11. (14 pts) Multithreading

Consider the following attempt to implement producer/consumer pattern w/ Java 1.4.

```
class Buffer {
    Buffer ( ) {
      Object buf = null;
      boolean empty = true;
    }
    void produce(o) {
      synchronize (buf) {
1.      if (!empty) wait( );
2.      empty = false;
3.      notifyAll( );
4.      buf = o;
       }
    }
```

```
    Object consume( ) {
      synchronize (buf) {
5.      if (empty) wait( );
6.      empty = true;
7.      notifyAll( );
8.      return buf;       // also releases lock
       }
    }
}
t1 = Thread.run { produce(1); }
t2 = Thread.run { produce(2); }
t3 = Thread.run { x = consume( ); }
t4 = Thread.run { y = consume( ); }
```

In the following, give schedules as a list of thread name/line number/range pairs, e.g., (t1, 1-4), (t2, 1), (t3, 5-8). For instance, one schedule under which x=1 and y=2 is (t1, 1-4), (t3, 5-8), (t2, 1-4), (t4, 5-8)

a. (2 pts) Give a schedule under which x = 2 and y = 1.

(t2, 1-4), (t3,5-8), (t1,1-4), (t4,5-8) etc…
Have t3 execute after t2, and t4 execute after t1

b. (4 pts) Give a schedule under which x = 2 and y = 2, or argue that no such schedule is possible.

(t4,5), (t2, 1-4), (t3,5-8), (t4,6-8) OR (t3,5), (t2, 1-4), (t4,5-8), (t3,6-8) etc…

Have one consumer thread (either t3 or t4) misbehave by waiting on 5, then returning and continuing execution even though condition is not valid (i.e., empty = true), causing it to read value 2 already read by other consumer

c. (8 pts) Explain why the given Java code allows data races and why deadlock may occur.

Because wait is not called in a while loop. Multiple threads may be woken, so the condition may not be true by the time a thread wakes up.

Deadlock:
(t1,1-4), (t2, 1), (t6, 1), (t3,5-8), (t2,2-4), (t6,2-4), (t5,5-8), (t4,5) etc…
Deadlock can occur by having producer thread t2 & t6 misbehave by waiting on 1, then returning and continuing execution even though condition is not valid (i.e., empty = false), for both, causing one producer to overwrite buf value already produced by other producer thread. One consumer will then hang because there are insufficient producers.

12. (22 pts) Ruby multithreading

Using Ruby monitors and condition variables, write a Ruby function simulate(M,N) that implements the following simulation of a dance club. M girls and N guys arrive at a club. Each guy is assigned a number between 0 and N-1, and each girl is assigned a number between 0 and M-1. Once at the club, each girl dances 10 times, each time picking any guy who is not currently dancing with another girl. Each dance lasts 0.01 seconds in real time (i.e., call sleep 0.01). Print out a message "X dancing with Y" for girl X and guy Y at the start of each dance. The action for each girl must be executed in a separate thread. You must allow multiple couples to dance at the same time (i.e., while calling sleep 0.01). Once all girls have finished dancing, the simulation is complete.

You must use monitors to ensure there are no data races, and condition variables to ensure girls efficiently wait if all guys are currently dancing. You may only use the following library functions.

Allowed functions:

```
n.times { |i| … }        // executes code block n times, with i = 0…n-1
a = [ ]                  // returns new empty array
a.empty?                 // returns true if array a is empty
a.push(x)                // pushes (adds) x to array a
x = a.pop                // pops (removes) element of a and assigns it to x
a.each { |x| … }         // calls code block once for each element x in a
m = Monitor.new          // returns new monitor
m.synchronize { … }      // only 1 thread can execute code block at a time
c = m.new_cond           // returns conditional variable for monitor
c.wait_while { … }       // sleeps while code in condition block is true
c.wait_until { … }       // sleeps until code in condition block is true
c.broadcast              // wakes up all threads sleeping on condition var c
t = Thread.new {… }      // creates thread, executes code block in new thread
t.join                   // waits until thread t exits
```

```
require "monitor"
Thread.abort_on_exception = true   # to avoid hiding errors in threads
class DanceHall
  def initialize
    @m = Monitor.new
    @c = @m.new_cond
    @num = 1;
    @guys = []
  end
  def goGirl
    me = 0
    g = 0
    @m.synchronize {
       me = @num
       @num = @num+1
    }
    10.times {
      @m.synchronize {
        @c.wait_while { @guys.empty? }
        g = @guys.pop
        puts "#{me} dancing with #{g}"
        $stdout.flush
      }
      sleep 0.01
      @m.synchronize {
        @guys.push(g)
        @c.broadcast
      }
    }
  end
  def simulate(numGirls,numGuys)
    numGuys.times { |i|
      @guys.push(i)
    }
    threads = []
    numGirls.times { |i|
      t = Thread.new { goGirl }
      threads.push(t)
      }
    threads.each { |t| t.join }
    puts "All done!"
  end
end

d = DanceHall.new
d.simulate(10,3)
```