CS1760: Multiprocessor Synchronization Due 22 October 2023

Homework 3

Professor: Maurice Herlihy *TA: cs1760tas@lists.brown.edu*

Problem 1. Imagine running a 64-bit system on a 32-bit system, where we simulate a single 64-bit memory location (register) using two atomic 32-bit memory locations (registers). A write operation is implemented by simply writing the first 32-bits in the first register, then the second 32-bits in the second register. A read, similarly, reads the first half from the first register, then the second half from the second register, and returns the concatenation. What is the strongest property that this 64-bit register satisfies?

- ☐ Regular register
- ☐ Safe Register
- ☐ Atomic register
- ☐ Does not satisfy any of these properties

Problem 2. This problem examines a stack implementation (Figure 1) whose push() method does not have a single fixed linearization point in the code.

```
1  public class AMGStack<T> {
2    AtomicReferenceArray<T> items;
3    AtomicInteger tail;
4    static final int CAPACITY = 1024;
5
6    public AMGStack() {
7      items = new AtomicReferenceArray<T>(CAPACITY);
8      tail = new AtomicInteger(0);
9    }
10   public void push(T x) {
11     int i = tail.getAndIncrement();
12     items.set(i,x);
13   }
14   public T pop() {
15     int range = tail.get();
16     for (int i = range − 1; i > −1; i−−) {
17       T value = items.getAndSet(i, null);
18       if (value != null) {
19         return value;
20       }
21     }
22     // Return Empty.
23     return null;
24   }
25 }
```

Figure 1: Afek/Morrison/Gafni stack.

The stack stores its items in an items array, which for simplicity we will assume is unbounded. 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 *null* with the current contents, returning the first non-*null* item it finds. If all slots are *null*, the method returns *null*, indicating an empty stack.

- Give an example execution showing that the linearization point for push() cannot occur at Line 11. Hint: give an execution where two push() calls are not linearized in the order they execute Line 11.
- Give another example execution showing that the linearization point for push() cannot occur at Line 12.
- 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?

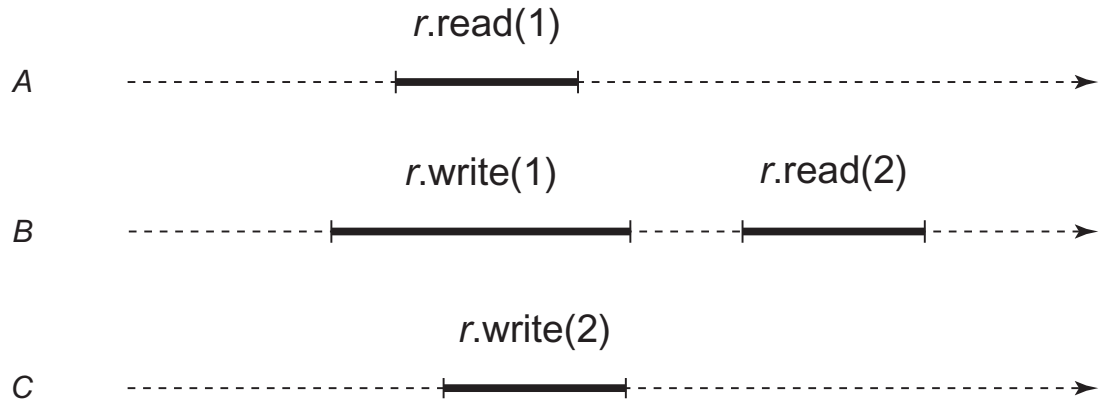

Figure 2: First history for Exercise 3.

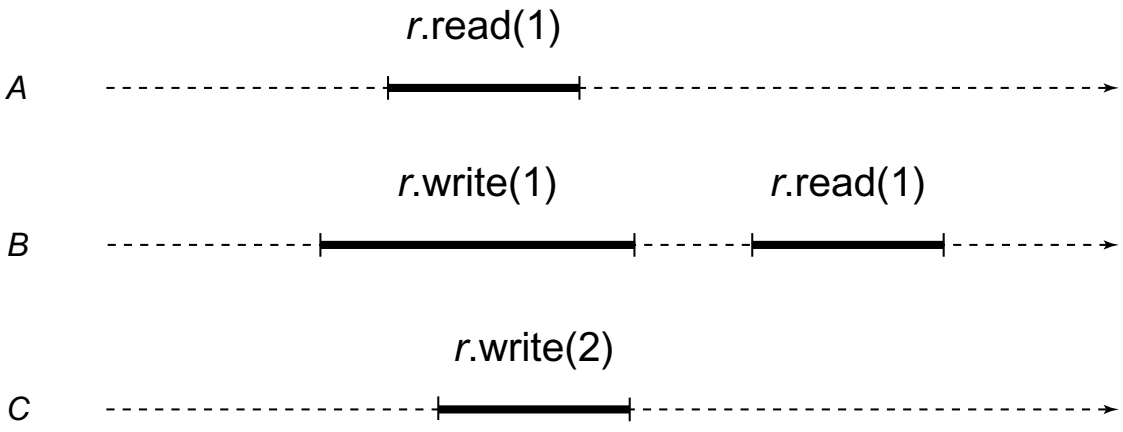

Figure 3: Second history for Exercise 3.

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

Problem 4. 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 Fig. 4. 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, 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.

```
1  class IQueue<T> {
2    AtomicInteger head = new AtomicInteger(0);
3    AtomicInteger tail = new AtomicInteger(0);
4    T[] items = (T[]) new Object[Integer.MAX_VALUE];
5    public void enq(T x) {
6      int slot;
7      do {
8        slot = tail.get();
9      } while (! tail.compareAndSet(slot, slot+1));
10     items[slot] = x;
11   }
12   public T deq() throws EmptyException {
13     T value;
14     int slot;
15     do {
16       slot = head.get();
17       value = items[slot];
18       if (value == null)
19         throw new EmptyException();
20     } while (! head.compareAndSet(slot, slot+1));
21     return value;
22   }
23 }
```

Figure 4: IQueue implementation.