

VPS5 - UE2

Stefan Kert

13. April 2016

1 Race Conditions

1.1 Was sind *Race Conditions*?

In Programmen kann es zu sogenannten Race conditions kommen, wenn Ergebnisse einer Operation von der zeitlichen Abfolge parallel ablaufender Threads abhängig sind und es zu einem unverhersebaren Ergebnis kommen kann. Ein Beispiel für eine solche Race Condition befindet sich in folgendem Listing:

```
public class RaceConditionExample
{
    private static readonly object LockObject = new object();
    static int result = 0;

    public static void IncreaseResult()
    {
        result++;
    }

    public static void Run(int numberOfIncrements, int threadCount)
    {
        var tasks = new Task[threadCount];
        var raceConditionCount = 0;
        result = 0;
        for (int i = 0; i < numberOfIncrements; i++)
        {
            for (int j = 0; j < threadCount; j++)
            {
                tasks[j] = new Task(() => IncreaseResult());
                tasks[j].Start();
            }

            Task.WaitAll(tasks);

            if (result != i * threadCount)
                raceConditionCount++;
        }

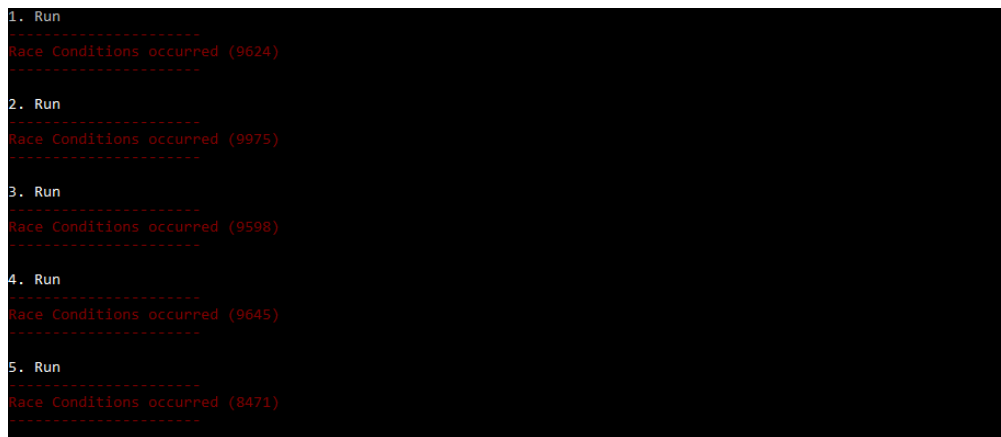
        .....
    }
}
```

```

    }
}

```

In der Methode *IncreaseResult()* wird die statische Variable *result* erhöht. Diese Methode werden schließlich mehrere Threads erzeugt, welche diese Methode gleichzeitig aufrufen. Hier kann es zu Race Conditions kommen, wodurch die Variable *result* falsch erhöht werden kann. Bei dem gegebenen Beispiel kommt es bei 5 Threads und 10000 Schleifendurchläufen zu folgendem Ergebnis:



```

1. Run
-----
Race Conditions occurred (9624)
-----

2. Run
-----
Race Conditions occurred (9975)
-----

3. Run
-----
Race Conditions occurred (9598)
-----

4. Run
-----
Race Conditions occurred (9645)
-----

5. Run
-----
Race Conditions occurred (8471)
-----

```

Abbildung 1: Ergebnis Race Conditions

1.2 Was kann getan werden um *Race Conditions* zu vermeiden?

Eine Möglichkeit Race Conditions zu vermeiden sind Locks. Im folgenden Beispiel wurden diese dazu verwendet um die Operation zu synchronisieren. Im folgenden Listing befindet sich eine Variante, durch welche die Race Conditions nicht mehr auftreten.

```

public class RaceConditionExample
{
    private static readonly object LockObject = new object();
    static int result = 0;

    public static void IncreaseResultWithLock()
    {
        lock (LockObject)
        {
            result++;
        }
    }

    public static void Run(int numberOfIncrements, int threadCount,
        Action method)
    {
        var tasks = new Task[threadCount];
        var raceConditionCount = 0;
    }
}

```

```

        result = 0;
        for (int i = 0; i < numberOfIncrements; i++)
        {
            for (int j = 0; j < threadCount; j++)
            {
                tasks[j] = new Task(method);
                tasks[j].Start();
            }

            Task.WaitAll(tasks);

            if (result != i * threadCount)
                raceConditionCount++;
        }
        .....
    }
}

```

Durch das Verwenden von *lock* in der Methode *IncreaseResultWithLock* wird das inkrementieren synchronisiert wodurch das erwartete Ergebnis erreicht wird:

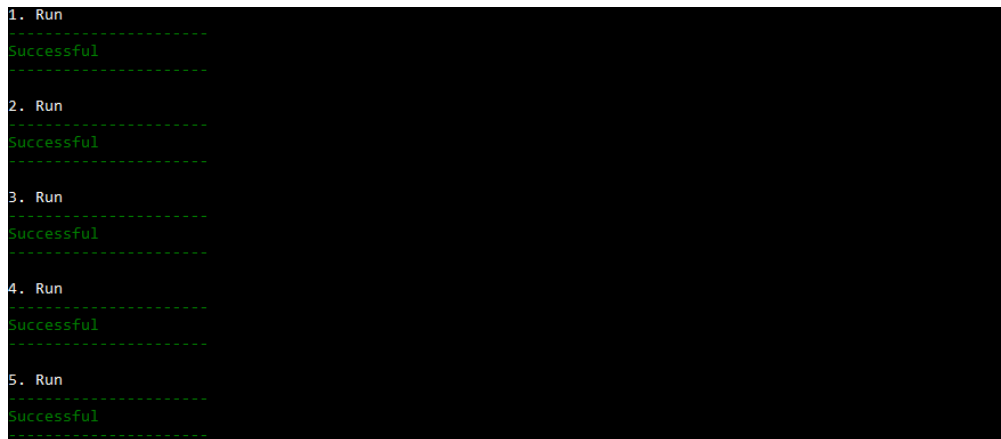


Abbildung 2: Ergebnis der behobenen Race Conditions

1.3 Wo befindet sich die Race Condition im folgenden Code und wie kann diese behoben werden?

Im folgenden Listing ist eine Klasse dargestellt die eine Race Condition enthält. Diese Race Condition zeigt sich in dem Ausmaß, dass der Writer Elemente überschreibt, welche sich im Buffer befinden, die vom Reader noch nicht gelesen wurden.

```

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;
    }
}
}

```

Gelöst werden kann diese Race Condition durch das Verwenden von *AutoResetEvents*. Im Writer wird auf das readerSignal gewartet, welches im Reader gesetzt wird. Danach wird das Element zum Buffer hinzugefügt und das Signal, auf welches im Reader gewartet wird, wieder freigegeben. Dies führt dazu, dass der reader das Element ausliest und wieder zum Anfang springt und die Synchronisierung von vorne beginnt.

```

class FixedRaceConditionExample
{
    private const int N = 1000;
    private const int BUFFER_SIZE = 10;
    private double[] buffer;
    private AutoResetEvent writerSignal;
    private AutoResetEvent readerSignal;

    public void Run()
    {
        buffer = new double[BUFFER_SIZE];
    }
}

```

```

        writerSignal = new AutoResetEvent(false);
        readerSignal = new AutoResetEvent(true);
        // 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++)
        {
            readerSignal.Set();
            writerSignal.WaitOne();
            Console.WriteLine(buffer[readerIndex]);
            readerIndex = (readerIndex + 1) % BUFFER_SIZE;
        }
    }
    void Writer()
    {
        var writerIndex = 0;
        for (int i = 0; i < N; i++)
        {
            readerSignal.WaitOne();
            buffer[writerIndex] = (double)i;
            writerIndex = (writerIndex + 1) % BUFFER_SIZE;
            writerSignal.Set();
        }
    }
}

```

2 Synchronization Primitives

2.1 Wie kann die Anzahl an erzeugten Threads begrenzt werden?

Mit der Klasse *SemaphoreSlim* kann die Anzahl an verwendeten Threads begrenzt werden. Folgendes Listing demonstriert, wie mit Hilfe dieser Klasse die Anzahl der Threads begrenzt werden kann. Dazu wird der Konstruktor mit dem gewünschten Limit (in dem Beispiel 10) instanziiert und in der in einem Thread ausgeführten Methode wird ein Signal aufgerufen, welches anzeigt, dass ein neuer Thread gestartet wird. Für jeden Aufruf von *syncSemaphore.Wait* wird in der Klasse *SemaphoreSlim* intern ein Counter erhöht und bei jedem Aufruf von *syncSemaphore.Release* wird dieser wieder verringert. Mit dieser Funktionalität kann die Anzahl an Aufrufen einer Methode reguliert werden und somit auch die Anzahl an Threads die verwendet werden.

```

public void DownloadFilesAsync(IEnumerable<string> urls)
{

```

```

        _syncSemaphore = new SemaphoreSlim(10, 10);
        _threads = new List<Thread>();
        foreach (var url in urls)
        {
            Thread t = new Thread(DownloadFile);
            _threads.Add(t);
            t.Start(url);
        }
    }

    public void DownloadFile(object url) {
        _syncSemaphore.Wait();
        Console.WriteLine($"Downloading {url}");
        Thread.Sleep(1000);
        Console.WriteLine($"finished {url}");
        _syncSemaphore.Release();
    }
}

```

2.2 Synchrone Implementierung der Methode DownloadFilesAsync

Im folgenden Listing wird die Methode *DownloadFiles* gezeigt, welche wartet bis alle Threads beendet sind. Dies geschieht mittels der Methode *Join*.

```

public void DownloadFiles(IEnumerable<string> urls)
{
    _threads = new List<Thread>();
    foreach (var url in urls)
    {
        Thread t = new Thread(DownloadFile);
        _threads.Add(t);
        t.Start(url);
    }
    foreach (var thread in _threads)
    {
        thread.Join();
    }
}

public void DownloadFile(object url) {
    _syncSemaphore.Wait();
    Console.WriteLine($"Downloading {url}");
    Thread.Sleep(1000);
    Console.WriteLine($"finished {url}");
    _syncSemaphore.Release();
}

```

2.3 Optimieren des Codes, der mittels Polling auf das fertigstellen der Threads wartet

Der folgende Codeabschnitt enthält eine Schleife, in der überprüft wird, ob alle Threads beendet wurden. Dass der Prozessor nicht zu stark belastet wird, wird in der Schleife immer ein paar Millisekunden gewartet und danach erneut überprüft.

```
while (resultsFinished < MAX_RESULTS) {  
    Thread.Sleep(10);  
}
```

Die in den neueren Version des .NET Frameworks vorhandene TPL bietet eine einfache Möglichkeit, wie ein solches Polling verhindert werden kann:

```
Task.WaitAll(tasks);
```

Mit der Methode *WaitAll* wird gewartet bis alle *Tasks* die übergeben wurden beendet sind.