EFREI M1: Distributed Algoruthms 2019: Solutions for Quiz 2

1 Hand-over-hand locking

Locking in *contains*

If we allow a *contains* operation C to proceed in the *wait-free* manner, every node it reads may turn out to be already unreachable from the *head*.

However, this is not an issue if we realize that if this is the case, then the *remove* operation R that unlinked the node from list must be concurrent with C. Moreover, the *linearization point* of R (the moment it updates *pred.next*) must lie within the interval of C. Indeed, if the liearization point of R preceeds the invocation of C in the *real-time* order, then the node is already unreachable at the moment when C is invoked and, thus, C cannot find it.

In fact, this observation holds for each of the algorithms we considered in the class: a complete wait-free *contains* operation can always be linearized at:

- the moment it performs its last read (in case the node it reads is reachiable from the *head*), or
- just before the linearization point of the successful *remove* operation that unlinks the last node it reads.

Checking *curr* before locking

The idea is to return *false* early, without grabbing locks. Indeed, an unsuccessful update does not need to protect data with locks, as it is not going to modify it.

We can see that the resulting algorithm is correct, as an unsuccessful update can be treated as a *contains* operation, and we have just shown that *contains* operations can be performed wait-free.

Locking one node at a time

Imagine that an operation $R = remove(1)$ keeps a lock on *pred*, reads *curr* = *pred.next* and releases the lock on *pred* before grabbing the lock on *curr*. Imagine further that *curr*.value == 1.

Then we can squeeze another $R' = remove(1)$ in the gap when no node is protected with locks that unlinks *curr* from the list updating *pred.next*.

R wakes up and successfully completes, which violates linearizability.

The morale here is that an update operation must at some point keep locks on *two* consecutive nodes when traversing the list.

Starvation-freedom

Immediate, once we realize that the underlying locks are starvation-free and each opearion may only perform a bounded number of steps. The bound comes from the parameter v of the operation, since the list is sorted, the number of shared-memory operations the operation performs is bounded by $O(v - MININT)$ (assuming that keys are integers).

2 Optimistic locking

The need for validation in updates

Without validation concurrent updates may overwrite each other. See the example of the *lost-update* problem on slide 8 on the lecture.

Validation in *contains*

Not necessary, check the discussion of the first question.

The lack of starvation-freedom

Suppose that the list is initially empty and consider an operation $I = insert(1)$ that is concurrent with an infinite series of alternating successful $insert(1)$ and $remove(1)$ operations, each of them scheduled just before I grabs locks and runs its validation procedure. As all validations fail, I never terminates, even though every lock is eventually released - *starvation-freedom* is violated.

But for this to heppen, infinitely many updates must take place, and *deadlock-freedom* is satisfied.

3 Lazy locking

Validation conditions

By "two conditions" here we mean (1) checking that *pred* is not marked for deletion and (2) checking that *pred* still points to *curr*.

The first check is needed to anticipate the scenario in which *pred* is removed by a concurrent update before we take a lock on it. If we do not do it, any further modification of *pred* will be "lost".

The second check is needed to make sure that a potential concurrent update that modified *pred.next* will not be "lost" because of our operation.

Checking $curr.marked$

Coming soon

Linearizability

A nice feature of a set abstraction is that we can prove linearizability of a history H by only considering separately, for each key k, the restriction of H to operations invoked with parameter k. Therefore, we can choose the linearization point of an operation in an execution of the lazy algorithm bazed on other operations with the same key. We only give a sketch below.

A successful update is linearized at the point it modifies *pred.next*. (Recall that an update is successful if it executes this instruction.)

An *incomplete* unsuccessful operation or *contains* is removed from the linearization (it is read-only, so no other operation is affected by its presence in a history).

The linearization point of a complete unsuccessful update is the moment it completes its validation.

To define the linearization point of *contains* see the first exercise.

Finishing the argument and showing that the sequential history resulting after placing operations of H in the order of their linearization points is left as an exercise. Check Chapter 7 of Herlihy-Shavit for details.