Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

FS 2018 Prof. R. Wattenhofer

Computer Engineering II

Solution to Exercise Sheet Chapter 11

Quiz

1 Quiz

a) Lock free.

b) Hash functions ...

- ... map variable size inputs to constant size hashes.
- ... ideally are fast. (perhaps unless used in cryptography)
- ... ideally have a uniform distribution of outputs, independent of the distribution of inputs. I.e., collisions should be rare!
- (... are hard to invert. [in cryptographic contexts])

c) One could use a binary search tree or any other search data structure within each bucket. This causes some overhead, but may save a lot of space compared to growing the hash map, if many buckets are filled far less. However, this scenario should only rarely occur given a good hash function with a uniform hash distribution.

d) Coarse Grained and Fine Grained locking are FIFO fair. Optimistic and lazy locking are not.

Basic

2 Livelock

a) Is there a scenario in which two (or more) threads deadlock?
No, as the locks are always acquired in the same order, with the first lock closer to the head of the list. More precisely, assume there is a deadlock. Then, there is a set of nodes that are locked but will never be unlocked. Let a be the locked node furthest down in the list and A be the thread that holds the lock on a. As thread A is deadlocked, it wants to lock another node b. This node b is further down the list and is unlocked. Thus, thread A will acquire the lock on b and cannot be deadlocked, a contradiction.

b) Is there a scenario in which one thread never succeeds in removing a node?
Consider a list with the four nodes a, b, c, and d. Thread A wants to remove node c (this thread will not succeed). Thread B wants to remove node b. Consider the following sequence of events:

(a) Thread A moves forward and finds node c.

(b) Thread B moves forward and finds node b.

(c) Thread B acquires the locks for node a and node b.

(d) Thread B removes node b and sets the pointer accordingly.

(e) Thread B releases the locks.

(f) Thread A acquires the locks for node b and c.

(g) Thread A calls validate and it fails.

The list has now only three nodes: a, c, and d. Thread C now wants to reinsert node b. Consider the following sequence of events.

(a) Thread A moves forward and finds node c.

(b) Thread C moves forward and finds node c.

(c) Thread C acquires the locks for node a and node c.

(d) Thread C inserts node b.

(e) Thread C releases the locks.

(f) Thread A acquires the lock for node a and node c (it still thinks that node a is the predecessor).

(g) Thread A calls validate and it fails.

The list now has four nodes again. This can repeat forever.

Advanced

3 Old Exam Question: Fine-Grained Locking

a) Coarse grained locking locks the entire tree at once for each operation, regardless of the location the change occurs. This can be achieved by always locking the same location for each operation. We could for example always lock the root by issuing LOCK(1) and UNLOCK(1). Coarse grained locking does not allow multiple threads to work concurrently on the heap.

b)

Algorithm 1 Insert value

```
i = 1
LOCK(i) ................
while A[i] != null do
    ....................
    next = smallestChild(i)
    LOCK(next) ............
    if if(A[i] > value) then
        exchange A[i] and value
    end if
    UNLOCK(i) ............
    i = next
    ....................
end while
.....................
A[i] = value
UNLOCK(i)...............
```

Algorithm 2 Remove smallest value

```
LOCK(1)................
ret = A[1]
i=1
A[1] = $\infty$
.....................
while A[i] != null do
    LOCK(2i), LOCK(2i+1) .......
    next = smallestChild(i)
    UNLOCK(4i+1-next).........
    if (A[next] != null) then
        exchange A[i] and A[next]
    else
        A[i] = null // Mark as not used
    end if
    UNLOCK(i) ............
    i = next
    ....................
end while
UNLOCK(i)...............
return ret
```

c) Both functions acquire locks in the same ordering: locks closer to the root are taken before the locks on descendants. This means that there is a global lock order, hence there cannot be a deadlock as seen in the lecture. Alternatively one might observe that descending in a tree is exactly the same as walking down a linked list, hence the same analysis from the lecture applies.

d) We could use optimistic locking to lock only at the subtree that is going to be modified. We would walk down from the root to the location the current value will be inserted at, lock that location and then check the consistency by walking down once more. We'd then start hand-over-hand locking at that location to propagate the changes down to the leafs. This has the advantage of not locking the root in the case of an insert. Notice that this would still lock the root in the case of a remove since the first modified location is the root.