Concurrent Systems — Exam

June 2012

Name: _____

Duration: 120 minutes — No document authorized

1.
a) What is false sharing?

_____

_____

_____

_____

_____

_____

b) We say that a concurrent system is “asynchronous”. What does that mean?

_____

_____

_____

_____

_____

_____

c) What is the difference between “safety” and “liveness” properties? Illustrate these concepts with an example.

_____

_____

_____

_____

_____

_____

d) What does Amdahl's law say? (You can give an informal explanation.)

_____

_____

_____

_____

_____

_____

e) Describe a typical cache hierarchy as found in a multi-core processor?

_____

_____

_____

_____

_____

_____

f) What are the semantics of read/write locks? What fairness properties are desirable for such locks?

_____

_____

_____

_____

_____

_____

2.
What are the "compareAndSet" and "getAndIncrement" operations? Describe their semantics informally or using pseudo-code.

Can we implement "getAndIncrement" using "compareAndSet"?
If the answer is "no", explain why.
If the answer is "yes", write the corresponding code and indicate if the implementation is lock-free, and/or wait-free.

3.
Consider the following 2-thread mutual exclusion algorithm seen in the course:

```
class MyLock implements Lock {
  private volatile boolean[] flag = new boolean[2];

  public void lock() {
    // Convention: i is local thread, j is other thread
    flag[i] = true;
    while (flag[j]) {}
  }

  public void unlock() {
    flag[i] = false;
  }
}
```

Does this algorithm ensure mutual exclusion?
What problem does it suffer from?
How can this problem be avoided?

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

4.

Consider a shared stack `S` and two threads `T1` and `T2`. Are the following histories linearizable and/or sequentially consistent? If so, write the equivalent sequential history. Note: It could be helpful to draw a graphical representation of the histories.

a)

```
T1 S.push(a)
T1 S:void
T2 S.push(b)
T1 S.pop()
T2 S:void
T1 S:b
T2 S.pop()
T2 S:a
```

b)

```
T1 S.push(a)
T1 S:void
T2 S.push(b)
T1 S.pop()
T2 S:void
T1 S:a
T2 S.pop()
T2 S:b
```

c)

```
T1 S.push(a)
T1 S:void
T2 S.pop()
T1 S.push(b)
T2 S:b
T2 S.pop()
T2 S:a
```

d)

```
T1 S.push(a)
T1 S:void
T2 S.pop()
T2 S:b
T1 S.push(b)
T2 S.pop()
T2 S:b
```

5.

Consider the following code that implements the well-known "double-checked locking" software design pattern.[1] Try to understand and explain the rationale of this approach. In particular, why is it better than a simpler implementation in which the **`getHelper()`** method would be synchronized?

```
class Foo {
    private volatile Helper helper = null;
    public Helper getHelper() {
        Helper result = helper;
        if (result == null) {
            synchronized(this) {
                result = helper;
                if (result == null) {
                    helper = result = new Helper();
                }
            }
        }
        return result;
    }
    // ...
}
```

[1] http://en.wikipedia.org/wiki/Double-checked_locking

6.

Write a simple *lock-based* queue that supports concurrent enqueues and dequeues (i.e., it must be possible for 2 threads to operate simultaneously on both ends of the queue if it is not empty). You are free to choose the design of your implementation (blocking or non-blocking, bounded or unbounded).