

YSC4231: Parallel, Concurrent and Distributed Programming

Theory Assignment 3

In this theory assignment, some of the problems refer to an accompanying repository.¹ Feel free to fork it and play with the code. For your solution, though, you only have to submit a PDF with explanations and proofs — no code is required.

Problem 1. The `AtomicInteger` Java class (in the `java.util.concurrent.atomic` package) is a container for an integer value. One of its methods is

```
def compareAndSet(expect: Int, update: Int): Boolean.
```

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 `get() : Int` which returns the object's actual value.

Consider the FIFO queue implementation given by the class `IQueue`. It stores its items in an array `items`. It has two `AtomicInteger` fields: `tail` is the index of the next slot from which to remove an item, and `head` is the index of the next slot in which to place an item. Give an example showing that this implementation is *not* linearizable. Use the tests to help your reasoning.

Problem 2. Consider the class `ReaderWriter`. According to what you have been told about the Java memory model, will the reader method ever divide by zero? Please, explain your reasoning.

Problem 3. For each of the histories shown in Figures 1 and 2, are they sequentially consistent? Linearizable? Justify your answer.

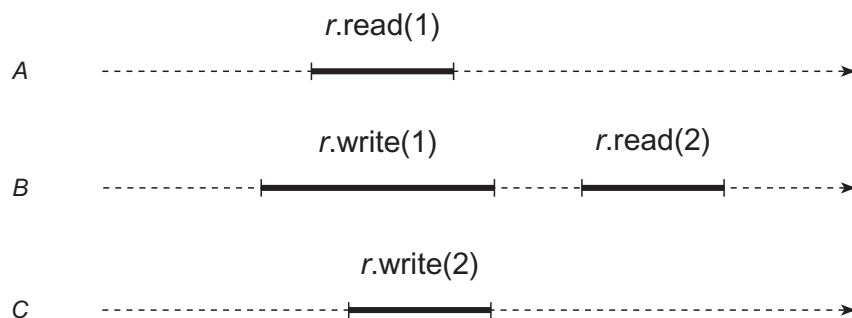


Figure 1: First history for Problem 3.

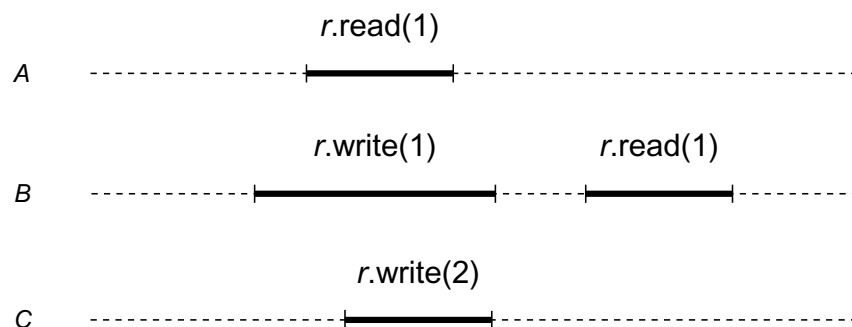


Figure 2: Second history for Problem 3.

¹<https://github.com/ysc4231/hw05-linearizability-examples>

Problem 4. This problem examines a stack implementation `AGMStack` whose `push()` method does not have a single fixed linearization point *in the code*.

The stack stores its items in an `items` array. The `tail` field is an `AtomicInteger`, initially zero. The `push()` 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 `pop()` method reads the value of `tail` and then traverses the array in descending order from the `tail` to slot zero. For each slot, it swaps `None` with the current contents, returning the first non-`None` item it finds. If all slots are `None`, the method returns `None`, indicating an empty stack.

- Give an example execution showing that the linearization point for `push()` cannot occur at the line

```
val j = tail.getAndIncrement()
```

Hint: give an execution where two `push()` calls are *not linearized* (i.e., take effect) in the order they execute this line.

- Give another example execution showing that the linearization point for `push()` cannot occur at the line
`items.set(j, Some(x))`
- Since these are the only two memory accesses in `push()`, we must conclude that `push()` has no single fixed linearization point. Does this mean `push()` is not linearizable? Can you then show an execution history of `AGMStack` that is not linearizable?