Concurrent Programming Collection

Question 1

In the bag-of-tasks pattern, there is normally a single process that holds the outstanding tasks. However, that process can then become a bottle-neck. The aim of this question is to investigate an alternative, where each node holds its own bag of tasks, but there is no central controller; i.e., the bag of tasks is distributed between the nodes. More precisely, each node is composed of two subprocesses, one implementing the bag, and the other working on data from the bag:

```
def Bag(me: Int, bagToWorker: ![Data], workerToBag: ?[Data],
        get: ?[Data], give: ![Data], done: ?[Unit]) = ...

def Worker(me: Int, workerToBag: ![Data],
           bagToWorker: ?[Data], done: ![Unit]) = ...
```

The Worker repeatedly gets a task from its bag (on channel bagToWorker) and, as a result, puts zero or more tasks back into the bag (via channel workerToBag), and then signals that it has completed that task (on channel done). You are *not* asked to provide an implementation of Worker for this question.

One way to implement the distributed bag of tasks involves arranging the nodes in a (logical) ring. The Bag can pass data to its clockwise neighbour on the channel give. It can accept a task from its anti-clockwise neighbour on the channel get, but only when it has no tasks in its own bag.

(a) Give an implementation of the Bag process. You do not need to consider termination for this part of the question. [15]

(b) Now explain how your answer to the previous part can be extended so that the system terminates when there are no outstanding tasks. You should give a precise description of your design, but need not produce code. [10]

Answer to question 1

Part a.

```
def Bag(me: Int, get: ?[Data], give: ![Data],
        bagToWorker: ![Data], workerToBag: ?[Data])
= proc("Bag"+me){
  // Store the tasks in a stack
  val stack = new scala.collection.mutable.Stack[Partial];
  /* Initialise stack in some way */

  var workerBusy = false; // Is the local worker busy?

  // Main loop
  while(true){
    if (!workerBusy)
      if (!stack.isEmpty){ // Send task to worker
```

```
          val p = stack.pop;
          prialt (
            toWorker -!-> { toWorker!p; workerBusy = true; }
            | give -!-> { give!p; }
          )
        }
        else{ val p = get?; toWorker!p; workerBusy = true; }
      else // workerBusy
        prialt (
          (!stack.isEmpty &&& give) -!-> { give!(stack.pop); }
          | fromWorker -?-> {
              val p = fromWorker?; stack.push(p);
            }
          | done -?-> { done?; workerBusy = false; }
        )
      }
    }
```

The priorities are chosen so as try to speed up distribution of tasks. [3 marks for some discussion of priorities.]

Part b. We arrange to send messages corresponding to the termination protocol on the get/give channels. This is best done by changing the type of those channels to some abstract type Msg, with two subtypes: Data (as above) and Terminating (for tokens in the termination protocol).

The termination protocol is initiated by Bag 0 when its stack is empty and its worker is idle. It sends a Terminating token round the ring, on the same channels as for the passing of partial solutions. Each subsequent node receives the token only when it is idle (the only time it receives on Get), and passes it on. If node 0 receives the token back and it has been idle ever since starting the termination protocol, then the whole system can terminate: it circulates the token again, with a flag set, to indicate this. If it becomes busy after initiating the termination protocol, it has to wait until it receives the token back before it re-tries the termination protocol. Therefore this node has to keep track of whether it is: not currently involved in the termination protocol; involved in the termination protocol; or has aborted the termination protocol and is waiting to receive the token back.

Code is not required, but the code below illustrates this, with an application to solve the Eight Queens Problem. [Note to tutors: I suggest you give this to students to study in their own time.] The class Partial (replacing Data) represents a partial solution to the puzzle. See comments.

```
// The 8 queens problem

import ox.CSO._;

abstract class Msg;

// Class to represent a partial solution

case class Partial(N:Int) extends Msg{
  // Represent partial solution by a list of Ints; the ith
  // entry represents the row number of the queen in column i
```

```
  // (0 <= i < len).  len is the length of board
  private var board : List[Int] = Nil;
  private var len = 0;

  def finished : Boolean = (len==N);

  // Is it legal to play in column len, row j?
  private def isLegal(j : Int) = {
    // is piece (i1,j1) on different diagonal from (len,j)?
    def otherDiag(p:(Int,Int)) = {
      val i1=p._1; val j1 = p._2; i1-len!=j1-j && i1-len!=j-j1;
    };

    (board forall ((j1:Int) => j1 != j)) &&
      // row j not already used
    (List.range(0,len) zip board forall otherDiag)
      // diagonals not used
  }

  // Return new partial resulting from playing in row j next
  private def doMove(j : Int) : Partial = {
    val newPartial = new Partial(N);
    newPartial.board = this.board ::: (j :: Nil);
    newPartial.len = this.len+1;
    return newPartial;
  }

  // Return list of successors, such that every solution to
  // this is a solution of one of the successors
  def successors : List[Partial] =
    for(j <- List.range(0, N) if(isLegal(j))) yield doMove(j);

  override def toString : String = {
    var st = "";
    for(i <- 0 until len) st = st + (i,board(i))+"\t";
    return st;
  }
}

// Class of messages used in testing for termination

case class Terminating(status:Boolean) extends Msg{ }
// Status of false means this is a probe if we can terminate;
// status of true means terminate now

// The main object
object EightQueens{
  val N = 8; // size of board

  // A worker.  This node can get a partial solution from its
  // anticlockwise neighbour on channel get; it can give a
  // partial solution to its clockwise neighbour on channel give
  def Node(me: Int, get: ?[Msg], give: ![Msg])
  = proc("Node"+me){

    // Process to maintain this bag of tasks
    def Bag(toWorker: ![Partial], fromWorker: ?[Partial],
```

```
        done: ?[Unit])
= proc("Bag"+me){
  // Store the tasks in a stack
  val stack = new scala.collection.mutable.Stack[Partial];
  if (me==0) stack.push(new Partial(N)); // Starting position

  var workerBusy = false; // Is the local worker busy?
  var finished = false; // Becomes true when we can terminate

  // Status of termination protocol, node 0 only
  val NONTERM = 0; val TRYING = 1; val ABORTED = 2;
  var terminationStatus = NONTERM;
  // NONTERM means not currently involved in termination
  // protocol; TRYING means termination protocol started,
  // and this node has been idle since then; ABORTED means
  // termination protocol was started, and this node then
  // became busy, but the protocol hasn't yet terminated.

  // Main loop
  while(!finished){
    if (!workerBusy)
      if (!stack.isEmpty){ // Send task to worker
        val p = stack.pop;
        prialt (
          toWorker -!-> { toWorker!p; workerBusy = true; }
          | give -!-> { give!p; }
        )
      }
      else{ // Idle, so wait for message from neighbour
        if (me==0 && terminationStatus==NONTERM){
          // Initiate termination protocol
          println ("Termination protocol starting");
          give!Terminating(false);
          terminationStatus = TRYING;
        }
        val m:Msg = get?;
        m match {
          case Partial(n) => {
            toWorker!m.asInstanceOf[Partial];
            workerBusy = true;
            if (me==0 && terminationStatus==TRYING){
              println ("Termination protocol aborted");
              terminationStatus = ABORTED;
            }
          }
          case Terminating(s) => {
            if (me==0){
              if (terminationStatus==TRYING){
                // We've received nothing since sending the
                // termination probe, so we can terminate
                give!Terminating(true);
                  // pass signal to next node
                toWorker.close; // Tell worker to close
                get?; // Receive termination signal back
                finished = true; // Can terminate
              }
              else if (terminationStatus==ABORTED){
```

```
                terminationStatus = NONTERM;
                println("Ready to retry termination protocol");
              }
            }
            else{ // me!=0
              give!m; // pass signal on
              if(s){ finished = true; toWorker.close; }
                // terminating
            }
          } // end of case Terminating(s)
        } // end of match
      }
      else // workerBusy
        prialt (
          (!stack.isEmpty &&& give) -!-> { give!(stack.pop); }
          | fromWorker -?-> {
              val p = fromWorker?; stack.push(p);
            }
          | done -?-> { done?; workerBusy = false; }
        )
    }
  }

  // Process to work on partial solutions
  def Worker(toBag: ![Partial], fromBag: ?[Partial],
             done: ![Unit])
  = proc("Worker"+me){
    repeat{
      val partial = fromBag?; // get job
      if(partial.finished){ println(partial); } // done!
      else // Generate all next-states
        for(p1 <- partial.successors) toBag!p1;
      // sleep(10);
      done!();
    }
  }

  // Put this node together
  val bagToWorker, workerToBag = OneOne[Partial];
  val done = OneOne[Unit];
  def ThisNode = (
    Bag(bagToWorker, workerToBag, done) ||
    Worker(workerToBag, bagToWorker, done)
  );
  ThisNode();
}

val p = 8; // Number of workers

// Put system together

val passPartial = OneOne[Msg](p); // indexed by recipient's id

def System =
  || ( for(i <- 0 until p) yield
       Node(i, passPartial(i), passPartial((i+1)%p)) )
```

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

Question 2

The *region labelling problem* is a problem that occurs in image analysis.

Assume that each pixel in an image is represented by a boolean: *true* indicates that the pixel is lit, and *false* represents that the pixel is unlit. The states of the pixels are stored in an array

```
val a = new Array[Array[Boolean]](N,N);
```

Let the *neighbours* of a pixel be the (up to four) pixels immediately above, below, left or right of it. (Note that the top and bottom rows are not considered adjacent, and the left and right columns are not considered adjacent.) Two pixels are in the same *region* if (and only if) there is a path from one to the other, where all pixels in the path are lit, and successive pixels in the path are neighbours. For example in the image below, there are three regions.

The aim of this question is to identify the distinct regions and to allocate each a unique label. Each label will be of type:

```
type Coord = Tuple2[Int, Int];
```

More precisely, the label for each region will be the coordinates of the left-most pixel in the top row of the region, i.e., the one that is smallest according to the lexicographic ordering.

```
def smaller(p1:Coord, p2:Coord) : Boolean = {
  val i1=p1._1; val j1 = p1._2;
  val i2=p2._1; val j2 = p2._2;
  (i1<i2 || i1==i1 && j1<j2)
}
```

An algorithm to calculate this is initially to label each pixel with its own coordinates; and then to repeatedly relabel each pixel with the smallest label of itself or its lit neighbours.

```
for((i1, j1) <- neighbours(i,j))
  if(a(i1)(j1) && smaller(label(i1)(j1), label(i)(j)))
    label(i)(j) = label(i1)(j1)
```

where `neighbours(i,j) : List[Coord]` gives the neighbours of `(i,j)`; you may assume a definition for `neighbours` has been provided.

(a) Write a concurrent program to solve the region labelling problem. The program should use `p` worker processes (you may assume `N%p = 0`); each process should work on a strip of `height = N/p` rows. (Each process may use an `N` by `N` array to store the labels, but will only work on a part of the array.)

Each worker should communicate only with the workers for the strips above and below theirs. There should be no central controller. The worker processes should communicate by message passing only: they may use no shared variables except they may all read the array a. They may use a barrier synchronisation to determine if they can terminate.

You should give the definition of a worker, and state carefully what you assume about any channels you use, but do not need to give an explicit definition of how the system is constructed. [18]

(b) Now suppose a barrier synchronisation operation is not available. Describe how to adapt your code. You should give a precise description, but need not produce code. Again, each worker should communicate only with the workers for the strips above and below theirs, and there should be no central controller or shared variables. [7]

Answer to question 2

Part a. I'll give the full code, but only the definition of the workers and the combining barrier are required. The channels need to be buffered to avoid deadlock.

```
// Region labelling problem

import ox.CSO._

object RegionLabelling{
  val N = 12; // size of array
  val a = new Array[Array[Boolean]](N,N);

  // Initialise array
  def Init =
    for(i <- 0 until N; j <- 0 until N)
      a(i)(j) = (i%3!=1 && j%3!=1)
//    a(i)(j) =
//      (i!=0 && j%2!=0 || i==1 && j%4==0 || i==N-1 && j%4==2)

  val p = 4; // number of workers
  assert (N%p==0);
  val height = N/p; // height of one strip

  type Coord = Tuple2[Int,Int]; // Cartesian coordinates

  def smaller(p1:Coord, p2:Coord) : Boolean = {
    val i1=p1._1; val j1 = p1._2; val i2=p2._1; val j2 = p2._2;
    (i1<i2 || i1==i1 && j1<j2)
  }

  // find neighbours of p
  def neighbours(p:Coord) : List[Coord] = {
    val i=p._1; val j=p._2; var ns = Nil : List[Coord];
    if(i>0) ns = (i-1,j)::ns; if(i<N-1) ns = (i+1,j)::ns;
    if(j>0) ns = (i,j-1)::ns; if(j<N-1) ns = (i,j+1)::ns;
    ns;
  }
```

```
// Type of messages, one row of array
type Msg = Array[Coord]

// Combining barrier
def and(b1:Boolean,b2:Boolean) = b1 && b2
val barrier = new CombiningBarrier(p, true, and);

// A worker; the channels are to the workers dealing with
// the strips above or below this strip.
def Worker(me: Int, sendUp: ![Msg], sendDown: ![Msg],
           receiveUp: ?[Msg], receiveDown: ?[Msg])
= proc("Worker"+me){
  // This worker deals with rows [start .. end)
  val start = me*height; val end = start+height;

  val myLabels = new Array[Array[Coord]](N, N);
  for(i <- 0 until N; j <- 0 until N)
    if(a(i)(j)) myLabels(i)(j) = (i, j);

  var done = false;

  while(!done){
    var myDone = true;
    // Update coordinates
    for(i <- start until end; j <- 0 until N){
      if(a(i)(j)){
        for( (i1, j1) <- neighbours(i,j) )
          if(a(i1)(j1) &&
              smaller(myLabels(i1)(j1), myLabels(i)(j))){
            myLabels(i)(j) = myLabels(i1)(j1); myDone = false;
          }
      }
    }
    // Distribute top and bottom rows
    if(me!=0) sendUp! myLabels(start).toArray;
    if(me!=p-1) sendDown! myLabels(end-1).toArray
    // Receive new top and bottom rows
    if(me!=0) myLabels(start-1) = receiveDown?;
    if(me!=p-1) myLabels(end) = receiveUp?;
    done = barrier.sync(myDone);
  }

  // Print results in order of process number,
  // using down channels as signals
  if(me!=0) receiveDown?; // Wait for previous process
  // Print my rows
  for(i <- start until end){
    print(i+"\t");
    for(j <- 0 until N) print(myLabels(i)(j)+"\t");
    println;
  }
  if(me!=p-1) sendDown!myLabels(height); // Signal to next row
}

// Put it together
val ups = Buf[Msg](1,p); val downs = Buf[Msg](1,p);
  // indexed by senders
```

```
  def System =
    || ( for(i <- 0 until p) yield
         Worker(i, ups(i), downs(i),
                ups((i+1)%p), downs((i+p-1)%p))
       )

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

Part b. We piggy-back the booleans indicating whether processes can terminate onto the data. Define the type of messages by

```
type Msg = Tuple2[Array[Coord], Boolean];
```

Instead of the data-passing and global synchronisation:

- In order, starting with the worker for the top strip, the bottom rows are passed down;
- Onto this message is piggy-backed a boolean which is the conjunction of all the myDone variables the workers for higher strips; thus this final done value is accumulated in this boolean;
- Then, in order, starting from the worker for the bottom strip, the top rows are passed up, together with the final done value.

Here's the code, although it's not required

```
if(me==0){
  // Send my bottom row
  sendDown! (myLabels(end-1).toArray, myDone);
  val (newBelow, b2) = receiveUp?;  // Receive new row below
  myLabels(end) = newBelow; done = b2;
}
else if(me==p-1){
  val (newAbove, b1) = receiveDown?; // Receive new row above
  myLabels(start-1) = newAbove; done = b1 && myDone;
  sendUp! (myLabels(start).toArray, done); // Send my top row
}
else{
  val (newAbove, b1) = receiveDown?; // Receive new row above
  myLabels(start-1) = newAbove;
  // Send my bottom row
  sendDown! (myLabels(end-1).toArray, myDone && b1);
  val (newBelow, b2) = receiveUp?;  // Receive new row below
  myLabels(end) = newBelow; done = b2;
  sendUp! (myLabels(start).toArray, b2); // Send my top row
}
```

Question 3

Consider the following synchronisation problem. A savings account is shared by N people, each represented by a process. Each process can make deposits and/or withdrawals. The account records the current balance; this may never go negative. As a result, a withdrawal may be blocked until sufficient funds become available.

The aim of this question is to implement the account using the three different types of concurrency primitive: message passing, monitors and semaphores. In each part of the question you should use only the concurrency primitive mentioned.

Throughout this question, the system does not need to be fair as to which pending deposit is unblocked once sufficient funds become available.

(a) Implement the account using a server process with message passing. Requests from clients should be made using messages of type Request:

```
type ClientId = Int;
type Amount = Int;
abstract class Request;
case class Deposit(n:Amount, c:ClientId)
        extends Request;
case class Withdrawal(n:Amount, c:ClientId)
        extends Request;
```

Assume that the clients share a channel req on which requests are made. In response to requests, the server should send a response giving the current balance; assume each client c has its own response channel resp(c). Hence the signature of the server process is:

```
def Server(req: ?[Request], resp: Seq[![Int]])
= proc{ ... }
```

(You do not need to implement the clients.) [8]

(b) Now implement the account as a monitor with the following signature:

```
object Account{
  def Credit(amount:Amount, c:ClientId) : Int = ...
  def Debit(amount:Int, c: ClientId) : Int = ...
}
```

[6]

(c) Finally, implement the account using semaphores; you should use an object with the same signature as in the previous part.

Hint: you might find it useful to use an array of semaphores, one for each client process, on which it waits when a withdrawal is blocked. [11]

Answer to question 3

Part a.

```
def Server(req: ?[Request], resp: Seq[![Int]]) = proc{
  var balance = 0; // current balance

  val withdrawals = new Array[Int](5);
    // amounts of queued withdrawals; -1 represents nothing queued
  for(i <- 0 until N) withdrawals(i) = -1;

  repeat(
    req? match {
      case Deposit(n,c) => {
        balance += n;
        resp(c)!balance;
        // Try to unblock withdrawals
        for(i <- 0 until N)
          if(withdrawals(i)>0 && withdrawals(i)<=balance){
            balance -= withdrawals(i); withdrawals(i) = -1;
            resp(i)!balance;
          }
      }
      case Withdrawal(n,c) =>
        if(n<=balance){ balance -= n; resp(c)!balance; }
        else withdrawals(c) = n;
    }
  )
}
```

Part b.

```
object Account{
  private var balance = 0; // current balance

  def Credit(amount:Int, c: ClientId) : Int = synchronized{
    balance += amount;
    notifyAll();
    return balance;
  }

  def Debit(amount:Int, c: ClientId) : Int = synchronized{
    while(balance<amount) wait();
    balance -= amount;
    return balance;
  }
}
```

Part c.

```
object Account{
  private var balance = 0; // current balance
```

```
  // Semaphores and amounts for pending withdrawals
  val sems = new Array[Semaphore](N);
  val amounts = new Array[Int](N);
  for(i <- 0 until N){
    sems(i) = new Semaphore; sems(i).down; amounts(i) = -1;
  }

  val mutex = new Semaphore; // For mutual exclusion

  // Wake up a waiting process, or lift mutex
  private def Signal = {
    var n = 0;
    // Find process to wake up
    while(n<N && (amounts(n)<0 || amounts(n)>balance)) n += 1;
    if(n<N){ // wake up waiting debit
      amounts(n) = -1; sems(n).up;
    }
    else mutex.up;
  }

  def Credit(amount:Int, id:Int) : Int = {
    mutex.down;
    balance += amount; val result = balance;
    Signal;
    return result;
  }

  def Debit(amount:Int, id:Int) : Int = {
    mutex.down;
    var result = 0;
    if(balance<amount){
      amounts(id) = amount;
      mutex.up; sems(id).down; // wait to be woken
      assert(amount <= balance);
      // Can now proceed
      balance -= amount; result = balance;
      Signal;
    }
    else{ // Can proceed immediately
      balance -= amount; result = balance;
      mutex.up; // No need to signal
    }
    return result;
  }
}
```