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

Deadline: 15th of April 2016 17:00

Name Roman Lumetsberger

Points _____ **Effort in hours** 8

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]);
    }
  }
```

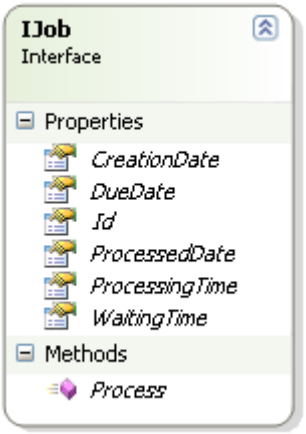
3. Toilet Simulation (4 + 4 + 4 Points)

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).

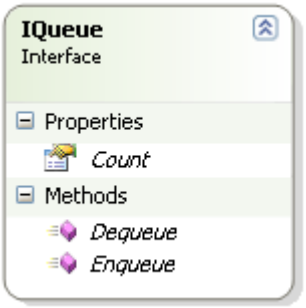
On Moodle you find a simple framework which already provides some parts of the simulation:

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.

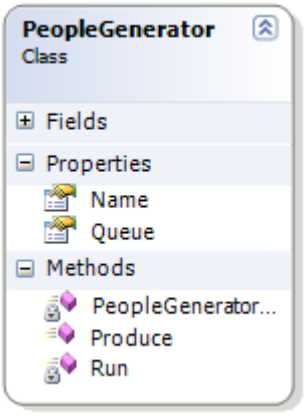


The class *Person* implements IJob. In the constructor of Person the time period available for processing is choosen randomly (normally distributed). Based on that time period the due date (*DueDate*) is set. Additionally the processing time (*ProcessingTime*) is also randomly set (normally distributed).

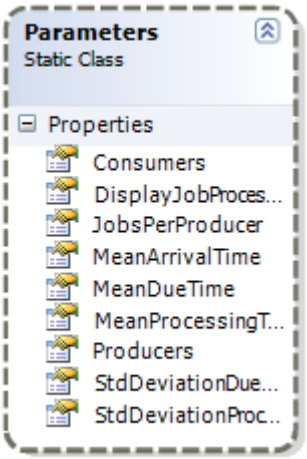
The interface *IQueue* defines the relevant methods for a queue which are used by the producer to enqueue jobs (*Enqueue*) and by the consumer to dequeue jobs (*Dequeue*).



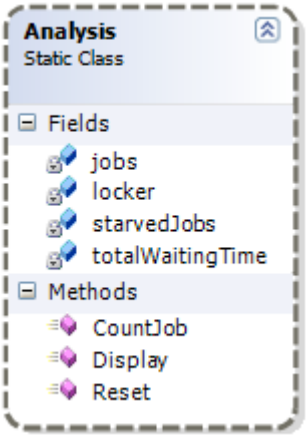
The producer *PeopleGenerator* uses a separate thread to create new jobs (instances of Person) and to enqueue them in the queue. The time between the creation of two Person objects is exponentially distributed (Poisson process).



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 treten bei Programmen auf, bei denen die Ergebnisse von der zeitlichen Abfolge der Threads abhängig und somit nicht vorhersehbar sind. Sie können entstehen, wenn mehrere Threads parallel auf den selben Speicherbereich (Variable) zugreifen und verändern.

```
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 condition in CSharp - Original version

Abbildung 1 zeigt die Ausgabe der implementierten race codition in CSharp. Dabei wird eine Variable `_counter` in der Methode `ThreadMethod` erhöht und dann mit dem vorherigen Wert verglichen. Diese Methode wird dann paralell von mehreren Threads verwendet. 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 überschrieben.

Soucecode siehe 1b. mit Parameter `useLock = false`

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.

1a./1b. Sourccode (Code/SimpleRaceCondition)

Soucecode mit Parameter `useLock = true`

```
1 using System;
2 using System.Threading;
3 using System.Threading.Tasks;
4
5 namespace SimpleRaceCondition
6 {
7     class SimpleRaceCondition
```

```
8       {
9           /// <summary>
10          /// Number of increment runs
11          /// </summary>
12          private const int NumberOfIncrements = 1000;
13
14          /// <summary>
15          /// Number of threads
16          /// </summary>
17          private const int ThreadCount = 5;
18
19          private static readonly Random Random = new Random();
20
21          private static int _counter;
22          private static bool _useLock;
23
24          private static readonly object LockObject=new object();
25
26          /// <summary>
27          /// Main method
28          /// </summary>
29          public static void Run(bool useLock)
30          {
31              _useLock = useLock;
32              _counter = 0;
33              Console.WriteLine($"Using {ThreadCount} threads " +
34                                $"and {NumberOfIncrements} increments");
35              Console.WriteLine("----------------------");
36              var tasks = new Task[ThreadCount];
37              for (int i = 0; i < ThreadCount; i++)
38              {
39                  tasks[i]=new Task(ThreadMethod);
40                  tasks[i].Start();
41              }
42              Task.WaitAll(tasks);
43              Console.WriteLine("----------------------");
44              Console.WriteLine($"Program finished, counter = {_counter}, " +
45                                $"race conditions occured = " +
46                                $"{_counter!=(ThreadCount*NumberOfIncrements)}");
47
48          }
49
50          /// <summary>
51          /// Method which runs parallel
52          /// </summary>
53          public static void ThreadMethod()
54          {
55              for (int i = 0; i < NumberOfIncrements; i++)
56              {
```

```
57                  Thread.Sleep(Random.Next(5));
58                  int oldCounter;
59                  int newCounter;
60                  if (_useLock)
61                  {
62                      //Fixed version using lock
63                      lock (LockObject)
64                      {
65                          oldCounter = _counter;
66                          newCounter = ++_counter;
67                      }
68                  }
69                  else
70                  {
71                      oldCounter = _counter;
72                      newCounter = ++_counter;
73                  }
74
75                  if ((oldCounter + 1) != newCounter)
76                      Console.WriteLine($"Racecondition occured " +
77                                        $"[oldCounter = {oldCounter}," +
78                                        $" newCounter = {newCounter}]");
79              }
80          }
81
82
83      }
84  }
```

1c. Race condition im Code

Die race condition im angegebenen Code ist die Tatsache, dass 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 mit Hilfe von zwei Semaphoren. Die `writerSemaphore` wird auf die `BUFFER_SIZE` initialisiert und kann somit diese Anzahl an Elementen schreiben, bevor sie warten muss, bis der `Reader` die Werte ausgegeben hat.

1c. Sourccode (Code/RaceConditionExample)

```
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.

2a.

Um nur 10 Dateien parallel herunterzuladen, kann eine Semaphore verwendet werden, die die gleichzeitige Anzahl an Downloads auf maximal 10 begrenzt. Dazu wird die Variable `_syncSemaphore` eingeführt und mit dem Wert 10 initialisiert. In der Methode `DownloadFile` wird dann `_syncSemaphore.Wait()` verwendet, um die Downloads zu begrenzen. Wenn ein Download fertig ist, wird die Semaphore wieder freigegeben `_syncSemaphore.Release()`.

2b.

Um auf alle Threads zu warten, müssen diese in einer Liste gespeichert und dann für jeden die Methode `Join` aufgerufen werden.

2a./2b. Sourccode (Code/SynchronizationPrimitives)

```
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);
30              }
31          }
32          public void DownloadFile(object url)
33          {
34              _syncSemaphore.Wait();
35              Console.WriteLine($"Downloading {url}");
36              Thread.Sleep(1000);
37              Console.WriteLine($"finished {url}");
38              _syncSemaphore.Release();
39          }
40
41          /// <summary>
42          /// Waits for all downloads to be finished
43          /// </summary>
44          public void DownloadFiles(IEnumerable<string> urls)
45          {
46              DownloadFilesAsync(urls);
47              foreach (var thread in _threads)
48              {
49                  thread.Join();
50              }
51              _syncSemaphore.Dispose();
52          }
53      }
54  }
```

2c. Aktives Warten

Um das Polling zu verhindern, können alle Tasks in einer Liste gespeichert und dann mit der Methode `Task.WaitAll` auf alle gewartet werden.

2.c Sourccode (Code/SynchronizationPrimitives)

```
1  using System;
2  using System.Threading.Tasks;
```

```
3
4  namespace SynchronizationPrimitives
5  {
6      internal class PollingExample
7      {
8          private const int MAX_RESULTS = 10;
9          private volatile string[] results;
10         private Task[] tasks;
11
12         public void Run()
13         {
14             results = new string[MAX_RESULTS];
15             tasks = new Task[MAX_RESULTS];
16             // start tasks
17             for (var i = 0; i < MAX_RESULTS; i++)
18             {
19                 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 }
```

3 Toilet Simulation

3a. Toilet Implementierung

Diese Aufgabe wurde bereits in der Übung implementiert. Der `Consumer` ist fertig, wenn alle Elemente der Warteschlange abgearbeitet sind. Dazu wurde das Interface `IQueue` um das Property `IsCompleted` erweitert.

Die Synchronisation wurde ebenfalls über die Warteschlange gelöst. Hier wurde das Inter-

face `IQueue` um die Methode `TryDequeue` erweitert. Diese blockiert, falls aktuell kein Element vorhanden ist, aber noch nicht alle Elemente hinzugefügt wurden.

3b. FifoQueue Implementierung

Die `FifoQueue` wurde mit Hilfe einer Semaphore implementiert. Dabei wird bei jedem `Enqueue` die Semaphore um Eins erhöht und beim `TryDequeue` wird diese wieder veringert. Damit wird der gewünschte Effekt erreicht, das die Methode `TryDequeue` blockiert. Dabei kann es vorkommen, dass die Queue bereits leer ist und trotzdem noch Threads in der Methode `TryDequeue` auf Elemente warten. Um das Warten zu beenden, wurde ein `CancellationToken` verwendet, der beim `Wait` mitgegeben wird. Damit können diese Threads aufgeweckt und beendet werden. Um race conditions zu vermeiden, wurde jeder Zugriff auf den Datenspeicher mit Locks geschützt.

3b. Sourcecode (Code/ToiletSimulationForStudents)

```
1  using System;
2  using System.Collections.Generic;
3  using System.Threading;
4
5  namespace VSS.ToiletSimulation
6  {
7      /// <summary>
8      /// Queue base class with common implementations
9      /// </summary>
10     public abstract class Queue : IQueue
11     {
12
13         protected IList<IJob> queue;
14         protected bool addingComplete;
15
16         private int producersComplete;
17
18         protected readonly object LockObject = new object();
19
20         /// <summary>
21         /// Current element count
22         /// </summary>
23         public int Count
24         {
25             get
26             {
27                 lock(LockObject)
28                 {
29                     return queue.Count;
30                 }
31             }
32         }
33
```

```
34          protected Queue()
35          {
36              queue = new List<IJob>();
37          }
38
39          /// <summary>
40          /// enqueue a new job
41          /// </summary>
42          /// <param name="job"></param>
43          public abstract void Enqueue(IJob job);
44
45
46          /// <summary>
47          /// fetch next job
48          /// </summary>
49          /// <param name="job"></param>
50          /// <returns></returns>
51          public abstract bool TryDequeue(out IJob job);
52
53
54          /// <summary>
55          /// Marks the queue as completed
56          /// </summary>
57          public virtual void CompleteAdding()
58          {
59              Interlocked.Increment(ref producersComplete);
60              if (producersComplete == Parameters.Producers)
61                  addingComplete = true;
62          }
63
64          /// <summary>
65          /// Gets the current completion status
66          /// </summary>
67          public bool IsCompleted => addingComplete && Count == 0;
68      }
69  }
```

```
1  using System;
2  using System.Linq;
3  using System.Threading;
4
5  namespace VSS.ToiletSimulation
6  {
7      /// <summary>
8      /// FIFO Queue implementation
9      /// </summary>
10     public class FifoQueue : Queue,IDisposable
11     {
```

```
12          private readonly SemaphoreSlim syncSemaphore;
13          private readonly CancellationTokenSource cancellationTokenSource;
14
15          public FifoQueue()
16          {
17              syncSemaphore = new SemaphoreSlim(0);
18              cancellationTokenSource= new CancellationTokenSource();
19          }
20
21          public override void Enqueue(IJob job)
22          {
23              if (addingComplete)
24              {
25                  throw new InvalidOperationException("Cannot insert " +
26                                                      "elements on already " +
27                                                      "complete marked queue");
28              }
29
30              lock (LockObject)
31              {
32                  queue.Add(job);
33              }
34              syncSemaphore.Release();
35          }
36
37          /// <summary>
38          /// fetch next job
39          /// </summary>
40          /// <param name="job"></param>
41          /// <returns></returns>
42          public override bool TryDequeue(out IJob job)
43          {
44              job = null;
45
46              if (IsCompleted)
47              {
48                  return false;
49              }
50
51              try
52              {
53                  syncSemaphore.Wait(cancellationTokenSource.Token);
54              }
55              catch (OperationCanceledException)
56              {
57                  //do nothing here because it is intended
58                  //it signals that the queue is empty and complete
59              }
60
```

```
61              if (!IsCompleted)
62              {
63                  lock (LockObject)
64                  {
65                      if (Count > 0)
66                      {
67                          var obj = GetNextJob();
68                          job = obj;
69                          queue.Remove(obj);
70                          if (IsCompleted)
71                          {
72                              Finished();
73                          }
74                          return true;
75
76                      }
77                  }
78              }
79              return false;
80          }
81
82          /// <summary>
83          /// Fetches next job
84          /// always returns first element
85          /// </summary>
86          /// <returns></returns>
87          protected virtual IJob GetNextJob()
88          {
89              lock (queue)
90              {
91                  return queue.First();
92              }
93
94          }
95
96          /// <summary>
97          /// Marks the queue as completed
98          /// Checks if all elements are processed
99          /// </summary>
100         public override void CompleteAdding()
101         {
102             base.CompleteAdding();
103             if (IsCompleted)
104             {
105                 Finished();
106             }
107         }
108
109         /// <summary>
```

```
110         /// Cleans the resources and
111         /// cancels waiting consumers
112         /// </summary>
113         private void Finished()
114         {
115             cancellationTokenSource.Cancel();
116         }
117 
118 
119         // Dispose() calls Dispose(true)
120         public void Dispose()
121         {
122             Dispose(true);
123             GC.SuppressFinalize(this);
124         }
125 
126         // The bulk of the clean-up code is implemented in Dispose(bool)
127         protected virtual void Dispose(bool disposing)
128         {
129             if (disposing)
130             {
131                 syncSemaphore.Dispose();
132                 cancellationTokenSource.Dispose();
133             }
134         }
135     }
136 
137 }
```

Parameter

- Producers: 2
- Jobs per produce: 200
- Consumers: 2
- Mean Arrival Time: 100ms
- Mean Due Time: 500ms
- Std. Dev. Due Time: 150ms
- Mean Processing Time: 100ms
- Std. Dev. Processing Time: 25ms

Ergebnisse

Tabelle 1: Ergebnisse FifoQueue

Simulation				
Lauf	Nicht erfolgreiche Jobs	Verhältnis	Wartezeit	Ø Wartezeit
1	222	0,555	00:03:07.4319	00:00:00.469
2	67	0,168	00:01:18.8208	00:00:00.167
3	142	0,356	00:02:13.6162	00:00:00.334
4	239	0,598	00:05:04.3024	00:00:00.761
5	310	0,775	00:07:09:1210	00:00:01.073
Ø	**196**	**0,490**	00:03:45.450	**00:00:00.530**
Std. Abw.	83,750	0,2092	00:02:05.926	00:00:00.361

3c. ToiletQueue Implementierung

Um die Abarbeitung der Jobs zu verbessern, kann der nächste Job anhand der Endzeit und Arbeitszeit ausgewählt werden. Dabei können die Jobs anhand der spätesten Startzeit (= Endzeit - Arbeitszeit) sortiert und dann der erste Eintrag der Liste verwendet werden. Als zusätzliche Optimierung können dann noch jene Jobs, bei denen die späteste Startzeit bereits in der Vergangenheit liegt, hinten angestellt werden (bei diesen Jobs ist es sowieso schon zu spät).

Um die Implementierung zu vereinfachen, wurden das Interface `IJob` um das Property `LatestStartTime` erweitert und in `Person` implementiert. Die `ToiletQueue` selbst stellt eine Ableitung der `FiFoQueue` und überschreibt die Methode `GetNextJob`. Diese Methode wählt dann den nächsten Job anhand des oben angeführten Algorithmus aus.

3c. Sourcecode (Code/ToiletSimulationForStudents)

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
```

```
4  using System.Runtime.Remoting.Metadata.W3cXsd2001;
5
6  namespace VSS.ToiletSimulation
7  {
8      /// <summary>
9      /// ToiletQueue which has a better scheduling algorithm
10     /// </summary>
11     class ToiletQueue:FifoQueue
12     {
13         /// <summary>
14         /// Returns the next job to execute
15         /// uses the job with the lowest time to scheduler
16         /// already too late jobs are scheduled at latest
17         /// </summary>
18         /// <returns></returns>
19         protected override IJob GetNextJob()
20         {
21             DateTime now = DateTime.Now;
22             IJob result;
23             lock (LockObject)
24             {
25                 //tries to find the next element where the latestStartTime > now
26                 //otherwise the first element of the queue is used,
27                 //because its alreay too late ;)
28                 result = queue.OrderBy(x => x.LatestStartTime)
29                               .FirstOrDefault(x => x.LatestStartTime > now)??
30                          queue.First();
31             }
32
33             return result;
34         }
35     }
36 }
```

Parameter

- Producers: 2
- Jobs per produce: 200
- Consumers: 2
- Mean Arrival Time: 100ms
- Mean Due Time: 500ms
- Std. Dev. Due Time: 150ms
- Mean Processing Time: 100ms
- Std. Dev. Processing Time: 25ms

Ergebnisse

Tabelle 2: Ergebnisse ToiletQueue

Simulation				
Lauf	Nicht erfolgreiche Jobs	Verhältnis	Wartezeit	Ø Wartezeit
1	35	0,0875	00:05:11.0704	00:00:00.469
2	30	0,075	00:03:40.0223	00:00:00.550
3	41	0,103	00:05:58.6884	00:00:00.103
4	40	0,100	00:05:30.3139	00:00:00.826
5	37	0,093	00:06:07:2078	00:00:00.918
Ø	**36,6**	**0,091**	00:05:17.461	**00:00:00.573**
Std. Abw.	3,029	0,00995	00:00:52.678	00:00:00.288

Vergleich

Die Durchschnittliche Anzahl an nicht rechtzeitig abgearbeiteten Jobs hat sich von 196 auf 36 reduziert und dadurch können mehr Kunden zufriedengestellt werden. Weiters hat sich das Verhältnis auf 0,49 auf 0,091 reduziert. Obwohl die Reihenfolge der Jobs anders ist, ist die durchschnittliche Wartezeit nahezu gleich geblieben.