Competitive and Multithreaded Programming

Lab1. Writer/Reader & Sleeping Barber

TYTGAT Karel

1. Goal

The objective of this laboratory was to write, a writer, reader algorithm and to implement the sleeping barber problem.

2. Writer and reader problem

a. Definition

The writer, reader problem is a typical problem in parallel programming. The goal is to create 2 entities, one writes into a file, and the other access theses written data. The multi access to a file at the same instant is the main difficulty of this problem because it needs to avoid accessing a file at the same instant.

b. Theory and implementation

The writer, reader problem can be implemented is many different ways. one of the best way is to use a pipe. A pipe is an OS concept, it is a one way communicating system in which every time the writer write data into the pipe, the reader will be able to read them and consume them at the same time. However, the pipe has a really limited space, that way it is also possible to use a temp file system to simulate this process.

In this implementation, I wanted to use a file pointer with which one would be in writing only and the other in reading only. Sadly, several problems appeared. The first one was that the reader thread needed to have access to the file in writing to be able to consume the data upon reading. Prior to that, the object used by C# to manage the file did not update the stream on change. So in this program, it is necessary to close the file stream every time the reader or the writer have to modify the file.

The program uses two threads. Each thread is respectively the writer and the reader.
To synchronise the process, we use semaphore. Semaphore is locks that can prevent another thread from executing to write data. The use of a semaphore create a critical section to be able to check and change the value of the semaphore in an atomic action (uninterruptible action). To implement the writer/reader problem, only 2 semaphore are needed, one for each thread.

Writer/Reader problem implementation :

```
public static Semaphore semReader;
public static Semaphore semWriter;
public static int maxData = 10;

static void Main(string[] args) {
```

```
 /*
  * Note :
  *  Due to probably the memory managment of c#, I didn't succeded in reading and
writing in the same stream.
  *  The buffer of the writing stream was never updated so the reader thread couldn't
read any data.
  *  To fix that, I have to open and close the file inside each thread instead of
opening the writer/reader stream only at the begining of each one.
  */

 semReader = new Semaphore(0, 1);
 semWriter = new Semaphore(1, 1);

 Thread reader = new Thread(Reader);
 Thread writer = new Thread(Writer);
 reader.Start( /*file*/ );
 writer.Start( /*file*/ );
 writer.Join();
 reader.Join();
 //file.Close();
}

public static void Writer(object fileStream) {
 for (int i = 0; i < maxData; i++) {
  semWriter.WaitOne();
  StreamWriter file = new StreamWriter(File.Open("./readerWriterFile.txt",
FileMode.OpenOrCreate));
  file.Write((char)(i + 65));
  Console.WriteLine("Writing");
  file.Flush();
  Thread.Sleep(1000);
  file.Close();
  semReader.Release();
 }
}

public static void Reader(object fileStream) {
 char[] buffer = new char[1];
 for (int i = 0; i < maxData; i++) {
  semReader.WaitOne();
  StreamReader file = new StreamReader(File.Open("./readerWriterFile.txt",
FileMode.OpenOrCreate));
  file.Read(buffer, 0, 1);
  Console.WriteLine("Reading : " + buffer[0]);
  //Thread.Sleep(1000);
  file.Close();

  semWriter.Release();
 }
}
```

3. Sleeping barber problem

a. Definition

"*In computer science, the sleeping barber problem is a classic inter-process communication and synchronization problem between multiple operating system*

processes. The problem is analogous to that of keeping a barber working when there are customers, resting when there are none, and doing so in an orderly manner."

The main difficulty of the sleeping barber problem is how to synchronize the different processes/threads making the barber shop run properly.

b. Theory and implementation

The main difficulty in implementing the sleeping barber problem is to synchronize all of the process to make them work together.

The problem can be implemented with 3 semaphore, one for the barber, one for the clients and one for the waiting room. Each semaphore has a maximum value defined by the number of seats in the waiting room, 6 in this implementation, plus 2 for the client and the barber semaphore. The waiting room semaphores represent the number of free seats in the waiting room. If this semaphore falls down to 0, it means that the waiting room is full. The client and the barber have a maximum of plus 2 compare to the waiting room to manage the overflow of client. One is for the barber working seat and the other one is for the last client, which enter with no places available.

The program is then divided in two different behaviors.

One thread is created at the beginning of the program. This thread represents the barber behavior. Upon creation, the barber will wait for a client to come into the shop. This mean that it will wait for the waiting room and the client semaphore to be unlock. When a client enters, the thread will consume the semaphore and start working. The working process takes 5 seconds to be completed. Once completed, the barber will restart its process and wait for the semaphore of the client and the waiting room to be unlocked.

Barber thread implementation :

```
public void BarberRoutine() {
 while (!close) {
  semClient.WaitOne();
  semWaitingRoom.WaitOne();
  room.freeSeat();

  semBarber.Release();
  semWaitingRoom.Release();

  /*Working*/
  barberChair.freeChair();
  workingChair.takeChair();
  for (int i = 0; i < 100; i++) {
   SetProgressBar(i);
   //progressBarBarber.Value = i;
   Thread.Sleep(50);
  }
  SetProgressBar(0);
  SetLabelClient("Client left with an haircut ! ");
  barberChair.takeChair();
```

```
  workingChair.freeChair();
 }
}
```

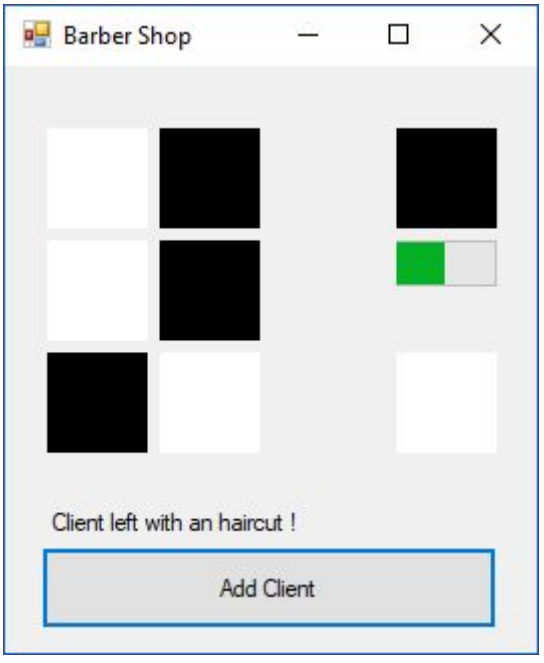
The second behavior is a client. When a client is created it will wait for the waiting room semaphore. If a place is available, the client will take a seat and be added into the waiting list of the barber. It will then wait until the barber consume the semaphore value. If no places are available, the client will leave without a haircut and release the waiting room semaphore

Client thread implementation :

```
public void clientRoutine() {
 SetLabelClient("New Client entered");
 semWaitingRoom.WaitOne();

 if (room.hasFreeSeat()) {
  room.takeSeat();
  semClient.Release();
  semWaitingRoom.Release();
  semBarber.WaitOne();
 } else {
  semWaitingRoom.Release();
  SetLabelClient("Client left without an haircut");
 }
}
```

The program was implemented in C# with an interactive interface where each square represents a chair in the barber shop. The square on the bottom right is the barber chair. The square on the left represent the waiting room. If the square is black, it means that the chair is occupied. The progress bar represents the process of getting a haircut by a client.



A client can be added by clicking on the “Add Client” button. Id the waiting room is full, no client will be added and a message will be displayed : “ Client left without an haircut ”

The full program is composed of the main program, which manage the user interfaces and the multithreading part, and 3 classes which help manage the interface display :

- **Chair** : manage the different seats display if taken or not
- **WaitingRoom** : manage the fullness of the waiting room and keep the number free seats

Program initialization :

```
public formBarberShop() {
 InitializeComponent();

 /*Waiting Room*/
 int nbOfSeat = 0;
 List < Chair > chairs = new List < Chair > ();
 Panel wR = (Panel) panelWaitingRoom;
 foreach(Control control in wR.Controls) {
  if (control.GetType() == typeof(PictureBox)) {
   nbOfSeat++;
   chairs.Add(new Chair((PictureBox) control));
  }
 }
 room = new WaitingRoom(chairs, nbOfSeat);

 /*Barber Seat*/
 barberChair = new Chair(pbBarberChair);
 barberChair.takeChair(); // On his seat at the begining

 /*Working Seat*/
 workingChair = new Chair(pbWorkingChair);

 /* Semaphore */
 semBarber = new Semaphore(0, nbOfSeat + 2);
 semWaitingRoom = new Semaphore(nbOfSeat, nbOfSeat);
 semClient = new Semaphore(0, nbOfSeat + 2);

 barber = new Thread(BarberRoutine);
 barber.Start();
}
```

Chair class implementation :

```
public class Chair {
 PictureBox chair;
 bool tacken;

 public Chair(PictureBox pb) {
  this.chair = pb;
  chair.BackColor = Color.White;
  tacken = false;
 }

 public bool takeChair() {
  if (tacken) {
   return false;
```

```
  }
  tacken = true;
  chair.BackColor = Color.Black;
  chair.Invalidate();
  return tacken;
 }

 public bool freeChair() {
  if (!tacken) {
   return false;
  }
  chair.BackColor = Color.White;
  chair.Invalidate();
  tacken = false;
  return true;
 }
}
```

WaitingRoom implementation :

```
public class WaitingRoom {
 List < Chair > seats;
 private int nbFreeSeat = 0;
 public int maxSeats = 0;

 public WaitingRoom(List < Chair > seats, int maxSeats) {
  this.seats = seats;
  this.maxSeats = maxSeats;
  nbFreeSeat = maxSeats;
 }

 public bool hasFreeSeat() {
  return nbFreeSeat > 0;
 }

 public bool takeSeat() {
  if (hasFreeSeat()) {
   foreach(Chair ch in seats) {
    if (ch.takeChair()) {
     nbFreeSeat--;
     return true;
    }
   }
  }
  return false;
 }

 public bool freeSeat() {
  foreach(Chair ch in seats) {
   if (ch.freeChair()) {
    nbFreeSeat++;
    return true;
   }
  }
  return false;
 }
}
```