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

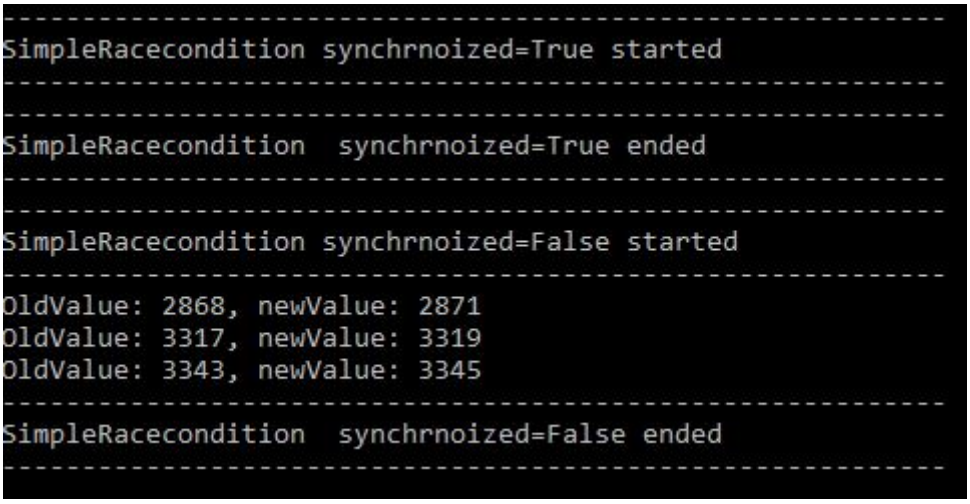
```
private int value = 0;
private readonly Random random = new Random();

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.

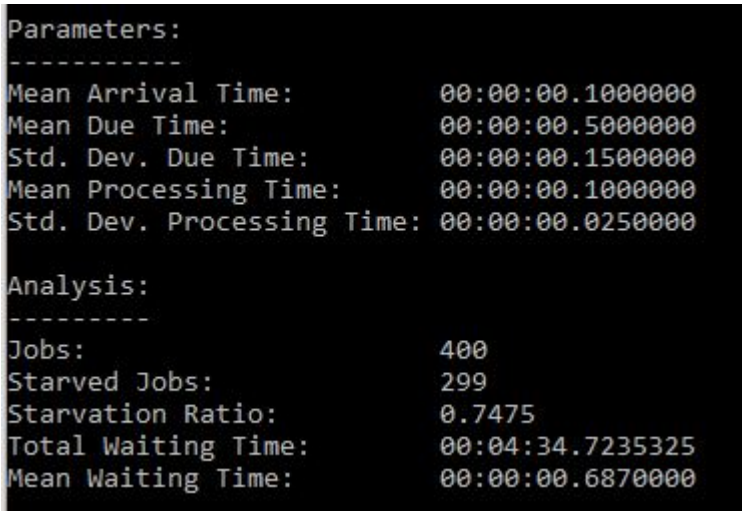


Figure 1. NetFIFOQueue

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.

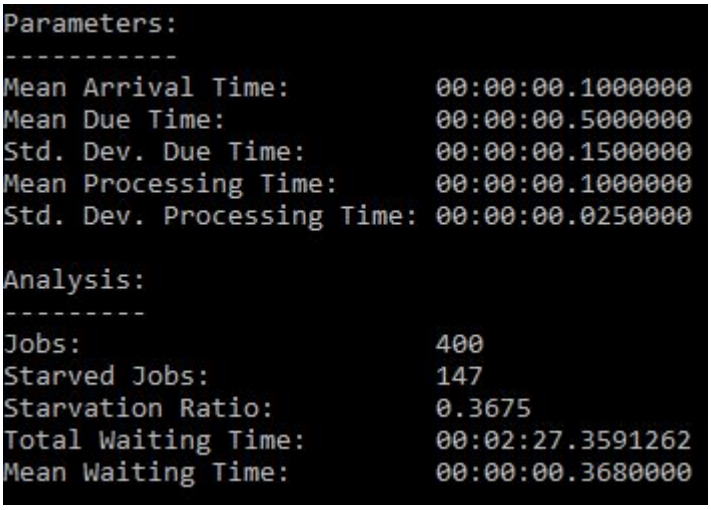


Figure 4. ToiletQueue with List and Semaphore

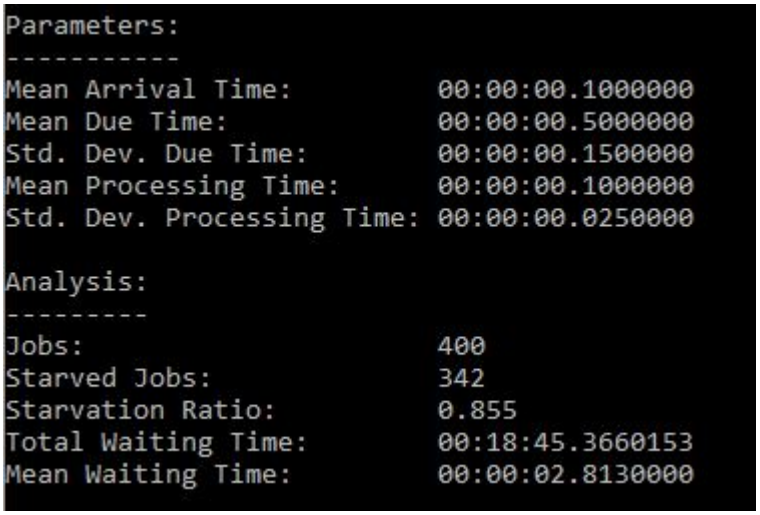


Figure 5. ToiletQueue with List and AutoResetEvent

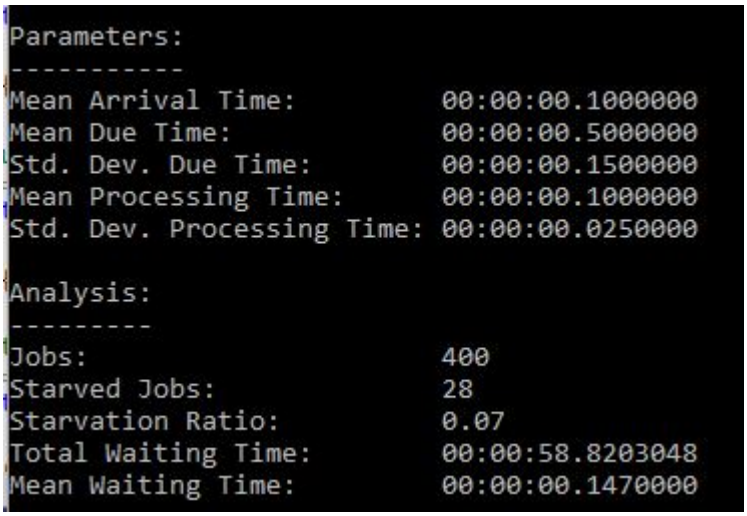


Figure 6. ToiletQueue with List and Thread.Sleep

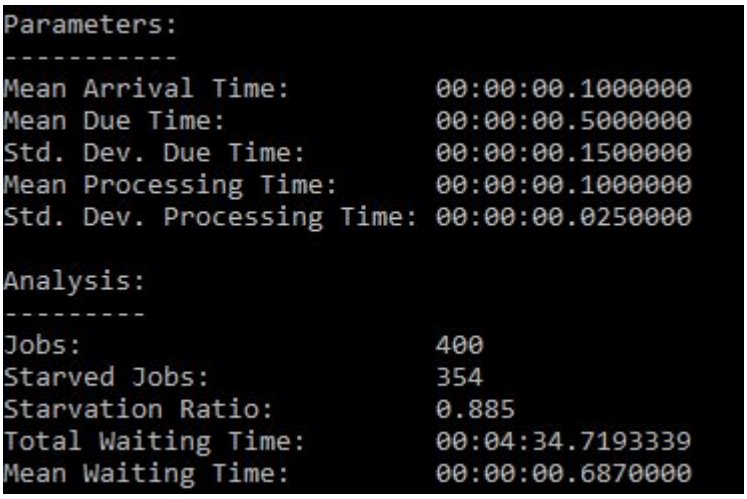


Figure 7. ToiletQueue with List and Thread.SpinWait

```
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

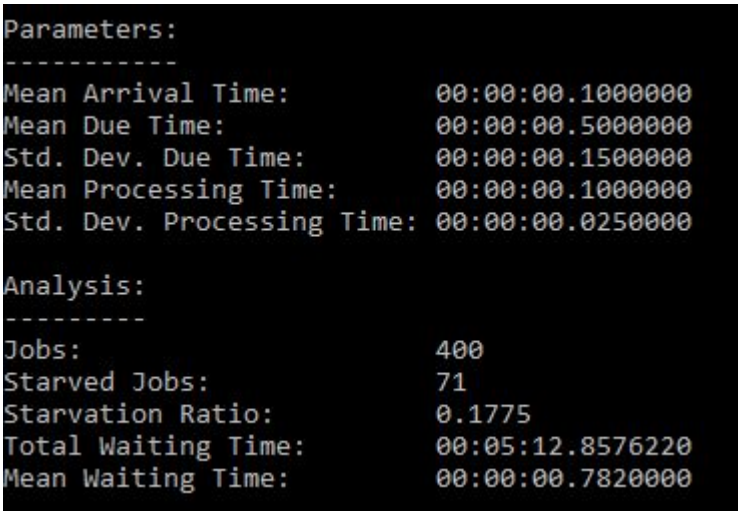


Figure 11. ToiletQueue with PriorityQueue and Thread.SpinWait

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