# Race Conditions:

1. 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()

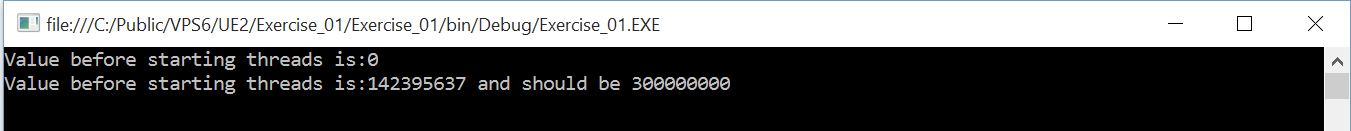
{

return x;

}

}

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



1. 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 warden. 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()

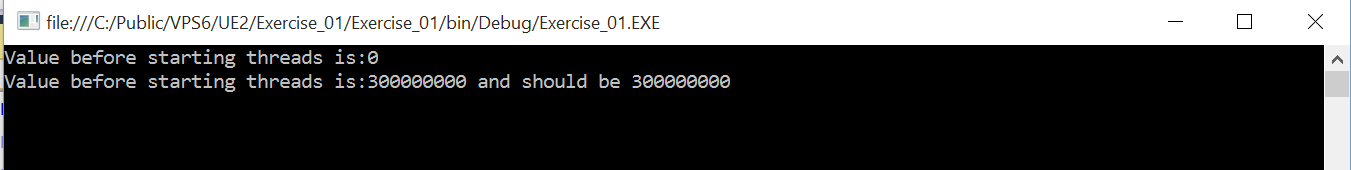
{

return x;

}

}

Das Ergebnis nun wie erwartet korrekt.



1. 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();

}

}

}

# Synchronization Primitives:

## Maximal 10 Files gleichzeitig downloaden:

Dies kann relaisiert 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();

}

}

## DownloadFiles sollte warten bis alle Downloads abgeschlossen sind bevor retouniert 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");

}

}

## 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;

}

}

# Toilet Simulation

## 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.

## 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;

}

}

}

## 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:

