Concurrent Programming: Exercise Sheet 2

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 will be placed on the course website. You should attempt these questions, mark them yourself, and hand them in.

Question 1 †

In the numerical integration example, we assumed that the number n of intervals was divisible by the number *ntasks* of tasks. How can we remove this assumption?

Question 2

[**Programming**] Write a concurrent program to multiply two N by N matrices a and b together, storing the result in matrix c. You should use the bag of tasks pattern. You should consider what a suitable "task" should be.

You will probably want to represent each matrix by a two dimensional array. Such an array can be initialised in Scala using, e.g.:

```
var a = new Array[Array[Int]](N,N);
```

The element in position (i, j) can be accessed using `a(i)(j)`.

Question 3 †

Suppose you are given two arrays, as follows:

```
val N = ...; // some large number
val a = new Array[Int](N);
val b = new Array[Int](N);
```

Neither array contains any repetitions. Describe the design for a concurrent program to count the number of distinct values that appear in both arrays. (You do not need to produce concrete code, unless you want to.) **Hint:** use a hash table (`scala.collection.mutable.HashSet`).

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

Question 4

Write a definition for a process

```
def Interleave(left: ?[Int], right: ?[Int], out: ![Int]) = proc ...
```

that inputs two streams of integers on left and right, interleaves them into a single stream (in some order), and outputs that on out. You should think carefully about what to do when one of the streams is closed. You may not use ox.cso.Components.merge.

Discuss whether your implementation is *fair* to the two input streams.

Question 5 †

Design a *change machine* process with signature:

```
def ChangeMachine(inPound: ?[Unit], out5p: ![Unit], out10p: ![Unit], out20p: ![Unit])
```

where communications on the channels correspond to the insertion of £1, and the output of a 5p, 10p or 20p piece, respectively. It should be willing to accept inPound whenever it had a zero balance. It should offer the environment the choice between how it wants the change, subject to the condition that it should not output coins of more value than it has received.

Question 6

Implement an unbounded buffer as a process:

```
def Buff[T](in: ?[T], out: ![T]) = proc{ ... }
```

The process should always be willing to input on in; it should output data on out in the order in which they were received, and should be willing to output whenever it has input data that have not yet been output.

Hint: you might want to use an instance of a scala . collection .mutable.Queue to store the current values held in the buffer.

Question 7 †

[**Programming**] The *readers and writers problem* is a classic synchronisation problem. Consider a collection of processes, each of which want to access a database concurrently, some for reading and some for writing. It is allowable for several readers to be accessing the database simultaneously, since their transactions will be independent. However, it is not allowable for two writers to have simultaneous access, nor for a reader and writer to have simultaneous access.

Implement a solution to the readers and writers problem using a server process: each reader and writer should communicate with the server when entering or leaving.

Is a reader or writer who is trying to gain entry to the database guaranteed to succeed, or could it be locked out forever? If the latter, how could this be avoided?

Question 8

[**Programming**] Consider the following synchronisation problem. There are two types of client 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. Design a server process to support this. Each client should send its name to the server, and receive back the name of its partner. Encapsulate the server within a class, with public methods

```
def ManSync(me: Name) : Name = ...
def WomanSync(me: Name) : Name = ...
```

Question 9

[**Programming**] Suppose N processes are connected in a ring. Each process i (for $i = 0, \ldots, N-1$) initially holds a value $x_i : T$. Write a concurrent program so that each process ends up in possession of the value

$$f(f(f(\ldots f(x_0, x_1), x_2) \ldots), x_{N-1}),$$

for some given function f. You should state properties that hold at key points of your program.

What property of f will allow a different program that achieves the same goal? Explain your answer. (An implementation is not necessary.)

Question 10

Consider a system constructed from N similar processes, with identities $0, \ldots, N-1$. Each process can send messages to other process, on synchronous channels.

Each process is either *active* or *passive*. When a process is passive, it simply waits to receive a message, at which point it becomes active. When a process is active, it can send messages to any other processes, but may eventually become passive.

The aim of this question is to consider a technique for determining whether all the processes are passive, in which case the system can terminate.

(a) Consider the following scheme. Process 0, when it is passive, sends a token to process 1, containing a boolean value, initially *true*. Each process k, when it receives a token of this form, sends a similar token to process $(k + 1) \bmod N$ (so the token circulates, as if the processes are arranged in a ring); process k sets the boolean to be *false* if it is active, or passes on the value it receives if it is passive. When the token returns to process 0, if the boolean is *true*, it assumes that all processes are passive, so sends a message to all processes telling them to terminate.

Explain why this scheme does *not* achieve the desired goal.

(b) Describe how to adapt this scheme so that it does achieve the desired goal. Explain why your scheme works. Make clear what (safety and liveness) properties your scheme achieves.

(c) Does it make a difference if the channels are asynchronous (i.e. buffered)?

Answer to question 1 †

The easiest thing to do is to make most tasks of size $\lceil n/ntasks \rceil$, and to make the last task appropriately smaller. Making the last task smaller is likely to make the overall computation finish faster.

It is possible to use a more sophisticated scheme, where all the tasks are of size $\lceil n/ntasks \rceil$ or $\lfloor n/ntasks \rfloor$, but, frankly, these don't give any benefits.

Answer to question 3 †

The obvious approach is to split a into aSegs segments, and b into bSegs segments, and to give a segment of a and a segment of b to each worker (possibly several pairs of segments, but see below). The worker can count the number of common values in these segments by putting the elements of the segment of b, say, into a hash table, and then testing whether each element of the segment of a is in that hash table. The counts from each worker can be passed to a controller,that adds them up.

Should we use the bag of tasks pattern, or should each worker deal with a single segment of each array? A bit of thought shows that taking aSegs∗bSegs greater than the number of processes leads to duplicated work: splitting a segment of a into multiple segments means that the hash table for the segment of b has to be created extra times; and splitting a segment of b into multiple segments means that each element of a has to be dealt with extra times. We therefore avoid the bag of tasks pattern, and allocate a single segment of each array to each worker.

Code is not compulsory, but I'll include mine below, anyway. My experiments suggest that this is fastest (on an 8 processor machine) when we take aSegs = 1 and bSegs = 8. This is an artefact of the relative speeds of different operations, and I don't think could have been anticipated in advance.

```
// Given arrays a, b, with no repetitions, count the
// number of distinct elements in both a and b.

import ox.CSO._;

object CountDups{
  val N = 500000; // size of arrays
  val a = new Array[Int](N);
  val b = new Array[Int](N);

  // Each worker will work on a segment of a and
  // a segment of b
  val aSegs = 1; // number of segments of a
  val aSegSize = N/aSegs
  val bSegs = 8; // number of segments of b
  val bSegSize = N/bSegs
  val numWorkers = aSegs*bSegs;

  // A single worker
  def Worker(me: Int, toController: ![Int]) = proc{
    // This worker will deal with a[aStart..aEnd)
    // and b[bStart..bEnd)
    def aStart = (me/bSegs) * aSegSize;
    def aEnd = (me/bSegs + 1) * aSegSize;
    def bStart = (me%bSegs) * bSegSize;
    def bEnd = (me%bSegs + 1) * bSegSize;

    // Put all b elements in hash table
    // val bHash = new java.util.HashSet[Int](bSegSize*2);
    // for(i <- bStart until bEnd) bHash.add(b(i));
    val bHash = new scala.collection.mutable.HashSet[Int];
    for(i <- bStart until bEnd) bHash += b(i);

    // Now iterate through segment of a, counting how
    // many entries are in bHash
    var count = 0;
    for(i <- aStart until aEnd)
      if(bHash.contains(a(i))) count+=1;

    // Send result to controller
    toController!count;
  }

  // The controller
  def Controller(toController: ?[Int]){
    var count = 0;
    for(i <- 0 until numWorkers) count += (toController?);
    println(count);
  }

  // Initialise arrays
  def Init{
    // We set a(i)=i, b(i)=2*i;
    for(i <- 0 until N){ a(i)=i; b(i)=2*i; }
  }

  // Construct system
  val toController = ManyOne[Int];
  def Workers =
    || ( for(i <- 0 until numWorkers) yield
           Worker(i, toController) )
  def System = Controller(toController) || Workers

  def main(args:Array[String]) = {
    Init;
    val t0 = java.lang.System.currentTimeMillis();
    System();
    println("Time␣taken:␣"+
      (java.lang.System.currentTimeMillis()-t0)/1000.0);
  }
}
```

Curiously, the Scala hashtable seems considerably faster than the Java one.

Answer to question 5 †

```
def ChangeMachine(inPound: ?[Unit], out5p: ![Unit], out10p: ![Unit], out20p: ![Unit])
= proc{
  var credit = 0; // credit in pence
  serve(
    (credit==0 &&& inPound) -?-> { inPound?; credit+=100; }
    | (credit>=5 &&& out5p) -!-> { out5p!(); credit-=5; }
    | (credit>=10 &&& out10p) -!-> { out10p!(); credit-=10; }
    | (credit>=20 &&& out20p) -!-> { out20p!(); credit-=20; }
  )
}
```

Answer to question 7 †

The server needs to keep track of the numbers of readers and writers currently accessing the database, and maintain the invariant

$$(readers = 0 \wedge writers \leqslant 1) \vee writers = 0$$

Avoiding starvation is tricky, since, say readers may always gain access, locking the writers out for ever. Below, priority is given to processes wishing to leave, and when $readers = 0 \wedge writers = 0$, it alternates between giving priority to a reader or a writer (according to the flag writerPri).

```
// The readers and writers problem
import ox.CSO._

object ReadersWriters{
  val random = new scala.util.Random;

  // A reader, which just repeatedly enters and leaves
  def Reader(me: Int, enter: ![Int], leave: ![Int]) = proc{
    repeat{
      sleep(random.nextInt(2000)); enter!me;
      sleep(random.nextInt(1000)); leave!me
    }
  }

  // A reader, which does likewise
  def Writer(me: Int, enter: ![Int], leave: ![Int])= proc{
    repeat{
      sleep(random.nextInt(2000)); enter!me;
      sleep(random.nextInt(1000)); leave!me
    }
  }

  // A server which controls the entering and leaving,
  // maintaining the invariant: (readers==0 && writers<=1) || writers==0
  // Priority is always given to leaving. When readers==0 and writers==0,
  // priority is given to a writer to enter next iff writerPri
  def Server(readerEnter: ?[Int], readerLeave: ?[Int],
             writerEnter: ?[Int], writerLeave: ?[Int])
  = proc{
    var readers = 0; var writers = 0;
    def Report = { println("readers="+readers+";␣writers="+writers); }
    var writerPri = true; // Does writer get priority?
    priserve(
      (readers>0 &&& readerLeave) --> {
        val id = readerLeave?; readers-=1; println("Reader␣"+id+"␣leaves"); Report
      }
      |
      (writers>0 &&& writerLeave) --> {
        val id = writerLeave?; writers-=1; println("Writer␣"+id+"␣leaves"); Report
      }
      |
      (writerPri && readers==0 && writers==0 &&& writerEnter) --> {
        val id = writerEnter?; writers+=1; writerPri=false;
        println("Writer␣"+id+"␣enters"); Report
      }
      |
      (writers==0 &&& readerEnter) --> {
        val id = readerEnter?; readers+=1; writerPri=true;
        println("Reader␣"+id+"␣enters"); Report
      }
      |
      (!writerPri && readers==0 && writers==0 &&& writerEnter) --> {
        val id = writerEnter?; writers+=1; writerPri=false;
        println("Writer␣"+id+"␣enters"); Report
      }
    )
  }

  // Number of readers and writers:
  val readers = 5; val writers = 5

  // Declare the channels
  val readerEnter, readerLeave, writerEnter, writerLeave = ManyOne[Int];

  def Readers =
    || ( for(i <- 0 until readers) yield Reader(i, readerEnter, readerLeave) )
  def Writers =
    || ( for(i <- 0 until readers) yield Writer(i, writerEnter, writerLeave) )
  def System =
    Readers || Writers || Server(readerEnter,readerLeave,writerEnter,writerLeave)

  def main(args : Array[String]) = { System() }
}
```

This does not guarantee starvation freedom, since the readers may try to enter sufficiently often that *readers* never reaches zero, so no writer can gain entry. This can be avoided by having writers register their interest (by another communication, that the server always accepts), and then not allowing any more readerEnter events before a writerEnter event.