Midterm Exam #1

CMSC 433
Programming Language Technologies and Paradigms
Spring 2011

March 3, 2011

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) We explained that concurrency brought both benefits and risks to program development. Give two examples the benefits of programming with concurrency. Give two examples, different from the benefits, of risks associated with concurrency.

   Answer:

   Advantages: Performance, Natural way to express some problems, Isolation of computations.
   Risks: Complexity, Overhead

   (b) What is thread scheduling? Describe briefly.

   Answer:

   Thread scheduling is the process of determining which thread(s) should be run at any given time.

   (c) What are the 3 functions of synchronization in Java? Describe each function briefly.

   Answer:

   Atomicity–exclusive operation
   Visibility–when are updates made visible to threads
   Ordering–which operations happen-before other operations

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 following code can deadlock. Show the smallest sequence of method calls you can that leads to deadlock.

```
class Obj {
  public synchronized void f1(Obj p1) {p1.f2(this);}
  public synchronized void f2(Obj p2) {}
}

public class Simple {
  public static void main(String[] args) {
    final Obj o1 = new Obj(), o2 = new Obj();
    // Thread T1
    (new Thread(new Runnable() {
      public void run() {while (true) {o1.f1(o2);}}})).start();

   // Thread T2
   (new Thread(new Runnable() {
     public void run() {while (true) {o2.f1(o1);}}})).start();
  }
}
```

   Answer:

   T1 calls o1.f1(o2)
   T2 calls o2.f1(o1)
   T1 calls o2.f2(o1) – blocks
   T2 calls o1.f2(o2) – blocks

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. Using the definitions of the Happens-Before relation given below prove the existence of a data race between T1 and T2 involving some field in 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
   they are unordered according to happens-before
```

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

    public void foo() {
      synchronized (x) {x = 1;}
      y = 2;
    }

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

Answer:

```
Example trace:

Lock(T1,x) < Write(T1,x,1) < Unlock(T1,x) < Lock(T2,x)
< Write(T1,y,2) < Read(T2,x,1) < Write(T2,y,1)

Happens-Before relation:

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

4. (Guarded Suspension (20 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 allows writers to proceed before readers whenever possible. Your implementation should work correctly for any number of reader and writer threads.

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



  public Buffer(int capacity) {
    contents = new LinkedList<Integer>();
    this.capacity = capacity;
  }

  public int read() {
  // FILL IN CODE HERE AS NECESSARY



    return contents.remove();
}
```

```
 public void write(int value) {
  // FILL IN CODE HERE AS NECESSARY





    contents.add(value);
  }
}
```

Answer:

```
public class ReaderWriter {

  static class Buffer {
    private Queue<Integer> contents;
    private int capacity, writers;

    public Buffer(int capacity) {
      contents = new LinkedList<Integer>();
      this.capacity = capacity;
    }

    public synchronized int read() {

      while (contents.size() <= 0 ||
            (writers > 0 && contents.size() < capacity)) {
        try {
          wait();
        } catch (InterruptedException e) {}
      }

      notifyAll();
      return contents.remove();
    }

  public synchronized void write(int value) {
    writers++;
    while (contents.size() >= capacity) {
      try {
        wait();
      } catch (InterruptedException e) {}
    }

    notifyAll();
    writers--;
    contents.add(value);
   }
}
```

5. Cyclic Barriers (25 Points). The MeetUp class simulates a set of particles that move one step at a time to a rendezvous point. For example, this might be used to model people that are meeting at a particular spot. or to simulate particles that are attracted by gravity towards a single large body.

   Each Particle contains a Location object representing its current location in a 2D plane. Particles support at least two methods: equals() and getCloser(). The equals(Object) method returns true if the Location has the same x and y coordinates as the parameter. Otherwise it returns false. The getCloser(Location) method will move the Particle one step closer to given rendezvous point. You can assume that these methods are provided for you.

   Fill in the following code skeleton to complete the MeetUp class. Once completed, the code should move the Particles one step at a time toward the rendezvous point. It should use a CyclicBarrier to coordinate the Particle's movements (all Particles take one step, then the process repeats). At each step the CyclicBarrier must execute a Runnable that determines whether all Particles have arrived at the rendezvous point. No Particle should exit its run() method until all Particles have arrived at the rendezvous point.

   The code for determining whether all Particles have arrived at the rendezvous point is encapsulated within the ConvergenceTester class. There are 2 CyclicBarrier methods that you will need to use.

   - public int await() throws InterruptedException,BrokenBarrierException. This method waits until all parties have invoked await on this barrier.
   - CyclicBarrier(int parties, Runnable barrierAction). This constructor method creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it, and which will execute the given barrier action when the barrier is tripped, performed by the last thread entering the barrier.

```
public class MeetUp {
  final private List<Particle> particles = new ArrayList<Particle>();
  final private Location center;
  final private CyclicBarrier barrier;
  final private int N;
  // FILL IN CODE AS NECESSARY HERE




  MeetUp(int N, Location center) {
    this.center = center;   this.N = N;
    // CREATE THE  BARRIER HERE
    barrier =



  }
  public static void main(String[] args) {
    // Location of rendezvous point
    Location center = new Location(/*x value*/, /* y value */);
    MeetUp mu = new MeetUp(/* number of particles */, center);
    mu.go();
  }

  // Starts each Particle in its own Thread
  void go() {
    for (int i = 0; i < N; i++) {
      Particle w = new Particle(new Location(/* x value */, */ y value */));
      particles.add(w);
      new Thread(w).start();
    }
  }
```

```
  // Determines whether all Particles have arrived at the rendezvous point
  class ConvergenceTester implements Runnable {
    // FILL IN CODE HERE
    public void run() {













    }
  }

  class Particle implements Runnable {
    Location myLoc;
    Particle(Location myLoc) { this.myLoc = myLoc; }

    // FILL IN CODE HERE. Use myLoc.getCloser(center)
    // to move the particle one step closer to the center
    public void run() {






















    }
  }
}
```

Answer:

```
public class MeetUp {
  final private List<Particle> particles = new ArrayList<Particle>();
  final private Location center;
  final private CyclicBarrier barrier;
  final private int N;
  volatile boolean done = false;

  MeetUp(int N, Location center) {
    this.center = center;
    this.N = N;
    barrier = new CyclicBarrier(N, new ConvergenceTester());
  }

  public static void main(String[] args) {
    Location center = new Location(/*x value*/, /* y value */);
    MeetUp mu = new MeetUp(/* number of particles */, center);
    mu.go();
  }

  void go() {
    Random r = new Random();
    for (int i = 0; i < N; i++) {
      Particle w = new Particle(new Location(r.nextInt(10), r.nextInt(10)));
      particles.add(w);
      new Thread(w).start();
    }
  }

  class ConvergenceTester implements Runnable {
    public void run() {
      for (Particle w : particles) {
        if (!w.myLoc.equals(center)) { return; }
      }
      done = true;
    }
  }

  class Particle implements Runnable {
    Location myLoc;
    Particle(Location myLoc) { this.myLoc = myLoc; }
    public void run() {
      while (!done) {
        myLoc.getCloser(center);
        try {
          barrier.await();
        } catch (InterruptedException e) {
        } catch (BrokenBarrierException e) {
        }
      }
    }
  }

}
```