LAST MINUTE TICKET - SYNCHRONIZATION

1 OBJECTIVES

The main goals of this assignment are:

- Studying race condition in absence of synchronization.
- Identifying critical sections for Mutual Exclusion (ME)
- Using locks for Mutual Exclusion in critical sections.

2 DESCRIPTION

There is an internet site that sell last minutes tickets to a very popular football game. The number of available tickets is very limited so as soon the site publishes its booking service, several clients are racing to get one or more of the available tickets. In this assignment we are going to simulate this scenario by creating a desktop application. The following assumptions are made.

- There are ten seat of which five are advertised as available seats and these are selected randomly.. You may use the following constants:

```
private const int maxSeats = 10;
private const int availableSeats = 5;
private const int maxClients = 15;
```

  The key word `const` in C# corresponds to Java's keyword `final`.

  Every time the program is re-run, five new seats are to be selected by using a Random object. For simplicity, seats can be numbered from 0 to maxSeats-1 or from 1 to maxSeats.

  Note: You can use a loop to get the five arbitrary seats, but it can happen that the random object generates the same seat more than once. Make sure to avoid duplicates.

- All of the clients are interested in purchasing a ticket. They are looking and reserving a ticket at the same time without any knowledge of each other. To begin with, every client is assumed to buy a **single** ticket. As an optional part of the this assignment (Part 3 , a client may reserve more than one seats.

Create an application that simulates clients as threads and the available seats as shared resources. The program needs no input from the user.

- Assume that each seat has an id, a status (available or occupied) and a client name if the seat is occupied.

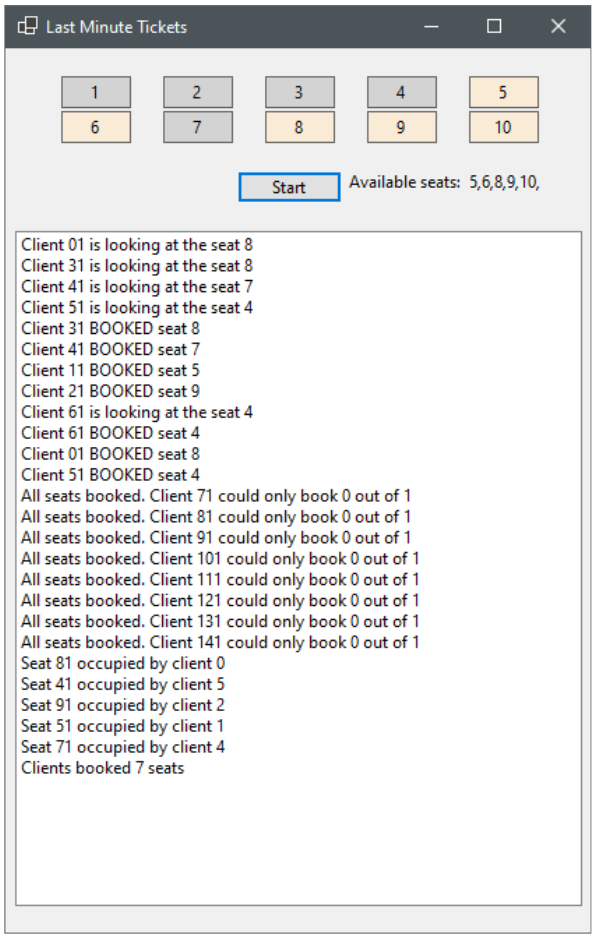
- A client has an id, an integer keeping track of how many seats the client has booked. A client may have booked more than one seats when no thread synchronization is implemented.

The program can be coded with a GUI or as a Console/Terminal text-based application. The image below shows an example of a GUI-based version. You may design your own user interface with or without graphic components.

Each seat has an attribute storing its status as "available" or "occupied". To begin with, all seats are in the "available" status. When a client is lucky to book a seat, the status of the seat is changed to occupied. A seat can only be booked if its status is "available". Despite this, it is fully possible to experience race condition when running the application.

3 Requirements for a Passing Grade

Part 1 and Part 2 are both mandatory. Part 3 is optional (not mandatory).



3.1 The image shown at the right side is a Windows Form object created in Visual Studio. The buttons are created dynamically. However, it is **NOT** mandatorty (but recommended) to use a GUI-application. You can provide the same features and functionalities using a Console/Terminal window.

3.2 In all parts, it is strictly forbidden to write messages to the user (using System.out in Java or Console.WriteLine or .Write in C#) from other classes than the user interface. All input/output operations are to be carried out from the user-interface object. As an example, the contents of the listbox in the figure here, comes from an array hosted inside the **SeatManagerShared** class as an instance variable. The information is printed from the Main thread after threads are started.

This array is the instance variable logbook in the code snippet below, showing the attributes declared in the SeatmanagerShared class.

```
class SeatManagerShared
{
   private List<Seat> seats;
   private Random random;
   private List<string> logBook = new List<string>();
```

Whenever you want to note a piece information in the SeatManagerShared class, or in the Client-class, or in the UI-class, you can add the information as a string to your **logBook** object.

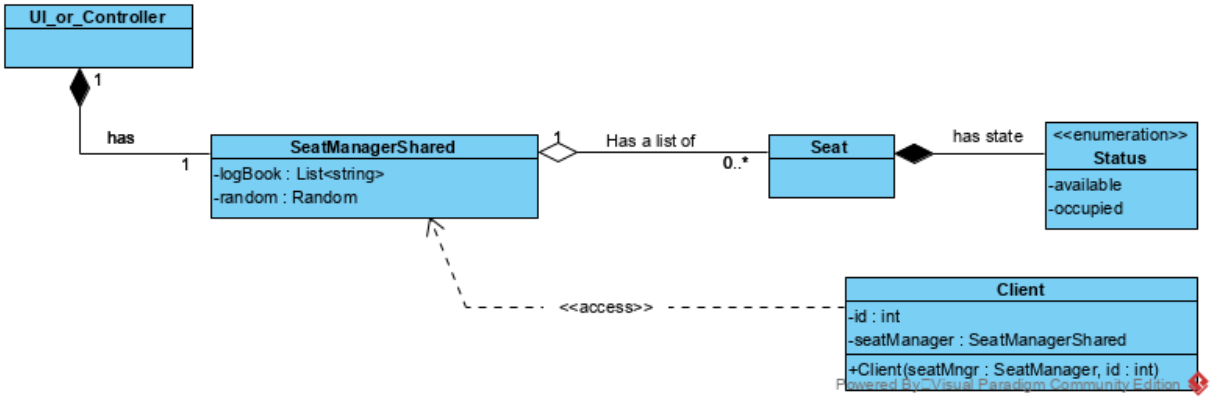
```
public String printStatus()
{
    String result = "";

    foreach (Seat seat in seats)
    {
        result = seat.toString();
        logBook.Add(result);

    }
    return result;
}
```

3.3 Write a ToString-method in the Seat class with data about the seat, e.g. id, its status (available or occupied and the client name. You can then use a loop to update the info about all available seats. The code is only an example. The method can be called by the UI-class as in the above listbox.

In each part there are a number of questions that you should be able to explain orally to your supervisor. You don't have to submit the answers in writing.

4 Part 1 – Race Condition

4.1 Write the classes that you find necessary to define a seat, a client and the shared resources. The class diagram shown below should give you an idea.



4.2 Run and test the application without any synchronization. As there are many more clients (threads) than available tickets (resources), race condition should happen.

You may have to run the program a number of times to experience race condition. It depends on how the threads are scheduled by CPU and the speed of your CPU, as well as how many other threads and processes are in action on your computer. One way to hasten the occurrence of a race condition is to use busy spinning, with a long-going loop. This is actually a match to reality as it normally takes a client a period of time to make a decision. You can first try testing your application without the loop!

```
//randomize a waiting time for every client.
int max = random.Next(1000000);
for (i = 0; i < max; i++)  //busy spin
{
    ;//do nothing
}
```

4.3 Use no sleep with any thread and no synchronization in this part.

4.4 Make sure to understand and see for yourself that the application encounters race condition and that the outcome is not correct. You should be able to answer the following questions:

- *Q1. How can you see the occurrence of a race condition when testing your program?*
- *Q2: Why can the same seat be booked by two or more threads although each seat only can be booked if it is not occupied?*

5 PART 2 - SYNCHRONIZATION

In this part, you are to solve the problem of race condition by implementing mutual exclusion (ME). Proceed as follows:

5.1 Identify the critical section(s) in your classes.

5.2 Use `lock` in **C#** and `synchronized` in Java to provide mutual exclusion. Apply synchronization only around the CS and use a local object as lock object.

```
private Object lockObj = new Object();
lock (lockObj) //comment out the line to test race condition
{
        //code
}
```

Java: Replace lock in the above code with synchronized and the rest is the same.

You should be able to answer the following questions:

Q3: How did you proceed to identify CS?

Q4: Does race condition can still happen?

Q5: Can startvation happen in your code?

6 PART 3 – DEADLOCK (NOT MANDATORY)

In order to see if you're your code can face the problem deadlock, you can let the clients to buy two seats. A certain client can be holding a lock to one seat and waiting for another seat to be released. In order to simulate this situation, you can use the random object to pick one or two clients wanting to purchase two (or more) seats.

Q6: Have you been able to experience a deadlock situation in your code? If yes, in which part of your code.

Q7: Does ME play a role in this case?

Q8: What would be a simple solution to avoid deadlock?

7 GRADING AND SUBMISSION

Compress all the files, folders and subfolders into a **zip**, **rar, 7z** file, and then upload it via the Assignment page on Canvas. Show your work to the teachers on the assignment-lab sessions.

The assignment is to be presented to the supervisor personally during the lab-sessions.

8 SOME HELP AND GUIDANCE

The following paragraphs are given only as example to let you get an idea. You don't have to follow the given solutions. You can use your own design and algorithms to solve the assignment.

8.1 **C#:** To update a control in the MainForm from another class, such as CharacterBuffer class, you need to proceed as follows:

a. Write the following line outside the class definition:

```
// Delegate for invoking main thread
private delegate void DisplayDelegate(string inpStr, ListBox lb);
```

b. When you want to send a text to a ListBox:

```
lstRead.Invoke(new DisplayDelegate(DisplayString), new object[] {
"Reading " + ch, lstRead });
```

c. Copy the following method in the CharacterBuffer class:

```
private void DisplayString(string s, ListBox lb)
 {
     lb.Items.Add(s);
 }
```

8.2 **The Client class:**

This class is in fact a thread in Java and in C# it should have method that is to be run by the thread created in the MainForm (or any other class where you create your client-threads).

```
class Client
{
    private int id;
    private SeatManagerShared seatManager;
    private int numberToBook;  //set to one for part 1 of the assignment
    private int numberBooked;
    private Random random = new Random();

    1 reference
    public Client(SeatManagerShared seatManager, int id)
    {
        this.seatManager = seatManager;
        this.id = id;
        numberToBook = 1;
    }
```

The field **numberBooked** is the number of seats that the client has booked (should be one in Part 1).

Good Luck!

Farid Naisan,
Course Responsible and Instructor