Guidelines

Put your name on each page before starting the exam. Write your answers directly on the exam sheets, using the back of the page as necessary. If you finish with more than 15 minutes left in the class, then bring your exam to the front when you are finished and leave the class as quietly as possible. Otherwise, please stay in your seat until the end.

If you have a question, raise your hand and I will come to you. Note, that I am unlikely to answer general questions however. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn’t need to do this at all, so be careful when making assumptions.
1. Futures (10 Points). The Count class shown below implements the Callable interface. Its call() method computes the number of times a given integer appears in a slice of an array of integers.

class Count implements Callable<Integer> {
    ...
    /**
     * @param data - integer array
     * @param startIndex - beginning index for searching
     * @param endIndex - ending index for searching
     * @param searchKey - key to be searched for
     */
    public Count(int[] data, int startIndex, int endIndex, int searchKey) {
        ...
    }

    /**
     * returns the number of times searchKey appears in data array
     */
    public Integer call() {
        ...
    }
}

The CountOccurrences class uses the Count class to compute, in parallel, the number of times an integer appears in a given array of integers. The CountOccurrences class includes a countOccurences() method which manages the parallel computations.

Specifically, the countOccurences() method breaks down the entire computation into multiple subcomputations which it then runs in parallel. To do this it creates several instances of Count, and submits them to an ExecutorService. The countOccurences() method then sums the results of each subcomputation and prints out the final result.

Fill out the countOccurences() method below to implement this behavior. You will need to use the Future.get() and the ExecutorService.submit() methods to do this. The Future.get() method returns the value of the Future. The ExecutorService.submit() submits the Callable to a thread pool and returns a Future representing the results of the Callable.

public class CountOccurrences {
    public static void main(String args[]) {
        ExecutorService executor = Executors.newCachedThreadPool();
        countOccurences(/* data array */, /* key */,
                         executor, /* number of subcomputations */);
        executor.shutdown();

        private static void countOccurences(int[] data, int searchKey,
                                             ExecutorService executor, int numFutures) {
            // FILL IN CODE ON NEXT PAGE
        }
    }
}
private static void countOccurences(int[] data, int searchKey, ExecutorService executor, int numFutures) {

}
2. Deadlock (15 points). Consider the following code. The run() method must acquire locks on two Fork objects, then call the eat() method, then release the two locks. As it is currently written, calling the main() method is likely to result in a deadlock. On the following page rewrite the run() method so that it uses a lock-ordering strategy that prevents deadlock from occurring.

```java
class Fork {}

public class DiningPhilosopher implements Runnable {
    Fork f1, f2;

    public DiningPhilosopher(Fork f1, Fork f2) {
        this.f1 = f1; this.f2 = f2;
    }

    private void eat() {} 

    public void run() {
        while (!Thread.interrupted()) {
            synchronized (f1) {
                synchronized (f2) {
                    eat();
                }
            }
        }
    }

    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        Fork f1 = new Fork(), f2 = new Fork(), f3 = new Fork();
        exec.execute(new DiningPhilosopher(f1, f2));
        exec.execute(new DiningPhilosopher(f2, f3));
        exec.execute(new DiningPhilosopher(f3, f1));
        exec.shutdown();
    }
}
```
public void run() {

Object tieLock = new Object();

int f1Hash = System.identityHashCode(f1);
int f2Hash = System.identityHashCode(f2);

while (!Thread.interrupted()) {
    if (f1Hash < f2Hash) {
        synchronized (f1) {
            synchronized (f2) {
                eat();
            }
        }
    } else if (f1Hash > f2Hash) {
        synchronized (f2) {
            synchronized (f1) {
                eat();
            }
        }
    } else {
        synchronized (tieLock) {
            synchronized (f1) {
                synchronized (f2) {
                    eat();
                }
            }
        }
    }
}
}
3. The Happens Before relation. (25 points). In class we discussed the Happens-Before relation. Some of the definitions we used are recreated below. On the next page there is a class called HappensBeforeClass. Assume that another class C creates an instance of the HappensBeforeClass, called hb, and also creates two different threads, T1 and T2. T1 calls hb.foo() and then exits, T2 calls hb.bar() and then exits. Using the definitions of the Happens-Before relation given below prove or disprove the existence of a data race between T1 and T2 involving some field in in hb. Be explicit and show all your work.

1. A trace is a sequence of events.

Events E ::= start(T) | end(T)  
               | read(T,x,v)  
               | write(T,x,v)  
               | lock(T,x)  
               | unlock(T,x)

2. Let E1 < E2 be the ordering of events as they appear in the trace.

3. Define happens-before ordering <: in a trace R as follows:
   E1 <: E2 iff E1 < E2 and one of the following holds:
   a) thread(E1) = thread(E2)  
   b) E1 is unlock(T1,x) and E2 lock(T2,x)  
   c) there exists E3 with E1 <: E3 and E3 <: E2

4. Updates are visible based on the following rules. For a trace R containing EW == write(T1,x,v1) and ER == read(T2,x,v2):
   EW "is not visible" to ER if
   - ER <: EW  
   - There exists some event EW2 == write(T,x,v3) such that EW <: EW2 <: R
   Otherwise EW is visible at ER

5. A data race takes place when there are two events in trace R that
   access the same memory location
   at least one is a write
   they are unordered according to happens-before
public class HappensBeforeClass {
    Integer x = new Integer(0);
    static Integer y = new Integer(0);

    public void foo() {
        synchronized (y) {y = x + 1;}
    }

    public void bar () {
        x = 3;
        synchronized (y) {y = 2;}
    }
}

Example trace:
Lock(T1,y) < Read(T1,x,0) < Write(T2,x,3) < Write(T1,y,1) < Unlock(T1,y)
< Lock(T2,y) < Write(T2,y,2) < Unlock(T2,y)

Happens-Before relation:
Lock(T1,y) Write(T2,x,3)
<: Read(T1,x,0) |------>
<: Write(T1,y,1) |------>
<: Unlock(T1,y) |------>
|------> <: Lock(T2,y)
<: Write(T2,y,2)
<: Unlock(T2,y)

Write and Read to x are unordered by Happens-Before relation, so there is a data race.
4. Guarded Suspension with Locks and Conditions (25 Points). Fill in the code below to create a thread-safe BoundedBuffer. The BoundedBuffer has a read() method that removes and returns an element from the BoundedBuffer. It has a write() method that inserts an element into the BoundedBuffer. Attempts to read from an empty BoundedBuffer or to write to a full BoundedBuffer should block. Additionally, the read() and write() methods should allow writers to proceed before readers whenever possible. Your implementation should work correctly for any number of reader and writer threads.

Your implementation should make use of the Lock and Condition interfaces to implement this functionality. Your implementation may not use any intrinsic locking methods, such as synchronized blocks, Object.wait(), Object.notify(), etc. Your implementation should use Conditions to minimize the number of Threads that must be put to sleep and woken up. You will likely need the Lock.lock(), Lock.unlock(), Lock.newCondition(), Condition.await() and Condition.signal().

```java
class Buffer {
    private Queue<Integer> contents;
    private int capacity;
    // FILL IN CODE AS NECESSARY

    public Buffer(int capacity) {
        contents = new LinkedList<Integer>();
        this.capacity = capacity;
    }
    // FILL IN CODE AS NECESSARY
    public int read() {
        readers++;
        while (contents.size() <= 0 || (writers > 0 && contents.size() < capacity)) {

        }

        readers--;
        Integer val = contents.remove();
    }
}
```
// FILL IN CODE AS NECESSARY

public void write(int value) {

    writers++;  
    while (contents.size() >= capacity) {

    }

    contents.add(value);  
    writers--;  

}  
}
5. Nonblocking Data Structures (25 Points). The code below implements a nonblocking queue. Threads can safely use the queue without holding a Java lock. Starting with an empty queue, show a trace that involves 2 threads and in which one thread’s operation interferes with the other thread’s operation. Your trace should be given in table form on the following page. Each row of the table represents one time step in the trace. At each time step indicate the line number that is currently being executed by each thread. If a thread gets blocked or takes more than one time step to execute a single line, repeat the line number over multiple rows.

```java
public class ConcurrentQueue<E> {
    private static class Node<E> {
        final E item;
        final AtomicReference<Node<E>> next;
        public Node(E item, Node<E> next) {
            this.item = item;
            this.next = new AtomicReference<Node<E>>(next);
        }
    }

    private final Node<E> dummy = new Node<E>(null, null);
    private final AtomicReference<Node<E>> head = new AtomicReference<Node<E>>(dummy);
    private final AtomicReference<Node<E>> tail = new AtomicReference<Node<E>>(dummy);

    public boolean put(E item) {
        Node<E> newNode = new Node<E>(item, null);
        while (true) {
            Node<E> curTail = tail.get();
            Node<E> tailNext = curTail.next.get();
            if (curTail == tail.get()) {
                if (tailNext != null) {
                    tail.compareAndSet(curTail, tailNext);
                } else {
                    if (curTail.next.compareAndSet(null, newNode)) {
                        tail.compareAndSet(curTail, newNode);
                        return true;
                    }
                }
            }
        }
    }

    public E take() {
        for (;;) {
            Node<E> oldHead = head.get();
            Node<E> oldTail = tail.get();
            Node<E> oldHeadNext = oldHead.next.get();
            if (oldHead == head.get()) {
                if (oldHead == oldTail) {
                    if (oldHeadNext == null) {
                        return null;
                    } else {
                        tail.compareAndSet(curTail, tailNext);
                    }
                } else {
                    if (head.compareAndSet(oldHead, oldHeadNext)) {
                        return oldHeadNext.item;
                    }
                }
            }
        }
    }
}
```
<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11