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]);
    }
  }
```

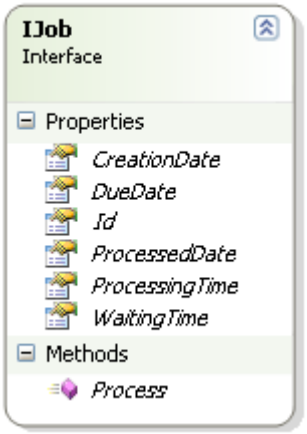
3. Toilet Simulation (4 + 4 + 4 Points)

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).

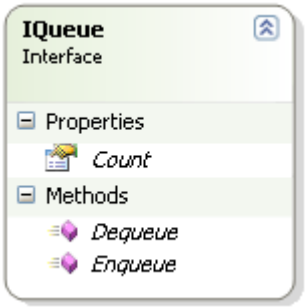
On Moodle you find a simple framework which already provides some parts of the simulation:

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.

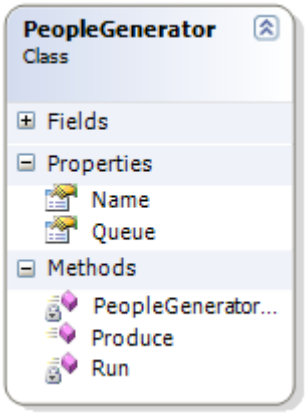


The class *Person* implements IJob. In the constructor of Person the time period available for processing is choosen randomly (normally distributed). Based on that time period the due date (*DueDate*) is set. Additionally the processing time (*ProcessingTime*) is also randomly set (normally distributed).

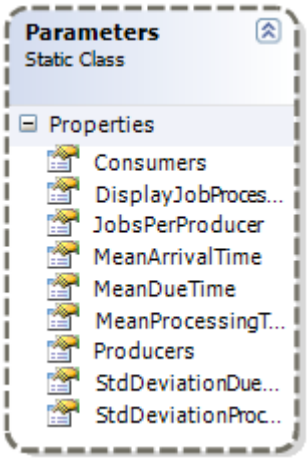
The interface *IQueue* defines the relevant methods for a queue which are used by the producer to enqueue jobs (*Enqueue*) and by the consumer to dequeue jobs (*Dequeue*).



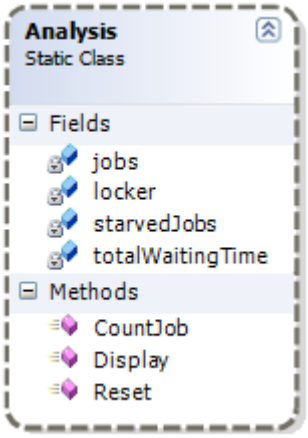
The producer *PeopleGenerator* uses a separate thread to create new jobs (instances of Person) and to enqueue them in the queue. The time between the creation of two Person objects is exponentially distributed (Poisson process).



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.