

Name Wolfgang Lumetsberger S1310307025Points           Effort in hours 6h**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).

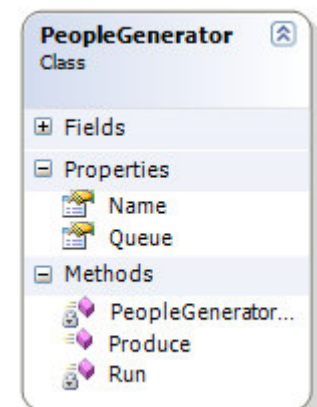
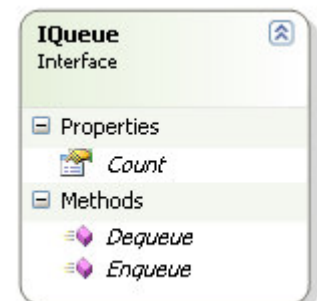
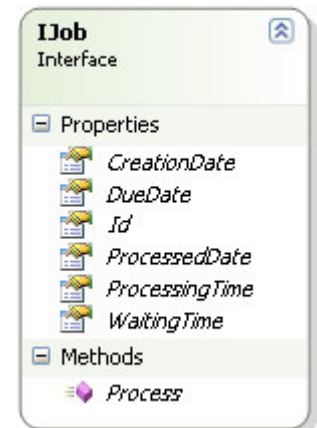
On Moodle you find a simple framework which already provides some parts of the simulation:

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.

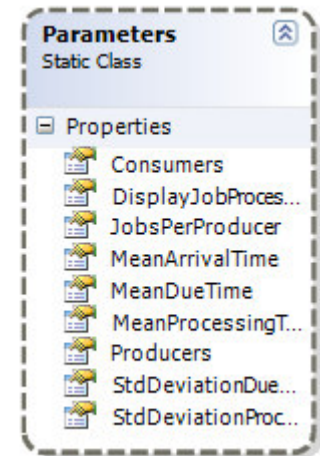
The class *Person* implements *IJob*. In the constructor of *Person* the time period available for processing is chosen randomly (normally distributed). Based on that time period the due date (*DueDate*) is set. Additionally the processing time (*ProcessingTime*) is also randomly set (normally distributed).

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*).

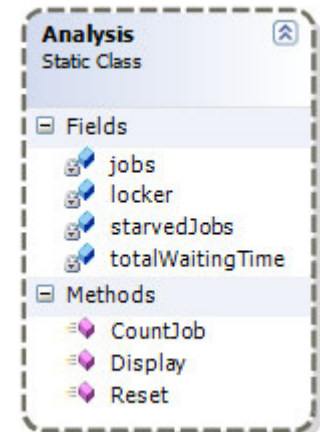
The producer *PeopleGenerator* uses a separate thread to create new jobs (instances of *Person*) and to enqueue them in the queue. The time between the creation of two *Person* objects is exponentially distributed (Poisson process).



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.

- 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?
- 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.

- 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:

a) What are race conditions? (Implement a simple .NET application):

Eine Race Condition existiert, wenn multithreaded oder parallel auf eine geteilte resource zugegriffen wird.

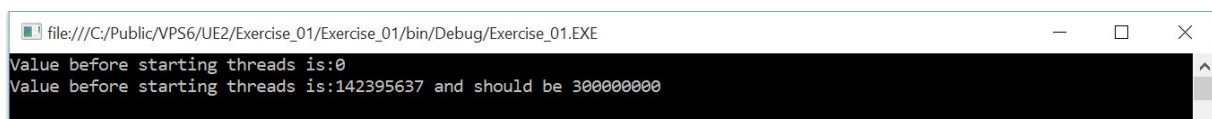
Es ist ein fehler, der aufgrund der zeitlichen reihenfolge der events hervorgerufen wird.

```
class Program
{
    static void Main(string[] args)
    {
        Exercise e = new Exercise();
        Thread t1 = new Thread(() => e.increaseForValueTimes(1, 100000000));
        Thread t2 = new Thread(() => e.increaseForValueTimes(1, 100000000));
        Thread t3 = new Thread(() => e.increaseForValueTimes(1, 100000000));
        Console.WriteLine("Value before starting threads is:" + e.getValue());
        t1.Start();
        t2.Start();
        t3.Start();
        t1.Join();
        t2.Join();
        t3.Join();
        Console.WriteLine("Value before starting threads is:" + e.getValue() + "
and should be " + (3 * 100000000));
        Console.ReadKey();
    }
}

class Exercise
{
    private int x = 0;
    public void increaseForValueTimes(int value, int times)
    {
        for (int i = 0; i < times; i++)
        {
            x = x + value;
        }
    }

    public int getValue()
    {
        return x;
    }
}
```

Das Ergebnis wie erwartet, ist ungleich dem erwarteten wert, da es zu race conditions kommt.



```
file:///C:/Public/VPS6/UE2/Exercise_01/Exercise_01/bin/Debug/Exercise_01.EXE
Value before starting threads is:0
Value before starting threads is:142395637 and should be 300000000
```

b) Improve the program to eliminate race condition

Um das Ergebniss von Oben richtig zu stellen, muss einfach ein Lock über die incrementation von x gebaut werden. Dazu wird im Program ein static object erzeugt, welches beim Lock angegeben wird. Die Threads nutzen dieses dann um zu Synchronisieren, und es kommt zu keiner race condition mehr.

```
class Program
```

```

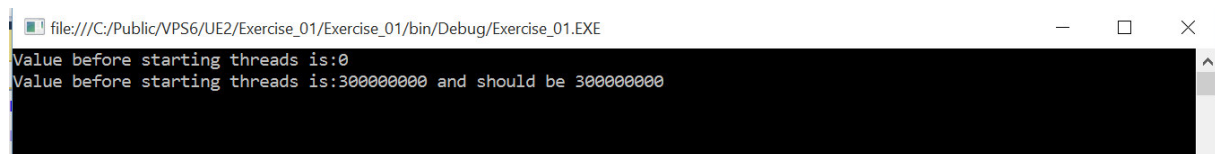
{
    public static object lockingObject = new object();
    static void Main(string[] args)
    {
        Exercise e = new Exercise();
        Thread t1 = new Thread(() => e.increaseForValueTimes(1, 100000000));
        Thread t2 = new Thread(() => e.increaseForValueTimes(1, 100000000));
        Thread t3 = new Thread(() => e.increaseForValueTimes(1, 100000000));
        Console.WriteLine("Value before starting threads is:" + e.getValue());
        t1.Start();
        t2.Start();
        t3.Start();
        t1.Join();
        t2.Join();
        t3.Join();
        Console.WriteLine("Value before starting threads is:" + e.getValue() + "
and should be " + (3 * 100000000));
        Console.ReadKey();
    }
}

class Exercise
{
    private int x = 0;
    public void increaseForValueTimes(int value, int times)
    {
        for (int i = 0; i < times; i++)
        {
            lock (Program.lockingObject)
            {
                x = x + value;
            }
        }
    }

    public int getValue()
    {
        return x;
    }
}

```

Das Ergebnis nun wie erwartet korrekt.



```

file:///C:/Public/VPS6/UE2/Exercise_01/Exercise_01/bin/Debug/Exercise_01.EXE
Value before starting threads is:0
Value before starting threads is:300000000 and should be 300000000

```

c) Where is the race condition and how can the race condition be removed?

Die beiden Threads verwenden den gemeinsamen Buffer. Da dieser nur Begrenzt ist, und writer und reader nicht Synchronisiert sind, passiert es, das der Writer mehr Werte als erlaubt erzeugt und somit alte Werte überschreibt. Um dies zu lösen, muss man beide Threads synchronisieren. Dies kann man Beispielsweise über Semaphoren lösen.

```

public class RaceConditionExample
{
    private const int N = 1000;
    private const int BUFFER_SIZE = 10;
    private double[] buffer;

```

```

private SemaphoreSlim readerSemaphore;
private SemaphoreSlim writerSemaphore;

public void Run()
{
    buffer = new double[BUFFER_SIZE];
    readerSemaphore = new SemaphoreSlim(0);
    writerSemaphore = new SemaphoreSlim(BUFFER_SIZE);

    var t1 = new Thread(Reader);
    var t2 = new Thread(Writer);
    t1.Start();
    t2.Start();
    t1.Join();
    t2.Join();
}
void Reader()
{
    var readerIndex = 0;
    for (int i = 0; i < N; i++)
    {
        readerSemaphore.Wait();
        Console.WriteLine(buffer[readerIndex]);
        readerIndex = (readerIndex + 1) % BUFFER_SIZE;
        writerSemaphore.Release();
    }
}
void Writer()
{
    var writerIndex = 0;
    for (int i = 0; i < N; i++)
    {
        writerSemaphore.Wait();
        buffer[writerIndex] = (double)i;
        writerIndex = (writerIndex + 1) % BUFFER_SIZE;
        readerSemaphore.Release();
    }
}
}

```

## 2. Synchronization Primitives:

### a) Maximal 10 Files gleichzeitig downloaden:

Dies kann realisiert werden, indem einfach eine Semaphore verwendet wird, welche mit 10 initialisiert wird.

```

class Program
{
    static void Main(string[] args)
    {
        LimitedConnectionsExample example = new LimitedConnectionsExample();
        IList<string> downloadList = new List<string>();
        for(int i=0; i<30; i++)
        {
            downloadList.Add("File_" + i);
        }
        example.DownloadFilesAsync(downloadList);
        Console.ReadKey();
    }
}

```



```

class LimitedConnectionsExample
{
    private SemaphoreSlim currentDownloadSemaphore;

    public void DownloadFilesAsync(IEnumerable<string> urls)
    {
        currentDownloadSemaphore = new SemaphoreSlim(10);

        foreach (var url in urls)
        {
            Thread t = new Thread(DownloadFile);
            t.Start(url);
        }
    }

    public void DownloadFile(object url)
    {
        currentDownloadSemaphore.Wait();
        Console.WriteLine("Start downloading: " + url);
        Thread.Sleep(100);
        Console.WriteLine("Downloaded: " + url);
        currentDownloadSemaphore.Release();
    }
}

```

b) DownloadFiles sollte warten bis alle Downloads abgeschlossen sind bevor retourniert wird.

Es muss in der Methode auf die im downloadFileAsync erstellten Threads ein join durchgeführt werden. Dazu einfach die Referenzen der erzeugten Threads merken.

```

class LimitedConnectionsExample
{
    private SemaphoreSlim currentDownloadSemaphore;
    private IList<Thread> threads;

    public void DownloadFilesAsync(IEnumerable<string> urls)
    {
        currentDownloadSemaphore = new SemaphoreSlim(10);
        threads = new List<Thread>();

        foreach (var url in urls)
        {
            Thread t = new Thread(DownloadFile);
            if (threads != null)
            {
                threads.Add(t);
            }
            t.Start(url);
        }
    }

    public void DownloadFile(object url)
    {
        currentDownloadSemaphore.Wait();
        Console.WriteLine("Start downloading: " + url);
        Thread.Sleep(100);
        Console.WriteLine("Downloaded: " + url);
        currentDownloadSemaphore.Release();
    }

    public void DownloadFiles(IEnumerable<string> urls)
    {
        Console.WriteLine("Starting downloading Files");
        this.DownloadFilesAsync(urls);
    }
}

```

```
        if (threads != null)
        {
            foreach (var t in threads)
            {
                t.Join();
            }
        }
        Console.WriteLine("Finished downloading Files");
    }
}
```

### c) Improve Code-Fragment:

In dem man sich einfach die Tasks in einem Array merkt, kann man mit Task.WaitAll dann einfach auf alle Warten, ohne ein aktives Polling betreiben zu müssen.

```
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 (int i = 0; i < MAX_RESULTS; i++)
        {
            tasks[i] = new Task((s) =>
            {
                int _i = (int)s;
                string m = Magic(_i);
                results[_i] = m;

            }, i);

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

    private string Magic(int i)
    {
        return "magic" + i;
    }
}
```

## 3. Toilet Simulation

- 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?

Wurde bereits im Unterricht durchgeführt.

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

```
public class FIFOQueue : Queue
{
    private readonly SemaphoreSlim semaphore;

    public FIFOQueue() {
        semaphore = new SemaphoreSlim(0);
    }

    public override void Enqueue(IJob job)
    {
        if (addingComplete)
        {
            throw new InvalidOperationException("Queue is already complete");
        }
        lock (queueLock)
        {
            queue.Add(job);
        }
        semaphore.Release();
    }

    public override bool TryDequeue(out IJob job)
    {
        job = null;
        if (IsCompleted)
        {
            return false;
        }
        semaphore.Wait();
        lock (queueLock)
        {
            if (!IsCompleted)
            {
                job = GetJob();
                queue.Remove(job);

                return true;
            }
        }
        return false;
    }

    private IJob GetJob()
    {
        lock (queueLock)
        {
            return queue.First();
        }
    }

    public override void CompleteAdding()
    {
        base.CompleteAdding();
    }
}

public abstract class Queue : IQueue
{
    protected IList<IJob> queue;
```

```

protected bool addingComplete;
protected object queueLock = new object();
private int complete;

public int Count
{
    get { return queue.Count; }
}

protected Queue()
{
    queue = new List<IJob>();
}

public abstract void Enqueue(IJob job);

public abstract bool TryDequeue(out IJob job);

public virtual void CompleteAdding()
{
    Interlocked.Increment(ref complete);
    if(complete == Parameters.Producers)
    {
        addingComplete = true;
    }
}

public bool IsCompleted
{
    get
    {
        return addingComplete && Count == 0;
    }
}
}

```

### c) Implement ToiletQueue:

Die neue Klasse erbt von der Queue, welche im obigen Beispiel erstellt wurde. Die Methode GetJob() wurde so geändert, dass sie nun überschrieben werden kann.

Es wird nun jener Job zurückgeliefert, dessen dueDate am kleinsten ist, denn das müsste der job sein, der am dringendsten abgearbeitet werden müsste.

```

public class ToiletQueue : FIFOQueue
{
    protected override IJob GetJob()
    {
        IJob job;
        lock (queueLock)
        {
            job = queue.OrderByDescending(q => q.DueDate).First();
        }
        return job
    }
    protected virtual IJob GetJob()
    {
        lock (queueLock)
        {
            return queue.First();
        }
    }
}

```

}

Das Liefert uns eine deutliche Verbesserung gegenüber der vorherigen Implementation:

```
Parameters:
-----
Mean Arrival Time:      00:00:00.1000000
Mean Due Time:          00:00:00.5000000
Std. Dev. Due Time:     00:00:00.1500000
Mean Processing Time:    00:00:00.1000000
Std. Dev. Processing Time: 00:00:00.0250000

Analysis:
-----
Jobs:                   400
Starved Jobs:           109
Starvation Ratio:       0,2725
Total Waiting Time:     00:01:48.1694769
Mean Waiting Time:      00:00:00.2700000
```

```
Parameters:
-----
Mean Arrival Time:      00:00:00.1000000
Mean Due Time:          00:00:00.5000000
Std. Dev. Due Time:     00:00:00.1500000
Mean Processing Time:    00:00:00.1000000
Std. Dev. Processing Time: 00:00:00.0250000

Analysis:
-----
Jobs:                   400
Starved Jobs:           55
Starvation Ratio:       0,1375
Total Waiting Time:     00:01:38.2342141
Mean Waiting Time:      00:00:00.2460000
```