

Thread-Synchronisation

Threadsynchronisation

- Alle Threads im System müssen Zugriff auf bestimmte Systemressourcen haben (z.B. Heaps, serielle Schnittstellen, Dateien, Fenster).
- Falls ein Thread exklusiven Zugriff auf eine Ressource benötigt, können andere Threads, die ebenfalls diese Ressource benutzen müssen, ihre Arbeit nicht erledigen.
- Andererseits kann nicht einfach jedem beliebigen Thread erlaubt werden, jederzeit auf jede beliebige Ressource zuzugreifen, sonst ist das Chaos vorprogrammiert.
- Die Synchronisation ist ein unabdingbarer Vorgang beim Multithreading.

Threadsynchronisation

- Um zu verhindern, dass eine gemeinsam genutzte Ressource durch den Zugriff mehrerer Threads *beschädigt* wird, müssen Threadsynchronisierungskonstrukte in den Code eingefügt werden.
- Microsoft Windows und die Common Language Runtime (CLR) bieten verschiedene Threadsynchronisierungskonstrukte an mit diversen Vor- und Nachteilen.
- Viele dieser Konstrukte der CLR sind nur objektorientierte Wrapperklassen um Win32-Threadsynchronisierungskonstrukte. Schließlich sind alle .NET-Threads Windows-Threads

Asynchrones Threading

```
class MyThread{
    public static int i=0;

    static void Main(string[] args){
        Thread[] ta = new Thread[100]; // Thread-Array
        for (int j = 0; j<100; j++) {
            ta[j] = new Thread(new ThreadStart(tuwas));
            Console.Write("Aufruf start von Thread " + j);
            // Threads werden erzeugt
            ta[j].Start(); // Threads werden gestartet
        }
        System.Console.ReadLine();
    }

    public static void tuwas(){
        String mystring = "Thread gestartet ";
        i++;
        mystring += i;
        System.Console.WriteLine(mystring);
    }
}
```

Asynchrones Threading

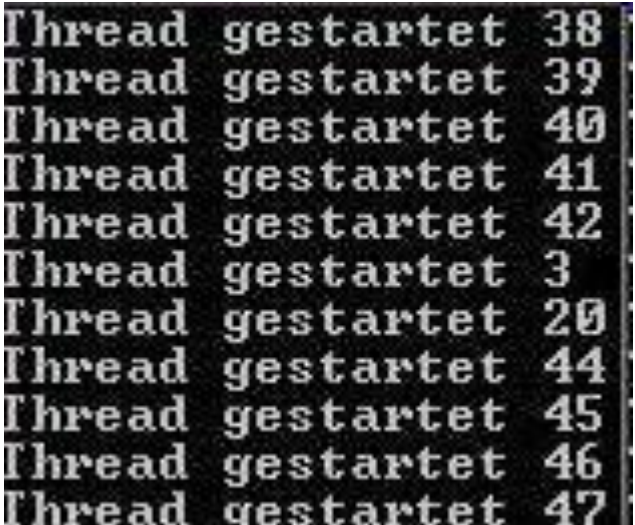
ta[j] = new Thread(new ThreadStart(tuwas));

erzeugt einen Thread, der die Methode **tuwas()** ausführt.

ta[j].Start();

startet der Thread letztendlich

Ein Aufruf des Programms könnte folgendes Ergebnis liefern:



```
Thread gestartet 38
Thread gestartet 39
Thread gestartet 40
Thread gestartet 41
Thread gestartet 42
Thread gestartet 3
Thread gestartet 20
Thread gestartet 44
Thread gestartet 45
Thread gestartet 46
Thread gestartet 47
```

Asynchrones Threading

Thread 3 und Thread 20 wurden zu Beginn scheinbar unterbrochen und bekommen erst nach der Ausführung von Thread 42 wieder eine Zeitscheibe zugeteilt.

Thread 43 wird wiederum scheinbar von Thread 3 und Thread 20 unterbrochen und wird wahrscheinlich auch erst zu einem späteren Zeitpunkt oder wurde evtl. schon vorher ausgeführt.

Synchrones Threading

```
class MyThread {
    public static int i=0;
    public static Object lockvar = ""; // Lock-Variable

    static void Main(string[] args){
        Thread[] ta = new Thread[100]; // Thread-Array
        for (int j = 0; j<100; j++){
            ta[j] = new Thread(new ThreadStart(tuwas));
            // Threads werden erzeugt
            ta[j].Start(); // Threads werden gestartet
        }
        System.Console.ReadLine();
    }

    public static void tuwas(){
        lock (lockvar){
            String mystring = "Thread gestartet ";
            i++;
            mystring += i;
            System.Console.WriteLine(mystring);
        }
    }
}
```

Synchrones Threading

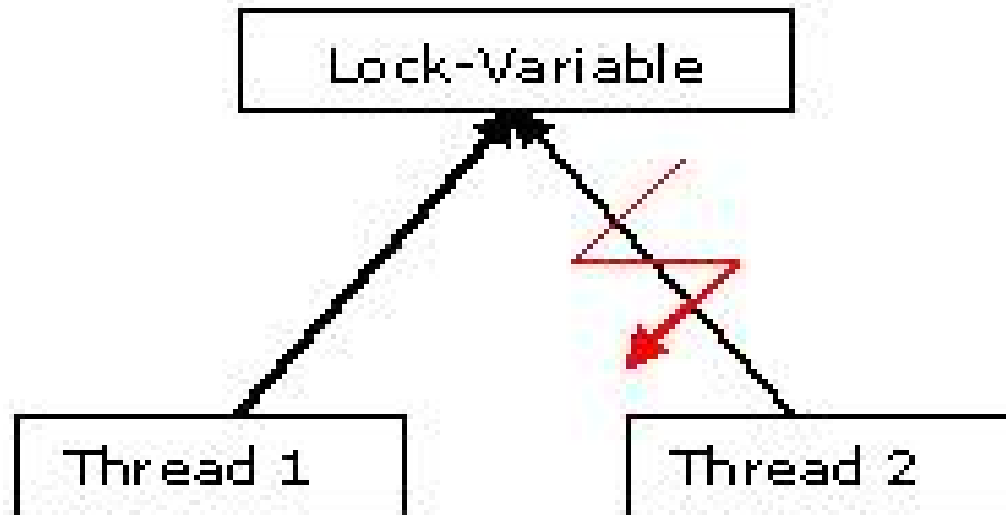
```
Thread gestartet 1  
Thread gestartet 2  
Thread gestartet 3  
Thread gestartet 4  
Thread gestartet 5  
Thread gestartet 6  
Thread gestartet 7  
Thread gestartet 8  
Thread gestartet 9  
Thread gestartet 10
```

Mit **public static Object lockvar = ""**; wird eine zur Synchronisation benötigte Lock-Variable erzeugt.

Der Ausdruck **lock(lockvar){...}** markiert einen atomaren (ununterbrechbaren) Block.

Während sich ein Thread innerhalb eines atomaren Blocks befindet, kann dieser nicht unterbrochen werden.

Synchrones Threading



Synchrones Threading

```
class MyThread {
    public static int i=0;
    public static Object lockvar = ""; // Lock-
    Variable

    static void Main(string[] args){
        Thread[] ta = new Thread[100]; // Thread-Array
        for (int j = 0; j<100; j++){
            ta[j] = new Thread(new ThreadStart(tuwas));
            // Threads werden erzeugt
            ta[j].Start(); // Threads werden gestartet
        }
        System.Console.ReadLine();
    }

    public static void tuwas(){
        Monitor.Enter(lockvar);
        String mystring = "Thread gestartet ";
        i++;
        mystring += i;
        System.Console.WriteLine(mystring);
        Monitor.Exit(lockvar);
    }
}
```

Synchrones Threading

```
class MyThread{
    private int i = 0;
    public void tuwas(){
        Monitor.Enter(this);
        String mystring = "Thread gestartet ";
        i++;
        mystring += i;
        System.Console.WriteLine(mystring);
        Monitor.Exit(this);
    }
    static void Main(string[] args){
        MyThread mythread = new MyThread();
        Thread[] ta = new Thread[100]; // Thread-Array
        for (int j = 0; j < 100; j++){
            ta[j] = new Thread(new ThreadStart(mythread.tuwas));
            ta[j].Start();
        }
        System.Console.ReadLine();
    }
}
```

Monitor

- Mechanismus, der den Zugriff auf Objekte synchronisiert.
- Die Klasse Monitor ist nicht instanziiierbar, da jedem Objekt nur ein Monitor zugeordnet werden kann.
- Mehrere Objekte können den Anspruch auf die Nutzung des Monitors eines anderen Objekts erheben.
- Nur einem Objekt aus der Warteschlange wird er zugestanden.

Monitor.Enter / Monitor.Exit

- Mit den Methoden Enter und Exit können kritische Codeabschnitte definiert werden, die zu einem gegebenen Zeitpunkt nur von einem Thread betreten werden dürfen.
- Beide Methoden sind statische Methoden der Klasse Monitor.
- Als Parameter erhalten beide Methoden die Referenz auf das zu synchronisierende Objekt, das kann auch this sein.

Monitor.Enter / Monitor.Exit

Enter(object)

Erhält eine exklusive Sperre für das angegebene Objekt.

Exit(object)

Hebt eine exklusive Sperre für das angegebene Objekt auf.

TryEnter(object)

Versucht, eine exklusive Sperre für das angegebene Objekt zu erhalten.

TryEnter(object,int)

Versucht über eine angegebene Anzahl von Millisekunden hinweg, eine exklusive Sperre für das angegebene Objekt zu erhalten.

TryEnter(object,System.TimeSpan)

Versucht über einen angegebenen Zeitraum hinweg, eine exklusive Sperre für das angegebene Objekt zu erhalten.

Monitor.Wait

public static bool Wait(object)

- Mit Wait wird der aktuelle Thread blockiert und gleichzeitig die Sperrung des Objekts aufgehoben. Damit kann ein anderer Thread das freigegebene Objekt nutzen.
- Parameter ist die Referenz auf das Objekt, dessen Sperrung aufgehoben werden soll.
- Der Rückgabewert ist true, wenn kein anderer Thread das Objekt sperrt und der akt. Thread die Sperrung übernimmt.

while (Monitor.Wait(obj)){

// Thread tritt in den synchronisierten Block ein

}

Monitor.Wait

Wait(object,int)

Hebt die Sperre für ein Objekt auf und blockiert den aktuellen Thread, bis er die Sperre erneut erhält. Wenn das angegebene Timeoutintervall abläuft, tritt der Thread in die Warteschlange für abgearbeitete Threads ein.

Wait(object,int,bool)

Wait(object,System.TimeSpan)

Wait(object,System.TimeSpan,bool)

Unterschied

- Thread, der mit Enter eine synchronisierte Methode betreten möchte → Zustand: bereit
- Thread, der mit Wait die Sperrung eines Objekts aufgehoben hat, befindet sich in einer Warteliste aller Threads, die den Zustand wartend haben.

Monitor.Pulse

Um einen Thread aus seinem Wartezustand zu holen, muss ein anderer Thread die Methode **Pulse** oder **PulseAll** auf dem gesperrten Objekt aufrufen.

Pulse(object)

Pulse wählt willkürlich einen wartenden Thread aus der Warteschlange auf und versetzt ihn in den Zustand bereit.

PulseAll(object)

Benachrichtigt alle wartenden Threads über eine Änderung am Zustand des Objekts und versetzt sie in den Zustand bereit.

Delegate ThreadStart

Das Delegate ThreadStart muss parameterlos sein. Manchmal ist es aber notwendig, der Threadmethode Daten zu übergeben. Das ist möglich mit dem Delegate ParameterizedThreadStart geboten:

```
public sealed delegate void ParameterizedThreadStart(object obj);
```

Die Erzeugung des Threads:

```
Thread thread =  
    new Thread(new ParameterizedThreadStart(ThreadMethod));
```

Die gewünschten Daten werden mit der Start-Methode an die Threadmethode geleitet:

```
thread.Start(IrgendEinObjekt);
```