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

Deadline: 15th of April 2016 17:00

Name Thomas Herzog

Points _____ **Effort in hours** 10

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]);
    }
  }
```

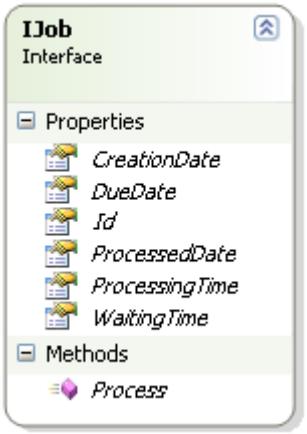
3. Toilet Simulation (4 + 4 + 4 Points)

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).

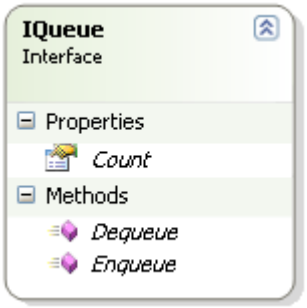
On Moodle you find a simple framework which already provides some parts of the simulation:

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.

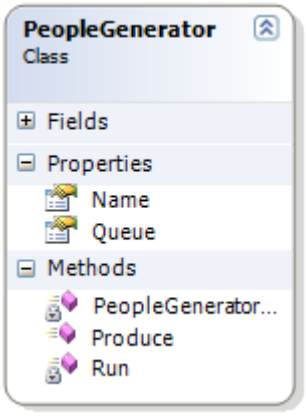


The class *Person* implements IJob. In the constructor of Person the time period available for processing is choosen randomly (normally distributed). Based on that time period the due date (*DueDate*) is set. Additionally the processing time (*ProcessingTime*) is also randomly set (normally distributed).

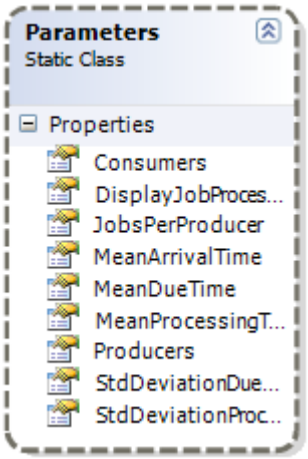
The interface *IQueue* defines the relevant methods for a queue which are used by the producer to enqueue jobs (*Enqueue*) and by the consumer to dequeue jobs (*Dequeue*).



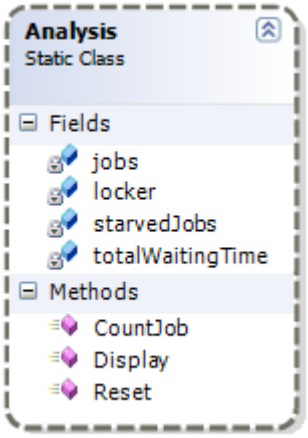
The producer *PeopleGenerator* uses a separate thread to create new jobs (instances of Person) and to enqueue them in the queue. The time between the creation of two Person objects is exponentially distributed (Poisson process).



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.

Synchronization

Ing. Thomas Herzog

Version 1.0, April 14 2016

Folgendes Dokument stellt die Dokumentation für die zweite Übung dar. Diese Übung ist in drei Teile unterteilt:

- Race conditions
- Synchronization primitves
- ToiletSimulation

Race conditions

Dieser Abschnitt beschäftigt sich mit *Race conditions*.
Race conditions tretten auf wenn mehrere *Threads* gleichzeitig auf eine Variable zugreifen, die in einer nicht atomaren Operation verändert wird. Dadurch können inkonsistente Datenbestände entstehen, da der Wert der Variable gleichzeit von mehreren Threads gelesen und geschrieben wird. Ebenso können Schreiboperationen auf diese Variable verloren gehen, wenn dieser Wert durch einen anderen *Thread* überschriben wird. Um *race conditions* zu vermeiden, müssen die nicht atomarn Operationen (*critical sections*), die in einem *Multithreading*-Kontext verwendet werden, synchronisiert werden und dadurch zu einer atomaren Operation zusammengeführt werden.

Folgendes Visual Studio Projekt enthält alle Implementierungen ***RaceConditions***.

Simple race condition

Folgender Beispielcode illustriert eine simple *race condition*.

Race condition example

```
private int value = 0;
private readonly Random random = new Random();

public void DoStuff()
{
    for (int j = 0; j < 100; j++)
    {
        int oldValue = value;
        // start: critical section
        int newValue = value = value + 1;
        // end: critical section

        if ((oldValue - newValue) != -1)
        {
            Console.WriteLine($"OldValue: {oldValue}, newValue: {value}");
        }

        // wait randomly
        Thread.Sleep(random.Next(100));
    }
}
```

Synchronized example

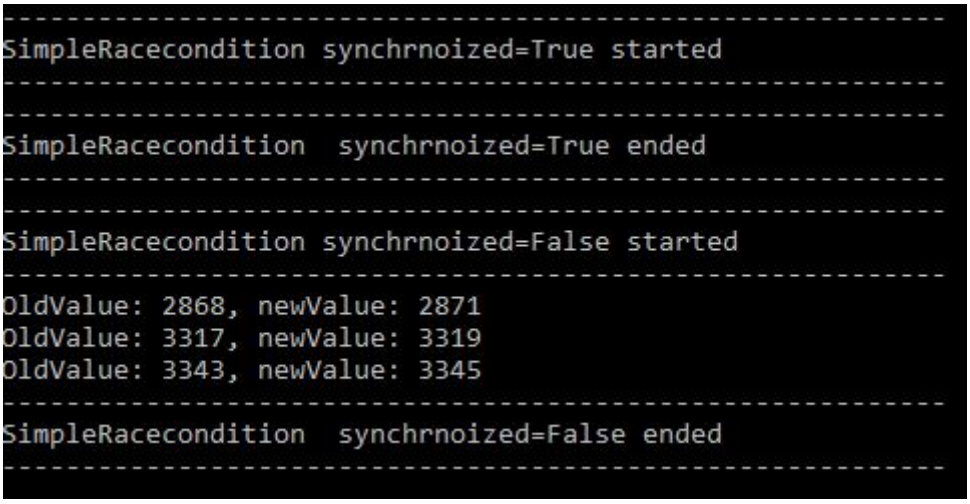
```
private int value = 0;
private readonly Random random = new Random();

private readonly object mutext = new object();

public void DoStuff()
{
    for (int j = 0; j < 100; j++)
    {
        int oldValue, newValue;

        // oldValue = value;
        // start: critical section
        // newValue = value = value + 1;
        // end: critical section

        // start: synchronization
        lock(mutex){
            oldValue = value;
            newValue = value = value + 1;
        }
        // end: synchronization

        if ((oldValue - newValue) != -1)
        {
            Console.WriteLine($"OldValue: {oldValue}, newValue: {value}");
        }

        // wait randomly
        Thread.Sleep(random.Next(100));
    }
}
```

Simple race condition Test

Folgender Abschnitt beschäftigt sich mit den Tests der implementierten *race condition*.
Es wurden zwei Tests durchgeführt, wobei je ein Test

- synchronisiert
- und nicht synchronisiert

durchgeführt wurde. Dieser Test illustriert wie in ein einem nicht snychronisierten Kontext *race conditions* auftretten können.
Es wurde folgende Konfiguration für den Test festgelegt: 20 *Threads* mit 100 Iteration / *Thread*



Nachdem nicht vorhergesagt werden kann zu welchem Zeitpunkt welcher Thread die Variable manipuliert, sind diese Art von Tests auch nit deterministisch und können daher auch nicht reproduziert werden. Es kann also vorkommen, dass bei Testdurchläufen keine *race condition* auftritt.

Für das implementierte Beispiel siehe bitte *Source* SimpleRacecondition.cs.

Fix race condition

Folgender Abschnitt beschäftigt sich mit dem Fix für das *Code*-Beispiel einer *race condition*.
Die *race condition* tritt beim indexierten Zugriff auf den Buffer auf, da hier gleichzeitig von *Threads* gelesen und geschrieben wird und der Buffer über alle *Threads* geteilt wird.

Dieses Problem lässt sich durch die Synchronisation des lesenden und schreibenden Zugriffs auf den Buffer lösen. Zusätzlich wurden Änderungen vorgenommen, die ein blockieren dieses Beispiels verhindern. (Blockierte in der Originalversion)

Für die implementierten Fixes siehe *Source* RaceConditionExampleFixed.cs

Synchronization primitves

Dieser Abschnitt beschäftigt sich mit der Aufgabe *Synchrnoization primitives*.
Die Aufgabe der **parallen *Doownloads*** von Dateien, wurde mit Hilfe von einer Semaphore gelöst, die maximal 10 *Threads* parallel zulässt. Die Threads erhalten eine Referenz auf die Semaphore und fragen diese am Beginn der Methode an. Steht die *Semaphore* zur Verfügung, wird die Methode abgearbeitet und wenn nicht, wird an diese Stelle blockiert. Die *Threads* werden sofort nach ihrer Erzeugung gestartet und werden in der synchronen Version gejoined und in der asynchronen Version nicht.

Für weitere Details und die anderen Aufgaben sei auf den *Source*, die *Tests* und die *Source* -Dokumentation im Projekt ***SynchrnoizationPrimitives*** verwießen.

ToiletSimulation

Folgender Abschnitt beschäftigt sich mit der Aufgabe *ToiletSimulation*.

Folgendes Visual Studio Projekt enthält alle Implementierungen ***SynchronizationPrimitives***.

FIFOQeuue

Folgender Abschnitt beschäftigt sich mit dem ersten Teil der Aufgabe *ToiletSimulation*, in der eine *FIFOQueue* implementiert werden musste.

Die *FIFOQueue* verwendet zwei *Semaphoren* wobei

- eine *Semaphore* für die *Producer*
- und eine *Semaphore* für die *Consumer*

verwendet wird. Die *Semaphoren* synchronisieren einerseits die *Producer* sowie die *Consumer*. Für die Implementierungsdetails sei auf die Klasse FIFOQueue.cs verwießen.

ToiletQueue

Folgender Abschnitt beschäftigt sich mit der implementierten *ToiletQueue*, die eine Verbesserung der implementierten *FIFOQueue* darstellen soll. Dazu wurden mehrere Synchronizationsmöglichkeiten implementiert wie

- mit *Semaphore*,
- mit *AutoResetEvent*,
- mit *Thread.Sleep*,
- und mit *Thread.SpinWait*,

die auf mehrere *Container* wie

- List
- und PriorityQueue

andwendbar ist. Der zu verwendene *Container* und die Synchrnizationsart kann über den Konstruktor definiert werden. Für weitere Implementierungsdetails sei auf den *Source* ToiletQueue.cs.

NetFIFOQueue Tests

Folgender Abschnitt beschäftig tsich mit den Tests der *NetFIFOQueue*. Diese *Queue-'Implementierung delegiert an die 'C# BlockingQueue*, welche die *C#* Implementierung einer *FIFOQueue* darstellt.

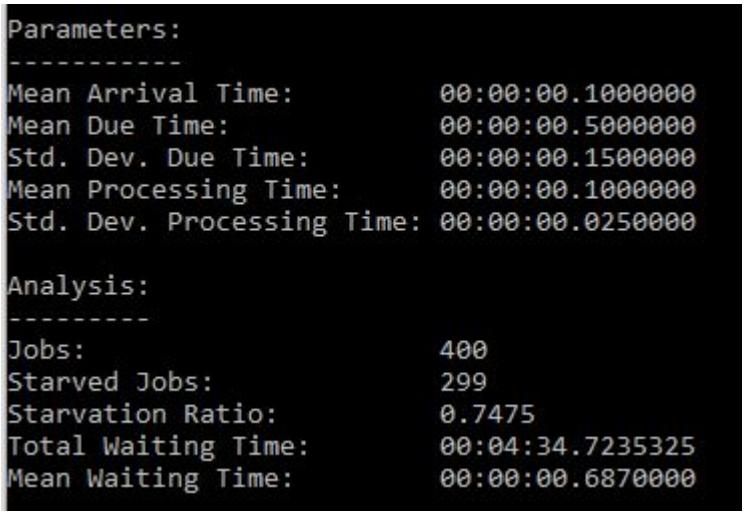


Figure 1. NetFIFOQueue

FIFOQueue Tests

Folgender Abschnitt beschäftig tsich mit den Tests der *FIFOQueue*. Diese Klasse representiert die eigens implementierte *FIFOQueue*.

Figure 2. FIFOQueue with List

Figure 3. FIFOQueue with PriorityQueue

ToiletQueue Tests

Folgender Abschnitt beschäftig tsich mit den Tests der *ToiletQueue*. Diese Klasse representiert die eigens implementierte *Queue*, die eine Verbesserung der implementierten *FIFOQueue* sein soll.

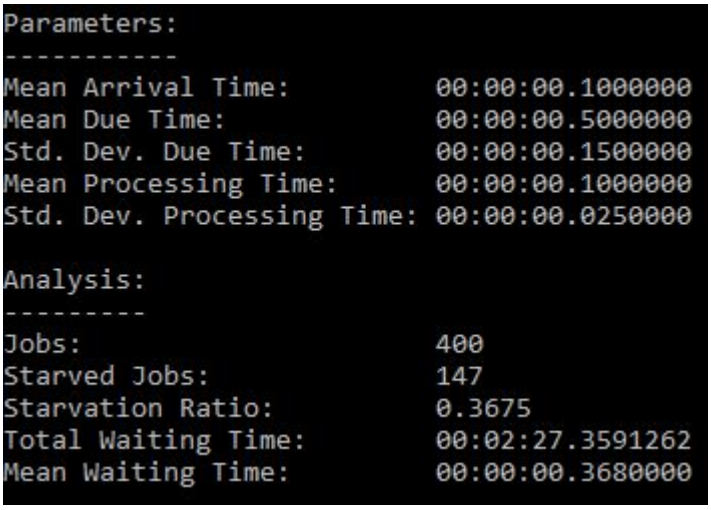


Figure 4. ToiletQueue with List and Semaphore

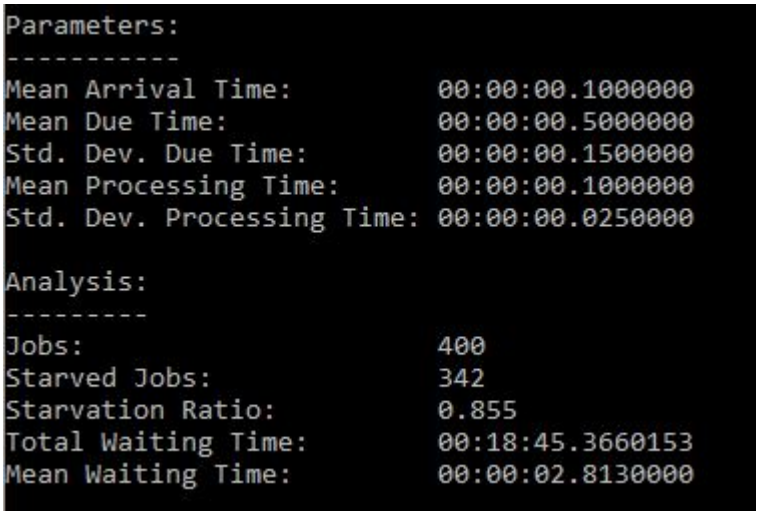


Figure 5. ToiletQueue with List and AutoResetEvent

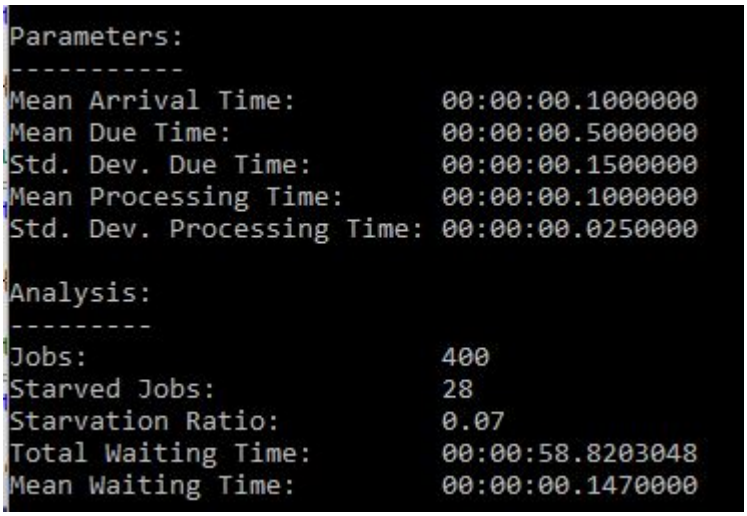


Figure 6. ToiletQueue with List and Thread.Sleep

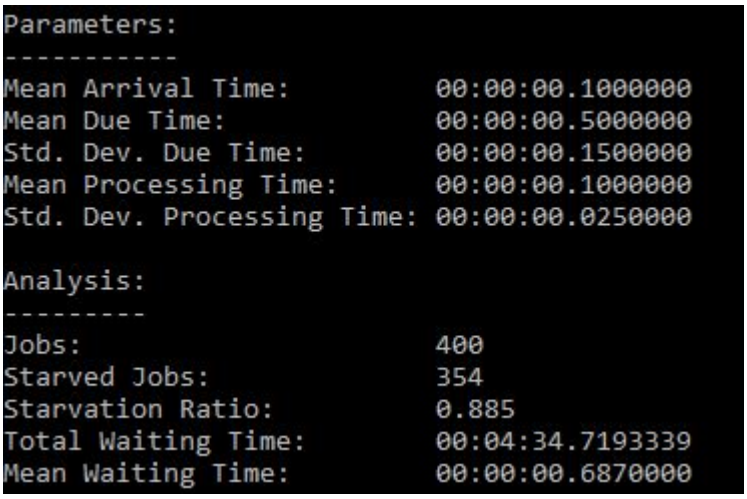


Figure 7. ToiletQueue with List and Thread.SpinWait

```
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:               28
Starvation Ratio:           0.07
Total Waiting Time:         00:02:20.6083298
Mean Waiting Time:          00:00:00.3520000
```

Figure 8. ToiletQueue with PriorityQueue and Semaphore

```
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:               82
Starvation Ratio:           0.205
Total Waiting Time:         00:19:29.6800436
Mean Waiting Time:          00:00:02.9240000
```

Figure 9. ToiletQueue with PriorityQueue and AutoResetEvent

```
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:               33
Starvation Ratio:           0.0825
Total Waiting Time:         00:00:56.7956123
Mean Waiting Time:          00:00:00.1420000
```

Figure 10. ToiletQueue with PriorityQueue and Thread.Sleep

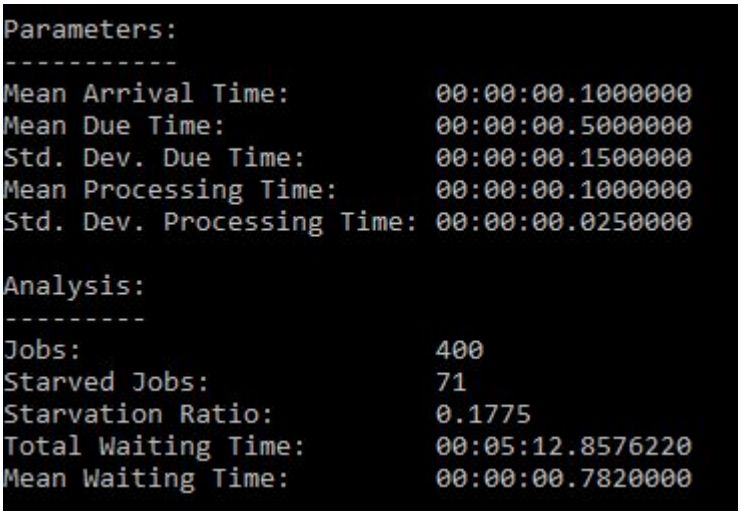


Figure 11. ToiletQueue with PriorityQueue and Thread.SpinWait

Es hat sich gezeigt, dass die Implementierung der *FIFOQueue* mit der Implementierung der *ToiletQueue* nicht verbessert werden konnte.