Concurrent Programming: Exercise Sheet 1

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.

In this and subsequent exercise sheets questions marked † will not normally be discussed in the class. Model answers are included below. You should attempt these questions, evaluate your attempts yourself, and hand them in with your other work.

I am very greatly indebted to Gavin Lowe for preparing all the questions and model answers for all the exercise sheets for the course this year.

Question 1 †

Consider a web browser that supports multiple tabs (i.e. different tabs for different web pages). Why might such a web browser be implemented using concurrency? Would this still make sense on a uni-processor computer? What problems might arise from such a design?

Question 2 †

Suppose process P performs m atomic actions (sequentially), and process Q performs n atomic actions (sequentially). When P and Q are run in parallel, how many interleavings of atomic actions are there?

Question 3

Consider the code below.

```
var x = 0;

def P = proc{ x = x+1; x = x+2; }

def Q = proc{ x = x+4; }

def System = P || Q
```

Suppose the atomic actions are reading or writing a variable. When System is run, what are the possible final values of x? Draw a diagram to illustrate the different possible execution paths.

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

Question 4

Consider the code below.

```
var x = 0; var y = 10;

def P = proc{ while(x!=y) x = x+1; }

def Q = proc{ while(x!=y) y = y-1; }

def System = P || Q
```

Will the process System terminate? Explain your answer.

Question 5

Consider an object representing a bank account, from the following class.

```
class Account{
  var balance = 0;

  def credit(value : Int) = balance += value;

  def canDebit(value : Int) : Bool = return(balance >= value);

  def debit(value : Int) = balance -= value;
}
```

Suppose each procedure is performed atomically (we will see how to do this in a later chapter).

A process that wants to perform a debit should first of all call canDebit, to avoid the account from going overdrawn; for example

```
def doDebit(value : Int) = {
  if(canDebit(value)) debit(value);
  else println("Debit not allowed!");
}
```

What can go wrong if two processes execute doDebit at the same time? Sketch a solution to this problem.

Question 6

[**Programming**] Write a definition for a process

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

that inputs two ascending non-empty streams of integers on left and right, merges them into a single ascending stream, and outputs that on out. The process should hold at most one value at a time from each of the input streams. You may assume that the input streams are never closed.

Question 7

CSO channels are rather like CSP events. Describe one or two ways in which they are different.

Question 8

[**Programming**] Implement a program to sort N integers using a pipeline of N components. Each component should receive a stream of values on its left channel, keep the largest, and pass the rest to the next process, via its right channel. Each process should hold at most two values at any time: the most recently received value and the largest seen so far.

What is the execution time of the program in terms of the number of sequentially-ordered messages?

Question 9

(a) Write a definition for a process

```
def sumStreams(in1: ?[Int], in2: ?[Int], out: ![Int]) = proc{ ... }
```

that repeatedly inputs a value from each of its input ports (in either order), and outputs the sum on the output port. (Functional programmers might like to think of this process as implementing zipwith (+).)

(b) The Fibonacci numbers fibs (0, 1, 1, 2, 3, 5, ...) can be defined using the equation

```
fibs = 0 : sumStreams( fibs , 1:fibs )
```

(i.e. the stream that starts with 0, and continues with the result of summing the stream fibs and the result of pre-pending fibs with 1).

Design and implement a circuit with signature

```
def Fibber(out: ![Int]) = ...
```

that outputs the Fibonacci numbers on out, making use of the above equation. Draw a picture to explain your design. Produce a simple test rig to test your implementation.

Question 10 †

[**Programming**]

(a) Write a definition for a component

```
def zipwith[L,R,O](f:(L, R)=>O)(lin:?[L], rin:?[R], out:![O]) = proc{ ... }
```

which repeatedly inputs values l and r on lin and rin, respectively, and outputs f(l,r) on out.

(b) An *integrator* is a process with signature

```
def integrator(in: ?[Int], out: ![Int]) = ...
```

that repeatedly inputs on in, and outputs on out the sums of the inputs so far. That is, if it receives the inputs x1, x2, x3, ... , it outputs x1, x1+x2, x1+x2+x3, ... Design and implement an integrator as a circuit using zipwith and other common components. Draw a diagram to explain your design. Produce a simple test rig to test the integrator.

Answer to question 1 †

Concurrency might be used to give better response to user actions (i.e. lower latency), and better overall performance (i.e. higher throughput); of these, the former seems more important. For example, we might choose to use a different thread for each tab. As an example of why a sequential implementation might be unsatisfactory, consider a user who has two tabs open, one viewing a page that automatically re-loads every few minutes, and another that the user is actively reading. Suppose the user tries to scroll down in the second page just after the first page starts a re-load; then the user has to wait until the re-load completes before the second page scrolls; this could take several seconds, and so annoy the user. (Firefox seems to act in this way on my computer.) If the different tabs used different processes, then they could run concurrently so the user wouldn't see this delay. This would still be the case on a uni-processor machine: the fact that the two processes are competing for the processor would slow things down a bit, but probably not enough for the user to notice.

The problem is, of course, that two processes might try to access some shared data (e.g. the history list) simultaneously, leading to race conditions. These race conditions can be avoided by careful programming: see the rest of the course!

Answer to question 2 †

There are a total of $m + n$ atomic actions. The number of ways of choosing which m of these belong to P is

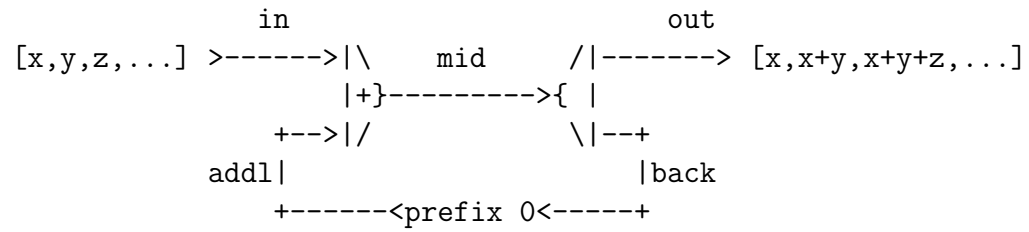
$$\binom{m+n}{m} = \frac{(m+n)!}{m!n!}$$

Answer to question 10 †

```
def zipwith[L,R,O](f:(L, R)=>O)(lin:?[L], rin:?[R], out:![O]) = proc{
  var l = null.asInstanceOf[L];
  var r = null.asInstanceOf[R];
  repeat {
    (proc { l = lin? } || proc { r = rin? })();
    out!f(l, r)
  };
  lin.close; rin.closein; out.closeout
}
```

In fact, this is defined as ox.cso.Components.zipwith.

For the integrator, we'll use a circuit like before (where the first component is a zipwith using adition.




```
import ox.CSO._, ox.cso.Components

object Integrator{

  def integrator(in: ?[Int], out: ![Int]) = {
    val mid, back, addl = OneOne[Int]
    (  Components.zipwith ((x:Int, y:Int)=>x+y) (in, addl, mid)
    || Components.tee (mid, List(out, back))
    || Components.prefix(0)(back, addl)
    )
  }

  // Produce stream of nats
  def nats(out: ![Int]) = proc("nats"){
    var n = 0;
    repeat{ out!n; n+=1; sleep(200) }
  }

  // Test rig, using the above
  def TestRig = {
    val in, out = OneOne[Int];
    ( integrator(in, out) || nats(in) || Components.console(out) )
  }

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

The test rig provides the integrator with naturals, and we can check that we get the triangular numbers out.