Parallel and Distributed Computing
DD2443 - Pardis24
Exercises for Lecture 3

Name: Casper Kristiansson

September 1, 2024

Exercise 1

Question

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.

Answer

From what we know to achieve quiescent consistency at each quiescent point a specific state could be reached by method calls to that starting point. In simple terms, this means that if all individual method calls follow a specific quiescent consistency rule then the whole system `H` also follows this rule.

As for the second statement if an entire history `H` is quiescent consistent then each object subhistory `H|x` is quiescent consistent. This means that if we look at it from the other perspective if the whole system follows a rule it also means that each method also follows this behaviour. Meaning if it applies to the whole system it also applies to the individual.

This means that both directions are true:

- If each object subhistory `H|x` is quiescent consistent, then the entire history `H` is quiescent consistent.
- If the entire history `H` is quiescent consistent, then each object subhistory `H|x` is quiescent consistent.

Exercise 2

Question

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 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 an 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.

Answer

To achieve linearizable it is required that each operation on a concurrent object takes into effect with other shared threads instantaneously. This means that an action such as an enqueue and dequeue operation should be seen to occur at a single point in time in history.

This means that for this FIFO queue to be linearizable we need that if an enqueue action has started any dequeuing actions after that must be able to retrieve the item that the enqueue action added.

A situation in which this won't be happening is for example as follows; Let's say we have two threads. The first thread calls the enqueue method with an empty list. While executing this method comes to line 9 which is while (!tail.compareAndSet(slot, slot+1)). What this action does is pretty much increment the tail index which on an empty list is from 0 to 1. Just after finishing executing this line a second thread started executing and both ¨does lines 16 and 17 which are both getting the head index and slot index. The issue here is that the value is still null because the first thread hasn't yet written the value to that specific location. This means that the second thread will throw an EmptyException error.

As stated before for this function to be linearizable means that if an action for example calling enqueue any following actions calling dequeue has to return the value that was used in the enqueue. In the example above this did not happen and the second thread instead received an error.

Exercise 3

Question

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.

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

Answer

1.

An example execution of this is let's say we have two threads executing the enqueue command. Let's say we have the following execution line:

- Thread1 executes `tail.getAndIncrement()` and gets i = 0.
- Thread2 executes `tail.getAndIncrement()` and gets i = 1.
- Thread2 than executes `items[1].set(Thread1Value)`
- Thread1 than executes `items[0].set(Thread1Value)`

This action shows that even though Thread2 is set in the queue before Thread1's value it doesn't match the order in the array. meaning this contradicts that the actions enqueue for thread1 should be completed before thread2's enqueue action.

2.

An example execution showing that linearization point for enqueue cannot occur at line 16 can happen using three threads as follows:

- Thread1 executes `tail.getAndIncrement()` and gets i = 0.
- Thread2 executes `tail.getAndIncrement()` and gets i = 1.
- Thread1 than executes `items[0].set(Thread1Value)`
- Thread3 than executes a deq() which retrieves Thread1Value
- Thread2 than executes `items[1].set(Thread1Value)`

With this situation, if linearization is implemented correctly the value thread3 should have retrieved should have happened after thread2 had executed and retrieved thread2's value instead.

3.

No, the enqueue method can still be linearizable even though it doesn't have a single linearization point. What is important is that the total order of operations in the enqueue method remains consistent with the queue implementation. This means that to make this function linearizable we need to focus on the logical order of the operations that might take place.