Dairon Andrés Benites Aldaz
DD2443 HT22 (52168) Parallel and Distributed Computing
September 8, 2022

Lecture 3. Exercise

1. Hand-in

Question 1. Prove that if a trace H|x is quiescent consistent for each object x then so is H. Does the converse implication hold? Prove this, or provide a counterexample.

One of the properties of quiescent consistency is that **Quiescent consistency is compositional**. Thus, a system composed of quiescently consistent objects is itself quiescently consistent.

Let's asume that:
$H \notin \mathrm{P}_{quiescently-consistent} \iff H|x_{1,2,3...}, \forall x_i \in H \bigwedge x_i$ is $quiescently - consistent$

$\exists H| \xrightarrow[H]{Domain}, \longrightarrow$Event$_x$ occurs at $Time_x \bigwedge$ Event$_y$ occurs at $Time_y, \iff C_y \to C_x, ...$ $\exists x_i$ which is not $quiescently - consistent$

By definition, this Events (or method calls) to x and y appear to take effect in real-time order when separated by a period of quiescence since any period of quiescence for x is a period of quiescence for y Thus, contradicting initial logic proposition of $\forall x_i \in H \bigwedge x_i$ is $quiescently - consistent$ when H is not.

Does the converse implication hold?

It does not. Since there may be a situation in which $x_i \notin P_{quiescently-consistent}$ in which global $H|x_i$ does not follow the Properties of quiescently consistency given that $\exists X_i$ in which a specification S if for all quiescent histories h there are no sequential histories hs of S such that qcons(h, hs).

Question 2. HSLS 3.4. For each of the histories shown in Figs. 3.11 and 3.12, are they quiescently consistent? Sequentially consistent? Linearizable? Justify your answer.

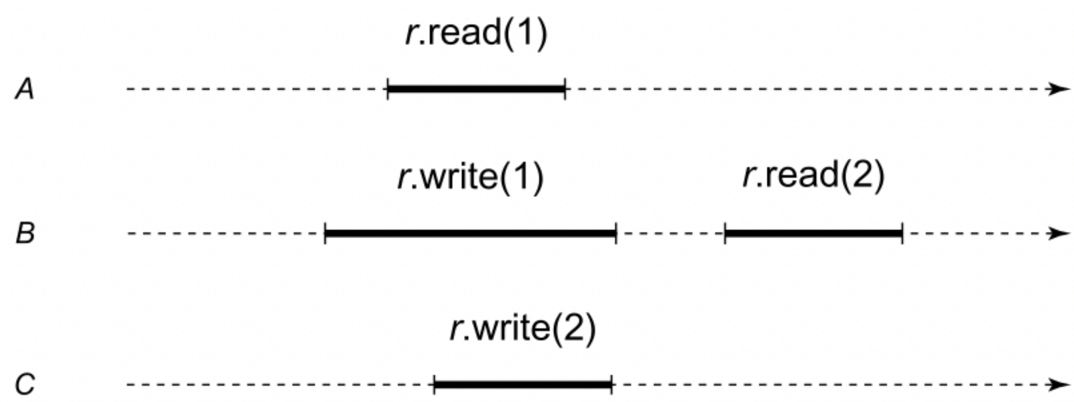

FIGURE 3.11

First history for Exercise 3.4.

Definitions

: quiescently consistent.- *Method calls separated by a period of quiescence should appear to take effect in their real-time order.*

: Sequentially consistent.- *Sequential consistency is a strong safety property for concurrent systems. Sequential consistency implies that operations appear to take place in some total order, and that that order is consistent with the order of operations on each individual process.*

: linearizable.- *Each method call should appear to take effect instantaneously at some moment between its invocation and response. This principle states that the real-time order of method calls must be preserved. We call this correctness property linearizability. Every linearizable execution is sequentially consistent, but not vice versa.*

According to linearizability, the order of the methods calls is in **sequential consistency** and the method calls are instantaneously effective at each invocation and response.
As follows:

: B's r.write (1) will be executed first.
: White executing B's r.write(1), A's r.read(1) will be executed.
: Afterwards C's r.write(2) is executed.
: At last, B's r.read(2) is executed.

The above history's events are in sequential execution of write and read on 1 and 2 which are instantaneously effective $\longrightarrow$ *linearization.*
Therefore, **it is a linearizable.**

quiescently consistent	Sequentially consistent	Linearizable
Yes	Yes	Yes

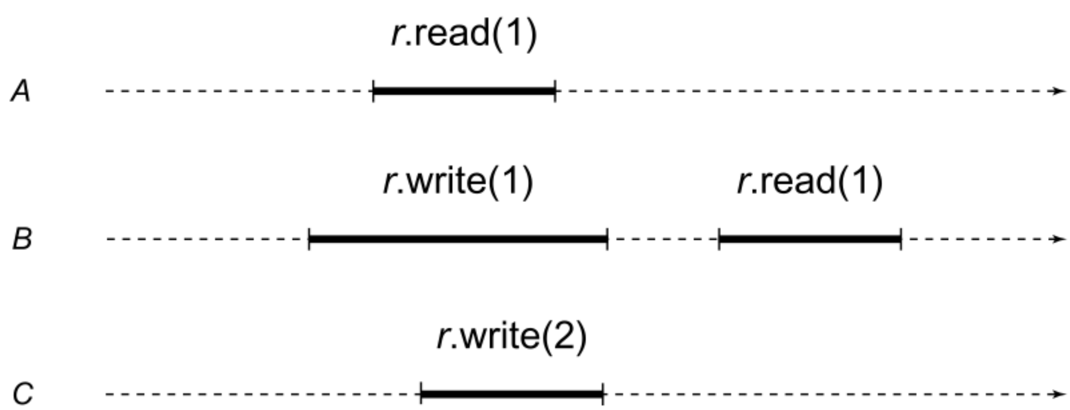

FIGURE 3.12

Second history for Exercise 3.4.

According to linearizability, the order of the methods calls is in **sequential consistency** and the method calls are instantaneously effective at each invocation and response. As follows:

: B's r.write (1) will be executed first.
: White executing B's r.write(1), A's r.read(1) will be executed.
: Afterwards C's r.write(2) is executed.
: At last, B's r.read(1) is executed.

The above history's events are not in sequential execution of write and read on 2 and 1 since they are not instantaneously effective and thus cannot be linearizable. Thus, the given history for figure 3.12 **is not a linearizable.**

Quiescently consistent	Sequentially consistent	Linearizable
Yes	No	No

Question 3. HSLS 3.7. The AtomicInteger class (in the java.util.concurrent.atomic pack- age) 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 with 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 value. Consider the FIFO queue implementation shown in Fig. 3.13. It stores its items in an array items, which, for simplicity, we 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.

```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;
  }
}
```

FIGURE 3.13

IQueue implementation for Exercise 3.7.

Definition

linearizable.- *Each method call should appear to take effect instantaneously at some moment between its invocation and response. This principle states that the real-time order of method calls must be preserved. We call this correctness property linearizability. Every linearizable execution is sequentially consistent, but not vice versa.*

Main uses of AtomicInteger

: Atomic counter

: Primitive that support compare-and-swap

Requirements for Linearizability

: Real-time order of a history

: Sequential consitency and Quiescently consistency

We can proof that this implementation of class IQueue<T>is not linearizable by showing an example in which it does not follow the Principles of Sequential Consistency:

- I would suggest that:

Given that .get() **from** head tail AtomicIntegers are atomic but an **«if operation»** is not.

Therefore, this non-atomic operation can be paused at any time. For example:

if... value == NULL   Allocation of $items[slot] \longleftarrow x$

Thus, we may encounter a situation in which:

$\exists(Thread_i)$ where $slot[value]_x \not\preceq slot[value]_y$ in same execution processor.

Question 4. HSLS 3.10. This exercise examines the queue implementation in Fig. 3.14, whose enq() method does not have a single fixed linearization point in the code. The queue stores its items in an items array, which, for simplicity, we assume is unbounded. The tail field is an AtomicInteger, initially zero. The enq() 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 deq() method reads the value of tail, and 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.

```
1  public class HWQueue<T> {
2    AtomicReference<T>[] items;
3    AtomicInteger tail;
4    static final int CAPACITY = Integer.MAX_VALUE;
5
6    public HWQueue() {
7      items =(AtomicReference<T>[])Array.newInstance(AtomicReference.class,
8          CAPACITY);
9      for (int i = 0; i < items.length; i++) {
10       items[i] = new AtomicReference<T>(null);
11     }
12     tail = new AtomicInteger(0);
13   }
14   public void enq(T x) {
15     int i = tail.getAndIncrement();
16     items[i].set(x);
17   }
18   public T deq() {
19     while (true) {
20       int range = tail.get();
21       for (int i = 0; i < range; i++) {
22         T value = items[i].getAndSet(null);
23         if (value != null) {
24           return value;
25         }
26       }
27     }
28   }
29 }
```

FIGURE 3.14

Herlihy–Wing queue for Exercise 3.10.

- Give an execution showing that the linearization point for enq() cannot occur at line 15. (Hint: Give an execution in which two enq() calls are not linearized in the order they execute line 15.)

A: 15, B: 15, B: 16, C: `deq() == B`.

- Give another execution showing that the linearization point for enq() cannot occur at line 16.

A: 15, B: 15, B: 16, A: 16, C: `deq() == A`.

- 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?

It does not. The linearization point occurs as soon as `getAndSet()` in line 22 returns a non-`null` value.