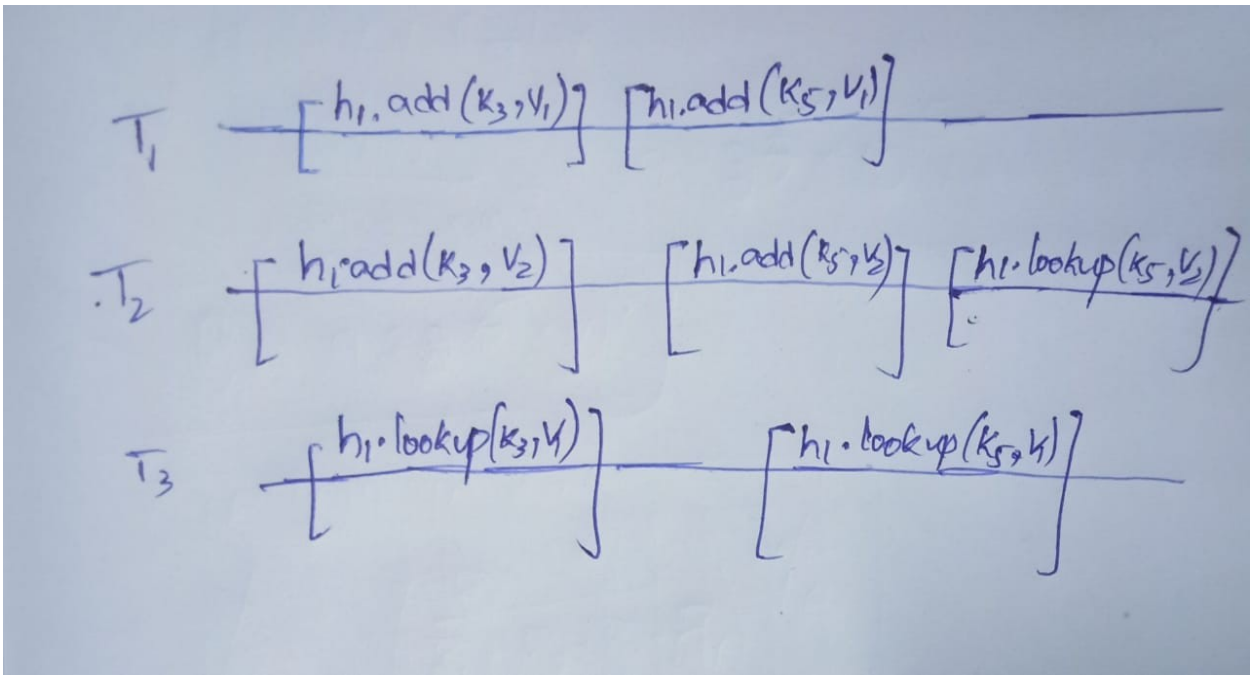
<w2(x,5) T2> -> <r1(x,5) T1> -> <w3(x,10) T3> -> <r2(x,10) T2>

Sequencial history of H2:

<w2(x,5) T2> -> <r1(x,5) T1> -> <r2(x,10) T2> -> <w3(x,10) T3>

The History H3 is non linearisable:

Converting to linearisable history: To change the history to linearisable change the sequenctial time part of T3 thread as shown in the figure.



Sequencial History of H3 after converting:

<h1.add(k3,v2) T2> -> <h1.add(k3,v1) T1> -> <h1.lookup(k3,v1) T3> -> <h1.add(k5,v1) T1> -> <h1.lookup(k5,v1) T3> -> <h1.add(k5,v2) T2> -> <h1.lookup(k5,v2) T2>

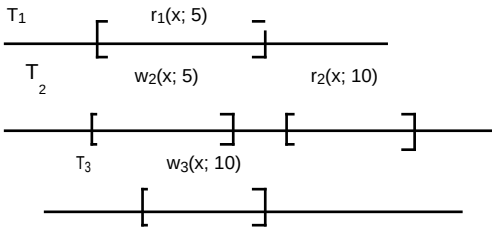


Figure 1: History H1

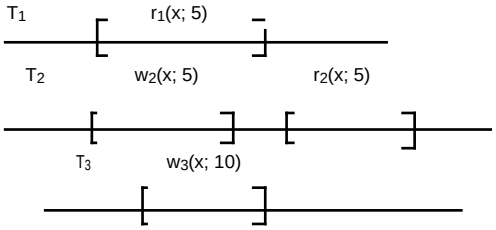


Figure 2: History H2

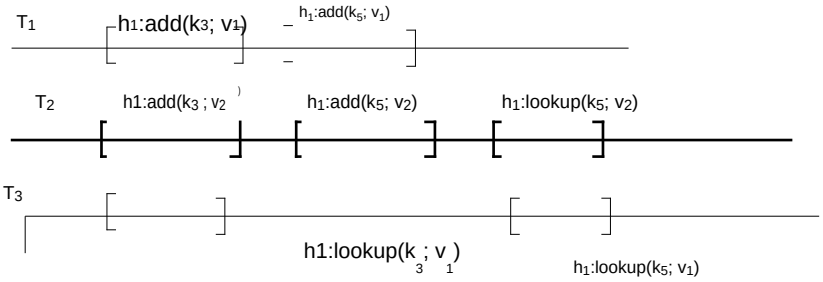


Figure 3: History H3

Q3 (7 points). Lamport's Bakery mutual exclusion algorithm assumed that all the shared registers are atomic. But will the algorithm work if the shared atomic registers are replaced by regular registers? Please give justification or give counter-example.

A. **True:** Lamport's Bakery Mutual Exclusion algorithm provides both mutual exclusion and FCFS even when the shared atomic registers are replace witth regular registers.

The difference between regular registers and atomic registers is that atomic register couldn't switch between old and new values when read during a write().

Now consider read and instructions in algorithm:

1. flag[i] = true; - write
2. label[i] = max(label[0], ...,label[n-1]) + 1; - read & write
3. while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {}; - read

We can see that the algorithm reads in both 2&3 where as writes in 1&2 lines.

Mutual Exclusion:

Suppose processes 1 and 2 are concurrently in the critical section. Let a and b be the last label weights. Suppose that (a, 1) << (b, 2).

When process 2 successfully completed the test in its waiting section, it must have read that flag[1] was false or that (b, 2) << (a, 1).

If process 2 read that flag[1] was false, then that read preceded or overlapped process 1's write to flag[1], which preceded process 1's read of flag[2] and write to flag[1], implying that label[1] > label[2], a contradiction.

So process 2 observed that (label[2], 2) << (label[1], 1). Since process 1 never wrote such a value, process 2 must have read label[1] at the time process 1 was updating it. But process 2 must have seen either the value being written, or the previous value, both of which are less than or equal to label[2], a contradiction.

FCFS:

Suppose processes 1 and 2 get labels a and b with a < b. Process 2 is faster and reaches the while loop and reads label[1] in the while loop at the same time process 1 writes to it. Process 2 can only read the value being written or the previous one, both of which are less or equal to b, so it will not enter the critical section, maintaining the FCFS property.

Q4 (8 points). Consider the following implementation of m-valued Atomic MRSW register. It is built from a Atomic MRSW boolean register and two Regular MRSW registers. You can assume that you already have Atomic MRSW boolean and Regular MRSW registers (In the class, we had discussed the construction of these registers).

Listing 2: m-valued Atomic MRSW register

```
1  public class AtomicMRSWRegister implements Register<Byte> f
2      private static boolean LEFT = 0;
3      private static boolean RIGHT = 1;
4
5      RegMRSWRegister leftReg, rightReg;          // Regular MRSW Registers
6      AtomicBooleanMRSWRegister gFlag;            // Atomic boolean flag
7
8      ThreadLocal<boolean> locFlag;               // Local value of gFlag used by the writer thread
9
10     public AtomicMRSWRegister() {
11         gFlag = LEFT;
12         locFlag = LEFT;
```

```
13      }
14      public void write(Byte x) {
15
16          if (locFlag == LEFT) {          // check if locFlag is LEFT
17              // In this case, set the rightReg
18              locFlag = RIGHT;
19              gFlag = RIGHT;
20              rightReg = x;
21          }
22          else {          // locFlag is RIGHT
23              // In this case, set the leftReg
24              locFlag = LEFT;
25              gFlag = LEFT;
26              leftReg = x;
27          }
28      }
29      public Byte read() {
30
31          if (gFlag == LEFT) {            // check if gFlag is LEFT
32              return leftReg;
33          }
34          else {          // gFlag is RIGHT
35              return rightReg;
36          }
37      }
38  }
```

As one can see, this implementation has two regular registers. The idea is that reader threads read from one register (for instance left register) while the write thread writes to the other register (which is right register in this case). Is this implementation correct? Please justify your answer by giving proof of regularity if you think it is correct. Otherwise, give a counter-example.

A. **False:** The above implementation is not a correct implementation for a m-valued MRSW atomic register.

Counter Example:

Let us consider that two threads A(writer thread), B(reader thread) are performing the following operations on a register.

1. A writes 10
2. A writes 20
3. A writes 30 and B reads concurrently

When the implementation is correct the m-valued MRSW register should return either 20(older value) or 30(newer value).

Now cosider the following order of execution:

1. A in line 16 – if condition is true
    1. line 18 - locFlag = Right
    2. line 19 - gFlag = Right
    3. line 20 - rightReg = 10;
2. A in line 16 – if condition is false
    1. line 24 - locFlag = Left
    2. line 25 - gFlag = Left
    3. line 26 – leftReg = 20;
3. A in line 16 – if condition is true
    1. line 18 - locFlag = Right
    2. line 19 - gFlag = Right
4. B in line 31 – if condition is false

1. line 35 – return **rightReg(10)**

5. A in line 20 – rightReg = 10

From the above implementation we can see that the B is returning 10 (reading the value of 10) which violates the actual functionality of m-valued MRSW register(should return either 20 or 30).

Q5 (10 points). Listing 3 provides an implementation of a stack object using compareAndSet() objects (that is, objects supporting the operations compareAndSet() and get()). It uses Java syntax.

Listing 3: Assign23 Implementation

```
1  import java.util.concurrent.atomic.;
2
3  @Th readSa fe
4  publicclassConcurrentStack <E> f
5
6       // The following line is Java Syntax. It is not necessary for you to undestand it completely
7       AtomicReference<Node<E>> top = new AtomicReference<Node<E> >();
8
9       // Push into the system
10      public void push(E item) f
11           Node<E> newHead = new Node<E>(item);        // Create a new head
12           Node<E> oldHead;
13           do f
14                oldHead = top.get(); // Read the old head
15                newHead.next = oldHead; // Set the next in the new head
16           gwhile (!top.compareAndSet(oldHead, newHead)); // Perform the CAS statement
17      g
18
19      // Pop from the system
20      public E pop() f
21           Node<E> oldHead;
22           Node<E> newHead;
23           do f
24                oldHead =top.get();      // Read the top
25                if (oldHead == null) f
26                     return null;            // Return null
27                g
28                newHead = oldHead.next; // Reference to the new head
29
30           // Atomically set the new head
31           gwhile (!top.compareAndSet(oldHead, newHead));
32           return oldHead.item;
33      g
34
35      // Node class used here.
36      private static class Node <E> f
37           public final E item;
38           public Node<E> next;
39                Node(E item) f
40           public
41                this.item = item;
42           g
43      g
44 g
```

Please answer the following:

(a) Correctness: Please explain if this implementation is correct, i.e., linearizable. If it is correct, please explain the linearization points along with the justification. If not, please give a counter-example.

(b) Progress: What is the progress criteria satisfied by this stack object? Is it blocking or non-blocking? Then specifically mention if it is: deadlock-free or starvation-free (blocking) or lock-free or wait-free (non-blocking).

A. **(a) True:** The above implementation of stack object using compareAndSet() object is linearisable.

Linearisation Point:

The linearisation point for the push() is in the line 16. after the update of old and new head, do-while(compareAndSet()) loop takes the responsibility of pushing only one element at a time and thus solving race condition.

The linearisation point for the pop() is in the line 26 if the stack is empty else in the line 32. The return value is the only dependent line where the effect takes place and thus serves as linearisation point.

(b) The implementation above is blocking, deadlock free but not starvation free.
Let us consider the following execution where A,B,C are trying to push the elements.
Let us consider the follwing order of execution:

1. A excutes until line 15.
2. B executes until line 15.
3. A pushes element to stack
4. C executes until line 15
5. B is still run in the loop as the stack head is updated when it calls compareAndSet().
6. C calles compareAndSet() but as the stack head is unchanged exits the loop pushing element to the stack.

Here the line 5 shows that the thread B is blocked by thread C and if it proceeds for a certain time provides starvation. But in the either case the threads make some progress and is thus deadlock free.

Q6 (7 points). In the class, we saw that a combination of fetch-and-add (faa) and test-and-set (tas) instructions on the same memory location can be used to solve n-thread binary consensus. Similarly atomic read, decrement and multiply on the same memory locations can also be used to solve n-thread binary consensus. For your reference the java code of read, decrement and multiply is shown here:

Listing 4: Shared Object supporting Test-and-Set (TAS) & Fetch-and-Add (FAA)

```
1        ShObj f
2  void
3
4        int shMem ; // the shared memory
5
6        // Constructor
7        shObj(int i) f
8              shMem = i ;
9        g
10
11       // Atomically read the memory location
12       int synchronized read() f
13             return shMem ;
14       g
```

```
15
16          / / Atomically gets the current value and decrements
17          v o i d  s y n c h r o n i z e d          d e c ( i n t  x )          f / / get-and-decrement
18                  / / Decrement ShMem by x
19                  shMem   = x ;
20          g
21
22          / / Atomically gets the current value and decrements
23          v o i d  s y n c h r o n i z e d          m u l t ( i n t          x )          f / / get-and-multiply
24                  / / Multiply ShMem by x
25                  shMem   =  x ;
26          g
27
28  g
```

Please explain how only using these instructions, one can solve n-thread binary consensus. You can assume that you know number of threads n for this solution. Note: Here the emphasis is again on binary consensus.