Midterm Exam #1

CMSC 433
Programming Language Technologies and Paradigms
Spring 2014

March 13, 2014

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.

Question	Points	Score
1	15	
2	15	
3	25	
4	20	
5	25	
Total	100	

1. Short answers (15 points). Give very short (1 to 2 sentences for each issue) answers to the following questions. **Longer responses to these questions will not be read.**

   (a) What is a data race? If a program has a data race will it always also have race condition? Why or why not.

   Answer:

   A data race occurs when two threads access the same memory location, at least one of the accesses is a write, and the accesses are not ordered by the happens-before relation. A program that has a data race will not always also have a race condition. This can happen because the data race does not affect the program's correctness.

   (b) In Java, what are the three functions of synchronization?

   Answer:

   (1) mutual exclusion, (2) ordering, and (3) visibility.

   (c) Java provides an intrinsic lock with every Object. Later versions of Java provide a Lock interface and Lock implementations that also allow programmers to define and use locks. Briefly describe one benefit and one drawback of using the newer Lock interface.

   Answer:

   Benefits – multiple wait sets per Object, higher performance in some cases, additional Lock styles, such as ReaderWriterLocks
   Drawbacks – user has to manually ensure that Locks are released

   (d) In class we explained that you should avoid starting a Thread within that Thread's constructor. Why is this the case?

   Answer:

   Until the constructor finishes, the Thread object has not been fully created and therefore not safely published.

(e) Explain the difference between the *guarded suspension* and optimistic trying strategies for dealing with state-dependent operations.

Answer:

Guarded suspension acquires a lock, waits until some condition holds, then performs the operation
Optimistic trying does not acquire locks, performs the operation and then chekcks to see whether the operation completed without interference. If there was interference, it rolls back the operation.

2. Deadlock. (15 points).

(a) What are the 4 necessary and sufficient conditions required for a deadlock to occur in a multithreaded program?

Answer:

i. *Mutual exclusion: a non-sharable resource exists*
ii. *Hold and wait: processes already holding resources may request new resources held by other processes.*
iii. *No preemption: No resource can be forcibly removed from a process holding it.*
iv. *Circular wait: two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds.*

(b) The Resource.main() method shown below is susceptible to deadlock. Examine this code to expose this potential deadlock. Based on your analysis show the shortest sequence of method calls and lock acquisition attempts you can find that leads to the deadlock.

```
public class Resource {
  static private final int MAX = 16;
  private int mCurrentAmount = new Random().nextInt(MAX);

  public void transfer(Resource otherResource) {
    synchronized (this) {
      synchronized (otherResource) {
        int amountToTransfer =
             Math.min(mCurrentAmount,
                      new Random().nextInt(MAX - otherResource.mCurrentAmount));
        mCurrentAmount -= amountToTransfer;
        otherResource.mCurrentAmount += amountToTransfer;
      }
    }
  }

  public static void main(String[] args) {
    final Resource c1 = new Resource(), c2 = new Resource();
    new Thread(new Runnable() {
      public void run() { c1.transfer(c2); }
    }).start();

    new Thread(new Runnable() {
      public void run() { c2.transfer(c1);}
    }).start();
  }
}

over...
```

```
// PUT METHOD CALL SEQUENCE HERE
```

Answer:

T1.run () c1.transfer(c2) T1 attempts to acquire lock on c1 T2.run() c2.transfer(c1) T2 attempts to acquire lock on c2 T1 attempst to acquire lock on c2 T2 attempst to acquire lock on c1

3. The Happens Before relation. (25 points). In class we discussed the Happens-Before relation. Some the 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. On the next page there is also an execution trace generated from a run of this program. Using the definitions of the Happens-Before relation given below, and the provided execution trace, prove the existence of a data race between T1 and T2 involving some field in hb. Show 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
```

```
public class HappensBeforeClass {
    Integer x = new Integer(0), y = new Integer(0);

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

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

Here is an execution trace generated from this program:

```
  Lock(T1,y)
  Write(T1,x,1)
  Unlock(T1,y)
  Read(T1,x,1)
  Write(T1,y,1)
  Lock(T2,y)
  Read(T2,x,1)
  Write(T2,y,1)
  Unlock(T2,y)
  Read(T2,y,1)
  Write(T2,x,1)
```

Answer:

```
Happens-Before relationships:

Lock(T1,y) <: Write(T1,x,1) <: Unlock(T1,y) <: Read(T1,x,1) <: Write(T1,y,1)
                                    :----> Lock(T2,y) <: Read(T2,x,1)
                                           <: Write(T2,y,1) <: Unlock(T2,y)
                                           <: Read(T2,y,1) <: Write(T2,x,1)
```

4. (Guarded Suspension (20 Points). Fill in the code below to create a thread-safe BoundedBuffer, that uses guarded suspension. The BoundedBuffer has a take() method that removes and returns an element from the BoundedBuffer. It has a put() 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, your implementation may only use Java intrinsic locks and intrinsic wait sets and your implementation should not deadlock.

```
class BoundedBuffer {
 private static final int CAPACITY = 10;
 private final List<Integer> mItems = new ArrayList<Integer>(10);

 public void put(Integer x) {
 // FILL IN CODE HERE









 }

 public Integer take() {
 // FILL IN CODE HERE









 }
}
```

```
class BoundedBuffer {
  private static final int CAPACITY = 10;
  private final List<Integer> mItems = new ArrayList<Integer>(10);

  public void put(Integer x) {
    synchronized (this) {
      while (CAPACITY == mItems.size()) {
        try { wait(); } catch (InterruptedException e) {e.printStackTrace();}
      }
      mItems.add(x);
      notifyAll();
    }
  }

  public Integer take() {
    synchronized (this) {
      while (0 == mItems.size()) {
        try { wait(); } catch (InterruptedException e) {e.printStackTrace();}
      }
      notifyAll();
      return mItems.remove(0);
    }
  }
}
```

5. Synchronizers (25 Points). The PrimeFinder class determines whether a given target number is or is not prime. To do this quickly the main() method starts three Threads, running them concurrently. Each of these threads examines a range of numbers, searching for a number that is a factor of the target. Only when all the Threads have finished, the main Thread examines their results to determine if any of them have found a factor of the target. If any are found, then the target is not prime.

   Fill in the code below to complete the PrimeFinder. **Note**: Don't worry if you can't remember the exact names of the required method calls. Just use a descriptive name and explain what the method is supposed to do.

```
class Divider implements Runnable {
  private final int mTarget, mMin, mMax;
  private boolean mResult = false;
  private final CountDownLatch mAllDone;

  Divider(int target, int min, int max, CountDownLatch allDoneLatch) {
    mTarget = target;
    mMin = min; mMax = max;
    mAllDone = allDoneLatch;
    }

  @Override
  public void run() {

    for (int idx = mMin; idx < mMax; idx++) {
      if (mTarget % idx == 0) {
      mResult = true;
      }
    }
    // FILL IN CODE HERE









  }

  public boolean isDivisible() {
    return mResult;
  }
}

over.....
```

```
public class PrimeFinder {

  public static void main(String[] args) {
    int target = Integer.parseInt(args[0]);
    if (target < 2) { return; }
    int sqrtTarget = (int) Math.sqrt(target) + 1;
    int[] breaks = { Math.max(2, sqrtTarget / 3),
                     Math.min(2 * sqrtTarget / 3, sqrtTarget) };
    // FILL IN CODE HERE
```

```
    if (divider1.isDivisible() ||
        divider2.isDivisible() ||
        divider3.isDivisible()) {

      System.out.println(``target:'' + target + `` is not prime'');
      } else {
      System.out.println(``target:'' + target + `` is prime'');
      }
    }
}
```

```
class Divider implements Runnable {
  private final int mTarget, mMin, mMax;
  private boolean mResult = false;
  private final CountDownLatch mAllDoneLatch;

  Divider(int target, int min, int max, CountDownLatch allDoneLatch) {
    mTarget = target;
    mMin = min;
    mMax = max;
    mAllDoneLatch = allDoneLatch;
  }

  @Override
  public void run() {

    for (int idx = mMin; idx < mMax; idx++) {
      if (mTarget % idx == 0) {
        mResult = true;
      }
    }

    mAllDoneLatch.countDown();
  }

  public boolean isDivisible() { return mResult; }

}
```

```
public class PrimeFinder {

  public static void main(String[] args) {

    int target = Integer.parseInt(args[0]);
    if (target < 2) { return; }

    int sqrtTarget = (int) Math.sqrt(target) + 1;
    int[] breaks = { Math.max(2, sqrtTarget / 3),
                     Math.min(2 * sqrtTarget / 3, sqrtTarget) };

    CountDownLatch allDoneLatch = new CountDownLatch(1);

    Divider divider1 = new Divider(target, 2, breaks[0], allDoneLatch);
    new Thread(divider1).start();

    Divider divider2 = new Divider(target, breaks[0], breaks[1], allDoneLatch);
    new Thread(divider2).start();

    Divider divider3 = new Divider(target, breaks[1], sqrtTarget, allDoneLatch);
    new Thread(divider3).start();

    try {
      allDoneLatch.await();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    if (divider1.isDivisible() || divider2.isDivisible() || divider3.isDivisible()) {
      System.out.println(‘‘target:’’ + target + ‘‘ is not prime’’);
    } else {
      System.out.println(‘‘target:’’ + target + ‘‘ is prime’’);
    }
  }
}
```