CS5300: Parallel and Concurrent Programming
Theory Assignment 2

Abburi Venkata Sai Mahesh - CS18BTECH11001

November 3, 2020

This document is generated by LaTeX

Q1 The book describes a wait-free queue implementation with only two threads: a single enqueuer/dequeuer. Suppose we add a new method `getElements()` as discussed follows:.

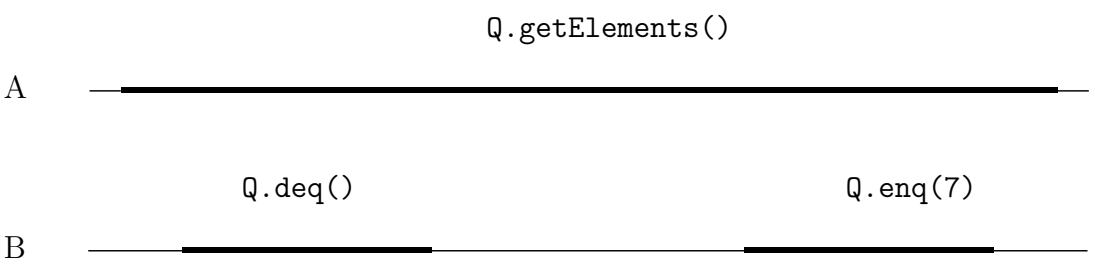
```
1  public T[] getElements () {
2    T[] rv = (T[]) new Object[capacity];
3
4    // Copy all the elments from head to tail
5    for (int i=head; i < tail; i++) {
6      rv[i] = item[i];
7    }
8
9    return rv;  // Return the copied elements
10
11 }
```

Now assume that you have only two threads that can execute two out of these three methods concurrently. Then, will the execution be linearizable? Please justify your answer.

A. Let an initial state of the queue be:

3	4	5	6

Let us assume two threads A, B are working on the present state. Now that thread A is calling `Q.getElements()` method and thread B is executing a `Q.deq()` and an `Q.enq(7)` to the queue concurrently.
Consider the Execution timeline as shown:



From the sequential exexution of B we can observe that

$$\text{<Q.deq() B>} \rightarrow \text{<Q.enq(7) B>} \qquad (1)$$

Consider the ouput returned by A to be the following queue:

3	4	5	6	7

We can observe that 3 is present in the output impling that the dequeue occurred after A reading head. So the A shall be executed before the dequeue of B.

$$\text{<Q.getElements() A>} \rightarrow \text{<Q.deq() B>} \qquad (2)$$

We can also observe that 7 is present in the output impling that the enqueue is completed before A reading modified tail. So A shall be executed after enqueue by B.

$$\text{<Q.enq(7) B>} \rightarrow \text{<Q.getElements() A>} \qquad (3)$$

When from (1), (2), (3) we can see that
<Q.getElements() A> → <Q.deq() B> → <Q.enq(7) B> → <Q.getElements() A>
The above execution is forming a cycle and thus is not linearisable.

Q2 Is the following property equivalent to saying that object x is lockfree?
For every infinite history H of x, an infinite number of method calls are completed.

A. *True*: The property is same as lockfree.
Let us assume that a thread acquires lock and executing critical section in x is preempted by the operating system and thus entering infinite history H of x. But the statement states that an infinite number of method calls are completed, which implies, other threads are able to run the method calls(entering critical section) which isn't possible if the object x is locked. Thus we can say that x is lockfree.

Q3 This exercise examines a queue implementation (Fig.1) whose `enq()` method does not have a linearization point.
The queue stores its items in an `items` array, which for simplicity we will assume is unbounded. The `tail` field is an `AtomicInteger`, initially zero. The `enq()` method reserves a slot by incrementing `tail`, and then stores the item at that location. Note that these two steps are not atomic: there is an interval after `tail` has been incremented but before the item has been stored in the array.
The `deq()` method reads the value of tail, and then traverses the array in ascending order from slot zero to the tail. For each slot, it swaps `null` with the current contents, returning the first non-null item it finds. If all slots are null, the procedure is restarted.
Give an example execution showing that the linearization point for `enq()` cannot occur at Line 15.
Give another example execution showing that the linearization point for enq() cannot occur at Line 16.
Since these are the only two memory accesses in `enq()`, we must conclude that enq() has no single linearization point. Does this mean enq() is not linearizable?

```
1  public class HWQueue<T> {
2    AtomicReference<T>[] items;
3    AtomicInteger tail;
4    static final int CAPACITY = 1024;
5
6    public HWQueue() {
7      items = (AtomicReference<T>[])Array.newInstance(AtomicReference.class,
8        CAPACITY);
9      for (int i = 0; i < items.length; i++) {
10       items[i] = new AtomicReference<T>(null);
11     }
12     tail = new AtomicInteger(0);
13   }
14   public void enq(T x) {
15     int i = tail.getAndIncrement();
16     items[i].set(x);
17   }
18   public T deq() {
19     while (true) {
20       int range = tail.get();
21       for (int i = 0; i < range; i++) {
22         T value = items[i].getAndSet(null);
23         if (value != null) {
24           return value;
25         }
26       }
27     }
28   }
29 }
```

Figure 1: Herlihy/Wing queue.

A. Let us consider thread A enqueues item x, B enqueues item y and thread C dequeues two times.
Now consider the following execution order:

1. A calling line 15, setting tail to 1 and returning $i_A = 0$
2. B calling line 15, setting tail to 2 and returning $i_B = 1$
3. B calling line 16, storing item y in items[1]
4. C finds items[0] as empty
5. C finds items[1] full and thus dequeues y
6. A calling line 16, storing item x in items[0]
7. C finds items[0] full and thus dequeues x

The above execution shows that the linearisation point of two `enq()` calls cannot occur at line 15.
Now consider the following execution order:

1. A calling line 15, setting tail to 1 and returning $i_A = 0$
2. B calling line 15, setting tail to 2 and returning $i_B = 1$
3. B calling line 16, storing item y in items[1]
4. A calling line 16, storing item x in items[0]

6. C finds items[0] full and thus dequeues x
7. C finds items[1] full and thus dequeues y

The above execution shows that the linearisation point of two `enq()` calls cannot occur at line 16.
The above executions resembles that we couldn't able to define a single linearisation point that would work for all method calls. This doesn't mean that the method is not linearisable.

Q4 Consider the safe Boolean MRSW construction shown in Fig. 2.
True or false: if we replace the safe Boolean SRSW register array with an array of regular M-valued SRSW registers, then the construction yields a regular M-valued MRSW register. Justify your answer.

```
1  public class SafeBooleanMRSWRegister implements Register<Boolean> {
2    boolean[] s_table; // array of safe SRSW registers
3    public SafeBooleanMRSWRegister(int capacity) {
4      s_table = new boolean[capacity];
5    }
6    public Boolean read() {
7      return s_table[ThreadID.get()];
8    }
9    public void write(Boolean x) {
10     for (int i = 0; i < s_table.length; i++)
11       s_table[i] = x;
12   }
13 }
```

Figure 2: The SafeBoolMRSWRegister class: a safe Boolean MRSW register.

A. *True*: If we replace the safe Boolean SRSW register array with an array of regular M-valued SRSW registers, then the construction yields a regular M-valued MRSW register.

If the `read()` call doesn't overlap with `write()` call, then the `read()` will return the recently written value stored in the s_table[i].

Let us consider a `read()` call is overlapping with k `write()` calls. Let x_0 be the value written by latest preceeding call and $x_1, x_2, x_3, ...x_k$ be the values to be written by the k `write()` calls respectively. Then the return value of read is given by the following cases:

(a) If the thread i reads s_table[i] after the writer writes x_i to s_table[i] but before writing x_{i+1}, then the `read()` call will return x_i.

(b) If the thread i reads s_table[i] while writing x_{i+1} to s_table[i], as the register is regular, the `read()` call will return either x_i or x_{i+1}.

Thus the construction resembles the M-valued MRSW register.

Q5 Does Peterson's two-thread mutual exclusion algorithm shown in Fig.3 work if the shared atomic flag registers are replaced by regular registers?

```
1  class Peterson implements Lock {
2    // thread-local index, 0 or 1
3    private boolean[] flag = new boolean[2];
4    private int victim;
5    public void lock() {
6      int i = ThreadID.get();
7      int j = 1 - i;
8      flag[i] = true;   // Im interested
9      victim = i;       // you go first
10     while (flag[j] && victim == i) {}; // wait
11   }
12   public void unlock() {
13     int i = ThreadID.get();
14     flag[i] = false;  // Im not interested
15   }
16 }
```

Figure 3: The Peterson lock algorithm.

A. *True*: The peterson's two-thread mutual exclusion algorithm work if the shared atomic flag registers are replaced by 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()`.

We can see that registers read only in line 10. Let us consider two threads are trying to acquire the lock at the sametime. Then the overlap occurs in line 8 and 9.

Consider A is working on line 10(completed lines 8,9) and B is working on line 8. Then, we can have the following cases:

(a) A reads the old value of flag[j](=false) and enters the critical section.

(b) A reads the new value of flag[j](=true) and still the victim is A so continues spinning.

When B is working on line 9(completing line 8), the case (b) is further having the following cases:

(a) A reads the old value of victim(=i) and spins until B finishes writing(line 9).

(b) A reads the new value of victim(=j) and enters the critical section.

In all the cases, for thread B as the flag[j] is true(set by thread A) and victim will become i(set by thread B in line 9), it continues spinning in line 10 while A enters critical section. So, the algorithm is correct even if we use regular registers.