Homework 3

Mathew Kurian (mk26473)
Kapil Gowru (krg766)

Question 1

To prove sequential consistency, we need to show that method calls appear to take effect in program order. In other words, we need to satify the logical sequence of program execution. For *Figure 3.13* we can arrange the methods so values are written and read while maintaing accuracy as follows:

```
r.write(1) >> r.read(1) >> r.write(2) >> r.read(2)
```

While for and *Figure 3.14* we can arrange them as follows:

```
r.write(1) >> r.read(1) >> r.read(1) >> r.write(2)
```

If you look at the figures, it is clearly possible to interleave the code to do the aforementioned cases thus both figures are sequentially consistent.

To prove linearizability, we must show sequential consistency and show that each operation appears to take effect atomically at some point between its invocation and completion. In other words, any read to a shared data item my return the value stored by the most recent write operation on that item. Upon analysis of the graph and taking into consideration the aforementioned sequential arrangements, we can see that both figures are also linearizable.

Question 2

This implementation is not linearizable. To prove this, consider the following case:

Time	Thread A	Thread B
1	`call enq(X)`	
2	`slot=0`	`call deq(X)`
3		`throws Exception`
4	`item[slot]=X`	
5	`return`	

In the case of `IQueue`, the write is not an atomic operation. With the linearizability constraint, every read expects to get the value of last write. With the Exception being thrown at `Time 4`, the queue fails to uphold the linearizability constraint.

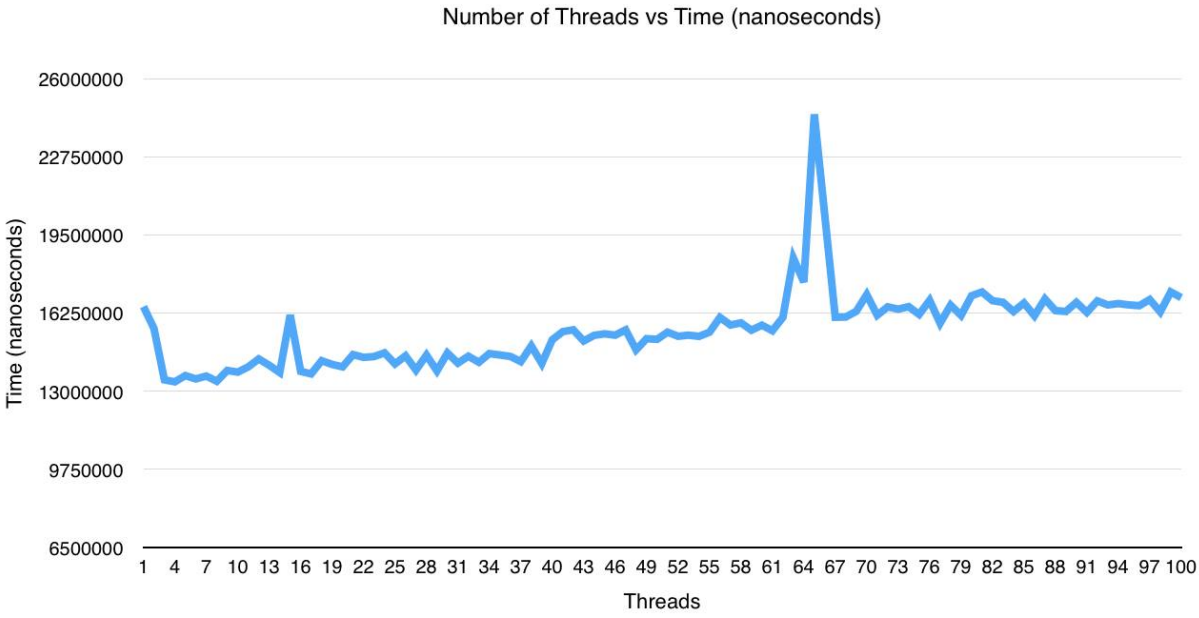
Question 3

Yes. It is possible for an `ArithmeticException`.

Notice that `int x` is not set as `volatile` which means that threads will access their cached values instead of directly accessing the memory. In other words, if `read()` is called, the new value of `x` (which in this case is `42`) can sometimes be propogated into other threads long after `v` (a volatile variable) is set to `true`. As a result, a thread that calls `write()` immediately can actually read `x` as `0`.

Question 4

A.Graph


Number of Threads vs Time (nanoseconds)
Time (nanoseconds)
26000000
22750000
19500000
16250000
13000000
9750000
6500000
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100
Threads