Concurrent Programming: Exercise Sheet 4

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.

Question 1 †

[**Programming**] Consider a binary tree containing integers, defined as follows:

```
abstract class IntTree
case class Leaf(n:Int) extends IntTree
case class Branch(l:IntTree, r:IntTree) extends IntTree
```

Write a concurrent program to calculate the sum of the values on the leaves of the tree, using the recursive parallel pattern.

Question 2

[**Programming**] Recall the Fibonacci numbers, which can be defined by

```
def fib(n:Int) : Int = if(n<=1) n else fib(n-1)+fib(n-2)
```

Implement this using recursive parallelism. Consider versions with no limit on the degree of recursion, and with some set limit. Carry out some informal timing tests on the two versions and on the sequential version above.

Question 3

[**Programming**] Recall the trapezium rule, used for estimating integrals, illustrated in the third chapter. *Adaptive quadrature* is an alternative approach that proceeds as follows. In order to calculate the integral of function f from a to b, first compute the midpoint $mid = (a + b)/2$. Then estimate three integrals, from a to mid, from mid to b, and from a to b, each using the trapezium rule with a single interval. If the sum of the former two estimates is within some value ϵ of the third, then we take that third estimate as being the result. Otherwise, recursively estimate the integrals over the two ranges a to mid and mid to b, and sum the results. The following sequential algorithm captures this idea:

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

```
def Estimate(a:Double, b:Double) : Double = {
  val mid = (a+b)/2.0;
  val fa = f(a); val fb = f(b); val fmid = f(mid)
  val larea = (fa+fmid)*(mid-a)/2;
  val rarea = (fmid+fb)*(b-mid)/2;
  val area = (fa+fb)*(b-a)/2;
  if (Math.abs(larea+rarea-area) < EPSILON) return(area);
  else{ return(Estimate(a,mid) + Estimate(mid,b)); }
}
```

Write a concurrent program to implement adaptive quadrature. The program should use a bag of tasks with replacement: the two recursive calls in the sequential version can be implemented by returning tasks to the bag.

Suggest ways in which your program can be made more efficient; optional: implement them.

Question 4

[**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 using semaphores. The consumer should be blocked until the array is full, and the producers should be blocked once the array is full.

Question 5 †

[**Programming**] Implement a bounded buffer, of size n, using two semaphores: one semaphore should prevent data from being added to the buffer when it is full; the other should prevent data from being removed from the buffer when it is empty.

Question 6

[**Programming**] Recall the following synchronisation problem from the previous sheet. 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. Use semaphores to implement a controller for this problem. The controller should have 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.

Question 7

[**Programming**] Implement a solution to the readers and writers problem using semaphores and the technique of passing the baton. Think carefully about whom the baton should be passed to in different scenarios in each of the following scenarios: when a reader enters; when a writer enters; when a reader leaves; when a writer leaves.

Answer to question 1 †

```
import ox.CSO._

abstract class IntTree
case class Leaf(n:Int) extends IntTree
case class Branch(l:IntTree, r:IntTree) extends IntTree

object TreeSum{
  // Add values on the tree, and send result on result
  def Adder(t:IntTree, result: ![Int]) : PROC = proc{
    t match {
      case Leaf(n) => result!n
      case Branch(l, r) => {
        // Fork off parallel processes to sum each subtree
        val res1 = OneOne[Int]; val res2 = OneOne[Int];
        ( Adder(l, res1) || Adder(r, res2) || proc{ result!(res1?)+(res2?) } )()
      }
    }
  }

  // Produce a random tree.  The parameter w represents
  // the reciprocal of the probability of producing a Leaf
  val random = new scala.util.Random;
  def MakeTree(w:Int) : IntTree = {
    if(random.nextInt(w)==0) return new Leaf(random.nextInt(10))
    else return new Branch(MakeTree(w-1), MakeTree(w-1))
  }

  def main(args : Array[String]) = {
    val t = MakeTree(3); println(t);
    val result = OneOne[Int];
    (Adder(t, result) || proc{ println(result?) })()
  }
}
```

Answer to question 5 †

```
// Bounded buffer, implemented using semaphore

import ox.CSO._

object BoundedBuffSemaphore{

  class Buff[T](n:Int){
    private val elements = new Array[T](n);
    private var first = 0; private var next = 0;
    // Elements held in elements[first..next) looping round;
    // 0 <= first,next < n
    private val mutex = new Semaphore; // for mutual exclusion
    private val size = new CountingSemaphore(0); // stores size of queue
    private val spaces = new CountingSemaphore(n); // stores # free spaces
```

```
    def add(v:T) = {
      spaces.down; mutex.down;
      elements(next)=v; next = (next+1)%n;
      size.up; mutex.up;
    }

    def remove:T = {
      size.down; mutex down;
      val result = elements(first); first = (first+1)%n;
      spaces.up; mutex.up; return result;
    }
  }

  val buff = new Buff[Int](5);

  def Producer = proc{
    var n = 0;
    repeat{ buff.add(n); println("Added "+n); n+=1; sleep(400) }
  }

  def Consumer = proc{
    val random = new scala.util.Random;
    repeat{ println(buff.remove); sleep(random.nextInt(1000)); }
  }

  def main(args : Array[String]) = (Producer || Consumer)();
}
```