EEEN30052 Concurrent Systems

Coursework Assignment

Part 2: Thread Synchronisation

Dr. Peter Green
Department of Electrical and Electronic Engineering
University of Manchester
Office: Sackville E8b
peter.green@manchester.ac.uk

1. Introduction

In this part of the assignment, the focus is on synchronisation – specifically designing and implementing synchronisation objects for the start of the race and for the baton exchange. The termination of the race is considered in the third and final part of the assignmemnt.

2. Track Layout

At the start of a relay race, all athletes must take up their correct positions and wait, either for the start of the race, or for their preceding team member to arrive at the exchange zone. In terms of the elements of the `theThreads` array:

- `theThreads[0][j], j` ∊ {0, 3} must be at the start
- `theThreads[1][j], j` ∊ {0, 3} must be at exchange zone 0
- `theThreads[2][j], j` ∊ {0, 3} must be at exchange zone 1
- `theThreads[3][j], j` ∊ {0, 3} must be at exchange zone 2

See Figure 1[1].

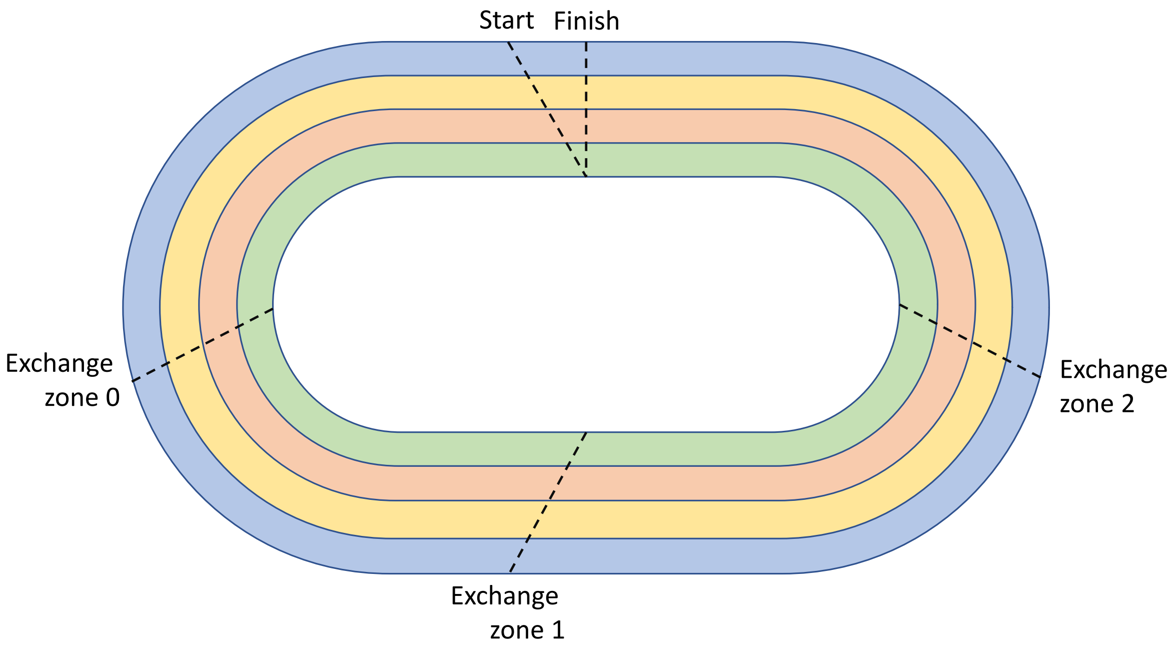

Figure 1 Track, start, finish, exchange zones

3. Synchronisation

Thread synchronisation is needed to implement the rules of a relay race in this simulation.

3.1 Start

All athletes must be in position before the race can start. In concurrent programming terms this means that all the threads must have started to execute but have become blocked in a synchronisation agent (object) before the simulated race can begin. Once this has happened,

[1] This is a simplification.

the race can start and theThreads[0][j], j ∈ {0, 3} are unblocked. They then enter a random time delay as discussed in the lab script for part 1.

3.2 Baton Exchange

At the end of the time delays of theThreads[0][j], j ∈ {0, 3}, each thread must unblock the next member of its team that is blocked at exchange zone 0 i.e.

- theThreads[0][j] must individually unblock the corresponding theThreads[1][j], j ∈ {0, 3}

The unblocked threads then enter their own time delay representing the running of their leg of the race. theThreads[0][j], j ∈ {0, 3} all then terminate after recording the time that it took to run their leg of the race i.e., their time delay. This should be stored in the appropriate in the ThreadMap, and will require a modification to the Competitor class.

The above protocol is executed by theThreads[1][j], j ∈ {0, 3} unblock theThreads[2][j], j ∈ {0, 3} that are waiting at exchange zone 1.

This is repeated at exchange zone 2, where theThreads[2][j], j ∈ {0, 3} unblock theThreads[3][j], j ∈ {0, 3}.

4. Coding

C++ mutexes, locks and condition variables should be used to provide the appropriate synchronisation described above.

The main task in this part of the assignment is to create synchronisation agents and to do this an inheritance hierarchy is defined. The base class is SyncAgent:

```
class SyncAgent {  //abstract base class
public:
   SyncAgent() {} //constructor
   // Declare  virtual methods to be overridden by derived classes
   virtual void pause() = 0;
   virtual void proceed() = 0;
}; //end abstract class SyncAgent
```

This is an **abstract** base class, as discussed in Microcontroller Engineering 2. In general, an abstract base class has at least one method (member function) that is declared to be **virtual**. In this class both member functions are virtual. The purpose of this abstract base class is to provide a framework for SyncAgents that are similar, but NOT the same.

The SyncAgent required at the start of the race has the same methods as the exchange zone SyncAgent, but the details of the pause and proceed functions in these two cases are different. The 'start' SyncAgent blocks several threads and releases them. The 'exchange zone' SyncAgents simply block and release a single thread.

4.1 `StartAgent`: Start SyncAgent

The skeleton of this class is as follows:

```
class StartAgent : public SyncAgent {  //concrete class that CAN be instantiated
public:
   StartAgent() {} //constructor
   void pause() {
      ...// insert code to implement pausing of all athlete threads
   }
   void proceed() {
      ...// insert code to implement releasing of all athlete threads
   }
private:
   // insert any necessary data members including variables, mutexes, locks, cond vars
}; //end class StartAgent
```

This is the most complex agent as it must implement the rules associated with the race starting.

- The athletes must all move into their allotted positions[2].
- The starter of the race must wait for all athletes to be in position.
- The starter then can start the race.

You are advised to implement this class in two stages:

(i) Code it in such a way that the race starts immediately the last athlete arrives in position.
(ii) Enhance the solution so that there is a random time delay between the last athlete getting into position and the race starting.

4.2 `EZAgent`: Exchange Zone SyncAgent

```
class EZAgent : public SyncAgent {  //concrete class that CAN be instantiated
public:
   EZAgent() {} //constructor
   void pause() {
      ...// insert code to implement pausing of next runner thread
   }
   void proceed() {
      ...// insert code to implement releasing of next runner thread
   }
private:
   // insert any necessary data members including variables, mutexes, locks, cond vars
}; //end class EZAgent
```

4.3 Athletes in Position

This has been mentioned earlier. Here it means that each athlete thread (i.e., each of theThreads) becomes blocked in the appropriate SyncAgent Object before the race starts. What SyncAgent objects are needed?

- One `StartAgent` that blocks `theThreads[0][j], j ∈ {0, 3}`

[2] The idea of the athletes moving into position in the simulation will be discussed below.

- One EZAgent for each team at each exchange zone. Hence 12 EZAgent objects are required. These are best organised as an array.
  ```
  const int NO_TEAMS = 4;            // number of teams in the race
  const int NO_MEMBERS = 4;          // number of athletes in the team
  const int NO_TEAM_EXCHANGES = 3;          // number of baton exchanges
  per team
  ...
  EZAgent exchanges[NO_TEAMS][ NO_TEAM_EXCHANGES];
  ```
- exhanges[m][n] blocks theThread[m][n+1], which is released by theThread[m][n]. See Figure 1.

Based on the above, each thread requires a SyncAgent. These should be created in main and passed to each run function by reference. This will require changes to run's prototype as follows:

```
void run(Competitor& c, SyncAgent& s);
```

One of the key advantages of using inheritance is that an object of any class derived from SyncAgent (in this case StartAgent and EZAgent) can be passed to run via parameter s.

5. Malpractice

There are well known solutions to some aspects of the synchronisation problems that feature in this part of the assignment. However, YOU ARE EXPECTED TO WRITE YOUR OWN CODE ON YOUR OWN and to follow the code structure outlined above. Therefore, each line of your code for this part of the assignment should be carefully commented, explaining its role in the synchronisation protocols. Omitted or inadequate comments will result in a substantial loss of marks, as will block copying from any published source. The unit leader's decision is final in the event of a dispute.