Concurrent Programming: Exercise Sheet 3

Bernard Sufrin*

January 12, 2012

You should complete all questions marked [**Programming**] at a computer, and include suitable testing code and test results with your answers.

Questions marked † will not normally be discussed in the class. Model answers are included below. You should attempt these questions, mark them yourself, and hand them in.

Question 1

Provide an implementation of barrier synchronisation that operates in $\Theta(\log N)$ time, for N processes. If you like, each process may pass its own identity to the sync function. Hint: base your implementation on one of the patterns from the "Interacting Peers" chapter.

Question 2

Recall the prefix sum example from the "Synchronous Data Parallel Programming" chapter. Explain why the barrier synchronisation is necessary.

Question 3

[**Programming**] Re-implement the prefix sum example, so as to use shared variables, rather than message-passing. You need to think carefully how to avoid race conditions.

Question 4 †

(a) Suppose N^2 worker processes are arranged in a N by N toroidal grid. Each process starts with an integer value v. The aim is for each process to end up in possession of the maximum of these N^2 values. Each process may send messages to the processes above it and to the right of it in the grid (where we treat the bottom row as being above the top row, and the left hand column as being to the right of the right hand column).

Write a definition of a process:

```
def Worker(x: Int, y: Int, readUp: ?[Int], writeUp: ![Int],
           readRight: ?[Int], writeRight: ![Int], v: Int) = proc{ ... }
```

*Gavin Lowe did **all** the work of preparing this sheet.

to solve this problem. You do not need to show how the overall system is constructed, but you should state any assumptions you make about the channels.

(b) Now consider the same scenario as in part (a), except assume each process can send messages to any other processes. We now want a solution that takes $O(\log N)$ rounds.

Give a definition for a process

```
def Worker(x: Int, y: Int, read: ?[Int], write: Seq[Seq[![Int]]], v:Int)
= proc{ ... }
```

to solve this problem. The process can send values to another process (x1, y1) on the channel write(x1)(y1), and can receive messages from other processes on the channel read. You should briefly explain your solution.

What is the total running time of the program (in $\Theta(\_)$ notation)?

Question 5

Extend the "slot" example from the lectures on monitors so as to hold a queue of data. The length of the queue should be bounded by some value N.

Question 6

[**Programming**] Consider the following producer-consumer problem. Several producers each produce one piece of (integer) data at a time, which they place into a buffer using a method Put(item:Int). These are accumulated into an array of size N. A consumer can receive that array, once it is full, using the method Get : Array[Int]. Implement the buffer as a monitor. The consumer should be blocked until the array is full, and the producers should be blocked once the array is full.

Question 7

[**Programming**] Consider the following synchronisation problem. Each process has an integer-valued identity. A particular resource should be accessed according to the following constraint: a new process can start using the resource only if the sum of the identities of those processes currently using it is divisible by 3. Implement a monitor with procedures Enter(id:Int) and Exit(id:Int) to enforce this.

Question 8

Consider the following synchronisation problem. There are two types of process, which we shall call Men and Women. These processes need to pair off for some purpose, with each pair containing one process of each type. A monitor is required, with a procedure for each type of process. The procedure should block processes until a process of the other type is ready, at which point both should continue.

(a) The following monitor was suggested. Explain why this does not meet the requirement.

```
object Monitor1{
  var menWaiting = 0; var womenWaiting = 0;
    // number of men, women currently waiting
  var manCommitted, womanCommitted = false;
    // Have the next man, woman to return been decided
    // but not yet paired?

  def ManEnter = synchronized{
    if(womenWaiting==0 && !womanCommitted){
      menWaiting += 1;
      while(womenWaiting==0 && !womanCommitted)
        wait(); // wait for a woman
      menWaiting -= 1; womanCommitted = false;
    }
    else{
      manCommitted = true; notify();  // wake up a man
    }
  }

  def WomanEnter = synchronized{
     if(menWaiting==0 && !manCommitted){
        womenWaiting += 1;
        while(menWaiting==0 && !manCommitted)
          wait(); // wait for a man
        womenWaiting -= 1; manCommitted = false;
     }
     else{
       womanCommitted = true; notify(); // wake up a woman
     }
  }
}
```

(b) Now the following monitor is suggested. Explain why this doesn't meet the requirement, either.

```
object Monitor2{
  var menWaiting = 0; var womenWaiting = 0;
    // number of men, women currently waiting
  var manCommitted, womanCommitted = false;
    // Have the next man, woman to return been decided
    // but not yet paired?

  def ManEnter = synchronized{
    if(womenWaiting==0 && !womanCommitted || manCommitted){
      menWaiting += 1;
      while(womenWaiting==0 && !womanCommitted
            || manCommitted)
        wait(); // wait for a woman
      menWaiting -= 1; womanCommitted = false;
    }
    else{
```

```
      manCommitted = true; notifyAll();  // wake up a man
    }
  }

  def WomanEnter = synchronized{
    if(menWaiting==0 && !manCommitted || womanCommitted){
      womenWaiting += 1;
      while(menWaiting==0 && !manCommitted
            || womanCommitted)
        wait(); // wait for a man
      womenWaiting -= 1; manCommitted = false;
    }
    else{
      womanCommitted = true; notifyAll(); // wake up a woman
    }
  }
}
```

(c) Suggest a definition for the module that does meet the requirement.

Answer to question 4 †

(a) We'll treat the channels as being buffered (alternatively, arrange for the reads and writes, below, to be concurrent).

```
def Worker(x: Int, y: Int,
           readUp: ?[Int], writeUp: ![Int],
           readRight: ?[Int], writeRight: ![Int], v:Int)
= proc{
  println((x,y)+" chooses "+v);
  var max = v;

  // propogate values along rows
  for(i <- 0 until N-1){
    writeRight!max; max = Math.max(max, readRight?)
  }

  // propogate values upwards
  for(i <- 0 until N-1){
    writeUp!max; max = Math.max(max, readUp?)
  }

  // Print result
  println((x,y)+" ends with "+max);
}
```

(b) The idea is that on round r, each process holds the maximum of the initial values from a 2^r by 2^r square (wrapping round in the obvious way), with itself at the bottom-left corner.

```
val barrier = new CombiningTreeBarrier(N*N);
```

```
def Worker(x: Int, y: Int, read: ?[Int],
           write: Seq[Seq[![Int]]], v:Int)
= proc("Worker "+(x,y)){
  println((x,y)+" chooses "+v);
  var max = v;
  var r = 0; // Round number
  var gap = 1; // 2^r
  // Invariant: max is the maximum of the values initially
  // chosen by nodes ((x+dx)%N, (y+dy)%N) for
  // 0 <= dx, dy < gap = 2^r

  while(gap < N){
    // Send max to processes gap positions left or down
    // from here
    val x1 = (x+N-gap)%N; val y1 = (y+N-gap)%N;
    write(x)(y1)!max; write(x1)(y1)!max; write(x1)(y)!max;
    // Receive from three processes gap right or above here
    for(i <- 0 until 3) max = Math.max(max, read?);
    // Update r, max
    r += 1; gap += gap;
    barrier.sync(N*x+y); // Sync before next round,
    // passing in unique "id"
  }

  // Print result
  println((x,y)+" ends with "+max);
}
```

The channels should be buffered with capacity (at least) 3.

There are $\Theta(\log N)$ rounds. On each round, each step takes constant time except the barrier synchronisation, which takes $\Theta(\log(N^2)) = \Theta(\log N)$. So the total time is $\Theta((\log N)^2)$.