VPS 5	Parallel and Distributed Software Systems	ST 16, Exercise 2

Deadline: 15th of April 2016 17:00

Name _____

Points _____ **Effort in hours** _____

1. Race Conditions (3 + 1 + 3 Points)

a) What are *race conditions*? Implement a simple .NET application in C# that has a race condition. Document the race condition with appropriate test runs.

b) What can be done to avoid race conditions? Improve your program from 1.a) so that the race condition is eliminated. Document your solution with some test runs again.

c) Where is the race condition in the following code? How can the race condition be removed?

```
class RaceConditionExample {
  private const int N = 1000;
  private const int BUFFER_SIZE = 10;

  private double[] buffer;

  private AutoResetEvent signal;
  public void Run() {
    buffer = new double[BUFFER_SIZE];
    signal = new AutoResetEvent(false);

    // start threads
    var t1 = new Thread(Reader); var t2 = new Thread(Writer);
    t1.Start(); t2.Start();

    // wait
    t1.Join(); t2.Join();
  }

  void Reader() {
    var readerIndex = 0;
    for (int i = 0; i < N; i++) {
      signal.WaitOne();
      Console.WriteLine(buffer[readerIndex]);
      readerIndex = (readerIndex + 1) % BUFFER_SIZE;
    }
  }

  void Writer() {
    var writerIndex = 0;
    for (int i = 0; i < N; i++) {
      buffer[writerIndex] = (double)i;
      signal.Set();
      writerIndex = (writerIndex + 1) % BUFFER_SIZE;
    }
  }
}
```

2. Synchronization Primitives (2 + 2 + 1 Points)

a) The following code starts multiple threads to download multiple files in parallel. Change the code so that only maximally ten files are downloaded concurrently.

```
class LimitedConnectionsExample {
  public void DownloadFilesAsync(IEnumerable<string> urls) {
    foreach(var url in urls) {
      Thread t = new Thread(DownloadFile);
      t.Start(url);
    }
  }

  public void DownloadFile(object url) {
    // download and store file here
    // ...
  }
}
```

b) Based on your version of the code in 2a) implement the synchronous method *DownloadFiles* that waits until all downloads are finished before returning.

c) In the following code one thread waits for the result of another thread in a polling loop. Improve the code fragment to remove the polling.

```
class PollingExample {
    private const int MAX_RESULTS = 10;
    private volatile string[] results;
    private volatile int resultsFinished;
    private object resultsLocker = new object();

    public void Run() {
      results = new string[MAX_RESULTS];
      resultsFinished = 0;

      // start tasks
      for (int i = 0; i < MAX_RESULTS; i++) {
        var t = new Task((s) => {
            int _i = (int)s;
            string m = Magic(_i);
            results[_i] = m;
            lock(resultsLocker) {
              resultsFinished++;
            }
          }, i);
        t.Start();
      }

      // wait for results
      while (resultsFinished < MAX_RESULTS) { Thread.Sleep(10); }

      // output results
      for (int i = 0; i < MAX_RESULTS; i++)
        Console.WriteLine(results[i]);
    }
  }
```

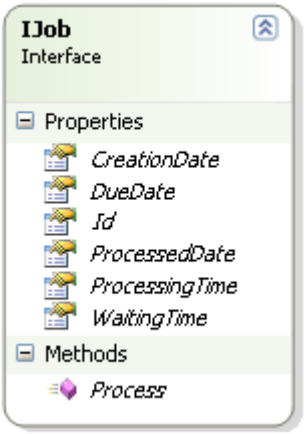
3. Toilet Simulation (4 + 4 + 4 Points)

Especially for simulation applications concurrent programming is very important, as real life is normally not sequential at all. So in order to simulate a realistic scenario as good as possible, parallel concepts are needed.

In this task you should implement a queue which handles jobs waiting to be processed (producer-consumer problem). In order to get the example a little bit more "naturalistic", imagine that the jobs are people waiting in front of a toilet (consumer).

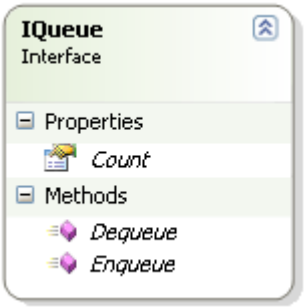
On Moodle you find a simple framework which already provides some parts of the simulation:

The interface *IJob* defines the data relevant for every job (id, creation date, due date, processing time, waiting time, time when the job was finally processed). It also has a method *Process* which is called by the consumer to process the job.

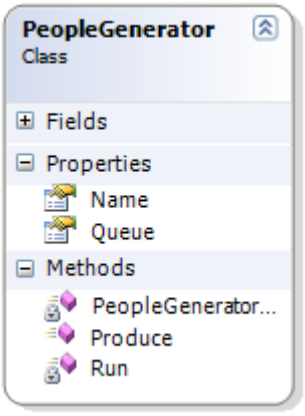


The class *Person* implements IJob. In the constructor of Person the time period available for processing is choosen randomly (normally distributed). Based on that time period the due date (*DueDate*) is set. Additionally the processing time (*ProcessingTime*) is also randomly set (normally distributed).

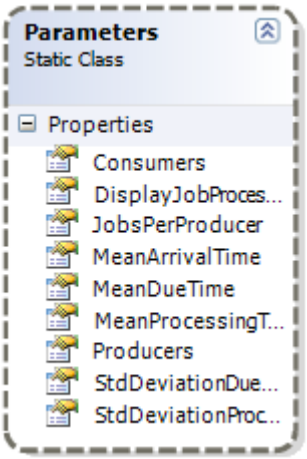
The interface *IQueue* defines the relevant methods for a queue which are used by the producer to enqueue jobs (*Enqueue*) and by the consumer to dequeue jobs (*Dequeue*).



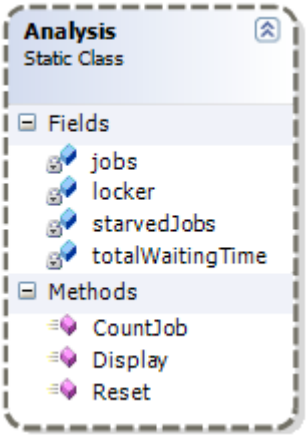
The producer *PeopleGenerator* uses a separate thread to create new jobs (instances of Person) and to enqueue them in the queue. The time between the creation of two Person objects is exponentially distributed (Poisson process).



The class *Parameters* contains all relevant parameters configuring the simulation. Especially, there is the number of producers and consumers, the number of jobs to generate per producer and the mean value and standard deviation of the arrival time, the due time and the processing time.



Analysis is used to analyze the job management in a queue. After a job is processed the job is counted by calling *CountJob*. The results of the analysis can be displayed with *Display* giving the total number of jobs, the number of "starved" jobs, the starvation ratio and the total and average waiting time.



The classes *NormalRandom* and *ExponentialRandom* are helper classes to create normally and exponentially distributed random variables.

ToiletSimulation contains the main method which is creating all required objects (producers, consumers, queue), starting the simulation and displaying the results.

a) Implement a simple consumer *Toilet* which is dequeuing and processing jobs from the queue in an own thread. Especially think about when the consumer should terminate. How can the synchronization be done?

b) Implement a first-in-first-out queue *FIFOQueue* and test it with the following parameter settings:

Producers	2
JobsPerProducer	200
Consumers	2
MeanArrivalTime	100
MeanDueTime	500
StdDeviationDueTime	150
MeanProcessingTime	100
StdDeviationProcessingTime	25

Execute some independent test runs and besides the individual results also document the mean value and the standard deviation.

c) As you can see from 2.b), the performance of FIFOQueue is not that good. "Starvation" occurs quite regularly, in other words many jobs are not processed in time. And what that means according to our simulation scenario … well you might know ;-).

Develop a better queue (*ToiletQueue*) which has a better performance according to the total number of starved jobs. Which strategy could be used to choose the next job from the queue that should be processed?

Repeat the test runs you have done in 2.b) for the improved queue and compare.

Note: Upload your report which contains all documentation and all changed or new source code of your program to Moodle.

Don't forget to give meaningful solution descriptions, so that one can easily get the main idea of your approach.

If necessary, you are allowed to extend or change the given classes. If you do so, please motivate and document such changes clearly in the solution description.

1 Race Conditions

1a. Was sind race conditions?

Race conditions können entstehen, wenn mehrere Threads paralell auf den selben Speicherbereich (Variable) zugreifen und verändern. Das Ergebnis ist abhängig von der zeitlichen Abfolge der Threads und somit nicht vorhersehbar.

```
Simple race condition
 -- Original version --
_____________________
1 run
Using 5 threads and 1000 increments
----------------------
Racecondition occured [oldCounter = 241, newCounter = 243]
Racecondition occured [oldCounter = 1835, newCounter = 1837]
Racecondition occured [oldCounter = 2159, newCounter = 2162]
Racecondition occured [oldCounter = 3310, newCounter = 3312]
----------------------
Program finished, counter = 4908, race conditions occured = True
======================
2 run
Using 5 threads and 1000 increments
----------------------
Racecondition occured [oldCounter = 22, newCounter = 24]
Racecondition occured [oldCounter = 377, newCounter = 379]
Racecondition occured [oldCounter = 850, newCounter = 852]
Racecondition occured [oldCounter = 1562, newCounter = 1564]
Racecondition occured [oldCounter = 2165, newCounter = 2167]
Racecondition occured [oldCounter = 2353, newCounter = 2355]
Racecondition occured [oldCounter = 3075, newCounter = 3077]
Racecondition occured [oldCounter = 4003, newCounter = 4005]
----------------------
Program finished, counter = 4896, race conditions occured = True
======================
3 run
Using 5 threads and 1000 increments
----------------------
Racecondition occured [oldCounter = 3801, newCounter = 3803]
Racecondition occured [oldCounter = 4610, newCounter = 4612]
----------------------
Program finished, counter = 4863, race conditions occured = True
======================
4 run
Using 5 threads and 1000 increments
----------------------
Racecondition occured [oldCounter = 2199, newCounter = 2201]
Racecondition occured [oldCounter = 3006, newCounter = 3008]
----------------------
Program finished, counter = 4890, race conditions occured = True
======================
5 run
Using 5 threads and 1000 increments
----------------------
Racecondition occured [oldCounter = 2450, newCounter = 2452]
----------------------
Program finished, counter = 4891, race conditions occured = True
======================
Drücken Sie eine beliebige Taste . . .
```

Abbildung 1: Race codition in CSharp - Original version

Abbildung 1 zeigt die Ausgabe der implementierten race codition in CSharp. Dabei wird eine Variable `_counter` erhöht und dann mit dem vorherigen Wert verglichen. Der neue Wert sollte genau um eins größer sein als der alte Wert. Ist dies nicht der Fall ist eine race condition

aufgetreten. Die Variable wurde also von einem anderen Thread geändert.

1b. Wie können race conditions vermieden werden?

Diese können mit Hilfe von Locks vermieden werden.

```
Simple race condition
 -- Fixed version --
_____________________
1 run
Using 5 threads and 1000 increments
----------------------
----------------------
Program finished, counter = 5000, race conditions occured = False
========================
2 run
Using 5 threads and 1000 increments
----------------------
----------------------
Program finished, counter = 5000, race conditions occured = False
========================
3 run
Using 5 threads and 1000 increments
----------------------
----------------------
Program finished, counter = 5000, race conditions occured = False
========================
4 run
Using 5 threads and 1000 increments
----------------------
----------------------
Program finished, counter = 5000, race conditions occured = False
========================
5 run
Using 5 threads and 1000 increments
----------------------
----------------------
Program finished, counter = 5000, race conditions occured = False
========================
Drücken Sie eine beliebige Taste . . .
```

Abbildung 2: Race condition in CSharp - Ohne race condition

Abbildung 2 zeigt die Ausgabe der verbesserten Version. Diese verwendet Locks um die race condition zu vermeiden.

1c. Race condition im Code

Die race condition im angegebenen Code ist die Tatsache, das das der Writer und Reader nicht korrekt miteinander synchronisiert sind und nur ein begrenzter Puffer zur Verfügung steht. Dadurch kann es sein, dass der Writer bereits mehr Werte erzeugt als der Puffer zulässt und somit die alten Werte überschreibt. Die Lösung ist die korrekte Synchronisation der beiden Threads.

```
1 using System;
2 using System.Threading;
3
4 namespace RaceConditionExample
5 {
```

```
6       class RaceConditionExampleFixed
7       {
8           private const int N = 1000;
9           private const int BUFFER_SIZE = 10;
10          private double[] buffer;
11
12          private SemaphoreSlim readerSemaphore;
13          private SemaphoreSlim writerSemaphore;
14          public void Run()
15          {
16              buffer = new double[BUFFER_SIZE];
17              //Reader semaphore starts with blocking
18              readerSemaphore = new SemaphoreSlim(0);
19
20              //Writer can produce BUFFER_SIZE values then he has to wait for the reader
21              //to consume it
22              writerSemaphore = new SemaphoreSlim(BUFFER_SIZE);
23
24              // start threads
25              var t1 = new Thread(Reader); var t2 = new Thread(Writer);
26              t1.Start(); t2.Start();
27              // wait
28              t1.Join(); t2.Join();
29
30              //check that buffer is loaded with the last produced values
31              for (int i = 0; i < BUFFER_SIZE; i++)
32              {
33                  if (!buffer[i].Equals(N - BUFFER_SIZE + i))
34                  {
35                      Console.WriteLine("Race condition occured :(");
36                  }
37              }
38
39          }
40          void Reader()
41          {
42              var readerIndex = 0;
43              for (int i = 0; i < N; i++)
44              {
45                  //wait for a value from the producer
46                  readerSemaphore.Wait();
47                  Console.WriteLine(buffer[readerIndex]);
48                  readerIndex = (readerIndex + 1) % BUFFER_SIZE;
49                  //signal producer that we have consumed a value
50                  writerSemaphore.Release();
51              }
52          }
53          void Writer()
54          {
```

```
55              var writerIndex = 0;
56              for (int i = 0; i < N; i++)
57              {
58                  //Wait until we can produce a new value
59                  writerSemaphore.Wait();
60                  buffer[writerIndex] = (double)i;
61                  writerIndex = (writerIndex + 1) % BUFFER_SIZE;
62                  //singal reader that we have produced a value
63                  readerSemaphore.Release();
64              }
65          }
66      }
67  }
```

2 Synchronization Primitives

2a. / 2b.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Threading;
4
5  namespace SynchronizationPrimitives
6  {
7      class LimitedConnectionsExample
8      {
9          private const int ConcurrentDownloads = 10;
10         /// <summary>
11         /// Semaphore used for allowing max download threads
12         /// </summary>
13         private SemaphoreSlim _syncSemaphore;
14
15         private List<Thread> _threads;
16
17         /// <summary>
18         /// Starts downloading files and returns
19         /// </summary>
20         /// <param name="urls"></param>
21         public void DownloadFilesAsync(IEnumerable<string> urls)
22         {
23             _syncSemaphore = new SemaphoreSlim(ConcurrentDownloads);
24             _threads = new List<Thread>();
25             foreach (var url in urls)
26             {
27                 Thread t = new Thread(DownloadFile);
28                 _threads.Add(t);
29                 t.Start(url);
```

```
            }
        }
        public void DownloadFile(object url)
        {
            _syncSemaphore.Wait();
            Console.WriteLine($"Downloading {url}");
            Thread.Sleep(1000);
            Console.WriteLine($"finished {url}");
            _syncSemaphore.Release();
        }

        /// <summary>
        /// Waits for all downloads to be finished
        /// </summary>
        public void DownloadFiles(IEnumerable<string> urls)
        {
            DownloadFilesAsync(urls);
            foreach (var thread in _threads)
            {
                thread.Join();
            }
            _syncSemaphore.Dispose();
        }
    }
}
```

2c. Aktives Warten

```
using System;
using System.Threading.Tasks;

namespace SynchronizationPrimitives
{
    internal class PollingExample
    {
        private const int MAX_RESULTS = 10;
        private volatile string[] results;
        private Task[] tasks;

        public void Run()
        {
            results = new string[MAX_RESULTS];
            tasks = new Task[MAX_RESULTS];
            // start tasks
            for (var i = 0; i < MAX_RESULTS; i++)
            {
                var t = new Task(s =>
```

```
20                  {
21                      var _i = (int) s;
22                      string m = Magic(_i);
23                      results[_i] = m;
24                  }, i);
25                  tasks[i] = t;
26                  t.Start();
27              }
28
29              Task.WaitAll(tasks);
30
31              // output results
32              for (var i = 0; i < MAX_RESULTS; i++)
33                  Console.WriteLine(results[i]);
34          }
35
36          private string Magic(int i)
37          {
38              return i.ToString();
39          }
40      }
41  }
```