

The Art of Multiprocessor Programming
Solutions to Exercises
Chapter 10

July 14, 2009

Exercise 119. Change the `SynchronousDualQueue<T>` class to work correctly with `null` items.

Solution The simplest solution is to introduce a level of indirection, having the `value` field of queue nodes refer to container objects:

```

1  class Value {
2      T value;
3 }
```

Exercise 120. Consider the simple lock-free queue for a single enqueueer and a single dequeuer, described earlier in Chapter 3. The queue is presented in Fig. 1. This queue is blocking, that is, removing an item from an empty queue or adding item to a full one causes the threads to spin. The surprising thing about this queue is that it requires only loads and stores and not a more powerful read-modify-write synchronization operation. Does it however require the use

```

1  class TwoThreadLockFreeQueue<T> {
2      int head = 0, tail = 0;
3      T[] items;
4      public TwoThreadLockFreeQueue(int capacity) {
5          head = 0; tail = 0;
6          items = (T[]) new Object[capacity];
7      }
8      public void enq(T x) {
9          while (tail - head == items.length) {};
10         items[tail % items.length] = x;
11         tail++;
12     }
13     public Object deq() {
14         while (tail - head == 0) {};
15         Object x = items[head % items.length];
16         head++;
17         return x;
18     }
19 }
```

Figure 1: A Lock-free FIFO queue with blocking semantics for a single enqueueer and single dequeuer. The queue is implemented in an array. Initially the `head` and `tail` fields are equal and the queue is empty. If the `head` and `tail` differ by `capacity`, then the queue is full. The `enq()` method reads the `head` field, and if the queue is full, it repeatedly checks the `head` until the queue is no longer full. It then stores the object in the array, and increments the `tail` field. The `deq()` method works in a symmetric way.

of a memory barrier? If not, explain, and if so, where in the code is such a barrier needed and why?

Solution The question is badly formulated because it does not state which memory model it uses. Nevertheless, if you can read an old value from the array, you might dequeue the same element twice, so any reasonable memory model, both the array elements and the head and tail counters require barriers.

Exercise 121. Design a bounded lock-based queue implementation using an array instead of a linked list.

1. Allow parallelism by using two separate locks for `head` and `tail`.
2. Try to transform your algorithm to be lock-free. Where do you run into difficulty?

Solution The first part is easy: the `BoundedQueue` class already ensures that no more than `capacity` entries are in use at any one time. Simply allocate an array of `capacity` entries, along with a `next` counter, initialized to 0, and increment modulo `capacity`. Every time an `enq()` method would have allocated a new entry, simply increment the `next` counter, and use that slot.

The most obvious way in which this approach fails to be lock-free is if one transaction, A , allocates the Entry at position i in the array, and then halts. Eventually, all the other processes will queue up waiting to claim that slot.

Exercise 122. Consider the unbounded lock-based queue's `deq()` method in Fig. ???. Is it necessary to hold the lock when checking that the queue is not empty? Explain.

Solution Yes, it is necessary to hold the lock. We can check, without holding the lock, whether `head.next` is `null`. If it is, then it is safe to throw `EmptyException`, but if not, we cannot assume that the queue is non-empty when we acquire the lock.

Exercise 123. In Dante's *Inferno*, he describes a visit to Hell. In a very recently discovered chapter, he encounters five people sitting at a table with a pot of stew in the middle. Although each one holds a spoon that reaches the pot, each spoon's handle is much longer than each person's arm, so no one can feed him- or herself. They are famished and desperate.

Dante then suggests “why do not you feed one another?”

The rest of the chapter is lost.

1. Write an algorithm to allow these unfortunates to feed one another. Two or more people may not feed the same person at the same time. Your algorithm must be, well, starvation-free.

2. Discuss the advantages and disadvantages of your algorithm. Is it centralized, decentralized, high or low in contention, deterministic or randomized?

Solution [[[Nir?]]]

Exercise 124. Consider the linearization points of the `enq()` and `deq()` methods of the lock-free queue:

1. Can we choose the point at which the returned value is read from a node as the linearization point of a successful `deq()`?
2. Can we choose the linearization point of the `enq()` method to be the point at which the `tail` field is updated, possibly by other threads (consider if it is within the `enq()`'s execution interval)? Argue your case.

Solution

1. Can we choose the point at which the returned value is read from a node as the linearization point of a successful `deq()`?

Yes. The most natural linearization point is the successful `compareAndSet()` that actually removes the item from the queue. The order of the reads that precede each successful `compareAndSet()` is the same as the order of those `compareAndSet()` calls. Each such read clearly falls within the interval of the call.

2. Can we choose the linearization point of the `enq()` method to be the point at which the `tail` field is updated, possibly by other threads (consider if it is within the `enq()`'s execution interval)? Argue your case.

No. The most natural linearization point is the successful `compareAndSet()` that actually appends the new node to the end of the list. Although the order of the `compareAndSet()` calls that update the `tail` field is the same as the order of the appending `compareAndSet()` calls, these updating `compareAndSet()` calls may not occur during the original `enq()` call.

Exercise 125. Consider the unbounded queue implementation shown in Fig. 2. This queue is blocking, meaning that the `deq()` method does not return until it has found an item to dequeue.

The queue has two fields: `items` is a very large array, and `tail` is the index of the next unused element in the array.

1. Are the `enq()` and `deq()` methods wait-free? If not, are they lock-free? Explain.
2. Identify the linearization points for `enq()` and `deq()`. (Careful! They may be execution-dependent.)

```

1  public class HWQueue<T> {
2      AtomicReference<T>[] items;
3      AtomicInteger tail ;
4      ...
5      public void enq(T x) {
6          int i = tail.getAndIncrement();
7          items[i].set(x);
8      }
9      public T deq() {
10         while (true) {
11             int range = tail.get();
12             for (int i = 0; i < range; i++) {
13                 T value = items[i].getAndSet(null);
14                 if (value != null) {
15                     return value;
16                 }
17             }
18         }
19     }
20 }
```

Figure 2: Queue used in Exercise 125.

Solution We assume the array is large enough that we never address an element beyond the array bounds.

1. Are the `enq()` and `deq()` methods wait-free?
 - The `enq()` method is clearly wait-free, because there are no loops or conditionals
 - If the queue is empty, the `deq()` method is not lock-free, because it will run forever. If the queue is always non-empty whenever `deq()` is called, then `deq()` is lock-free, but not wait-free.
2. Identify the linearization points for `enq()` and `deq()`.
 - The linearizaton point for `enq()` is when it stores its item in the array.
 - The linearizaton point for `deq()` is when it calls a `getAndSet()` that returns a non-*null* value