

## Home Assignment 1

Due date: **December 11th, 23:59** (submit via moodle)

### Submission instructions

- In this assignment, you can submit either in pairs or individually. Even if working in a pair, each member should submit the assignment through the moodle.
- Submissions must be PDF, at most 3 pages, minimum 11 pt font.
- A solution submitted  $t$  seconds late will have its grade multiplied by  $1 - \left(\frac{t}{60 * 60 * 24 * 7}\right)^4$ .

## 1 Linearizability

A  $B$ -bounded queue object is a FIFO queue object that can hold up to  $B$  values (assume  $B \geq 2$ ). It has the following sequential specification: A state  $\sigma$  is a (possibly empty) sequence of values  $v_1 \dots v_m$ , where  $|\sigma| = m \leq B$  is called the *size* of the queue. The initial state is the *empty* state,  $\varepsilon$ . An `enqueue( $v$ )` operation in state  $\sigma$  returns  $\perp$  without changing the state if  $|\sigma| = B$ ; otherwise, if  $|\sigma| < B$ , the `enqueue( $v$ )` operation changes the state to  $\sigma v$  and returns OK. A `dequeue()` operation in state  $\sigma$  returns  $\perp$  if  $\sigma = \varepsilon$ ; otherwise, if  $\sigma = v \sigma'$ , the `dequeue()` operation returns  $v$  and changes the state to  $\sigma'$ . (Assume that the value  $\perp$  is never enqueued.)

Let  $Q$  be a linearizable  $B$ -bounded queue algorithm. Figure 1 shows a new  $B$ -bounded queue algorithm, which uses  $Q$  as a subroutine. Is this new algorithm necessarily linearizable? Explain or give a counter-example. To show that the algorithm is linearizable, describe how to assign linearization points to an arbitrary execution history of the algorithm, and why this assignment describes a legal sequential history of a  $B$ -bounded queue. To show that the algorithm is not linearizable, describe an execution for which **no** assignment of linearization points would produce a legal sequential history.

<pre> 1 // Queue is an instance of the Q 2 // algorithm that implements a 3 // B-bounded queue. 4 Q Queue; // initially empty 5 // Count is an unbounded signed 6 // integer. 7 int Count = 0; // initially zero </pre>	<pre> 8 enqueue(v) { 9   n = Count 10  while (true) { 11    if (n ≥ B) 12      return ⊥ 13    if (FAA(Count, 1) &lt; B) { 14      r = Queue.enqueue(v) 15      if (r ≠ ⊥) 16        return r 17    } 18    n = FAA(Count, -1) - 1 19  } 20 } </pre>	<pre> 21 dequeue() { 22   n = Count 23   while (true) { 24     if (n ≤ 0) 25       return ⊥ 26     if (FAA(Count, -1) &gt; 0) { 27       r = Queue.dequeue() 28       if (r ≠ ⊥) 29         return r 30     } 31     n = FAA(Count, 1) + 1 32   } 33 } </pre>
---	---	---

**Figure 1:** A new  $B$ -bounded queue algorithm that uses  $Q$  (a linearizable  $B$ -bounded queue algorithm) as a subroutine. Reminder: a `FAA( $v, x$ )` operation atomically adds  $x$  to variable  $v$  and returns the value that  $v$  held before the addition.

## 2 Cache coherence

### 2.1

In class, we described the high-level idea of how to execute an atomic instruction like CAS. Now, fill in the details. Design a state machine for the processor execution of CAS, and extend the state machines of the cache and memory controllers as necessary. For a baseline, use the simple snooping MSI cache coherence protocol with atomic transactions but **without** atomic requests (Tables 7.8 and 7.9 in the book *A Primer on Memory Consistency and Coherence*). **For a full score, (1) your design should not add new message types to the coherence protocol, and (2) should not allow the processors to deadlock.**

### 2.2

Now we want to support a DCAS atomic operation, which does two CASes atomically:

---

```
bool DCAS( $a_1, o_1, n_1, a_2, o_2, n_2$ ) atomically {  
    // we assume DCAS is always executed with  $a_1 \neq a_2$   
    if (memory[ $a_1$ ] ==  $o_1$  and memory[ $a_2$ ] ==  $o_2$ ) {  
        memory[ $a_1$ ] =  $n_1$   
        memory[ $a_2$ ] =  $n_2$   
        return true  
    } else {  
        return false  
    }  
}
```

---

Design a DCAS implementation: a state machine for the processor execution of DCAS, and any extensions necessary to the state machines of the cache and memory controllers. (Again, use the simple snooping MSI cache coherence protocol as a baseline.) **For a full score, (1) your design should not add new message types to the coherence protocol, and (2) should not allow the processors to deadlock.**