Assignment 8: Concurrent Objects

ETH Zurich

1 Comparing Histories: Linearizability

Figures 1 and 2 show two different histories for three threads. Each time line corresponds to one thread.

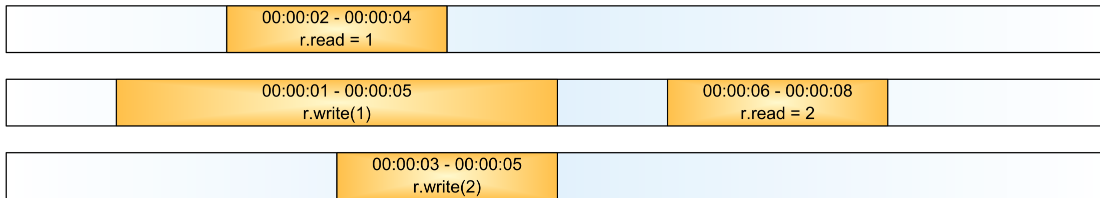

Figure 1: first history

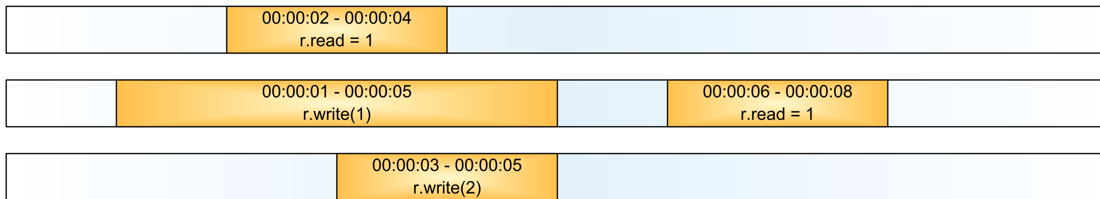

Figure 2: second history

1.1 Task

Decide whether the histories are sequentially consistent. Decide whether the histories are linearizable. Justify your answer.

2 FIFO Queue: Linearizability

The *AtomicInteger* class 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 following FIFO queue implementation. It stores its items in an array *items*, which, for simplicity, we will assume has unbounded size. It has two *AtomicInteger* fields. *tail* is the index of the next slot from which to remove an item. *head* is the index of the next slot in which to place an item.

```
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;
  }
}
```

2.1 Task

Give an example showing that this implementation is not linearizable.

3 Lock-free Deque

3.1 Background

This exercise is a version of the algorithm given in [1].

The deque is identified by a triple, $(LeftEnd, RightEnd, Status)$ where $LeftEnd$ and $RightEnd$ are pointers to nodes, and $Status$ is a flag denoting the current stability or operation. The possible values of the status flag are $Stable$, $LPush$, and $RPush$, denoting a stable state, in the middle of a left-push operation, or in the middle of a right-push operation.

The nodes themselves also have a $Left$ and $Right$ componenet, as well as some associated data.

We say that the deque is $left$-$incoherent$ if the property $node.Right.Left \neq node$ holds for the left-most node. Likewise a deque is $right$-$incoherent$ if the property $node.Left.Right \neq ndoe$ holds for the right-most node.

We say that a deque is stable if it has the status-tag $Stable$, and:

- Has 0 or 1 nodes.
- Has 2 or more nodes, and it is both left- and right-coherent.

Our deque has four operations: $push_right$, $push_left$, pop_right, and pop_left.

3.2 Task

The task is to implement, in pseudo-code and using compare and swap, the *push_right* operation for a lock-free deque. This operation takes some data as an argument and adds a new node on the right-side of the deque to hold it.

The deque is the global variable *Anchor*.

You can assume that a procedure *Stabilize* exists. *Stabilize* when given an argument deque (l, r, s) where s indicates incoherence (either $LPush$ or $RPush$) will either successfully make the *Anchor* stable with left and right nodes set to l and r, or return.

Every operation (*push_right*, *push_left*, *pop_right*, and *pop_left*) only proceeds to modify the deque after the deque is stable. If the deque is not stable, each operation tries to make it be stable before it proceeds.

You can also assume that the CAS operation is able to operate properly on the deque triple.

References

[1] CAS-Based Lock-Free Algorithm for Shared Deques. 9th Euro-Par Conference on Parallel Processing. Maged M. Michael 2003.