What To Turn In

Please hand in work in two pieces, one for the Problems and one for the Programming (Partner Optional):

Problems: Turn in handwritten or typed answers by the due date. Be sure your work is stapled and that your answers are clearly marked and in the correct order.

Programming (Partner Optional): This part involves writing Scala code. You are welcome to work with a partner but are not required to do so. I can also assist in matching pairs — simply send me email and I will pair you with someone else also looking for a partner. As usual please submit your code separate from the written answers and via turnin.

Reading

1. Read Mitchell, Chapter 14.1 – 14.2, 14.4 (up through page 461)
2. Scala Actors Tutorials, as needed. (A good starting point is http://www.scala-lang.org/node/242.)

Problems

Q1. (15 points) ......................... Race Conditions and Atomicity

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

```java
class DoubleCounter {
    protected int x = 0;
    protected int 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. You may assume that x++ and y++ execute atomically (although this is not always guaranteed...).

(a) What are the possible return values of getDifference if there are two threads that each invoke incrementBoth exactly once at the same times as a third thread invokes getDifference?

(b) What are the possible return values of getDifference if there are n threads that each invoke incrementBoth exactly once?

(c) Data races can be prevented by inserting synchronization primitives. One option is to declare
public synchronized int getDifference() {...}
public int 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.

(c) 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() {...}

(f) Atomicity is another concurrency control concept that is used in some newer language designs. If a block is declared atomic, then the implementation guarantees that the program output is equivalent to either completing the atomic block without interference from any other threads, or not running the block at all. Atomic blocks can be implemented by a variety of methods, including locking and/or mechanisms that allow a thread to rollback to a previous state if needed. However, one goal of atomicity as a language construct is to separate reasoning about correctness from details of how atomicity is achieved in a particular instance.

Using atomic blocks, we could declare incrementBoth atomic as follows:

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

If a thread calling incrementBoth is interrupted in the middle by a change to the value of x or y, the run-time system can rollback to the state before entering the atomic block. Then when the thread runs again it will continue from the beginning of the atomic block. Will declaring incrementBoth atomic ensure that getDifference always returns 0? Explain.

---

**Programming (Partner Optional)**

**P1. (30 points) ...................................................... Sieve of Eratosthenes**

The ML function, primesto n, given below, can be used to calculate the list of all primes less than or equal to n by using the classic “Sieve of Eratosthenes”.

```ml
(*
 * 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.
 *)
fun sieve [] = []
| sieve (fst::rest) =
    let fun filter p [] = []
        | filter p (h::tail) = if (h mod p) = 0 then filter p tail
                                    else h::(filter p tail);
    val nurest = filter fst rest
    in
```
fst::(sieve nurest)
end;

(*
 * Returns list of integers from i to j
 *)
fun fromto i j = if j < i then [] else i::(fromto (i+1) j);

(*
 * Return list of primes from 2 to n
 *)
fun primesto n = sieve(fromto 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 ML, one must wait until
the entire tail has been filtered before you can start the sieve on the resulting list. However, 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, which uses the Java Buffer class.

The main program should begin by creating a Buffer object, with perhaps 5 slots. It should then
successively insert the numbers from 2 to n into the Buffer, followed by -1 to signal that there
are no more input numbers.

After the creation of the Buffer object, but before starting to put the numbers into it, the program
should create a Sieve object (using the Sieve class described below) and pass it the Buffer 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 Buffer object has been stored in an instance
variable, in, its run method should get the first item from in:

• If that number is negative then the run method should terminate.
• Otherwise, it should print out the number (using System.out.println) and then create a
  new Buffer object, out. A new Sieve should be created with Buffer 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.
  If the number is divisible by the first number that was obtained from in, it is discarded.
  Otherwise, it is added to the out buffer. This reading and filtering continues until a negative
  number is read. When the negative number is read, that number is put into the out buffer
  and then the run method terminates.

In this way, the program will eventually have created a total of \( p + 1 \) Sieve objects (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.

Write this program in Java using the Buffer class from lecture. Each of the buffers used should
be able to hold at most 5 items.

P2. (30 points) Sieve of Actors

Note: As I mentioned in lecture on Thursday, the version of scala installed on the lab machines
does not support the actor library we'll be using. To use the version with actors, please run the
following command in the terminal window before using scalac or scala:

export PATH=~/freund/shared/scala-2.10.7/bin/:$PATH
You’ll need to do this each time you open a new terminal window. If you are using your own computer, you can download the correct version here: http://www.scala-lang.org/download/2.10.7.html. You can verify you are the correct version if running “scala -version” prints out:

Scala code runner version 2.10.7 -- Copyright 2002-2017, LAMP/EPFL

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.

Write the code in the Sieve.scala file in the starter project.

**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 Num(n) and Stop.

The operation of each Sieve actor, once started, is similar to the previous problem. If it receives a message of the form Num(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 PickANumber example) rather than react and loop (as in the Parrot and Account 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 when one of your actors receives this message it replies with the list of all primes it knows about. 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 — this may result in an “unchecked” warning from the compiler for a fairly subtle Java compatibility reason, but you can ignore that.)

I have provided a sample program ProducerConsumer.scala in the starter code. You don’t need to use this directly, but you may wish to refer to this code as a simple example of using Actors.

What To Turn In for the Programs.

- Your code for the programming questions should be documented, and include your ID (and that of your partner) at the top of each file.
- Turn in a printout of your code file separate from the answers to the written problems. If you worked with a partner, please turn in only one copy.
- Also, commit and push all of your code to our Gitlab server.