DEGREE OF MASTER OF SCIENCE

Computer Science

Concurrent Programming

HILARY TERM 2016

Your answers for this assignment are to be handed in at the Examination Schools, High Street, by **12:00 noon on Monday, 18th April, 2016**. *You must not discuss this assignment with anyone before that time.*

*Answer **all** questions*

All programs (and parts of programs) you write should be submitted both on paper and in computer-readable form (on a USB memory-stick, SD Card, CD, or DVD clearly identified with your candidate number) together with a script (or other means) of compiling and (where appropriate) executing them.

Program code that is included with the paper submission of your answers must be neatly laid-out and documented in such a way that the examiners can understand your reasons for believing that it is correct.

You may assume that processes are cheap to implement. Unless a question states otherwise your implementations may use any component or type provided by the CSO API, or the Scala collection API. You are advised to use the most recent version/revision of the standard CSO library; this is published on the course website along with its souce code and documentation.

Question 1

A `ZipSlot` is an implementation of the following trait:

```
trait ZipSlot[T,U] {
  def write0(v0: T)
  def write1(v1: U)
  def read: (T, U)
}
```

It is intended to implement communication between a pair of producer processes writing streams of data of (respectively) types `T` and `U`, and a consumer process, reading streams of pairs of type `(T,U)`.

If `s` is a `ZipSlot[T,U]` then the n^{th} call of `s.read` does not terminate until both the n^{th} calls of `s.write0` and `s.write1` have been made by their respective producers; and the termination of the producers' n^{th} write calls is synchronized with that of the consumer's n^{th} read call. Producers must not write to a slot before the previous writes have terminated.

If the n^{th} producer calls were `s.write0(`E_0`)` and `s.write1(`E_1`)` then the n^{th} `s.read` yields the pair (v_0, v_1) (where the v_i are the values of the E_i evaluated in their respective producer processes).

(a) Construct and explain a monitor-style implementation of a `ZipSlot`, and comment briefly on its efficiency. If your solution uses a CSO `Monitor` with more than one `Condition` then explain what benefit that brings.

You may assume that the producers' streams are infinite. (14 marks)

(b) Construct and explain a semaphore-style implementation of a `ZipSlot`, and comment briefly on its efficiency as compared with your solution to part (a).

You may assume that the producers' streams are infinite. (14 marks)

(c) Construct a simple test-rig suitable for demonstrating appropriate behaviour of your slot implementations, and explain what inappropriate behaviours it is intended to catch. Demonstrate its operation on your implementations. (5 marks)

(d) In this part of the question the assumption that both producers' streams are infinite will no longer be valid.

Either producer may signal termination of the sequence it is sending, by writing an appropriately typed `null` value: for example `s.write0(null.asInstanceOf[T])`. In this case we say that the slot has been *closed*, and an invocation of `read` on a channel that has (just) closed should throw the `Stopped` exception by invoking the CSO method `stop`. A producer attempting to write to a closed slot should be stopped – by throwing the same exception, as should one whose "partner" in the slot has just closed it.

Modify one or both of your slot implementations to exhibit this behaviour, and modify your test rig accordingly if necessary. (7 marks)

Question 2

A *boid* is a simulation of a bird-like object. The aim of this question is to simulate and visualise, by a concurrent program, a collection of boids moving over a 2-dimensional toroidal surface. A snapshot of a visualisation is shown in Figure 1.

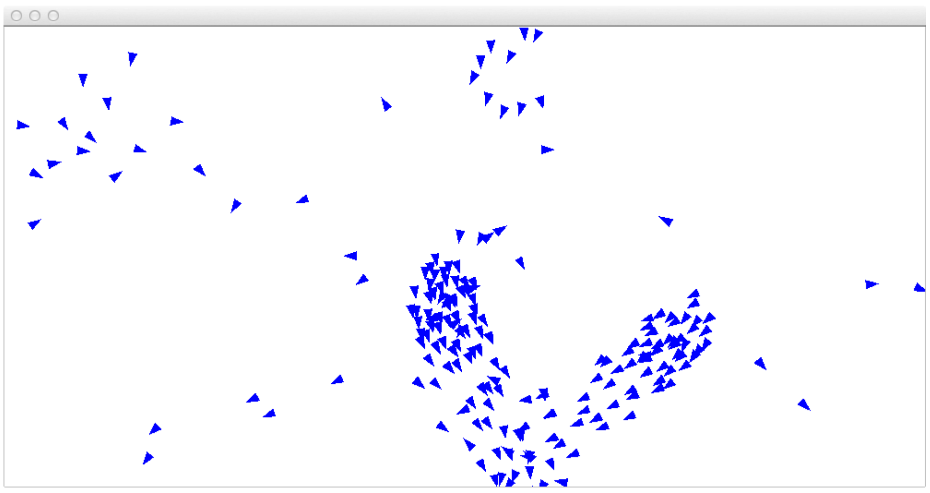

Figure 1: Snapshot of a visualised simulation of a collection of Boids

Boids interact according to certain simple rules, such as:

- ◇ separation: boids steer to avoid being crowded by other boids;
- ◇ cohesion: boids steer to move toward the average position of other nearby boids;
- ◇ alignment: boids steer towards the average direction of other nearby boids.

Here, "nearby boids" means those other boids within this boid's field of vision (a sector of a circle defined by an angle and a radius). These rules lead to emergent behaviour, such as flocking. There are several articles on the Web about boids, and you might like to read a couple (although doing so will not be necessary for you to be able to complete this question).

Most of the code you need is provided for you in the three files described below, and available via the course website.

- ◇ `Boid.scala` defines a *class* `Boid`, each object of which represents a single boid; it also defines the *object* `BoidParams` that, specifies (default) parameters of the whole simulation such as cohesiveness, maximum speed, drag, *etc.*

  A boid's current state is defined by its position variables (`x` and `y`), and its velocity variables (`speed` and `theta`). Its most important procedure is `newState`: this takes an array containing all the boids in the system, and returns the next state of *this* boid as the tuple `(x, y, speed, theta)`. There are also procedures to initialise the boid randomly, to set its state to a new value (also specified by the tuple `(x, y, speed, theta)`), and to return its current state.

- ◇ `Display.scala` defines a class that implements the display of the flock and is initialised with an iterable collection that contains all the boids in the flock. The procedure `draw` redraws the display.[1]
- ◇ `BBox.scala` defines the dimensions of the surface that the boids move over.

You can complete this question without changing any of this code; if you do change the code, you should explain your changes.

Your task is to implement two concurrent programs to perform a discrete-time simulation of a flock of boids. At each step, each boid's state should be updated depending on the state of all the other boids at the start of that step, as defined by `newState`. You should use one process for each boid; you are likely to use a distinct process to update the display at appropriate times.

(a) Write a concurrent program to implement the simulation, using a shared array that stores the state of the boids. The array should be accessed in a disciplined way, avoiding race conditions. (15 marks)

(b) Now write a concurrent program to implement the simulation that uses no shared variables, where the states of boids are passed over channels. Comment briefly on its efficiency relative to that of your answer to part (a). (15 marks)

(c) Recall that the simulation parameters defined in the `BoidParams` object are used by `newState` in each boid's computation of its state in the next frame of the simulation. Suggest how you could arrange to set the values of these parameters interactively during the evolution of a simulation, and discuss any potential pitfalls and their remedies. Provide an implementation that works with one or both of your simulations. (10 marks)

[1]To be precise, it arranges for the Java GUI thread to redraw the flock, and synchronizes with that thread so that it returns only when the flock has been redrawn. This detail should have no consequence for the rest of the simulation.

Question 3

Consider the following administrative problem: students, professors, and teaching assistants are modelled by processes, each of which has its own distinct identity. A class takes place in a classroom only when a *quorum* of s students, p professors, and t teaching assistants have shown up; and when they have done so, each participant in the class discovers the identitities of all the others. Participants may leave the classroom at any time after a class has started, and when no participant remains in the room, it is free to become the venue for another class. Anybody who shows up at a classroom in which a class is not already taking place must wait for a quorum to be formed; and anybody who shows up when a class has already started must wait until it finishes, and then wait for a quorum to be formed.

Design and test an implementation of the following abstract class suitable for solving the synchronization problem(s) inherent in administering a classroom.

```
abstract class Classroom(s: Int, p: Int, t: Int)
{ def S(id: Int): List[Int]
  def P(id: Int): List[Int]
  def T(id: Int): List[Int]
  def Leave(id: Int): Unit
}
```

Your implementation can be in the semaphore/flag style, or the channel style[2]; but you should not mix styles. (20 marks)

[2]Using an internally forked server process.