Multicore Semantics & Programming

Exercise sheet (Tim Harris' section)

Each question carries equal marks, for a total of 50% of the course total

Core exercises (Part II and ACS/Part III)

1. Locks and performance

A machine has 4 processors, each with its own cache. A program runs with 1 thread per processor, and 1 mutual exclusion lock used by all of the threads. Each thread must acquire the lock, execute a short critical section, and then release the lock.

For each of the following locks (i) describe the cache line transfers that occur in a simple MESI cache protocol in the best-case execution in which threads happen to only attempt to acquire the lock when it is available, and (ii) describe any additional cache line transfers that may occur if threads attempt to acquire the lock at the same time.

- Test-and-set lock.
- Test-and-test-and-set lock.
- MCS lock.

You may assume that, by chance, the thread on processor 1 successfully acquires the lock first, then processors 2, 3, and 4. Also, you may assume that each QNode structure for the MCS lock is a single cache line in size.

2. Lock implementation

(i) In the slides, the pseudo-code for the MCS acquireMCS operation shows that the new QNode is only linked into the queue after performing a CAS operation on lock->tail. This makes the releaseMCS operation more complicated because it may need to wait in the while loop watching qn->next.

Therefore it is tempting to look for ways to update the links between the QNodes *before* updating the tail of the list in lock->tail. Show why it would be *incorrect* to do this, even if a CAS is used:

```
void acquireMCS(mcs *lock, QNode *qn) {
  QNode *prev;
  qn->flag = false;
  qn->next = NULL;
  while (true) {
     prev = lock->tail;
     if (prev == NULL || (CAS(&prev->next, NULL, qn) &&
                          CAS(&lock->tail, prev, qn))) break;
  }
  if (prev != NULL) {
    while (!qn->flag) { } // Spin
} }
```

(ii) A "ticket lock" is implemented using two shared counters, T and C, both initially 0. A thread wanting to acquire the lock uses an atomic fetch-and-add on T to obtain a unique sequence number. The thread then waits until C is equal to this sequence number. After releasing the lock, the thread increments C.

What are the advantages / disadvantages of this ticket lock compared with a test-and-test-and-set lock, and compared with the MCS queue lock?

3. Linearizability

Consider the following history of operations on a set implemented over a linked list. The set is initially empty. A call to insert(X) returns true if it succeeds in adding X to the set. A delete_ge(X) operation deletes the next value above or equal to X. It returns the value deleted, or -1 if there is no such value.

Thread 1 : Calls delete_ge(10)

Thread 2 : Calls insert(30)

Thread 2 : insert(30) returns true

Thread 3 : Calls insert(20)

Thread 3 : insert(20) returns true

Thread 4 : Calls insert(30)

Thread 4 : insert(30) returns false

Thread 1 : delete_ge(10) returns 30

Show that this concurrent history *is not* linearizable. Then, for each of the following alternatives, show that the resulting history *would be* linearizable:

(i) If the delete_ge(10) operation had returned -1 to thread 1.
(ii) If the delete_ge(10) operation could delete any value greater than or equal to 10 in the set.
(iii) If the operations in thread 4 executed before those in thread 3.

4. Lock freedom

Consider a simple shared counter that supports an "Increment" operation. Each increment advances the counter's value by 1 and returns the counter's new value – i.e., 1, 2, 3, etc.

(i) Explain whether or not the following history is linearizable:

- Time 0 : Thread 1 invokes Increment
- Time 10 : Thread 1 receives response 1
- Time 11 : Thread 1 invokes Increment
- Time 20 : Thread 2 invokes Increment
- Time 21 : Thread 1 receives response 3
- Time 22 : Thread 1 invokes Increment
- Time 30 : Thread 2 receives response 2
- Time 31 : Thread 1 receives response 4

(ii) In pseudo-code, give a lock-free implementation of "Increment" using an atomic compare and swap operation.

(iii) Explain whether or not your implementation is also wait-free.

5. Lock-free lists and memory management

Consider a lock-free linked list of integers, held in sorted order and shared between a large number of threads. Threads perform search, insert, and delete operations on the list.

Initially, assume that a garbage collector is used to reclaim storage automatically. Describe workloads (i) where the lock-free list is likely to perform better than a list protected by a single well-implemented mutual exclusion lock, and (ii) where the lock-free list is likely to perform worse than the lock-based list. In each case, describe the number of threads involved, the size of the list, and the mix of operations being performed on the list.

Suppose that a per-list-node reference counting scheme is used instead of garbage collection in the lock-free list. Are there now any cases where the lock-free list would still be preferable to the lock-based list? (Note that reference counting *would not* be needed in the lock-based list.)

Additional exercises (ACS/Part III only)

6. Work-stealing queues

The array-based deque in the slides supports one thread on the "bottom" and, and multiple threads stealing from the "top" end (slides 41—46).

Consider instead the case of a simpler array-based queue supporting a fixed maximum number of elements in the queue at any one time (N) and only a single producer (calling "pushTop") and a single consumer (calling "popBottom"). A push should return "true" if it succeeds (adding the item to the queue), and "false" otherwise (if the queue is full). A pop should return a data item if there is one in the queue, or NULL if the queue is empty.

In pseudo-code, give a lock-free linearizible implementation of this queue building on atomic compare and swap, read, and write.

7. Transactional memory

Transactional memory implementations are often classified as making eager or lazy updates and performing eager or lazy conflict detection.

Describe two workloads, one of which would perform well under eager-eager, and one which would perform well under lazy-lazy. Justify your answer in terms of (i) the series of reads and writes that are being attempted within the transactions, (ii) the amount of work executing the transactions initially, (iii) the amount of additional work attempting to commit the transactions, and (iv) the amount of additional work caused by transactional re-execution.

8. Alternatives

An enthusiastic researcher writes that "In the future all shared memory data structures will be lock-free because they are fast and scalable". With the aid of example data structures, and possible uses of these data structures, describe three cases in which you would agree with using lock-free data structures, and three cases where you would suggest using locking or using transactional memory instead.