Semaphores

In class we gave the following definitions using semaphores:

- **Wait(S):**
  - if \( S > 0 \) then \( S := S - 1 \) else suspend in queue assoc. w/ \( S \)
- **Signal(S):**
  - if processes waiting then wake them up, else \( S := S + 1 \);

Suppose we instead used the following definitions:

- **Wait(S):**
  - \( S := S - 1 \); if \( S < 0 \) then suspend in queue assoc. w/ \( S \)
- **Signal(S):**
  - \( S := S + 1 \); if \( S \geq 0 \) then wake up a waiting process

Is there any difference in behavior in programs using Wait and Delay if the first definitions are replaced by the second?

Atomicity & Race Conditions

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, y = 0;

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

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

There is a potential data race between **incrementBoth** and **getDifference** if **getDifference** is called between the increment of \( x \) and the increment of \( y \).

(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 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() {...}

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() {...}

3. (10 points) ................................................................. Actor computing
Mitchell, Problem 14.3

4. (10 points) ................................................................. Concurrent Access to Objects
Mitchell, Problem 14.6

5. (10 points) ................................................................. Java synchronized objects
This question asks about the Java implementation of a bounded buffer given in class.

(a) What is the purpose of while (BufferSize == MaxBuffSize) 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 situation 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 BufferStart and BufferEnd can be used to decide whether insert and delete can proceed concurrently?

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

6. (20 points) ................................................................. Using Bounded Buffer
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 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 that will use a slightly modified version
of the Java Buffer class defined in the lecture notes. The only modifications necessary are to
have the store instance variable hold an array of int, rather than char, and then change the
appropriate types of parameters, return types, and local variables in the insert and delete
methods from char to int.

The main program should begin by creating a Buffer object (say with 5 slots) and then should
successively insert the numbers from 2 to n (for some fixed n) into the Buffer using the insert
method, and finally put in -1 to signal that it is the last element. After the creation of the
Buffer object, but before starting to put the numbers into the Buffer, 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 stored in an instance variable,
in, its run method should get the first item from in using the delete 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 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, 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 Buffer class in Lecture 37, modified as suggested
above so that it holds ints rather than chars. Each of the buffers used should be able to hold at
most 5 items.