Midterm Exam #1

CMSC 433
Programming Language Technologies and Paradigms
Spring 2012

March 6, 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) What is a race condition? If a program has a data race will it always also have race condition? Why or why not.

   Answer:

   a) A race condition is a condition that occurs when a program's correctness unexpectedly depends on the ordering of events.
   b) 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) 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) 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:

   Some examples include:

   Benefits – multiple wait sets per Object, higher performance in some cases, additional locking semantics, such as ReaderWriterLocks

   Drawbacks – user has to manually ensure that Locks are released

2. Deadlock. (15 points).

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

   Answer:

   Mutual exclusion: a non-sharable resource exists
   Hold and wait: processes already holding resources may request new resources held by other processes.
   No preemption: No resource can be forcibly removed from a process holding it.
   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 Cup.pour() method is susceptible to deadlock. Fill in the Cup.main() method such that it can expose this potential deadlock. Based on the code you write show the smallest sequence of method calls you leads to the deadlock.

```
public class Cup {
  private Random r = new Random();
  private int MAX = 8, level = r.nextInt(MAX) + 1;

  public void pour(Cup otherCup) {
    int transfer = 0, space = 0;
    synchronized (this) {
      synchronized (otherCup) {
        // transfer some liquid from this to otherCup
        // without overfilling otherCup
      }
    }
  }
}
```

   over...

```
  public static void main(String[] args) {
    ExecutorService exec = Executors.newFixedThreadPool(2);
    // FILL IN CODE HERE

    Cup c1 = new Cup(), c2 = new Cup();


    exec.execute(new Runnable() {
      public void run() {
        // FILL IN CODE HERE

       c1.pour(c2);


      }
    });
    exec.execute(new Runnable() {
      public void run() {
        // FILL IN CODE HERE


       c2.pour(c1);


      }
    });
    exec.shutdown();
  }
}


// PUT EXECUTION TRACE HERE

  c1.pour(c2) // acquires lock on c1, then attempts to acquire lock on c2
  c2.pour(c1) // acquires lock on c2, then attempts 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. 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 (y) {x = 1;}
      x = y;
    }

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

Answer:

```
Initial trace:
Assume x=y=0 initially

lock (T1,y) < write (T1,x,1) < unlock (T1,y) < read (T1,y,0) < write (T1,x,0) <
lock (T2,x) < read (T2,x,0) < write (T2,y,0) < unlock (T2,x)

Happens-Before:

(E1) lock (T1,y) <: (E2) write (T1,x,1) <: (E3) unlock (T1,y) <:
  (E4) read (T1,y,0) <: (E5) write (T1,x,0)

(E6) lock (T2,x) <: (E7) read (T2,x,0) <: (E8) write (T2,y,0) <:
  (E9) unlock (T2,x)

Data races: Any one of:

1) E2 and E7
2) E4 and E8
3) E5 and E8
```

4. (Guarded Suspension (20 Points). Fill in the code below to create a thread-safe BoundedBuffer. 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, 1) your implementation cannot use Java intrinsic locks or intrinsic wait sets, 2) your implementation cannot wake up more than 1 waiting Thread at a time, and 3) your implementation should not deadlock if there are client Threads with requests that can be serviced (e.g., there is a client that wants to take() and BoundedBuffer is not empty).

   You will need to use the following methods in your solution: void Lock.lock(), void Lock.unlock(), Condition Lock.newCondition(), void Condition.await(), void Condition.signal().

```
class Buffer <T> {
  private Queue<T> contents;
  private int capacity;
  private Lock lock = new ReentrantLock();
  // FILL IN CODE HERE

  private Condition notFull = lock.newCondition();
  private Condition notEmpty = lock.newCondition();

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

  public T get() {
  // FILL IN CODE HERE

    lock.lock();
    try {
      while (contents.size() <= 0) {
        try {
          notEmpty.await();
        } catch (InterruptedException e) {
      }
    }
    notFull.signal();
    return contents.remove();
    } finally {
      lock.unlock();
   }
  }
over...
```

```
  public void put(T value) {
  // FILL IN CODE HERE

    lock.lock();
      try {
        while (contents.size() >= capacity) {
          try {
            notFull.await();
          } catch (InterruptedException e) {
        }
      }
      contents.add(value);
      notEmpty.signal();
      } finally {
        lock.unlock();
      }
  }
}
```

5. Futures (25 Points). The FutureBasedFactorizer class determines whether a given target number is or is not prime. To do this the main() method uses a FutureTask to acquire the set of all prime numbers up to some given stopping point. This work is done asynchronously outside the main Thread. The main thread must eventually acquire this set of prime numbers and then determine whether any of these prime numbers evenly divides the target. If yes, then the target is not prime; if no, then the target is prime. Fill in the code below to complete the FutureBasedFactorizer. **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 Primes {
  public static List<Integer> getPrimes() {
    return Arrays.asList(new Integer[] { 2, 3, 5, 7, 9, 11, 13, 17, 19, 23,
      27, 29, 31, 37, 41 });
    }
}

public class FutureBasedFactorizer {

  private static FutureTask<List<Integer>> mFutureTask =
      new FutureTask<List<Integer>>(new Callable<List<Integer>>() {
        public List<Integer> call() {
        // FILL IN CODE HERE
          return Primes.getPrimes();
        }
      });

  public static void main(String args[]) {
    boolean isPrime = true;
    try {

    // SET UP THE FUTURETASK

      new Thread(mFutureTask).start();

      int target = Integer.parseInt(args[0]);

    // USE THE RESULTS OF THE FUTURETASK


      for (int i : mFutureTask.get()) {
        if (bigNumber % i == 0) {
        isPrime = false;
        break;
        }
      }
    } catch (InterruptedException e) {
    e.printStackTrace();
    } catch (ExecutionException e) {
    e.printStackTrace();
    }
  System.out.println(isPrime);
  }
}
```