Concurrent Systems — Exam

June 2014

Name: ______________________________________

Duration: 120 minutes — No document authorized

1.

a) What does it mean for a lock to be *reentrant*?

b) What is a *Future*?

c) Explain informally the notions of *work* and *critical path length* (as used when studying the parallelism of a multi-threaded program)?

d) How do `AtomicStampedReference` and `AtomicMarkableReference` differ from a classical `AtomicReference` in Java? What problem can they help solve?

e) Explain informally the principle of *exponential backoff*.

f) Give 3 examples of read-modify-write operations.

2.
A semaphore[1] is an abstract data type used for controlling access, by multiple processes, to a common resource in a parallel programming environment. A useful way to think of a semaphore is as a record of how many units of a particular resource are available, coupled with operations to safely (i.e., without race conditions) adjust that record as units are required or become free, and, if necessary, wait until a unit of the resource becomes available. Semaphores that allow an arbitrary resource count are called counting semaphores.

Consider the following implementation of a counting semaphore:

```
public class CountingSemaphore {
  private int available;

  public CountingSemaphore(int n) {
    available = n;
  }

  public synchronized void acquire() {
    while(available == 0)
      try { wait(); } catch(InterruptedException e) { }
    available--;
  }

  public synchronized void release() {
    available++;
    notify();
  }
}
```

Propose an equivalent implementation of a counting semaphore that is lock-free.

[1] https://en.wikipedia.org/wiki/Semaphore_(programming)

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

3.
Consider the simple barrier seen in the course:

```
public class Barrier {
  AtomicInteger count;
  int size;

  public Barrier(int n) {
    size = n;
    count = new AtomicInteger(size);
}

  public void await() {
    if (count.getAndDecrement() == 1)
      count.set(size);
    else
      while (count.get() != 0) { }
  }
}
```

What is the problem (limitation) with this implementation? Describe a scenario where this would problematic. Explain informally how one can fix this problem by modifying the barrier implementation.

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

____

4.

Consider a shared queue `Q` and two threads `T1` and `T2`. Are the following histories linearizable and/or sequentially consistent? If so, write the equivalent sequential history.

a)

```
T1 Q.enq(a)
T2 Q.deq()
T1 Q:void
T2 Q:b
T1 Q.deq()
T2 Q.enq(b)
T1 Q:a
T2 Q:void
```

b)

```
T1 Q.enq(a)
T2 Q.deq()
T1 Q:void
T2 Q:a
T1 Q.deq()
T2 Q.enq(b)
T1 Q:b
T2 Q:void
```

c)

```
T1 Q.enq(a)
T1 Q:void
T2 Q.enq(b)
T2 Q:void
T1 Q.deq()
T2 Q.deq()
T1 Q:a
T2 Q:b
```

d)

```
T1 Q.enq(a)
T1 Q:void
T2 Q.enq(b)
T1 Q.deq()
T1 Q:b
T2 Q:void
T2 Q.deq()
T2 Q:a
```

e) Can you think of a history that would be linearizable but not sequentially consistent?

5.

Consider the code below. Explain what it does, which properties it provides, and how a well-behaved client should use it.

```
import java.util.concurrent.atomic.*;

public class C {
  private AtomicInteger c, n;

  public C() {
    c = new AtomicInteger();
    n = new AtomicInteger();
  }

  public void f() {
    int i = n.getAndIncrement();
    while (c.get() != i);
  }

  public void g() {
    c.getAndIncrement();
  }
}
```

6.

A read-write lock allows either a single writer or multiple readers to execute in a critical section.

a) Provide an implementation of a read-write lock in Java. You can use synchronized methods and the wait/notify mechanism if you wish. The class should provide the 4 methods **`lockRead()`**, **`unlockRead()`**, **`lockWrite()`**, and **`unlockWrite()`**. This implementation does not need to be FIFO, starvation-free, nor reentrant.
HINT: You might want to keep track of the number of readers and writers.

```
public class ReadWriteLock {













}
```

b) Modify the implementation of your read-write lock so that it ensures that writers do not starve (i.e., readers cannot prevent writers from acquiring the lock infinitely). This implementation does not need to be FIFO nor reentrant.

```
public class ReadWriteLock {

}
```