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]

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]

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]