VPS 5	Parallel and Distributed Software Systems	WT 21/22, Exercise 2

Deadline: 06.12.2021 15:00

Name _____

Points _____ **Effort in hours** _____

1. Race Conditions (3 + 3 Points)

a) What are *race conditions*? Implement a simple .NET application in C# that contains a race condition. Document the race condition with appropriate tests. Then improve your program, so that the race condition is removed. Document your solution with appropriate tests again.

b) Where is the race condition in the following code? Explain how the race condition can be removed and provide a fixed version of the code.

```
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 for threads
    t1.Join(); t2.Join();
  }

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

  private 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 (3 + 3 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);
    }
  }

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

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

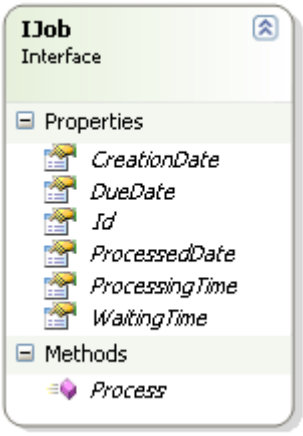
3. Toilet Simulation (4 + 4 + 4 Points)

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).

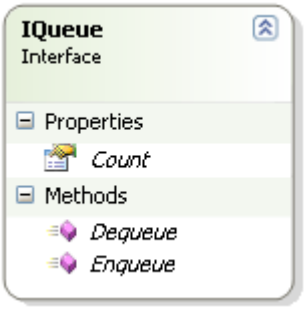
On Moodle you find a simple framework which already provides some parts of the simulation:

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.

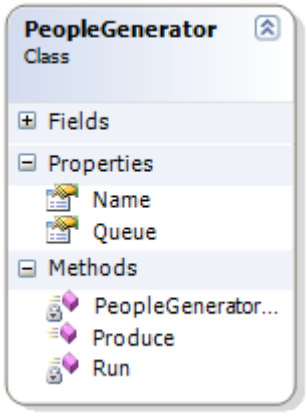


The class *Person* implements IJob. In the constructor of Person the time period available for processing is choosen randomly (normally distributed). Based on that time period the due date (*DueDate*) is set. Additionally the processing time (*ProcessingTime*) is also randomly set (normally distributed).

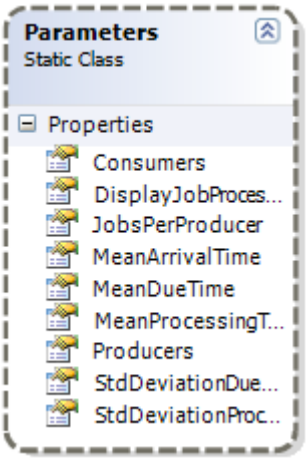
The interface *IQueue* defines the relevant methods for a queue which are used by the producer to enqueue jobs (*Enqueue*) and by the consumer to dequeue jobs (*Dequeue*).



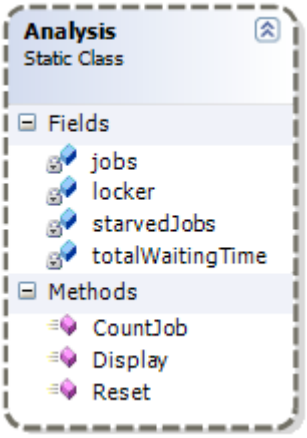
The producer *PeopleGenerator* uses a separate thread to create new jobs (instances of Person) and to enqueue them in the queue. The time between the creation of two Person objects is exponentially distributed (Poisson process).



The class *Parameters* contains all relevant parameters for 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 values.

ToiletSimulation contains the main method which creates all required objects (producers, consumers, queue), starts the simulation and displays the results.

a) Implement a simple consumer *Toilet* which dequeues and processes jobs from the queue in a separate 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 parameters:

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

Execute some tests and besides the individual results also document the mean value and the standard deviation.

c) As you can see in 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 tests you have done in 2.b) with your improved queue and compare.

Note: 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 and in the source code.

VPS5 Exercise 2

Race Conditions

A race condition occurs when two threads try to change the same variable. Since the threads have no way of communicating their changes to the other one, one will overwrite the changes the other made. In this example, the buffer writer will overwrite values that have not yet been read.

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

    private double[] buffer;

    public void Run() {
        buffer = new double[BUFFER_SIZE];

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

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

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

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

The above Race condition can be fixed with an AutoResetEvent. The Writer notifies the Reader whenever a variable is read and vice versa.

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

    private double[] buffer;
    private AutoResetEvent canWrite;
    private AutoResetEvent canRead;

    public void Run() {
        buffer = new double[BUFFER_SIZE];
        canWrite = new AutoResetEvent(true);
        canRead = new AutoResetEvent(false);

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

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

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

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

Synchronisation primitives

To synchronise the downloads, a semaphore with capacity 10 is used. When a download is complete, the semaphore releases the thread.

```
class LimitedConnectionsExample {
    private Semaphore semaphore;
    public LimitedConnectionsExample() {
        semaphore = new Semaphore(10, 10);
    }

    public void DownloadFilesAsync(IEnumerable<string> urls) {
        foreach (var url in urls) {
            semaphore.WaitOne();
            Thread t = new Thread(() => DownloadFile(url));
            t.Start();
            //semaphore.Release();
        }

        Console.WriteLine("Finished");
    }

    private void DownloadFile(object url) {
        // download and store file ...
        Thread.Sleep(2000);
        Console.WriteLine($"File downloaded using Thread #
{Thread.CurrentThread.ManagedThreadId} from url {url.ToString()}");
        semaphore.Release();
    }
}
```

For the synchronised method, all created threads are saved in a list. At the end of the function, all threads are joined back into the caller thread.

```
public void DownloadFiles(IEnumerable<string> urls) {
    List<Thread> threads = new List<Thread>();

    foreach (var url in urls) {
        semaphore.WaitOne();
        Thread t = new Thread(() => DownloadFile(url));
        threads.Add(t);
        t.Start();
    }
    foreach (var t in threads) {
        t.Join();
    }
}
```

Toilet Simulation

The consumer Toilet and FiFoQueue were already implemented during the lecture and produce the following results.

	Jobs	Starved Jobs	Starvation Ratio	Mean waiting time	Total waiting time
1	400	109	27.25%	00:00.2	01:32.8
2	400	110	27.50%	00:00.2	01:35.5
3	400	117	29.25%	00:00.2	01:37.3
4	400	118	29.50%	00:00.2	01:37.3
5	400	116	29.00%	00:00.2	01:38.9
		Mean Starvation Ratio	28.50%		
		Standard deviation	0.009		

A starvation rate of almost 30% is not great. This can be improved by using a priority queue. This means that new Jobs added to the queue will be inserted sorted by their WaitingTime.

```
public static void InsertSorted(this IList<IJob> target, IJob newJob) {
    var index = 0;
    for (var i = 0; i < target.Count; i++) {
        if (target[i].WaitingTime < newJob.WaitingTime) {
            index++;
        } else {
            break;
        }
    }
    target.Insert(index, newJob);
}
```

This improved the starvation rate by about 18%.

	Jobs	Starved Jobs	Starvation Ratio	Mean waiting time	Total waiting time
1	400	42	10.50%	00:00.2	01:10.6
2	400	36	9.00%	00:00.2	01:07.2
3	400	38	9.50%	00:00.2	01:14.4
4	400	42	10.50%	00:00.2	01:20.8
5	400	42	10.50%	00:00.2	01:10.7
		Mean Starvation Ratio	10.00%		
		Standard deviation	0.006		