Exam Concurrency & Multithreading

VU University Amsterdam, 22 October 2014, 15:15-18:00

(At this exam, you may use the textbook of Herlihy and Shavit. Answers can be given in English or Dutch. Use of the slides or a laptop is not *allowed.)*

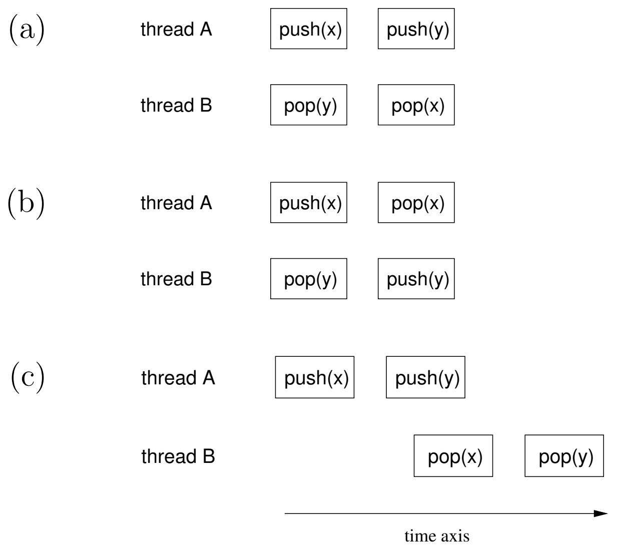
(The exercises in this exam sum up to 90 points; each student gets 10 points bonus.)

1. Does Peterson's two-thread mutual exclusion work if we replace the atomic registers `flag[0]`, `flag[1]` and `victim` by regular registers? (Explain your answer.) (10 pts)

2. Why can we in general not define first-come-first-served in a mutual exclusion algorithm based on the order in which the first instruction in the `lock()` method was executed on an atomic register?

   Argue your answer in a case-by-case manner, based on the nature of the first instruction executed by concurrent `lock()` calls: an atomic read or write, to separate locations or to the same location. (10 pts)

3. Consider the stack data structure, with the standard `push()` and `pop()` methods. Explain for each of the following three executions, by two threads, whether it is linearizable and/or sequentially consistent. (12 pts)



4. Suppose that in the 2-thread consensus protocol using a FIFO queue (Figure 5.7 on page 107), a thread first dequeues an element from the queue, and then writes its value in its slot in the `proposed` array (i.e., lines 13 and 14 are swapped). Would the resulting protocol be wait-free? (Explain your answer.) (8 pts)

5. Consider the following (blocking but lock-free) queue for a single enqueuer and a single dequeuer.

```
class TwoThreadLockFreeQueue<T> {
   int head, tail;
   T[] items;
   public TwoThreadLockFreeQueue(int capacity) {
      head = 0; tail = 0;
      items = T[] new Object[capacity]; }
   public void enq(T x) {
      while (tail - head == items.length) {};
      items[tail % items.length] = x;
      tail++; }
   public T deq() {
      while (tail == head) {};
      T x = items[head % items.length];
      head++;
      return x; }
}
```

Introduce a volatile Boolean variable `barrier`, and add one or more lines of code where this variable is set to `true`. Explain why this memory barrier is needed here, and how your adaptation guarantees that the queue works correctly. (12 pts)

6. Adapt the dissemination barrier to make it reusable. *(Hint: You may want to keep track of the sense and parity of the current phase.)*

   Consider two threads, where one thread is very slow in detecting that it can leave the barrier, while the other thread leaves the barrier and reaches it again. Argue that your reusable dissemination barrier works correctly (also) in this case. (14 pts)

7. Give a (Java pseudocode) implementation of the dequeue method for the unbounded transactional FIFO queue (using atomic blocks). (12 pts)

8. Wall clock time measures the actual elapsed time and includes waiting, while cpu time only measures the time used to actively process instructions.

   In the practical assignment you implemented multiple concurrent data structures and ran measurements using the provided framework. This framework allows you to set a "worktime" parameter: for each add and remove call, some additional (fully parallelizable) work is simulated. This is implemented using wall clock time:

   ```
   long endTime = System.nanoTime() + workTime;
   while (System.nanoTime() < endTime); // busy wait
   ```

   An alternative approach is to implement this using cpu time:

   ```
   long endTime = System.nanoTime() + workTime;
   while (bean.getCurrentThreadCpuTime() < endTime); // busy wait
   ```

   Explain which of the two would be the more realistic way of simulating the additional work. How does this influence the results if you are running more threads than there is parallelism on the machine? (12 pts)