Exercises for Nonblocking Data Structures

Michael L. Scott

July 2019

1. Write an atomic_counter class in C++11. Provide a constructor that initializes the counter to zero. Provide two methods to increment the counter, one that uses fetch_add internally and the other that uses compare_exchange_strong. Benchmark these methods in a loop on an x86 processor, for varying numbers of concurrent threads and explain the observed performance.

   In a similar vein, provide two methods to read the counter, one that uses memory_order_seq_cst to order itself with respect to previous and subsequent operations and the other that uses memory_order_memory_relaxed to avoid the cost of doing so. Again, benchmark these methods in a loop on an x86 processor, for varying numbers of concurrent threads and explain the observed performance.

2. Consider the code for the Treiber stack, shown on a slide in class and repeated below. (Note that this code defers the problem of memory management to the caller of push and pop.) What memory ordering annotations are required to make the code correct on a machine with a relaxed memory model?

```
class stack
    ⟨node*, int⟩ top

void stack.push(node* n):
    repeat
        ⟨o, c⟩ := top
        n→next := o
    until CAS(&top, ⟨o, c⟩, ⟨n, c⟩)
```

```
node* stack.pop():
    repeat
        ⟨o, c⟩ := top
        if o = null return null
        n := o→next
    until CAS(&top, ⟨o, c⟩, ⟨n, c+1⟩)
    return o
```

   Answer: It suffices for the CAS in push to be a release (to be ordered after all previous loads and stores). For pop, the CAS should be both an acquire and a release, so it is ordered after all previous loads and stores (in particular, those that read the previous state of the stack) and so it is ordered before all subsequent loads and stores (in particular, those that might use the popped node). It's probably safe for the CAS in push not to be an acquire, because (presumably) the subsequent code in the calling thread makes no further use of the pushed node.

3. Consider the following code for the M&S queue. What are the linearization points? What memory ordering annotations are required to make this code correct on a machine with a relaxed memory model?

```
type ptr = ⟨node* p, int c⟩                          // counted pointer
type node
    value val
    ptr next
class queue
    ptr head
    ptr tail

void queue.init()
    node* n := new node(⊥, null)                     // initial dummy node
    head.p := tail.p := n

void queue.enqueue(value v):
    node* w := new node(v, null);                    // allocate node for new value
    ptr t, n
    loop
        t := tail.load()                             // counted pointers
        n := t.p→next.load()
        if t = tail.load()                           // are t and n consistent?
            if n.p = null                            // was tail pointing to the last node?
                if CAS(&t.p→next, n, ⟨w, n.c+1⟩)     // try to add w at end of list
                    break                            // success; exit loop
            else                                     // tail was not pointing to the last node
                (void) CAS(&tail, t, ⟨n.p, t.c+1⟩)   // try to swing tail to next node
    (void) CAS(&tail, t, ⟨w, t.c+1⟩)                 // try to swing tail to inserted node

value queue.dequeue():
    ptr h, t, n
    loop
        h := head.load()                             // counted pointers
        t := tail.load()
        n := h.p→next.load()
        value rtn
        if h = head.load()                           // are h, t, and n consistent?
            if h.p = t.p                             // is queue empty or tail falling behind?
                if n.p = null return ⊥               // empty; return failure
                (void) CAS(&tail, t, ⟨n.p, t.c+1⟩)   // tail is falling behind; try to update
            else                                     // no need to deal with tail
                // read value before CAS; otherwise another dequeue might free n
                rtn := n.p→val.load()
                if CAS(&head, h, ⟨n.p, h.c+1⟩)       // try to swing head to next node
                    break                            // success; exit loop
    free_for_reuse(h.p)                              // type-preserving
    return rtn                                       // queue was nonempty; return success
```

Answer: Execution of the enqueue method linearizes on the CAS of p→next. If the dequeue method returns ⊥, execution linearizes on the load of tail (assuming that was ordered after the load of head). If dequeue returns an actual value, execution linearizes on the CAS of tail.

There is no single correct set of memory order annotations. The safest approach is to put atomic labels on head, tail, and all val and next fields, and access them with memory_order_seq_cst in all cases. Some of the resulting orderings might safely be relaxed via careful reasoning, but the performance gain is probably not worth the risk of making a mistake. In addition, the ordinary accesses of the

new call in enqueue should be ordered before subsequent accesses (probably with a release fence), and the ordinary accesses of the release call in dequeue should be ordered (perhaps with an acquire fence) after the CAS of the final loop iteration.

4. The code in the previous example was written to use counted pointers and a type-preserving allocator. Modify it to use hazard pointers.

   Answer: The counter portions of the pointers can all be elided, of course. Since head and tail are statically allocated, they don't need hazard pointers.

   The dequeue method needs to set a hazard pointer to the value read from head before defererencing it, and a second to the value read from head→next before dereferencing that. These reservations can be dropped at the end of the body of the while loop.

   The enqueue method needs to set a hazard pointer to the value read from tail before defererencing it, and a second to the value read from tail→next before defererencing that. These reservations can again be dropped at the end of the body of the while loop. The free_for_reuse routine will presumably be written to delay reclamation of any node for which a hazard pointer is outstanding.

5. (Hard) Consider the bounded obstruction free deque of Herlihy, Luchangco, and Moir, which was sketched in class. (The original was published at ICDCS 2003.) How might you extend this code to create an unbounded nonblocking deque comprising a linked list of arrays?

   Answer: A solution to this problem can be found in the paper by Graichen, Izraelevitz, and Scott at ICPP 2016.