The Art of Multiprocessor Programming
Solutions to Exercises
Chapter 15

July 14, 2009

Exercise 173. Give an example of a quiescently consistent priority queue execution that is not linearizable.

Solution.

$$< q.enq(0)A >$$
$$< q.enq(1)B >$$
$$< q.OK()B >$$
$$< q.enq(2)C >$$
$$< q.OK()C >$$
$$< q.OK()A >$$
$$< q.deq()A >$$
$$< q.OK(2)A >$$

In this execution, A's enqueue of 0 is very slow. In the meantime, B enqueues 1 and later C enqueues 2. Because the object is not quiescent, quiescent-consistency does not impose a

Exercise 174. Implement a quiescently consistent Counter with a lock-free implementation of the boundedGetAndIncrement() and boundedGetAndDecrement() methods using a counting network or diffracting tree.

Solution. See the BoundedCounter class in the code folder. Each increment request goes through the counting network to pick a value. If that value is greater than the bound, it undoes its effect by executing a decrement, and returning the bound minus one. Decrement requests work in a similar way.

Exercise 175. In the SimpleTree algorithm, what would happen if the boundedGetAndDecrement() method were replaced with a regular getAndDecrement()?

Solution. The boundedGetAndDecrement() call is intended to take possession of an item lower down in the tree by decrementing the counter if it is non-zero. Applying a boundedGetAndDecrement() call to a zero counter has no effect (there is no item to acquire). Applying a regular getAndDecrement() call to a zero counter would set that counter to be negative. We could try to compensate by applying getAndIncrement(), but an item inserted in that subtree in the interval beween the getAndDecrement() and getAndIncrement() would go undetected, because the counter would not be non-zero.

Exercise 176. Devise a SimpleTree algorithm with bounded capacity using boundedGetAndIncrement() methods in treeNode counters.

Solution.

Exercise 177. In the SimpleTree class, what would happen if add(), after placing an item in the appropriate Bin, incremented counters in the same *top-down* manner as in the removeMin() method? Give a detailed example.

Solution. Suppose an add() call that inserts a high-priority item increments the counters in a top-down order, but suspends itself half-way through. A removeMin() call that arrives while such an add() call was suspended would move half-way down the tree to the last incremented counter, but then might move in the wrong direction when it visited a counter not yet visited by the suspended add() call.

Exercise 178. Prove that the SimpleTree is a quiescently consistent priority queue implementation.

Solution. See Shavit, N. and Zemach, A. 1999. Scalable concurrent priority queue algorithms. In Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing (Atlanta, Georgia, United States, May 04 - 06, 1999). PODC '99. ACM, New York, NY, 113-122. DOI= http://doi.acm.org/10.1145/301308.301339

Exercise 179. Modify FineGrainedHeap to allocate new heap nodes dynamically. What are the performance limitations of this approach?

Solution. See the FineGrainedHeapAlloc class in the code folder. We still keep references to nodes in an array (so we can find empty slots in constant time), but we allocate a new node for each new item, and we swap nodes by swapping pointers instread of copying fields.

The advantage of this approach is that we do not need to copy fields, the disadvantage is that we need to allocate new nodes, and the garbage collector has to reclaim freed nodes.

Exercise 180. Fig. 1 shows a *bit-reversed* counter. We could use the bit-reversed counter to manage the next field of the FineGrainedHeap class. Prove the following: for any two consecutive insertions, the two paths from the leaves to the root have no common nodes other than the root. Why is this a useful property for the FineGrainedHeap?

Solution. For two consecutive insertions, the non-reversed counter values have opposite parities (one LSB is 0, the other 1). When the bits are reversed, one has MSB 0, and the other 1, so the paths through the tree (constructed by right-shifting the reversed values) are disjoint.

Disjoint paths are good because they imply that traversing one path will not invalidate cached entries for the other path (ignoring false sharing).

Exercise 181. Provide the code for the PrioritySkipList class's add() and remove() methods.

Solution. See code folder.

```
public class BitReversedCounter {
  int counter, reverse, highBit;
  BitReversedCounter(int initialValue) {
    counter = initialValue;
    reverse = 0;
    highBit = -1;
  }
  public int reverseIncrement() {
    if (counter++ == 0) {
      reverse = highBit = 1;
      return reverse;
    }
    int bit = highBit >> 1;
    while (bit != 0) {
      reverse ^= bit;
      if ((reverse & bit) != 0) break;
      bit >>= 1;
    }
    if (bit == 0)
      reverse = highBit <<= 1;
    return reverse;
  }
  public int reverseDecrement() {
    counter--;
    int bit = highBit >> 1;
    while (bit != 0) {
      reverse ^= bit;
      if ((reverse & bit) == 0) {
        break;
      }
      bit >>= 1;
    }
    if (bit == 0) {
      reverse = counter;
      highBit >>= 1;
    }
    return reverse;
  }
}
```

Figure 1: A bit-reversed counter.

Exercise 182. The PrioritySkipList class used in this chapter is based on the LockFreeSkipList class. Write another PrioritySkipList class based on the LazySkipList class.

Solution. See the LazyPrioritySkipList class in the code folder.

The find() method is wait-free: it acquires no locks and reads through logically deleted nodes.

The add() method locks predecessors and successors and then validates, checking that no predecessor or successor is deleted, and that each predecessor points to its successor, before it adds the new node. The remove() and contains() methods are similar.

Exercise 183. Describe a scenario in the SkipQueue implementation in which contention would arise from multiple concurrent removeMin() method calls.

Solution. Each removeMin() call invokes findAndMarkMin(). Multiple concurrent findAndMarkMin() calls could all attempt to mark the same minimal element at the same time. One wins, and the rest go on to contend for the next, and soon.

Exercise 184. The SkipQueue class is quiescently consistent but not linearizable. Here is one way to make this class linearizable by adding a simple time-stamping mechanism. After a node is completely inserted into the SkipQueue, it acquires a timestamp. A thread performing a removeMin() notes the time at which it starts its traversal of the lower level of the SkipQueue, and only considers nodes whose timestamp is earlier than the time at which it started its traversal, effectively ignoring nodes inserted during its traversal. Implement this class and justify why it works.

Solution. See the PrioritySkipListTS class in the code folder. The class has a static AtomicInteger clock field, incremented each time a new node is added to the list. The Node<> class has an integer timestamp field. The add() method adds the node, sets its timestamp field to the current clock value, and advances the clock. The findAndMarkMin() method reads the current clock and ignores nodes with later timestamps.