Homework 12

Due Tuesday, 12/9/2014

Please turn in your homework solutions online at `http://www.dci.pomona.edu/tools-bin/cs131upload.php` before the beginning of class on the due date.

1. (10 points) **Atomicity & Race Conditions**

   The `DoubleCounter` class defined below has methods `incrementBoth` and `getDifference`. Assume that `DoubleCounter` will be used in multi-threaded applications.

```
class DoubleCounter {
   protected int x = 0, y = 0;

   public int getDifference() {
      return x - y;
   }

   public void incrementBoth() {
      x++;
      y++;
   }
}
```

   There is a potential data race between `incrementBoth` and `getDifference` if `getDifference` is called between the increment of x and the increment of y. For the following questions, assume that the "++" operators are atomic (in reality they are not) and that each thread calls incrementBoth exactly once.

   (a) What are the possible return values of `getDifference` if there are 2 threads?

   (b) What are the possible return values of `getDifference` if there are n threads?

   (c) Data races can be prevented by inserting synchronization primitives. One option is to declare

```
public synchronized int getDifference() {...}
public void incrementBoth() {...}
```

   This will prevent two threads from executing method `getDifference` at the same time. Is this enough to ensure that `getDifference` always returns 0? Explain briefly.

   (d) Is the following declaration

```
public int getDifference() {...}
public synchronized int incrementBoth() {...}
```

   sufficient to ensure that `getDifference` always returns 0? Explain briefly.

(e) What are the possible values of `getDifference` if the following declarations are used?

```
public synchronized int getDifference() {...}
public synchronized int incrementBoth() {...}
```

2. (10 points) **Concurrent Access to Objects**

   Please do problem 14.6 from Mitchell, page 471.

3. (10 points) **Java synchronized objects**

   Please do problem 14.7, parts a,b,and c from Mitchell, page 472.

4. (20 points) **Using Bounded Buffer**

   The Haskell function, `primesto n`, given below, can be used to calculate the list of all primes less than or equal to n by using a variant of the classic "Sieve of Eratosthenes".

```
{- Sieve of Eratosthenes: Remove all multiples of first element from list,
   then repeat sieving with the tail of the list.  If start with list [2..n]
   then will find all primes from 2 up to and including n. -}
sieve :: (Integral a) => [a] -> [a]
sieve [] = []
sieve (fst:rest) = let
        nurest = filter (\n -> (n `mod` fst) /= 0) rest
     in
        fst:(sieve nurest)

-- return list of primes from 2 to n
primesto :: (Integral t) => t -> [t]
primesto n = sieve [2..n];
```

   Notice that each time through the sieve we first filter all of the multiples of the first element from the tail of the list, and then perform the sieve on the reduced tail. In an eager language, one must wait until the entire tail has been filtered before you can start the sieve on the resulting list. In Haskell, it will perform the calculation on demand and only calculate as much as is needed. For example if we displayed only the first element of the result of primesto 100, then only that element would be calculated. If we have spare processors, one could use parallelism to have one process start sieving the result before the entire tail had been completely filtered by the original process.

   Here is a good way to think of this concurrent algorithm that will use the Java `BoundedBuffer` class defined in the previous problem. The main program should begin by creating a `BoundedBuffer` object (say with 5 slots) and then should successively insert the numbers from 2 to n (for some fixed n) into the `BoundedBuffer` using the `put` method, and finally put in -1 to signal that it is the last element. After the creation of the `BoundedBuffer` object, but before starting to put the numbers into the `BoundedBuffer`, the program should create a `Sieve` object (using the `Sieve` class described below) and pass it the `BoundedBuffer` object (as a parameter to `Sieve`'s constructor).

The `Sieve` object should then begin running in a separate thread while the main program inserts the numbers in the buffer.

After the `Sieve` object has been constructed and the `BoundedBuffer` object stored in an instance variable, `in`, its `run` method should get the first item from `in` using the `get` method. If that number is negative then the `run` method should terminate. Otherwise it should print out the number (System.out.println is fine) and then create a new `BoundedBuffer` object, `out`. A new Sieve should be created with `BoundedBuffer` `out` and started running in a new thread. Meanwhile the current Sieve object should start filtering the elements from the `in` buffer. That is, the `run` method should successively grab numbers from the `in` buffer, checking to see if each is divisible by the first number that was obtained from `in`. If a number is divisible, then it is discarded, otherwise it is put on buffer `out`. This reading and filtering continues until a negative number is read. When the negative number is read then it is put into the `out` buffer and then the `run` method terminates.

If all of this works successfully then the program will eventually have created a total of p + 1 objects of class `Sieve` (all running in separate threads), where p is the number of primes between 2 and n. The instances of `Sieve` will be working in a pipeline, using the buffers to pass numbers from one `Sieve` object to the next.

Please write this program in Java using the `BoundedBuffer` class from the previous problem. Each of the buffers used should be able to hold at most 5 items.

5. (15 points) **Sieve with Actors**

   Rewrite the Sieve of Eratosthenes program from above in Scala using Actors (and no `BoundedBuffer`). In the Java program you wrote, you created a new Thread for every prime number you discovered. This time, you will create a new `Sieve` actor for each prime `Int`.

   Hints: Your `Sieve` actor should keep track of the prime it was created with, and it should have a slot that can hold another "follower" `Sieve` actor that will handle the next prime found. (The mailbox of this other Actor will play the role of the `BoundedBuffer` from the previous problem in that it will hold the numbers being passed along from one actor to the next.) Each `Sieve` actor should be able to handle messages of the form `Number(n)` and `Stop`.

   You will get a warning that `Actor` is deprecated in Scala. While you are welcome to try to figure out Akka Actors, I recommend that you stay with plain `Actor` and ignore the warnings.

   The operation of each `Sieve` actor, once started, is similar to the previous problem. If it receives a message of the form `Number(n)` then it checks if it is divisible by its prime. If so, it is discarded. If not, then if the follower Actor has not yet been created, create it with the number. If not, then send the number to the follower `Sieve` actor. When the Stop `message` is sent, pass it on to the follower (if any), and exit.

   This program works best with a `receive` and a `while` loop (like in the Auction example) rather than `react` and `loop` (as in the `PingPong` and `BoundedBuffer` actor examples from class).

   Your program should print (in order) all of the primes less than 100. You can print each prime as it is discovered (e.g., when you create the corresponding `Sieve` actor), but it would be even better to return a list of all of the primes, and then print those out. To do this, you can send the message Stop synchronously with "!?" and get back the list of all primes. The "!?" operator returns an object of type `Any` (equivalent to Java's `Object`), so you'll have to use `match` to decode it as a list of `Int`.