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

Deadline: 06.12.2021 15:00

Name Andreas Wenzelhuemer

Points ________ **Effort in hours** 5

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.

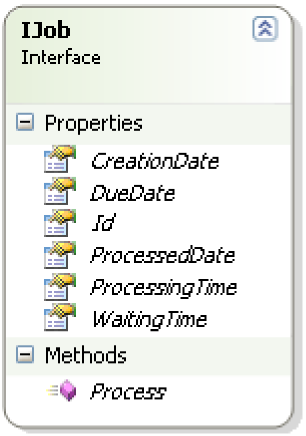
3. Toilet Simulation (4 + 4 + 4 Points)

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.

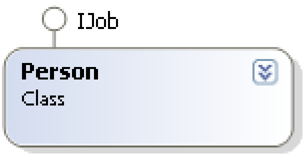
In this task you should implement a queue which handles jobs waiting to be processed (producer-consumer problem). In order to get the example a little bit more "naturalistic", imagine that the jobs are people waiting in front of a toilet (consumer).

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

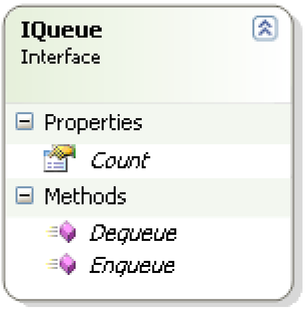
The interface *IJob* defines the data relevant for every job (id, creation date, due date, processing time, waiting time, time when the job was finally processed). It also has a method *Process* which is called by the consumer to process the job.



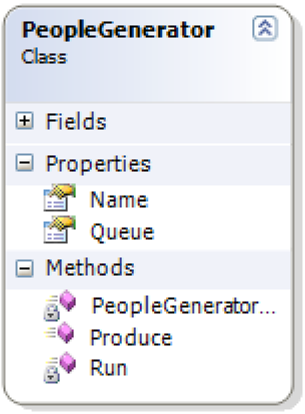
The class *Person* implements IJob. In the constructor of Person the time period available for processing is choosen randomly (normally distributed). Based on that time period the due date (*DueDate*) is set. Additionally the processing time (*ProcessingTime*) is also randomly set (normally distributed).



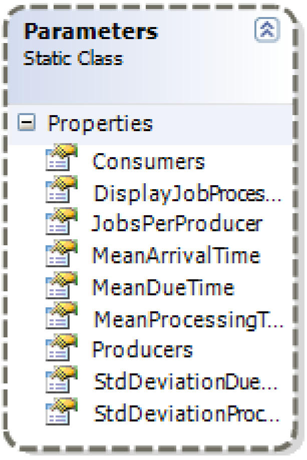
The interface *IQueue* defines the relevant methods for a queue which are used by the producer to enqueue jobs (*Enqueue*) and by the consumer to dequeue jobs (*Dequeue*).



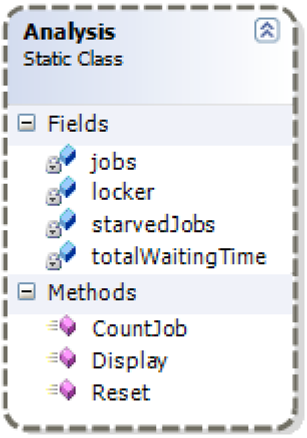
The producer *PeopleGenerator* uses a separate thread to create new jobs (instances of Person) and to enqueue them in the queue. The time between the creation of two Person objects is exponentially distributed (Poisson process).



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.

<u>*Note:*</u> 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.

Übung 2

Table of Contents

1. Race conditions

1.1. Create an simple race condition

Simple race condition where x gets incremented. Because twice threads increment x at the same time, a race condition occurs and x is 1 instead of two. To prevent this a lock object gets created and the lock operation gets called each time x is incremented.

Listing 1. Program with simple race condition

```
static readonly DateTime startTime = DateTime.Now.AddSeconds(1);

private static void RaceCondition()
{
    Console.WriteLine("With race condition");

    int x = 0;
    void incrementX()
    {
        while (DateTime.Now < startTime)
        {
            //do nothing
        }

        x++;
    }

    Thread worker1 = new(incrementX);
    Thread worker2 = new(incrementX);

    worker1.Start();
    worker2.Start();

    worker1.Join();
    worker2.Join();

    Console.WriteLine($"x => {x}");

}
```

Listing 2. Fixed race condition with lock object

```
static readonly DateTime startTime = DateTime.Now.AddSeconds(1);
private static void FixedRaceCondition()
{
    Console.WriteLine("With fixed race condition");

    int x = 0;
    object locker = new();

    void incrementX()
    {
        while (DateTime.Now < startTime)
        {
            //do nothing
        }

        // Locking for x to prevent race condition
        lock (locker)
        {
            x++;
        }
    }

    Thread worker1 = new(incrementX);
    Thread worker2 = new(incrementX);

    worker1.Start();
    worker2.Start();

    worker1.Join();
    worker2.Join();

    Console.WriteLine($"x => {x}");
}
```

1.2. Find the race condition

The race condition occurs because the writer continuously adds new values to the buffer and old ones gets overwritten, because the writer doesn't wait for the reader.

One possible solution would be to use two events, one for the reader and one for the writer. Here both threads could be synchronized. The buffer would be useless with this solution, thats why two semaphores were used instead. One to signal the reader, that items are available to read and one to signal that an item was read successful and the writer can continue writing. Both semaphores are initialized with a capacity of 10 to work correctly with the buffer.

Listing 3. Semaphores

```
using System;
using System.Threading;

namespace RaceConditions2
{
    internal class Program
    {
        class RaceConditionExample
        {
            private const int N = 1000;
            private const int BUFFER_SIZE = 10;

            private double[] buffer;
            private Semaphore empty, full;

            public void Run()
            {
                buffer = new double[BUFFER_SIZE];
                empty = new Semaphore(BUFFER_SIZE, BUFFER_SIZE);
                full = new Semaphore(0, 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++)
                {
                    full.WaitOne(); // Wait until item available
                    lock (buffer)
                    {
                        Console.WriteLine(buffer[readerIndex]);
                        readerIndex = (readerIndex + 1) % BUFFER_SIZE;
                    }
                    empty.Release(); // Signal place is free
                }
            }

            private void Writer()
            {
                var writerIndex = 0;
                for (int i = 0; i < N; i++)
                {
                    empty.WaitOne(); // Wait until place is free

                    lock (buffer)
                    {
                        buffer[writerIndex] = i;
                        writerIndex = (writerIndex + 1) % BUFFER_SIZE;
                    }
                    full.Release(); // Signal item available
                }
            }
        }

        static void Main(string[] args)
        {
            var condition = new RaceConditionExample();
            condition.Run();
        }
    }
}
```

2. Synchronisation primitives

The synchronisation is possible by using a semaphore with capacity 10 and initial count of 10. In the DownloadFile-Method `Release()` gets called for each call. To prevent the method from blocking the main thread, an additional thread is placed around the foreach-Loop.

Listing 4. Combine 10 downloads

```
readonly Semaphore semaphore = new(10, 10);

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

Instead of placing a thread around, every started thread gets added to a list. In a second loop all threads of the list get joined with the main thread, which synchronizes them.

Listing 5. Combine all downloads

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

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

            foreach (var thread in threads)
            {
                thread.Join();
            }
        }
```

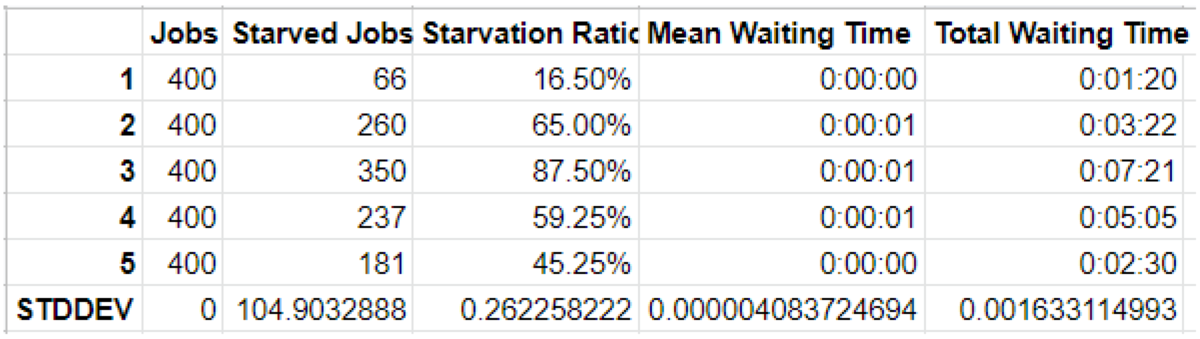
Listing 6. DownloadFile

```
        private void DownloadFile(object url)
        {
            Thread.Sleep(1000);
            Console.WriteLine($"Download file from {url}");
            semaphore.Release();
        }
    }
```

3. Toilet simulation

3.1. FIFO queue

A and b were already implemented during the lesson. In the following table you can see the test results:



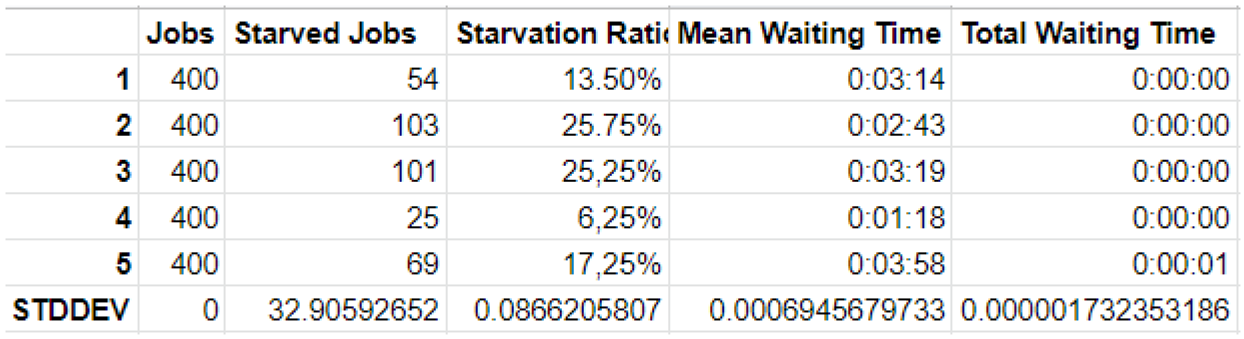
	Jobs	Starved Jobs	Starvation Ratic	Mean Waiting Time	Total Waiting Time
1	400	66	16.50%	0:00:00	0:01:20
2	400	260	65.00%	0:00:01	0:03:22
3	400	350	87.50%	0:00:01	0:07:21
4	400	237	59.25%	0:00:01	0:05:05
5	400	181	45.25%	0:00:00	0:02:30
STDDEV	0	104.9032888	0.262258222	0.000004083724694	0.001633114993

Figure 1. Test results

3.2. Priority queue

Instead of a fifo queue, a priority queue is used. Jobs that are due earlier get prioritized and jobs where the due date is in the past get executed last because it's already to late.

Here you can see that the starvation rate is way lower than with the fifo queue.



	Jobs	Starved Jobs	Starvation Ratic	Mean Waiting Time	Total Waiting Time
1	400	54	13.50%	0:03:14	0:00:00
2	400	103	25.75%	0:02:43	0:00:00
3	400	101	25,25%	0:03:19	0:00:00
4	400	25	6,25%	0:01:18	0:00:00
5	400	69	17,25%	0:03:58	0:00:01
STDDEV	0	32.90592652	0.0866205807	0.0006945679733	0.000001732353186

Figure 2. Test results

3.2.1. Code

```
using System;
using System.Threading;

namespace VPS.ToiletSimulation {
  public class ToiletQueue : Queue {

        private readonly SemaphoreSlim items = new SemaphoreSlim(0, Parameters.JobsPerProducer *
Parameters.Producers + Parameters.Consumers);

        public override void CompleteAdding()
        {
            base.CompleteAdding();
            items.Release(Parameters.Consumers);
        }

        public override void Enqueue(IJob job)
        {
            lock (queue)
            {
                int i = 0;
                while(i < queue.Count && job.DueDate.CompareTo(queue[i].DueDate) >= 0) { i++; }
                queue.Insert(i, job); // Insert sorted
            }
            items.Release(); // Semaphore is thread safe
        }

        public override bool TryDequeue(out IJob job)
        {
            items.Wait();
            lock (queue)
            {
                if (queue.Count > 0)
                {
                    for (int i = 0; i < queue.Count; i++)
                    {
                        if(queue[i].DueDate <= DateTime.Now)
                        {
                            continue;
                        }
                        job = queue[i];
                        queue.RemoveAt(i);
                        return true;
                    }
                    job = queue[0];
                    queue.RemoveAt(0);
                    return true;
                }
            }
            job = null;
            return false;
        }
    }
}
```