100 points total.

Contents

```java
class VolatileExample {
    int x = 0;
    volatile boolean v = false;
    public void writer() {
        x = 42;
        v = true;
    }
    public void reader() {
        if (v == true) {
            int y = 100/x;
        }
    }
}
```

Listing 1: Volatile field example from section 1

1 Java memory model [6 points]

Consider the class shown in listing 1. According to what you have been told about the Java memory model, will the *reader* method ever divide by zero?

2 Sequential Consistency (1) [2 points]

The following is a history of a FIFO queue. The operations are enq(x)/void and deq()/x. Is this history a valid sequential history? Why not?

```
A: q.enq(x)
B: q.enq(y)
A: q:void
B: q:void
```

3 Sequential Consistency (2) [6 points]

P1	**P2**	**P3**
x = 1;	y = 1;	z = 1;
print(y,z);	print(x,z);	print(x,y);

The variables x, y and z are stored in a memory system that offers sequential consistency. Every operations including the print statements is atomic[1]. Explain whether the following are legal output sequences or not.

1. `001011`
2. `001111`
3. `001110`

For a legal output, show one possible interleaving of the instructions that leads to the said output. For an illegal output, explain why there is no possible interleaving that may leads to the said output.

4 Consistency (1) [6 points]

Consider a *memory object* that encompasses two register components. We know that if both registers are quiescently consistent, then so is the memory object. Does the converse hold? If the memory object is quiescently consistent, are the individual registers quiescently consistent? Outline a proof or give a counterexample.

5 Consistency (2) [6 points]

Give an example of an execution that is quiescently consistent but not sequentially consistent, and another that is sequentially consistent but not quiescently consistent.

6 Linearizability (1) [6 points]

The following is a history of a stack. The operations are push(x)/void and pop()/x. Is the history linearizable? If yes, then find a legal sequential history.

```
A: s.push(x)
B: s.push(y)
B: s:void
B: s.pop()
```

[1]no operation can overlap with other operations

```
A: s:void()
A: s.pop()
A: s:y
```

7 Linearizability (2) [6 points]

The following is a history of a FIFO queue. The operations are enq(x)/void and deq()/x. Is the history linearizable? If yes, then prove. Is it sequentially consistent?

```
A: r.enq(x)
A: r:void
B: r.enq(y)
A: r.deq()
B: r:void
A: r:y
```

8 Linearizability (3) [6 points]

The following is a history of a FIFO queue. The operations are enq(x)/void and deq()/x. Is the history linearizable? If yes, then prove.

```
A: q.enq(x)
B: q.enq(y)
A: q:void
B: q:void
A: q.deq()
C: q.deq()
A: q:y
C: q:y
```

9 Compositional Linearizability [6 points]

Proove the "only if" part of Theorem 3.6.1, reprinted below.

H is linearizable if, and only if, for each object x, $H|x$ is linearizable.

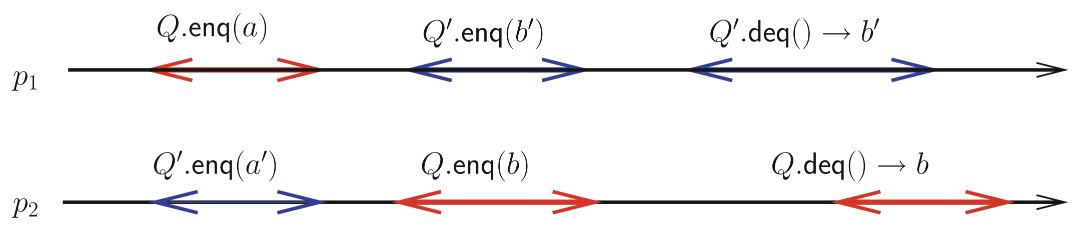

Figure 1: History H.

10 More histories (1) [10 points]

Figure 1 shows a History H with two processes: p_1 and p_2. The processes are accessing two concurrent queues Q and Q'. Answer the following questions:

1. Is $H|Q$ sequentially consistent?
2. Is $H|Q'$ sequentially consistent?
3. Is the entire history H sequentially consistent? If not, then why? If yes, then explain.

11 More histories (2) [6 points]

Assume you have a multiprocessor system with three processors. Each processor does not stall when it encounters a last-level cache (LLC) miss; i.e., each processor continues executing instructions that are independent of the data that needs to be fetched from DRAM due to the LLC miss.

W stands for write instruction. R stands for read instruction. Assume the data in the memory locations a and b is initialized to zero. The three processors execute the following sequence:

```
Processor 1:  W(a, 1); R(b, 0);
Processor 2:  W(b, 1); R(b, 1); R(a, 1);
Processor 3:  R(b, 1); R(a, 0);
```

Is this multiprocessor system sequentially consistent?

12 More Histories (3) [6 points]

Is this a linearizable execution?

Time	Task *A*	Task *B*
0	Invoke `q.enq(x)`	
1	Work on `q.enq(x)`	
2	Work on `q.enq(x)`	
3	Return from `q.enq(x)`	
4		Invoke `q.enq(y)`
5		Work on `q.enq(y)`
6		Work on `q.enq(y)`
7		Return from `q.enq(y)`
8		Invoke `q.deq()`
9		Return `x` from `q.deq()`

13 More Histories 4) [6 points]

Is this a linearizable execution?

Time	Task *A*	Task *B*
0	Invoke `q.enq(x)`	
1	Work on `q.enq(x)`	Invoke `q.enq(y)`
2	Work on `q.enq(x)`	Return from `q.enq(y)`
3	Return from `q.enq(x)`	
4		Invoke `q.deq()`
5		Return `x` from `q.deq()`

14 More Histories (5) [6 points]

Is this a linearizable execution?

Time	Task *A*	Task *B*
0	Invoke `q.enq(x)`	
1	Return from `q.enq(x)`	
2		Invoke `q.enq(y)`
3	Invoke `q.deq()`	Work on `q.enq(y)`
4	Work on `q.deq()`	Return from `q.enq(y)`
5	Return `y` from `q.deq()`	

15 AtomicInteger [7 points]

The `AtomicInteger` class (in the `java.util.concurrent.atomic` package) is a container for an integer value. One of its methods is `boolean compareAndSet (int expect, int update)`.

This method compares the object's current value to `expect`. If the values are equal, then it atomically replaces the object's value with `update` and returns `true`. Otherwise, it leaves the object's value unchanged and returns `false`. This class also provides `int get()`, which returns the object's actual value.

Consider the FIFO queue implementation shown in listing 2. It stores its items in an array `items`, which, for simplicity, we will assume has unbounded size. It has two `AtomicInteger` fields: `head` is the index of the next slot from which to remove an item, and `tail` is the index of the next slot in which to place an item. Give an example showing that this implementation is not linearizable.

16 Herlihy/Wing queue [9 points]

This exercise examines a queue implementation (listing 3) whose `enq()` method does not have a linearization point.

The queue stores its items in an `items` array, which for simplicity we will assume never hits `CAPACITY`. The `tail` field is an `AtomicInteger`, initially zero. The `enq()` method reserves a slot by incrementing `tail` and then storing 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.

```java
class IQueue<T> {
    AtomicInteger head = new AtomicInteger(0);
    AtomicInteger tail = new AtomicInteger(0);
    T[] items = (T[]) new Object[Integer.MAX_VALUE];
    public void enq(T x) {
        int slot;
        do {
            slot = tail.get();
        } while (!tail.compareAndSet(slot, slot+1));
        items[slot] = x;
    }
    public T deq() throws EmptyException {
        T value;
        int slot;
        do {
            slot = head.get();
            value = items[slot];
            if (value == null)
                throw new EmptyException();
        } while (!head.compareAndSet(slot, slot+1));
        return value;
    }
}
```

Listing 2: `IQueue` implementation

The `deq()` method reads the value of `tail`, 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 14[2]. Give another example execution showing that the linearization point for `enq()` cannot occur at line 15. 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,
              CAPACITY);
 8          for (int i = 0; i < items.length; i++) {
 9              items[i] = new AtomicReference<T>(null);
10          }
11          tail = new AtomicInteger(0);
12      }
13      public void enq(T x) {
14          int i = tail.getAndIncrement();
15          items[i].set(x);
16      }
17      public T deq() {
18          while (true) {
19              int range = tail.get();
20              for (int i = 0; i < range; i++) {
21                  T value = items[i].getAndSet(null);
22                  if (value != null) {
23                      return value;
24                  }
```

[2]Hint: give an execution where two `enq()` calls are not linearized in the order they execute line 14.

```
                }
            }
        }
}
```

Listing 3: Herlihy/Wing queue