Concurrent Systems — Exam

June 2016

Name: _____

Duration: 120 minutes — No document authorized

1.

a) What is the difference between **`Callable`** and **`Runnable`**?

_____

_____

_____

_____

_____

_____

b) What is the difference between **`wait()`** and **`sleep()`**?

_____

_____

_____

_____

_____

_____

c) Can we emulate **`getAndSet()`** using just **`compareAndSet()`** (without using locks)? If no, justify. If yes, provide pseudo-code of a possible implementation.

_____

_____

_____

_____

_____

_____

d) The **`ReentrantReadWriteLock`** class provided by the **`java.util.concurrent.locks`** package does not allow a thread holding the lock in read mode to then access that lock in write mode (the thread will block). Justify this design decision by sketching what it would take to permit such lock upgrades.

2.
Consider the classical producer-consumer problem: a group of *P* producer threads and a group of *C* consumer threads share a bounded circular buffer. If the buffer is not full, producers are allowed to add elements; if the buffer is not empty, consumers can consume elements. Let's assume that the system uses the buffer implementation below, where head and tail point to the ends of the buffer where the items can be consumed, respectively produced.

```
public class BoundedBuffer {
  int head = 0, tail = 0;
  Object[QSIZE] items;

  public synchronized void put(Object x) {
    while (tail - head == QSIZE)
      this.wait();
    items[tail % QSIZE] = x;
    tail++;
    this.notifyAll();
  }
  public synchronized Object get() {
    while (tail == head)
      this.wait();
    Object x = items[head % QSIZE];
    head++;
    this.notifyAll();
    return x;
  }
}
```

a) Is this implementation deadlock-free? Explain.

b) What if we replace **`notifyAll()`** with **`notify()`**?

3.
Consider the Bakery algorithm presented in the lecture. Does the algorithm still provide mutual exclusion if the labels are bounded and can overflow (i.e., if **`label+1 > MAX_VALUE`**, then **`label+1 = 0`**)? What about deadlock-freedom?

As a reminder, the pseudo-code of the lock function for thread **`i`** is given below:

```
public void lock() {
  flag[i] = true;
  label[i] = max(label[0], …, label[n-1]) + 1;
  while ((∃k | flag[k]) && (label[i],i) > (label[k],k)) {}
}
```

4.

Consider an integer set **s**, initially empty, and two threads **T1** and **T2**. We assume that methods **add()** and **remove()** do not return any result and, upon completion, leave the set with one copy, respectively no copy, of the integer passed as parameter. 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)

```
T1 s.contains(1)
T2 s.add(1)
T2 s:void
T1 s:false
T2 s.remove(1)
T2 s:void
```

b)

```
T2 s.contains(1)
T1 s.contains(1)
T2 s:false
T1 s:true
T2 s.add(1)
```

c)

```
T1 s.add(1)
T2 s.contains(1)
T2 s:true
T2 s.contains(1)
T2 s:false
```

5.
Consider the following code:

```
Arrays.asList("Arlington",
              "Berkeley",
              "Clarendon",
              "Dartmouth",
              "Exeter")
    .stream()
    .forEach(s -> printf("%s\n", s));
```

Is this code allowed to output the elements of the list out of order (e.g., **`"Berkeley -> Arlington -> Clarendon -> Exeter -> Dartmouth"`**)? If yes, justify. If not, show how one can modify the code so that such an output is possible.

6.
Consider the following implementation of a read/write lock:

```
public class ReadWriteLock{
  private int readers = 0;
  private int writers = 0;

  public synchronized void lockRead() throws InterruptedException {
    while (writers > 0) wait();
    readers++;
  }
  public synchronized void unlockRead() {
    readers--;
    notifyAll();
  }
  public synchronized void lockWrite() throws InterruptedException {
    while(readers > 0 || writers > 0) wait();
    writers++;
  }
  public synchronized void unlockWrite() throws InterruptedException {
    writers--;
    notifyAll();
  }
}
```

Modify this implementation so that it becomes starvation-free on the writer side (i.e., readers cannot prevent writers from acquiring the lock infinitely).