Homework 10

Due 10 May (THURSDAY)

Handout 23
CSCI 334: Spring, 2012

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

1. (*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.

```
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?

(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.

(d) Is the following declaration

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

sufcient 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:

```
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.

2. (*10 points*) .... Bounded Buffers

This question asks about the Java implementation of a bounded buffer given in class. Its code is available on the handouts page for your reference.

(a) What is the purpose of `while (elementCount == elementData.length) wait()` in `insert`?

(b) What does `notifyAll()` do in this code?

(c) Describe one way that the buffer would fail to work properly if all synchronization code is removed from `insert`.

(d) Suppose a programmer wants to alter this implementation so that one thread can call `insert` at the same time as another calls `delete`. This causes a problem in some situations but not in others. Assume that some locking may be done at entry to `insert` and `delete` to make sure the concurrent-execution test is satisfied. You may also assume that increment or decrement of an integer variable is atomic and that only one call to `insert` and one call to `delete` may be executed at any given time. What test involving `start` and `end` can be used to decide whether `insert` and `delete` can proceed concurrently?

(e) The changes in the previous part will improve performance of the buffer. Describe one reason that leads to this performance advantage. Despite this win, some programmers may choose to use the original method anyway. Desribe one reason why they might make this choice.

3. (*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".

```
(*
 * 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 fitered 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.

Include a printout of your `Sieve` class with your homework, and use `turnin` to submit it electronically as well. You need not print out or submit the other Java files.

4. (*30 points*) .... Sieve of 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`.

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` — 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` on the course handouts page. You may wish to refer to this code as a simple example of using Actors.

Turn in your code with `turnin`.

5. (*10 points*) .... Java Bytecode Analysis

Java programs are compiled into bytecode, a simple machine language that runs on the Java Virtual Machine. Note that it is possible that bytecode could be written by hand, or that compiled bytecode could get corrupted when transmitted over the network. So, when a class is loaded by the JVM, it is first examined by the bytecode verifier. The verifier performs static analysis of the class to ensure that a program will not cause an unchecked runtime type error (which could then lead to a security violation). One kind of bytecode error that the verifier should catch is use of an uninitialized variable. Here is a code fragment in Java and some corresponding bytecode. We have added comments to the bytecode to help you figure out the effects of the instructions.

```
// Java Source Code
Point p = new Point(3);
p.print();

// Compiled Bytecode
1: new #1 <Class Point>                // Allocate space for Point
2: dup                                 // Duplicate top entry on stack
3: iconst 3                            // Push integer 3 onto stack
4: invokespecial #4 <Method Point(int)>   // Invoke constructor, which pops
                                       //  its argument (3) and one of
                                       //  the pointers to p off stack
5: invokevirtual #5 <Method void print()> // Invoke print method, popping
                                       //  the other pointer to p.
```

The first line of the Java source allocates space for a new `Point` object and calls the `Point` constructor to initialize this object. The second line invokes a method on this object and therefore can be allowed only if the object has been initialized. It is easy to verify from this Java source that `p` is initialized before it is used.

Checking that objects are initialized before use in the bytecode is more difficult. The Java implementation creates objects in several steps: First, space is allocated for the object. Second, arguments to the constructor are evaluated and pushed onto the stack. Finally, the constructor is invoked. In the bytecode for this example, the memory for p is allocated in line 1, but the constructor isn't invoked until line 4. If several objects are passed as arguments to the constructor, there could be an even longer code fragment between allocation and initialization of an object, possibly allocating multiple new objects, duplicating pointers, and taking conditional branches.

To account for pointer duplication, some form of aliasing analysis is needed in the bytecode verifier. This problem will consider a simplified form of initialize-before-use bytecode verification that keeps track of pointer aliasing by keeping track of the line number at which an object was first created. When a pointer to an uninitialized object is copied, the alias analysis algorithm copies the line number where the object was created, so that both pointers can be recognized as aliases for a single object. Of course, if an instruction creating a new object is inside a loop, then there may be many different uninitialized objects created at the same line number. However, the bytecode verifier does not need to work for this case, since Java compilers do not generate code like this. We won't consider any cases with conditional branches in this problem.

Let's examine the contents of stack and any associated line numbers for alias detection after execution of each of the bytecode instructions from above. Note that we draw the stack growing downward in this diagram:

After line:	Stack Contents	line # where created	initialized
1	`Point p`	1	no
2	`Point p` alias for `p`	1 1	no no
3	`Point p` alias for `p` int 3	1 1 3	no no yes
4	`Point p`	1	yes
5	(empty)		

By using line numbers as object identifiers associated with each pointer on the stack, we keep track of the fact that both pointers on the stack point to the same place after line 2. So when the constructor invoked on line 4 initializes the alias for `p`, we recognize that `p` is initialized as well. Thus the `print()` method is applied to a properly initialized `Point` object, and this example code fragment passes our simple initialize-before-use verifier.

(a) Consider the following bytecode:

```
1: new #1 <Class Point>
2: new #1 <Class Point>
3: dup
4: iconst 3
5: invokespecial #4 <Method Point(int)>
6: invokevirtual #5 <Method void print()>
```

When line 5 is reached, there will be more than one uninitialized `Point` object on the stack. Use the static verification method described above to figure out which one gets initialized, and whether the subsequent invocation of the `print()` method occurs on an initialized `Point`. You should draw the state of the stack after each instruction, using the original line number associated with any pointer to detect aliases.

(b) So far we have only considered bytecode operations that use the operand stack. For this part, we introduce two more bytecode instructions, `load` and `store`, which allow values to be moved between the operand stack and local variables.

- `store x` removes a variable from the top of the stack and stores it into local variable `x`.
- `load x` loads the value from local variable `x` and places it on the top of the stack.

We have begun applying the initialize-before-use verification procedure described above to the following code fragment in the table that follows. Note that we are maintaining information about aliasing and initialization state for local variable 0 as well as the contents of the operand stack.

```
1: new #1 <Class Point>
2: new #1 <Class Point>
3: store 0
4: load 0
5: load 0
6: iconst 5
7: invokespecial #4 <Method Point(int)>
8: invokevirtual #5 <Method void print()>
9: invokevirtual #5 <Method void print()>
```

After Line:	local variable 0			stack		
	contents	line #	init?	contents	line #	init?
1:	n/a	n/a	no	`Point`	1	no
2:	n/a	n/a	no	`Point` `Point`	1 2	no no
3:	`Point`	2	no	`Point`	1	no
4:						
5:						
6:						
7:						
8:						
9:						

Continue applying the procedure to fill in the missing information on lines 4-9.

Based on the state information you have after instruction 7, is the `print()` method on line 8 applied to a properly initialized object? What about the `print()` method on line 9?

6. (*30 points*) .... (Bonus Question) Software Transactions

Read the paper "Language Support for Lightweight Transactions" from the course readings page. It describes how to support transactions at the software level in Java. Briefly describe the implementation. What conclusions can be drawn from their experiments? What open issues are remain? There has been much work on this topic in the last ten years.

If you're curious, you can find many other papers on the subject on the web. A good place to start is the paper list for "TRANSACT 2006", a workshop on the subject.