CMSC330 Fall 2009 Practice 7 Solutions (Java 1.4)

1. Multithreading, Data Races, and Deadlock
   a. For the following program, give two schedules under which the final value of i differs in the two schedules. Give the schedule as a list of line numbers, and in each case, also give the final value of i.

```
l = new Object();
m = new Object();
i = 0;

Thread 1
1. synchronized(l) {
2.   i = 3;
3. }

Thread 2
4. synchronized (m) {
5.   i = i + 1;
6. }
```

Since l and m are different locks, they do not provide any mutual exclusion in this example, and the threads may be interleaved arbitrarily.

Some example solutions:

1, 2, 3, 4, 5, 6 - i = 4
1, 4, 5, 2, 3, 6 - i = 3

b. For the following program, give one schedule under which there will be a deadlock, and give one schedule under which there will not be a deadlock. Give the schedule as a list of line numbers.

```
l = new Object();
m = new Object();
n = new Object();

Thread 1
1. synchronized (l) {
2.   synchronized (m) {
3.   }
4. }

Thread 2
5. synchronized (m) {
6.   synchronized (n) {
7.   }
8. }

Thread 3
9.   synchronized (n) {
10.   synchronized (l) {
11.   }
12. }
```

Some example solutions:

No deadlock - 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

Deadlock - 1, 5, 9

Because thread 1 holds l, thread 2 holds m, and thread 3 holds n, so none of the threads can make progress

c. Using language notation similar to above, write a program that has no data races, but nonetheless may produce different output under different schedules.

There are many possible answers, such as:

```
l = new Object();
i = 0;

Thread 1
synchronized (l) {
  i = 2;
}

Thread 2
synchronized (l) {
  i = 3;
}
```

Here the final value of i is either 2 or 3, but there is no data race.

d. List all possible outputs from the following program. Indicate next to each possible output whether all threads complete at the end, or, if they do not, which threads remain blocked.

```
l = new Object();

Thread 1
synchronized (l) {
  System.out.print("a ");
  l.wait();
  System.out.print("b ");
}

Thread 2
synchronized (l) {
  System.out.print("c ");
  l.notifyAll();
  System.out.print("d ");
}
```

Possible outputs:

a c d b - all threads complete
c d a - thread 1 remains blocked, waiting to be woken up

2. Multithreading Code

a. Using Java Conditions, you must implement a synchronization construct called MyBarrier. A MyBarrier object is created with a certain value n. When a thread calls the method enter( ), it enters the barrier and blocks until a total of n threads have entered the barrier. When the n^{th} threads enters the barrier, all the threads waiting at the barrier wake up and unblock, and the n^{th} thread continues without blocking. When a thread calls the method reset( ), the barrier is reset so that it starts fresh in counting up to n (i.e., n more threads must enter the MyBarrier). You may start by modifying the following code fragment:

```
public class MyBarrier {
        public void MyBarrier (int n) { … }
        public enter( ) { … }
        public reset( ) { … }
}
```

```
public class MyBarrier {
        int num;                // shared read-only data
        int current = 0;        // shared modifiable data
        Object lock = new Object();

        public MyBarrier (int n) {
           num = n;
        }

        public void enter( ) throws InterruptedException {
           synchronized (lock) {       // prevent data race on current
           current++;                  // incr # of threads at barrier

           if (current == num) {       // enough threads at barrier
                lock.notifyAll();      // wake up other threads
           }                           // continue execution
           else {
                while (current < num) { // wait for more threads to enter
                   lock.wait();        // sleep until enough threads enter
                }                      // use while ( ) in case reset( ) called
           }
           } // releases lock
        }

        public void reset( ) {
           synchronized (lock) {       // prevent data race on current
              current = 0;
           } // releases lock
        }
}
```

b. Implement MyBarrier using Ruby monitors.

```
require "monitor.rb"

class MyBarrier
  def initialize n
    @num = n
    @current = 0
    @myLock = Monitor.new
    @myCondition = @myLock.new_cond
  end

  def enter
    @myLock.synchronize {
      @current = @current + 1
      if @current == @num then
        @myCondition.broadcast
      else
        @myCondition.wait_while { @current < @num }
      end
    }
  end

  def reset
    @myLock.synchronize {
      @current = 0
    }
  end
end
```

c. Write a Ruby program that creates a barrier for 2 threads, then creates 2 threads that each print out “hello”, enters the barrier, then prints out “goodbye”.

```
bar = MyBarrier.new 2

t1 = Thread.new {
  puts "hello"
  bar.enter
  puts "goodbye"
}

t2 = Thread.new {
  puts "hello"
  bar.enter
  puts "goodbye"
}
```

3. Garbage collection

Consider the following Java code.

```
Object a, b, c;
public foo( ) {
        a = new Object( );      // object 1
        b = new Object( );      // object 2
        c = new Object( );      // object 3
        a = b;
        b = c;
        c = a;
}
```

a. What object(s) are garbage when foo( ) returns? Explain why.

Object 1 is garbage there are no longer any references to it within the program. After foo( ) returns, a → object 2, b → object 3, c → object 2.

b. Describe the difference between mark-and-sweep & stop-and-copy.

Mark-and-sweep stops the program to determine what objects are still reachable. Stop-and-copy in addition will move reachable objects to new locations.