Concurrent Systems — Exam

June 2017

Name: ____

Duration: 120 minutes — No document authorized

1.

a) Explain informally what is a *deadlock*. Describe a simple strategy to make sure that multiple threads that share a set of locks never deadlock when acquiring the locks.

b) Explain what is the principle of a *future* and describe briefly how one can use it (e.g., using pseudo-code).

c) The **`atomicAdd(int n)`** operation adds value **`n`** to an atomic integer. Can we implement this operation using **`compareAndSet()`**? If yes, provide a (pseudo-code) implementation.

d) Explain informally what is the issue with the simple "test-and-set" algorithm for implementing a spinlock, and why "test-and-test-and-set" typically performs better.

2.
Assume you have two threads, T_1 and T_2, executing the following code:

```
T₁:
  inst1₁
  inst2₁
T₂:
  inst1₂
  inst2₂
```

Implement a *rendezvous* point that guarantees that $\texttt{inst1}_1$ happens before $\texttt{inst2}_2$ and $\texttt{inst1}_2$ happens before $\texttt{inst2}_1$. There is no order constraint between $\texttt{inst1}_1$ and $\texttt{inst1}_2$. Use only semaphores in your implementation.

> **Reminder:** A counting semaphore has two main methods: **`acquire()`** and **`release()`**. It is initialized with a given number of permits. For each call to **`acquire()`** a permit is taken by the calling thread. For each call to **`release()`** a permit is returned to the semaphore. Thus, at most *N* threads can pass the **`acquire()`** method without any **`release()`** calls, where *N* is the number of permits the semaphore was initialized with. If there are no permits left, a thread calling **`acquire()`** will block until new permits are available.

3.
Consider the following barrier implementation pseudocode:

```
Lock lock = new ReentrantLock();
Semaphore sem = new Semaphore(0);  // Semaphore initially 0
int count = 0, n = NUM_THREADS;

// Reaching the barrier
lock.lock();
count = count + 1;
lock.unlock();
if (count == n) {
  sem.release();    // Increment semaphore
}
sem.acquire();      // Decrement semaphore
// Critical section follows
```

The variable **count** is used to keep track of how many threads reached the barrier. It is protected by a lock. **NUM_THREADS-1** threads will block when trying to acquire the semaphore. The last thread releases the semaphore and all threads can continue to the critical section.

Is this a correct barrier implementation? If not, explain what can go wrong and suggest a solution.

4.

a) Consider the following history for two threads **`T1`** and **`T2`** and two FIFO queues **`p`** and **`q`**:

```
T1 p.enq(x)
T2 q.enq(y)
T1 p:void
T1 q.enq(x)
T2 q:void
T2 p.enq(y)
T1 q:void
T1 p:deq()
T2 p:void
T2 q:deq()
T1 p:y
T2 q:x
```

Is it linearizable? ______________________________

Is it sequentially consistent? ______________________________

b) If we change the last line of the history to:

```
...
T2 q:y
```

Is it linearizable? ______________________________

Is it sequentially consistent? ______________________________

c) Consider again the original history of **a)**. Prepend one operation (method invocation and response) in front of the history so that it becomes both linearizable and sequentially consistent.

5.
Consider the following code:

```
class MyStream {

  static long n0 = 1;
  static long n1 = 1;

  static Stream<Long> get() {
    return Stream.generate(
            () -> {
              long r = n0 + n1;
              n0 = n1;
              n1 = r;
              return r;
            });
  }
}
```

a) Explain what this stream does.

____

____

____

____

____

____

b) What will be the output of the following code?

```
MyStream.get().limit(8).forEach(System.out::println);
```

____

____

c) Can we modify the code as follows? If not, explain why and propose a possible solution.

```
MyStream.get().parallel().limit(8).forEach(System.out::println);
```

____

____

____

____

____

____

6.
A **`CountDownLatch`** is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes. A **`CountDownLatch`** is initialized with a given count. The **`await()`** methods block until the current count reaches zero due to invocations of the **`countDown()`** method, after which all waiting threads are released and any subsequent invocations of await return immediately. This is a one-shot phenomenon; the count cannot be reset. Note that the **`CountDownLatch`** does not require that threads calling **`countDown()`** to wait for the count to reach zero before proceeding, it simply prevents any thread from proceeding past an **`await()`** until all threads could pass.

Provide a simple implementation of **`CountDownLatch`** using Java monitors.