EE 382C/361C: Multicore Computing **Fall 2016**

Questions for Concurrent Queues, Stacks and Linked Lists

Lecturer: Vijay Garg *Scribe: Weihao Xu*

1 Questions

1. To implement lock-free concurrent queue in java, why do we need to use class AtomicStampedReference instead of AtomicReference?

2. Fine-grained synchronization and coarse-grained synchronization, which one costs more memory? Which one is more parallel?

(A) fine-grained synchronization, fine-grained synchronization

(B) coarse-grained synchronization, coarse-grained synchronization

(C) fine-grained synchronization, coarse-grained synchronization

(D) coarse-grained synchronization, fine-grained synchronization

3. Suppose we are using AtomicReference class to implement a non-blocking synchronization linked list. The linked list is head $\rightarrow$ a $\rightarrow$ b $\rightarrow$ c. What will happen if two threads A and B try to remove node a and b respectively using CAS operation at the same time?

(A) both a and b will be removed

(B) neither a nor b will be removed

(C) a will be removed, b will not be removed

(D) a will not be removed, b will be removed

4. How to solve the problem revealed in Q3?

5. What is the assumption when we adopt optimistic synchronization for concurrent linked list?

6. In Michael and Scotts Lock Free Queue Algorithm, what will thread A do as next step in the following two scenarios when it is executing enque(a) operation?

(a) tail.next.get() == b != null

(b) tail.next.get() == null

7. For lock-based concurrent linked list, at least how many times of traverse does it need to complete an add() function when using Optimistic Synchronization and lazy synchronization, respectively?

2 Answers

1. To avoid ABA problem, we need to use a integer stamp that is atomically updated whenever a node is updated.

2. A

3. C

4. Use AtomicMarkableReference instead of AtomicReference. Atomically set the mark while removing a node.

5. In optimistic synchronization, if a synchronization conflict causes the wrong nodes to be locked, then the thread need to release the locks and start over. Therefore, when adopting optimistic synchronization, we suppose this kind of conflict is rare, otherwise the performance would be poor.

6.

(a) The next field of tail points to node b, indicating that node b is in the process of enque(b) but not finished. Therefore, thread A need to help swing the tail to b and start over.

(b) The next field of tail points to null, indicating that no other enque() action happens before A. Then thread A need to try to point the next field of tail to node a and then swing tail to a.

7.

(a) The add() operation of optimistic synchronization needs at least two traversals, one for navigating and locking the pred and curr nodes, and the other for validating these nodes.

(b) The add() operation of lazy synchronization needs at least one traversal.

References

[1] V. K. Garg, Introduction to Multicore Computing

[2] Herlihy, Maurice, and Nir Shavit, The art of multiprocessor programming." PODC. Vol. 6. 2006.