# Concurrent Programming

## Question 1

The aim of this question is to write concurrent programs, using CSO, to update the $N$-by-$N$ array a to hold values defined as follows:

$$\begin{array}{rcll} \mathsf{a}(0)(j) & = & j, & \text{for } 0 \leq j < N, \\ \mathsf{a}(i)(0) & = & \mathsf{a}(i-1)(0), & \text{for } 1 \leq i < N, \\ \mathsf{a}(i)(j) & = & f(\mathsf{a}(i-1)(j-1), \mathsf{a}(i-1)(j)), & \text{for } 1 \leq i < N,\ 1 \leq j < N, \end{array}$$

where $f$ is some given function.

Note that you will need to ensure that your programs calculate the entries of a in a suitable order.

(a) Write a concurrent program that uses $N$ workers to solve this problem. The $j$th worker should be responsible for calculating $\mathsf{a}(i)(j)$ for $0 \leq i < N$. Explain your design. (10 marks)

(b) Write a concurrent program that uses $p$ worker processes, where $p < N$, to solve this problem. Your program should use the bag-of-tasks pattern, where each task should involve calculating a *single* entry in a. Explain your design. (15 marks)

As always, you should aim to make your programs reasonably efficient.



**TURN OVER**

## Question 2

Consider a collection of nodes that are connected together in a directed graph. Each node has a variable

```
neighbours: List [NodeId]
```

listing its neighbours in the graph, where

```
type NodeId = Int
```

is the type of node identifiers.

If Node n1 has a neighbour n2, then n1 can send n2 messages on the channel toNode(n2), and n2 can send messages back on toNode(n1). However, nodes can send messages only to their neighbours.

The aim of this question is to produce concurrent programs, using CSO, to find a spanning tree of the graph. For each part, you should give a brief description of your design. You should state any assumptions you make about the toNode channels.

(a) Write a concurrent program to find a spanning tree. The protocol should be initiated by Node 0, which should become the root of the spanning tree. Each node except Node 0 should end with a variable parent holding the identity of its parent in the tree. Node 0 should start by sending a message to each of its neighbours. The first time that a node receives a message, it should accept the sender of the message as its parent, and should send a message on to all of its other neighbours. (7 marks)

(b) Now adapt your answer to part (a) so that, in addition, each process ends up with a variable children : List [NodeId] that holds the identities of its children, in some order. Further, the spanning tree should be constructed by exploring the graph in a depth-first order. (12 marks)

(c) Now describe how to adapt the program so that the spanning tree is constructed by exploring the graph in a *breadth-first* order. Code is not required, but you should give a clear explanation of how to achieve this goal. (6 marks)

## Question 3

Consider a single-lane road bridge, crossing a river from north to south. Cars going in the same direction can cross the bridge simultaneously, but cars going in different directions may not. The aim of this question is to implement classes, using CSO, to control the bridge, with the following signature:

```
class Bridge{
  /** Car arrives at north end. */
  def arriveNorth = {...}
  /** Car leaves at south end. */
  def leaveSouth = {...}
  /** Car arrives at south end. */
  def arriveSouth = {...}
  /** Car leaves at north end. */
  def leaveNorth = {...}
}
```

A car arriving at the north end of the bridge calls arriveNorth; this call should block until the car may enter the bridge. When the car leaves the south end of the bridge it calls leaveSouth. Cars travelling from south to north call arriveSouth and leaveNorth, similarly.

The code should enforce both the safety condition that cars are never simultaneously crossing the bridge in different directions, and also the liveness condition that each car that calls one of the arrive procedures may eventually cross the bridge. Hint: one way to achieve the liveness condition is to not allow a car that arrives at the bridge to immediately enter it if there is a car waiting at the other side.

You may find that your code for cars travelling from south to north is very similar to the code for cars travelling from north to south. If so, you should give the code for one direction in full, but may omit the code for the other direction.

The physical characteristics of the bridge will prevent one car from overtaking another while on the bridge. Your code does *not* need to enforce this.

(a) Implement the Bridge class by encapsulating a server process. (9 marks)

(b) Now implement the Bridge class as a monitor. (8 marks)

(c) Finally implement the Bridge class using semaphores. (8 marks)

Note that in each part you should use *only* the concurrency primitive mentioned. You should explain your designs.

# Concurrent Programming: Model answers

## Answer to question 1

The difficulty is in ensuring that processes calculate values in a suitable order.

(a) The solution below arranges for each worker $j$, for $j < N-1$, to signal on the channel flag(j+1) to worker $j+1$ after writing each value; the $i$th such signal (counting from 0) indicates that $a(i)(j)$ has been written. Worker $j+1$ will calculate the next value only after receiving this signal. We use buffered channels for efficiency.

```
object Recurrence1{
  val N = 10
  val a = new Array[Array[Int]](N,N)

  val flag = Array.fill(N)(OneOneBuf[Unit](1)) // indexed by recipients

  def f(m:Int, n:Int) = m+n // for testing

  def worker(j:Int) = proc("Worker"+j){
    a(0)(j) = j
    for(i <- 1 until N){
      if(j < N-1) flag(j+1)!() // signal to j+1
      if(j > 0){
        flag(j)?() // wait for signal
        a(i)(j) = f(a(i-1)(j-1), a(i-1)(j))
      }
      else a(i)(j) = a(i-1)(j) // j = 0 doesn't need to wait
    }
  }

  def main(args: Array[String]) = {
    run(|| (for(j <- 0 until N) yield worker(j)))
    for(j <- 0 until N) println(a(N-1)(j))
  }
}
```

(b) In order to ensure that values are calculated in a suitable order, the controller has an array of booleans indicating which values have been calculated; tasks are added to its queue when the necessary prior tasks have been completed.

```
object Recurrence2{
  val N = 10
  val a = new Array[Array[Int]](N,N)

  type Task = (Int, Int)
  // the task (i,j) tells the worker to calculate a(i)(j)

  val toWorkers = OneMany[Task] // channel for distributing tasks
```

```
  val done = ManyOne[Task] // channel to tell controller that task is done
  def f(m:Int, n:Int) = m+n // for testing

  def worker = proc{
    repeat{
      val(i,j) = toWorkers?()
      if(i == 0) a(i)(j) = j
      else if (j > 0) a(i)(j) = f(a(i-1)(j-1) , a(i-1)(j))
      else a(i)(j) = a(i-1)(j)
      done!(i,j)
    }
  }

  def controller = proc{
    val b = Array.fill[Boolean](N,N) // indicates which values are done; initialised to all false
    // queue stores the tasks that are ready to be done
    val queue = new scala.collection.mutable.Queue[Task]
    for(j <- 0 until N) queue += (0,j)
    var busyWorkers = 0 // number of busy workers

    // Main loop
    serve(
      (queue.nonEmpty && toWorkers) =!=> { busyWorkers += 1; queue.dequeue }
      |
      (busyWorkers > 0 && done) =?=> { case (i,j) =>
        b(i)(j) = true; busyWorkers -= 1
        // add tasks to queue if possible
        if(i < N-1){
          if(j == 0 || b(i)(j-1))
            // (i,j-1) and (i,j) done, so can do task (i+1,j)
            queue += (i+1,j)
          if(j < N-1 && b(i)(j+1))
            // (i,j) and (i,j+1) done so can do task (i+1,j+1)
            queue += (i+1,j+1)
        }
      }
    )
    // Loop terminates when queue.isEmpty && busyWorkers = 0

    toWorkers.close
  }

  def main(args: Array[String]) = {
    val p = 4 // number of workers
    val workers = || (for(j <- 0 until p) yield worker)
    run(workers || controller)
    for(j <- 0 until N) println(a(N-1)(j))
  }
}
```

## Answer to question 2

(a) Node 0 sends a Req request message to each of its neighbours. The first time that a node receives a message, it accepts the sender of the message as its parent, and sends a Req message on to all of its other neighbours.

Here's a complete program, although only about the first half is required.

```
object SpanningTree0{

  val N = 10 // number of nodes
  type NodeId = Int // identities of nodes

  abstract class Msg
  case class Req(n: NodeId) extends Msg
    // suggestion from potential parent n

  val toNode = Array.fill(N)(OneOneBuf[Msg](N-1)) // channels to each node

  def node(me: Int, neighbours: List[NodeId]) = proc{
    var parent = 0

    if(me != 0){ // Wait to receive notification from parent
      toNode(me)?() match{ case Req(p) => parent = p }
    }
    println("Node "+me+" chooses parent "+parent)

    // Send out request to potential children
    for(n <- neighbours; if n != parent){ toNode(n)!Req(me) }

    // There's no real need to consume other requests here
  }

  /** Generate neighbours randomly. */
  val allNeighbours = new Array[List[NodeId]](N)
  def generateNeighbours = {
    // Generate links randomly
    val links = new Array[Array[Boolean]](N,N)
    for(i <- 0 until N; j <- i+1 until N){
      if(scala.util.Random.nextDouble < 0.5){
        links(i)(j) = true; links(j)(i) = true
      }
    }
    // Now build lists of neighbours
    for(i <- 0 until N){
      allNeighbours(i) = List[NodeId]()
      for(j <- 0 until N) if(i != j && links(i)(j)) allNeighbours(i) ::= j
    }
  }
```

```
  def main(args:Array[String]) = {
    generateNeighbours
    for(i <- 0 until N) println("neighbours("+i+")␣=␣"+allNeighbours(i))
    run( || (for (i <- 0 until N) yield node(i, allNeighbours(i))) )
  }
}
```

(b) When a node first receives a Req message, it sends Req messages to each of its neighbours in turn, waiting for a response from each before sending to the next; it then sends a response back to its parent. Each Req message is responded to with a Resp message, than indicates whether the sender of the Resp has accepted the sender of the Req as its parent. Note that a node needs to reply to subsequent Req messages negatively, until it has sent a Resp to each of its neighbours.

```
abstract class Msg
case class Req(n: NodeId) extends Msg
  // suggestion from potential parent n
case class Resp(n: NodeId, ok: Boolean) extends Msg
  // response, either positive or negative, from node n

val toNode = Array.fill(ManyOne[Msg]) // channels to each node

def Node(me:Int, neighbours: List[NodeId]) = proc{
  var parent = 0

  if(me != 0){ // Wait to receive notification from parent
    toNode(me)?() match{ case Req(p) => parent = p }
  }
  println("Node␣"+me+"␣chooses␣parent␣"+parent)

  var children = List[NodeId]()
  var reqs = if(me == 0) 0 else 1 // number of requests so far

  // Send out requests to potential children, and wait for reply
  for(n <- neighbours; if n != parent){
    toNode(n)!Req(me)
    var done = false
    while(!done){
      toNode(me)? match{
        case Req(n1) => // refuse new invitations
          reqs += 1; toNode(n1)!Resp(me, false)
        case Resp(n1,b) =>
          assert(n1 == n); done = true // received response
          if(b) children ::= n // add to list of children if positive response
      } // end of match
    } // end of while
  } // end of for

  println("Node␣"+me+"␣has␣children␣"+children)
```

```
    if (me != 0) toNode(parent)!Resp(me,true) // send agreement to parent

    // Consume outstanding requests
    val len = neighbours.length-children.length // # expected reqs
    while(reqs < len){
      toNode(me)? match{
        case Req(n1) => // refuse new invitations
          reqs += 1; toNode(n1)!Resp(me, false)
      }
    }
  }
```

Most students will fail to consume outstanding Req messages at the end, which will normally lead to deadlock [3 marks for this bit].

(c) The protocol proceeds in rounds, with the $n$th round exploring the graph to a depth of $n$. The first time a node receives a Req, it just sends back a response, accepting its parent. The next time it receives a Req, it sends a Req on to all its other neighbours, to find if they will be its children. In order to achieve termination, the responses back up the tree have to include a field indicating whether the search has bottomed-out. Perhaps the easiest way to implement this is for the root node to detect when all branches have bottomed out, and to send a final "terminate" signal down the tree. [A better way is for a node whose sub-tree has bottomed-out to terminate, and for its parent to not subsequently send it messages; with this approach, nodes also need to avoid sending messages to neighbours from whom they've previously received invitations that they've rejected (in case those nodes have terminated).]

## Answer to question 3

Note: students are expected to elide about 45% of the code below.

(a) Each call to an Arrive procedure causes a reply channel to be sent to the server. The server sends a reply on this channel when the car should proceed.

```
class BridgeServer{
  private type ReplyChan = OneOne[Unit]
  // channels to server
  private val arriveNorth, arriveSouth = ManyOne[ReplyChan]
  private val leaveSouth, leaveNorth = ManyOne[Unit]

  def arriveNorth = {
    val c = OneOne[Unit]
    arriveNorth!c // register with server
    c?() // wait for acknowledgement to proceed
  }

  def leaveSouth = {
    leaveSouth!() // tell server we're done
```

```
  }

  def arriveSouth = {
    val c = OneOne[Unit]
    arriveSouth!c // register with server
    c?() // wait for acknowledgement to proceed
  }

  def leaveNorth = {
    leaveNorth!() // tell server we're done
  }

  private def server = proc{
    // Queues of ReplyChans for waiting cars
    val northQueue = new scala.collection.mutable.Queue[ReplyChan]
    val southQueue = new scala.collection.mutable.Queue[ReplyChan]
    var fromNorth = 0; var fromSouth = 0 // number on bridge
    // Invariant: fromNorth=0 or fromSouth=0

    serve(
      arriveNorth =?=> { c =>
        if(fromSouth > 0 || southQueue.nonEmpty) // need to wait
          northQueue += c
        else{ fromNorth += 1; c!() } // can go
      }
      |
      arriveSouth =?=> { c =>
        if(fromNorth > 0 || northQueue.nonEmpty) // need to wait
          southQueue += c
        else{ fromSouth += 1; c!() } // can go
      }
      |
      leaveSouth =?=> { () =>
        fromNorth -= 1
        if(fromNorth == 0) // release cars queuing from south
          while(southQueue.nonEmpty){ southQueue.dequeue!(); fromSouth += 1 }
      }
      |
      leaveNorth =?=> { () =>
        fromSouth -= 1
        if(fromSouth == 0) // release cars queuing from south
          while(northQueue.nonEmpty){ northQueue.dequeue!(); fromNorth += 1 }
      }
    ) // end of serve
  }

  server.fork // fork off server process
}
```

(b) Here's a version using a JVM monitor.

```
class BridgeMonitor{
  private val mon = new Monitor
  // Conditions for signalling that the route from south, resp. north, is clear.
  private val southClear, northClear = mon.newCondition

  private var fromNorth = 0; private var fromSouth = 0 // number on bridge
  private var waitingNorth = 0; private var waitingSouth = 0// numbers waiting

  def arriveNorth = monitor.withLock{
    if(fromSouth > 0 || waitingSouth > 0){ // need to wait
      waitingNorth+=1
      northClear.await
      assert(fromSouth == 0)
      waitingNorth-=1
    }
    // proceed
    fromNorth += 1
  }

  def arriveSouth = monitor.withLock{
    if(fromNorth > 0 || waitingNorth > 0){ // need to wait
      waitingSouth+=1
      southClear.await
      assert(fromNorth == 0)
      waitingSouth-=1
    }
    // proceed
    fromSorth += 1
  }

  def leaveSouth = monitor.withLock{
    fromNorth -= 1
    if(fromNorth == 0 && waitingSouth > 0) southClear.signalAll()
  }

  def leaveNorth = monitor.withLock{
    fromSouth -= 1
    if(fromSouth == 0 && waitingNorth > 0) northClear.signalAll()
  }
}
```

Note that once a waiting car $c$ receives a signal, no car will enter from the other direction until (at least) $c$ has left the bridge.

(c)

```
class BridgeSemaphore{

  private var fromNorth = 0; private var fromSouth = 0 // number on bridge
  private var waitingNorth = 0; private var waitingSouth = 0// numbers waiting
```

```
  private val mutex = MutexSemaphore() // for mutual exclusion
  // Semaphores to signal to cars waiting at each end
  private val north = SignallingSemaphore()
  private val south = SignallingSemaphore()

  def arriveNorth = {
    mutex.down
    if (fromSouth > 0 || waitingSouth > 0){ // need to wait
      waitingNorth += 1; mutex.up; north.down // wait for signal
      waitingNorth -= 1
    }
    fromNorth+=1
    signalNorth // maybe signal to next car at this end
  }

  // Signal to cars waiting in the north, if any
  private def signalNorth = if(waitingNorth > 0) north.up else mutex.up

  def arriveSouth = {
    mutex.down
    if (fromNorth > 0 || waitingNorth > 0){ // need to wait
      waitingSouth += 1; mutex.up; south.down // wait for signal
      waitingSouth -= 1
    }
    fromSouth+=1
    signalSouth
  }

  // Signal to cars waiting in the south, if any
  private def signalSouth = if(waitingSouth > 0) south.up else mutex.up

  def leaveSouth = {
    mutex.down
    fromNorth -= 1
    if (fromNorth == 0) signalSouth else mutex.up
  }

  def leaveNorth = {
    mutex.down
    fromSouth -= 1
    if (fromSouth == 0) signalNorth else mutex.up
  }
}
```