# Concurrent Systems — Exam

**June 2015**

**Name:** _____

**Duration: 120 minutes — No document authorized**

**1.**

**a)** Explain informally what is a *barrier* and how it can be used?

_____

_____

_____

_____

_____

_____

**b)** Explain informally the difference between the various cache associativity designs discussed in the course (*fully associative*, *direct mapped*, *k-way set associative*)?

_____

_____

_____

_____

_____

_____

**c)** The *Anderson* and *CLH* locks solve some major limitations of spin locks. Which are these limitations?

_____

_____

_____

_____

_____

_____

**d)** The class **`AtomicMarkableReference<V>`** provides a method **`V get(boolean[] mark)`** that "returns the current values of both the reference and the mark". Can you explain why it takes an array as parameter?

**e)** What do we mean when we talk about *linear scalability* for a concurrent algorithm?

**f)** Explain informally what is the *ABA problem*.

**2.**
Consider the following interface for a *Consensus* object:

```
public interface Consensus {
  // Propose value v and return agreed-upon value
  Object decide(Object v);
}
```

The consensus requires each thread to start with a non-null input value, given as parameter to the `decide()` method in this simplified interface. All threads must eventually agree on a common input value, which is returned by the `decide()` method.

We can easily implement an algorithm using locks that allows an arbitrary number of threads to reach consensus, as follows:

```
public class LockConsensus implements Consensus {
  Object decision = null;

  synchronized Object decide(Object v) {
    if (decision == null)
      decision = v;
    return decision;
  }
}
```

**a)** Can you propose a *lock-free* consensus algorithm between an arbitrary number of threads with compare-and-swap?

**b)** Can you propose a *lock-free* consensus algorithms between 2 threads using get-and-set?

**3.**

In computer science, load-link and store-conditional (LL/SC) are a pair of instructions used in multithreading to achieve synchronization. Load-link returns the current value of a memory location, while a subsequent store-conditional to the same memory location will store a new value only if no updates have occurred to that location since the load-link. Together, this implements a lock-free atomic read-modify-write operation.[1]

Consider the following class that represents an atomic integer with support for LL/SC operations.

```
public class AtomicValue {
  private int value;

  public AtomicValue (int v) {
    value = v;
  }

  // Return current value
  public native int loadLinked();              // Code not shown

  // Store new value if not modified since last loadLinked()
  // Return true on success, false otherwise
  public native bool storeConditional(int v); // Code not shown

  // Insert your code here
}
```

**a)** Can you extend this class by adding methods: `int getAndIncrement()`, `int getAndSet(int value)`, and `bool compareAndSet(int expectedValue, int newValue)`, implemented using only `loadLinked()` and `storeConditional()`? If so, write the corresponding code. Otherwise, explain why this is not possible.

[1] https://en.wikipedia.org/wiki/Load-link/store-conditional

**b) [Bonus]** Can you think of a reason why LL/SC would be more powerful than compare-and-swap?

**4.**

Consider an integer register (i.e., variable) `r` and three threads `T1`, `T2`, and `T3`. Draw a graphical representation of the following histories and indicate if they are linearizable and/or sequentially consistent? If so, write the equivalent sequential history.

**a)**

```
T2 r.write(1)
T1 r.read()
T3 r.write(2)
T1 r:1
T3 r:void
T2 r:void
T2 r:read()
T2 r:2
```

**b)**

```
T2 r.write(1)
T1 r.read()
T3 r.write(2)
T1 r:1
T3 r:void
T2 r:void
T2 r:read()
T2 r:1
```

**5.**

Consider the code below. According to the Java memory model, can the reader method ever divide by zero? If so, how can we fix the program with a minimal amount of changes?

```
class Example {
  int x = 0;
  boolean v = false;
  public void writer() {
    x = 42;
    v = true;
  }
  public void reader() {
    if (v) {
      int y = 100 / x;
    }
  }
}
```

**6.**
The following code shows an incorrect implementation of *CLHLock* in which a thread reuses its own node instead of its predecessor node. Explain how this implementation can go wrong.

```
public class BadCLHLock implements Lock {
  AtomicReference<QNode> tail = new AtomicReference<QNode>(new QNode());
  ThreadLocal<QNode> myNode = new ThreadLocal<QNode>() {
    @Override protected QNode initialValue() { return new QNode(); }
  };

  public void lock() {
    QNode node = myNode.get();
    node.locked = true;
    QNode pred = tail.getAndSet(node);
    while (pred.locked) { }
  }

  public void unlock() {
    myNode.get().locked = false;
  }

  static class QNode {
    public boolean locked = false;
  }
}
```