Seminar aus Softwareentwicklung

(Inside Java and .NET)

Threading unter .NET

Daniel Grünberger
9855066
880

Inhaltsverzeichnis

1. Vorwort & Motivation 3

2. Überblick über Threads 3

   2.1. Was ist ein Thread? 3

   2.2. Funktionsweise & Eigenschaften von Threads 4

      2.2.1. Erstellen und Starten 5

      2.2.2. Vereinigen 5

      2.2.3. Suspendieren 5

      2.2.4. Abbrechen 5

      2.2.5. Zustandsdiagramm eines Threads 5

   2.3. Datenübergabe an einen Thread 6

   2.4. Synchronisation 8

      2.4.1.Interlocked 10

      2.4.2. Monitor & Lock 11

      2.4.3. Race Condition & Deadlocks 12

   2.5. Verwaltung 14

3. Applikation Domain 16

   3.1. Einführung 16

   3.2. Unterschiede zwischen AppDomain und Thread 17

   3.3. Verwendung 18

   3.4. AppDomain Events 19

   3.5. Code Management 20

4. Literaturliste 21

1. Vorwort & Motivation

Am Anfang der Softwareentwicklung basierten die meisten Applikationen auf einem bzw. mehrere Prozess(e) (kleinste Einheit, mit der ein Betriebssystem arbeitet). Doch im Laufe der Zeit wurde die Software immer komplexer und man war gezwungen mehrere Prozesse, oft auch gleichzeitig, zu verwenden und abzuarbeiten. Diese hatte aber meist einen großen Aufwand und die Kommunikation war meist schwierig und ineffizient. Weiters war dann auch noch das Problem der Synchronisation.
Um diese in den Griff zu bekommen versuchten von nun an die Betriebssysteme eine Art von „Preemptive Scheduling“ zu verwenden – d.h.: Das Betriebssystem kontrolliert, wie lange ein Prozess Systemressourcen beanspruchen darf und im Falle zu langer Beanspruchung wird der Prozess für eine kurze Zeit unterbrochen, damit nicht ein Prozess die gesamte CPU-Leistung für sich beansprucht, sondern auch andere Prozesse zum Zug kommen können. Algorithmen für Events, Mutexes, Semaphores, Read/Write-Locks und Critical-Sections sind entstanden.
Aber nach wie vor machte das gleichzeitige abarbeiten von Prozessen noch immer Probleme. Darum wurden Threads ins Leben gerufen. Diese können auf den gleichen Adressraum wie ihr Vater-Prozess zugreifen und erlauben ohne viel Aufwand mehrere Dinge zum gleichen Zeitpunkt abzuarbeiten. Somit konnte man nun auch „echte“ Echt-Zeit-Anwendungen schreiben, welche bekannt dafür sind, dass sie viele Threads benötigen.

2. Überblick über Threads

2.1. Was ist ein Thread?

Ein Thread bezeichnet eine selbständig ausführbare Programm-Funktion, also einen Teil eines Programms, der unabhängig vom übrigen Programm ausgeführt werden kann bzw. die in Multitaskingumgebungen unabhängig von und zeitgleich mit anderen ausgeführt werden können. Ein Programm kann so dutzende von Teilaufgaben scheinbar gleichzeitig ausführen. Auch auf Einprozessorsystemen kann das sinnvoll sein, wenn die einzelnen Threads den Prozessor jeweils nur gering auslasten. Die Anzeige der Bilder in einem Webbrowser zum Beispiel ist meistens durch Multithreading realisiert. Alle Bilder werden scheinbar gleichzeitig geladen und angezeigt, obwohl jedes Bild von einem eigenständigen Programmteil verarbeitet wird. Threads sind eine Art leichtgewichtiger Prozesse, mit deren Hilfe das Multitasking innerhalb von Applikationen möglich ist.

Wenn ein Programm mehrere Dinge gleichzeitig machen soll, setzt man typischerweise Threads ein. Angenommen man will die Zahl PI (3,14159265…) bis auf die Millionste Stelle genau berechnen. Der Prozessor beginnt zu rechnen und läuft

und läuft, jedoch wird man für die nächsten paar Millionen Jahre keine Ausgabe erhalten, denn solange würde die Berechnung dauern. Um das ganze etwas benutzerfreundlicher zu gestallten, kann man einen Thread dafür verwenden, ständig den aktuellen Stand der Berechnung anzuzeigen, und möglicherweise einen zweiten Thread für einen STOP-Button, damit man die Berechnung jederzeit unterbrechen kann.

Falls ein Thread, z.B.: auf eine Event, eine Benutzerinteraktion warten muss, kann er in der Zwischenzeit seine Ressourcen für andere Berechnungen verwenden, und es scheint, dass das Programm schneller läuft. Aber nicht immer tritt dieser Effekt auf. Wenn ein zwei Threads gleichzeitig mehrere Berechnungen machen, kann dies auf einer Multiprozessor-Maschine zwar von Vorteil sein, kann jedoch auf einem gängigen Heim-Computer (Einzelprozessor-Maschine) das Gegenteil bewirken – das ganze Programm wird langsamer, denn der Prozessor muss ständig zwischen beiden Threads hin und her schalten und verursacht somit zusätzlichen Aufwand.

2.2.Funktionsweise & Eigenschaften von Threads:

2.2.1. Erstellen und Starten:

Der Namensraum *System.Threading* bietet eine große Anzahl von Klassen, Interfaces und Funktionen mit denen es sehr leicht ist Applikationen mit Threads zu erstellen.

```
using System;
using System.Threading;

namespace ThreadHelloWorld
{
	class ThreadHelloWorldTest
	{
		static void ThreadEntry()
		{
			Console.WriteLine("Hello Threading World");
		}

		static void Main(string[] args)
		{
			Thread t = new Thread (new ThreadStart(ThreadEntry));
			t.Name = "Hello World Thread";
			t.Priority = ThreadPriority.AboveNormal;
			t.Start();
			t.Join();
		}
	}
}
```

In diesem Beispiel wird eine neue Instanz der Klasse Thread angelegt. Der Konstruktor-Aufruf verlangt einen einzigen Parameter von Typ *delegate.* Die Methode *public delegate void ThreadStart()* wird speziell für diesen Zweck von der

CLR zur Verfügung gestellt und verweist auf eine Methode, die den Eintrittspunkt des Threads darstellt und beim Starten des Threads aufgerufen wird. Die Methode die aufgerufen wird darf jedoch ***keinen Parameter*** enthalten und muss den Typ *void* zurückliefern. Wie allerdings jetzt die Daten an einen Thread übergeben werden können, wird im ***Kapitel 2.3*** behandelt.
Dadurch dass nur eine Instanz der Klasse erstellt wurde, muss der Thread explizit mit der Methode *Start()* gestartet werden. Weiters kann man auch Name und Priorität (5 Stufen – *Hoch, höher als normal, normal, niedriger als normal, Niedrig*) angegeben. Die Thread-Priorität jedoch sagt aus, wie der Thread vom Betriebssystem abgearbeitet wird. Standardmäßig ist die Priorität auf *Normal* eingestellt.

2.2.2. Vereinigen:

Man kann einen Thread dazu veranlassen, seine aktuelle Aufgabe zu unterbrechen und auf anderer Threads zu warten - dies wird als Vereinigung (*Join*) bezeichnet. Man kann z.B.: in der *Main()* für jeden Thread ein *Join()* aufrufen und anschließend eine Meldung „alle Threads beendet“ ausgeben.

2.2.3. Suspendieren:

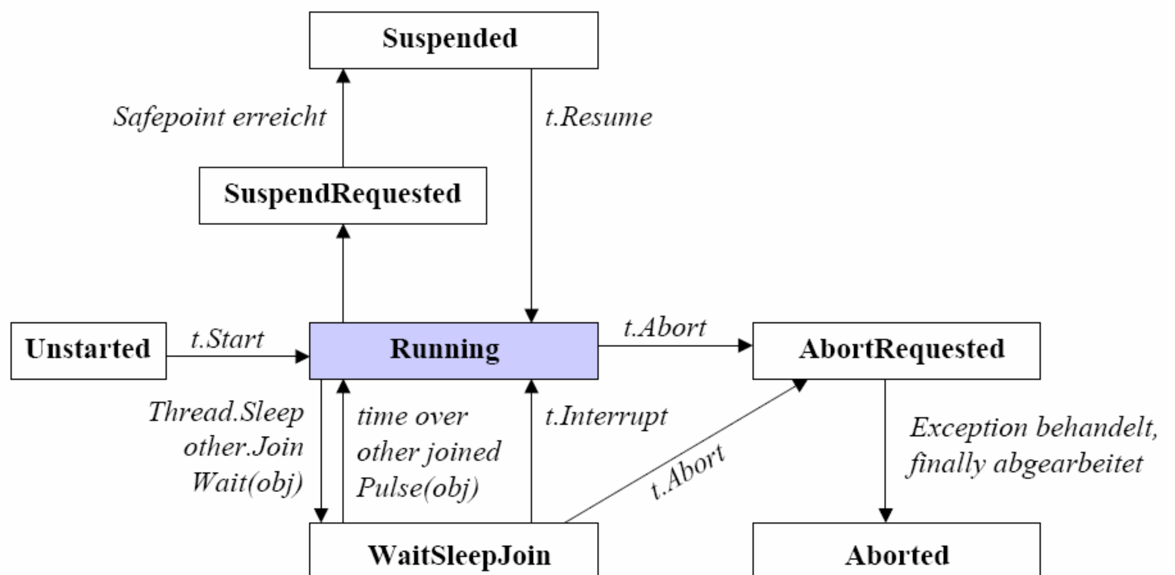
In Programmen werden oft Threads verwendet, bei denen es nicht nötigt ist, dass diese ständig aufgerufen werden, und so unnötig Ressourcen verbrauchen. Darum gibt es die Möglichkeit, Threads für kurze Zeit zu unterbrechen und auf Eis zu legen. Beispielsweise soll der Bildschirm für eine Überwachungsanlage nur alle 10 Sekunden aktualisiert werden. Somit wird der Thread nur einmal kurz für den Aktualisierungsvorgang eingeschalten und anschließend unterbrochen, und somit viel Ressourcen gespart. Die Klasse Thread bietet hierfür die Methode *Sleep( )* die eine Zeitangabe in Millisekunden erwartet.

2.2.4. Abbrechen:

Normalerweise verabschiedet sich ein Thread nach erledigter Arbeit. Aber oft ist es auch wichtig, besonders bei Echtzeitapplikationen, einen Thread vorzeitig zu unterbrechen. Dies kann mit der Funktion *Abort()* bewerkstelligt werden und löst eine ThreadAbortException aus. Falls man diese Exception abfängt, kann man im Catch-Block die verwendeten Ressourcen freigeben.

2.2.5. Zustandsdiagramm eines Threads:

Ein Thread kann wie ein Automat mit mehreren Zuständen verstanden werden. Mit den oben genannten Methoden kann man sich zwischen den einzelnen Zuständen bewegen. Folgende Grafik veranschaulicht diese Zustände in einem Zustandsdiagramm.



2.3. Datenübergabe an einen Thread

Es gibt drei Möglichkeiten Informationen/Daten an einen Thread zu übergeben. Die einfachste Lösung besteht darin, die benötigten Daten in **globale Daten** zu speichern.

- Auf diese globalen Daten kann ein Thread bzw. auch alle anderen Programmteile zugreifen, wobei man sich bewusst sein muss, dass wenn mehrere Threads diese Daten benötigen und auch verändern, die Daten synchronisiert werden müssen.

```
using System;
using System.Threading;

namespace ThreadHelloWorldStaticInformation
{
	class ThreadHelloWorldTest
	{
		static string message;
		static void ThreadEntry()
		{
			Console.WriteLine(message);
		}

		static void Main(string[] args)
		{
			message = "Hello World";
			Thread t = new Thread (new ThreadStart(ThreadEntry));
			t.Start();
			t.Join();
		}
	}
}
```

- Die zweite Möglichkeit besteht darin, dass man den „**Entrypoint**“ in eine eigene Klasse kapselt und somit diesem die gewünschten Daten mit übergeben kann. D.h.: Es wird eine eigene Klasse mit einem/mehreren ‘**Properties**“ erstellt, welche vor dem Thread-Start zugewiesen und manipuliert werden können. Der Thread verwendet anschließend diese Daten und ist auf keine Synchronisation mit einem anderen Thread angewiesen.

```
using System;
using System.Threading;

namespace ThreadHelloWorld
{
	class HelloWorld
	{
		private string message;
		public string Message
		{
			get
			{
				return message;
			}
			set
			{
				message = value;
			}
		}

		public void ThreadEntry()
		{
			Console.WriteLine(message);
		}
	}

	class DynamicThreadInformation
	{
		static void Main(string[] args)
		{
			HelloWorld first = new HelloWorld();
			first.Message = "Hello World";
			Thread t = new Thread (new
					ThreadStart(first.ThreadEntry));
			t.Start();
			t.Join();
		}
	}
}
```

- die dritte und letzte Möglichkeit, Daten einen Thread zu übergeben, besteht darin das ganze über **TLS** (thread local storage) abzuwickeln. Das folgende Beispiel zeigt, das die ganzen Daten sich immer nur auf den lokalen Thread beziehen.

```
using System;
using System.Threading;
```

```
namespace ThreadHelloWorld
{
	class ThreadLocalStorage
	{
		static LocalDataStoreSlot slot;

		static void ThreadEntry()
		{
			Console.WriteLine("The data in the slot is: {0}",
			                     (string)Thread.GetData(slot));
		}

		static void Main(string[] args)
		{
			String message = "Hello World";
			slot = Thread.AllocateDataSlot();
			Thread.SetData(slot,message);
			Thread t = new Thread (new ThreadStart(ThreadEntry));
			t.Start();
			t.Join();
		}
	}
}
```

2.4. Synchronisation

Wie man in den bereits gezeigten Beispielen sehen kann, ist das erstellen eines Threads nicht sehr schwer. Auch die Datenübergabe ist noch sehr einfach, wobei hier die ersten Probleme auftreten können – Die Verwaltung und Synchronisation von „Shared Data". Dies ist genau dann wichtig, wenn nur begrenzte Ressourcen zur Verfügung stehen und mehrere Threads diese benötigen. Ein klassisches Beispiel ist hierfür eine Tankstelle mit nur einer Zapfseule. Kommt man zu dieser Tankstelle und die Zapfseule wird bereits verwendet, kann man entweder weiterfahren, oder man warten geduldig mit anderen ein einer Reihe, die ebenfalls tanken wollen. Sobald die Zapfseule frei ist, kommt der nächste aus der Reihe dran.

Das folgende Beispiel zeigt, welche Probleme auftreten können wenn man **keine** Synchronisation bei mehreren Threads verwendet. Hier sollen 5 Threads hintereinander ein Fibonat'sche Zahl ausrechnen.

```
using System;
using System.Threading;

namespace MultiblethreadedQueue
{
	class Worker
	{
		private int Fib(int x)
		{
			return ((x<=1)?1:(Fib(x-1)+Fib(x-2)));
		}
```

```
            public void doWork()
            {
                  string item =
                    Convert.ToString(Thread.CurrentThread.Name) + " ";
                  for (int i =0; i < 10; i++)
                  {
                        Console.WriteLine("Thread {0} {1} {2}",
                                                item,i,Fib(30));
                  }
            }
      }

      class UnsynchronizedTest
      {

            static void Main(string[] args)
            {
                  Worker wq = new Worker();
                  Thread a = new Thread( new ThreadStart(wq.doWork));
                  a.Name = "a";
                 Thread b = new Thread( new ThreadStart(wq.doWork));
                  b.Name = "b";
                 Thread c = new Thread( new ThreadStart(wq.doWork));
                  c.Name = "c";
                 Thread d = new Thread( new ThreadStart(wq.doWork));
                  d.Name = "d";
                 Thread e = new Thread( new ThreadStart(wq.doWork));
                  e.Name = "e";

                 a.Start();
                 b.Start();
                 c.Start();
                 d.Start();
                 e.Start();
            }
      }
}
```

Das Ergebnis ist nicht sehr Zufrieden stellend. Dadurch dass keine Synchronisation eingesetzt wird, werden je nach dem, wie die Threads CPU-Zeit zugewiesen bekommen, aufgerufen. Unabhängig ob ein Thread bereits sich in der For-Schleife befindet oder nicht, wird ein Output im Konsolenfenster getätigt. Es kann somit vorkommen, dass gewisse Threads sehr oft CPU-Zeit bekommen. Das gleiche gilt auch für den umgekehrten Fall: es kann Threads geben welche nur ganze selten bzw. niemals CPU-Zeit bekommen und so niemals ihre Berechnung durchführen können.

Dies ist ein noch sehr *einfaches Beispiel* gewesen, aber man kann sich demnach auch vorstellen dass solche Probleme weit reichende Folgen haben können. Z.B.: Verwendung von Threads für die Lagerverwaltung. Ein Kunde sucht nach einem bestimmten Produkt und möchte dieses falls vorhanden auch kaufen. Der erste Thread liest den Inhalt und stellt fest, dass noch ein Exemplar im Lager vorhanden ist und überprüft die eingegeben Kundeninformationen. Währenddessen sucht ein weiterer Thread nach dem gleichen Produkt. Dadurch dass der erste Thread den Datensatz noch nicht aktualisiert hat, findet der zweite Thread den gleichen Datensatz und beginnt ebenfalls mit dem Verkaufsvorgang. Beide Threads setzen ihre Aktivität fort und verkaufen das gleiche Produkt zweimal.
Deshalb ist es wichtig alle Ressourcen welche mehrmals benötigt werden zu synchronisiere.

Um solche Ergebnisse entgegenzuwirken, gibt es mehrere Möglichkeiten Daten zu synchronisieren.

2.4.1 Interlocked:

Das Inkrementieren und Dekrementieren eines Wertes ist ein derart verbreitetes Programmiermuster und eines, das so häufig einen Schutz durch Synchronisation benötigt, dass C# speziell für diesen Zweck eine eigene Klasse integriert hat: *Interlocked*. Diese Klasse verfügt über zwei Methoden, *Increment* und *Decrement*, die nicht nur einen Wert erhöhen oder erniedrigen, sondern auch synchronisieren.

```
public void doWork()
{
   try
   {
      while (counter < 100)
```

```
        {
           int temp = Interlocked.Increment(ref counter);
           Thread.Sleep(1);
           Console.WriteLine("Thread {0}. Incrementer: {1}",
                             Thread.CurrentThread.Name,temp);

        }
     }
  }
```

Wenn es darum geht, einen Wert hoch- oder herunterzuzählen, kann man ruhig das *Interlocked*-Objekt verwenden. Geht es aber darum, komplexere Objekte zu synchronisieren muss der Zugriff anders geregelt werden.

2.4.2 Monitor and Lock:

Die **Monitor**-Klasse steuert den Zugriff auf Objekte, indem Sie einem einzelnen Thread eine Sperre für ein Objekt zuteilt. Mit Objektsperren kann der Zugriff auf einen Codeblock eingeschränkt werden, der allgemein kritischer Abschnitt genannt wird. Während ein Thread die Sperre für ein Objekt besitzt, kann kein anderer Thread diese Sperre erhalten.

Um beim vorhergehenden Beispiel das gewünschte Ergebnis zu erzielen, muss man nur in der Funktion doWork() ganz am Anfang und am Schluss jeweils die entsprechende Monitor-Funktion einfügen. d.h.: wird diese Funktion aufgerufen, so ist diese solange für den aktuellen Thread besperrt, bis die Funktion abgearbeitet ist.

```
public void doWork()
{
     Monitor.Enter(this);
     string item =
        Convert.ToString(Thread.CurrentThread.Name) + " ";
     for (int i =0; i < 10; i++)
     {
          Console.WriteLine("Thread {0} {1} {2}",
                            item,i,Fib(30));
     }
     Monitor.Exit(this);
}
```

Man kann auch statt Monitor die Funktion Lock verwenden d.h.:

```
Monitor.Enter(this)
	Console.WriteLine("Hello World");
Monitor.Exit(this);

bzw.

try
{
	Monitor.Enter(this);
	Console.WriteLine("Hello World");
}
Finally
{
	Monitor.Exit(this)
}


entspricht:

lock(this)
{
	Console.WriteLine("Hello World");
}
```

2.4.3 Race Condition & Deadlocks:

Besonders bei komplexen Programmen ist die Synchronisation von Threads eine verzwickte Angelegenheit und man muss sich mit den üblichen Problemen der Thread-Syncrhonisation auseinandersetzen, zu denen Race Condition und Deadlocks gehören.

- **Race Condition:**
  Eine Race Condition liegt dann vor, wenn die zeitlich unkoordinierte Arbeit zweier Threads den erfolgreichen Ablauf eines Programms gefährdet. z.B.: Es werden zwei Threads verwendet – einer ist dafür zuständig eine Datei zu öffnen, und der andere ist dafür zuständig, in diese Datei zu schreiben. Nun muss der zweite Thread so überwacht werden dass mit Sicherheit der erste Thread die Datei schon geöffnet hat. Falls dies nicht der Fall sein sollte, kann im Grunde genommen das Programm funktionieren, sofern Thread eins immer vor Thread zwei startet. Aber es kann auch vorkommen, dass Thread zwei vor Thread eins startet und dann will dieser in eine Datei schreiben, obwohl diese nicht geöffnet ist und somit eine Exception ausgelöst wird. Das ganze Programm bricht zusammen. So etwas wird als Race Condition bezeichnet und kann ausgesprochen schwer zu debuggen sein. Um solchen Problemen vorzubeugen sollte man die oben genannten Methoden (Monitor & Lock) verwenden.

```
public void ThreadProcA(int index)
{
   lock(GlobalList)
   {
      GlobalList.Items.RemoveAt(index);
   }
}

public void ThreadProcB()
{
   For (int x =0; x < GlobalList.Items.Count; x++)
   {
     //… mehrere zeitaufwändige Operationen
     lock(GlobalList)
     {
         GlobalList.Items[x] = … // irgendeine Manipulation
     }
   }
}
```

In diesem Beispiel ist Thread A für das Anfügen von Elementen an eine Liste und Löschen von Elementen aus einer Liste zuständig. Thread B für das Bearbeiten der Listenelemente. Theoretisch sollte das Beispiel funktionieren, jedoch in der Praxis nicht. Das Lock-Objekt ist im Thread B viel zu eng angesetzt worden. Es kann vorkommen, dass Thread B auf ein leeres Listenelement gelangen kann. Bzw. es kann ein und dasselbe Element auch zweimal oder öfters bearbeitet werden. Lösung: die Lock-Anweisung muss noch vor die For-Schleife gezogen werden, um richtig funktionieren zu können.

- **Deadlocks:**
  Wenn man darauf wartet, dass eine Ressource frei wird, riskiert man einen Deadlock. Bei einem Deadlock warten zwei oder mehrere Threads aufeinander, und keiner kann sich aus dieser Situation befreien.

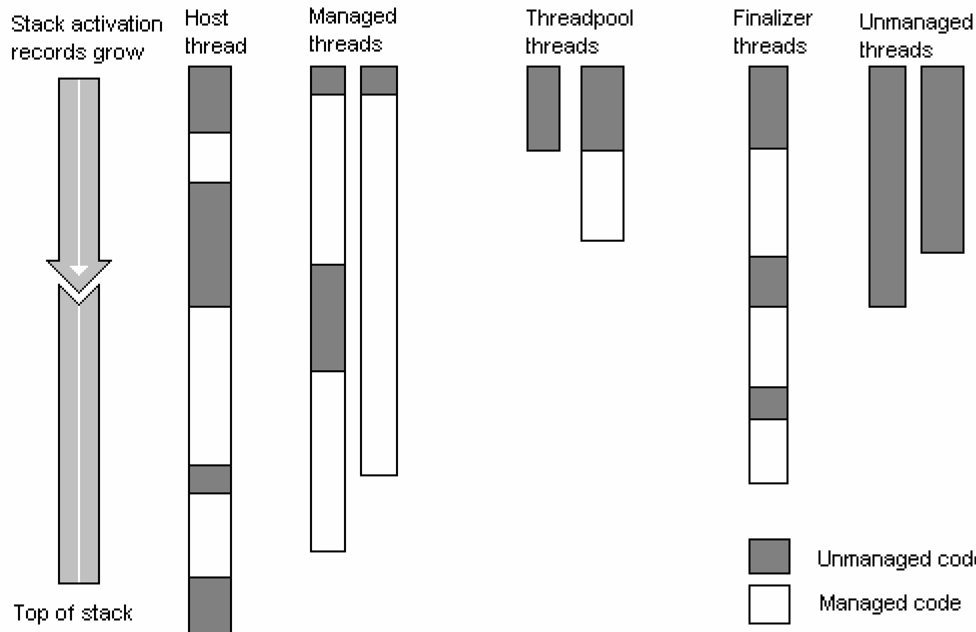
```
public void ThreadProcA()
{
   lock(RefA)
   {
     //…
     lock(RefB)
     {
        //…
     }
   }
}

public void ThreadProcB()
{
   lock(RefB)
   {
     //…
     lock(RefA)
     {
        //…
     }
   }
}
```

Die ist ein Beispiel, wie zwei Threads auf die jeweilige Freigabe des anderen warten und das bis in alle Ewigkeit. Der Ausweg ist allerdings erheblich schneller beschrieben als das Problem selbst. Wenn zwei Threads mehrere sperren anfordern, dann sollten sie das grundsätzlich in derselben Reihenfolge tun – im gegeben Beispiel muss Thread B also genauso wie Thread zuerst RefA und erst dann RefB anfordern.

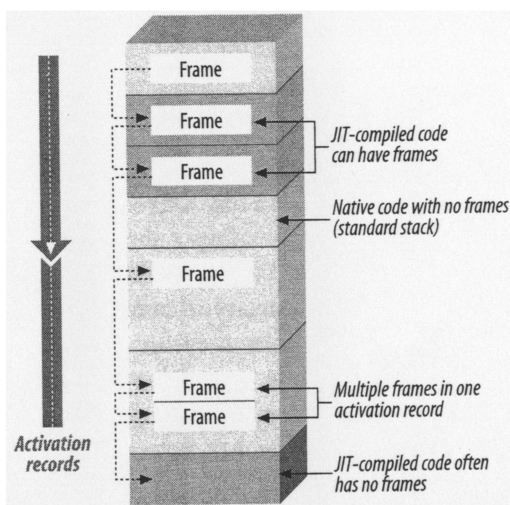
2.5. Verwaltung:

Threads in der CLI haben eine sehr wichtige Rolle – sie verwalten in einer primären Datenstruktur Informationen des Ausführungscodes. Z.B.: Buchhaltungsinformationen, Sicherheitsanmerkungen, Garbage-Collection-Eintragungen, Variablen usw. Innerhalb der Execution Engine sind Threads aufbauend auf **PAL-Threads** (sind nicht verwaltete Threads (werden nicht vom Garbage-Collector berücksichtigt) implementiert.
Die Execution Engine verwaltet „managed“ und „unmanaged code“ nahezu gleich. Der Stack ist also eine Mixtur aus beiden Arten von Code:



Bevor JIT-Compiled Code ausgeführt wird, werden zwei wichtige Operationen am Stack durchgeführt. Einerseits wird ein Exception-Handler für den „managed code“ Bereich eingefügt und andererseits werden mehrere Frames, welche Ausführungsinformationen, Buchhaltungsinformationen und Begrenzungsinformationen beinhalten, eingefügt.
Der SSCLI-Stack ist, wenn man ihn genauer betrachtet, eine Mixtur aus execution-engine-frames, exception-handling-frames und Aktivierungssätzen, wobei die einzelnen Frames in späterer Folge weiters unterteilt und entsprechend interpretiert werden.

Wenn man durch den Stack (dieser beinhaltet „managed“ und „unmanaged-code“) laufen möchte, kann dies mit Hilfe von den einzelnen Aktivierungssätzen (activation-records) und dem Wissen der Exectuion-Engine über den Aufbau der JIT-compiled-methodes geschehen.



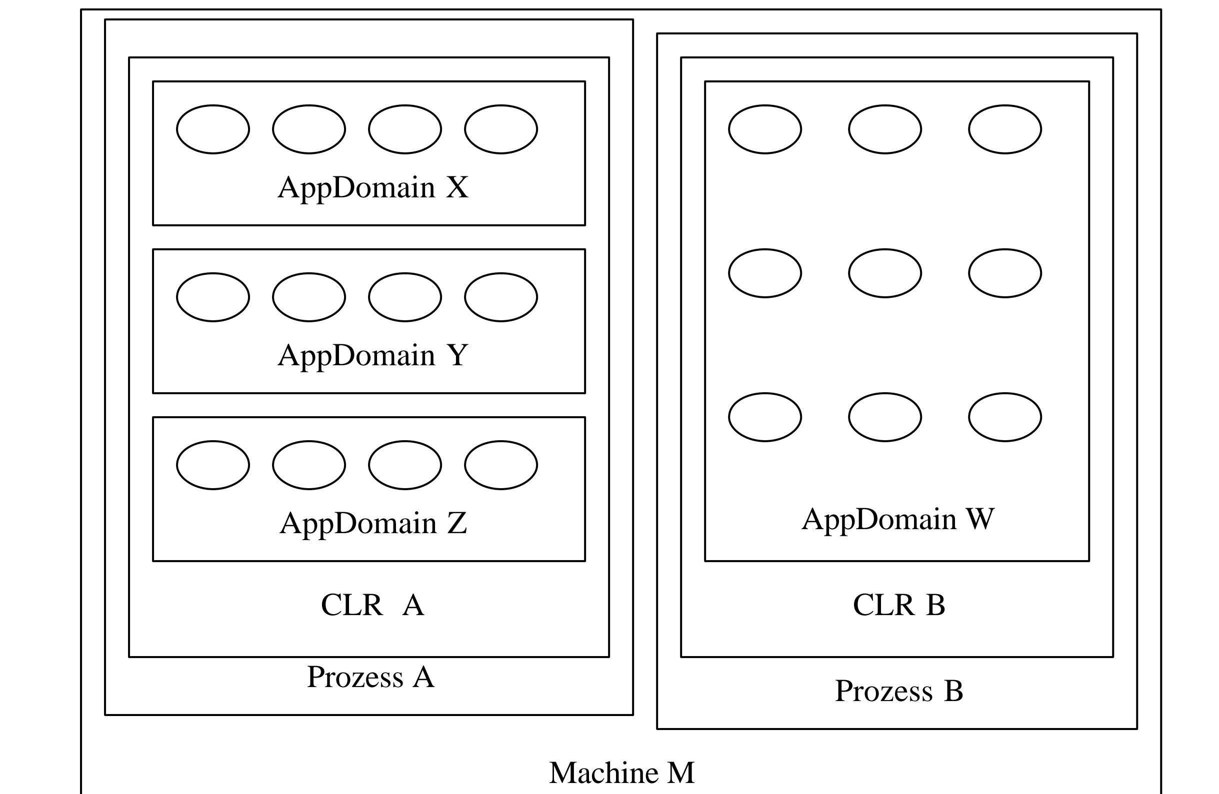
Threads sind die primäre Abstraktion für Ausführung und Parallelität in der CLI. Die CLI erlaubt die unabhängige Ausführung von Programmteilen und in Folge dessen auch deren Kommunikation und Serialisierung/Synchronisation.

3. Applikation-Domain (AppDomain)

3.1.Einführung

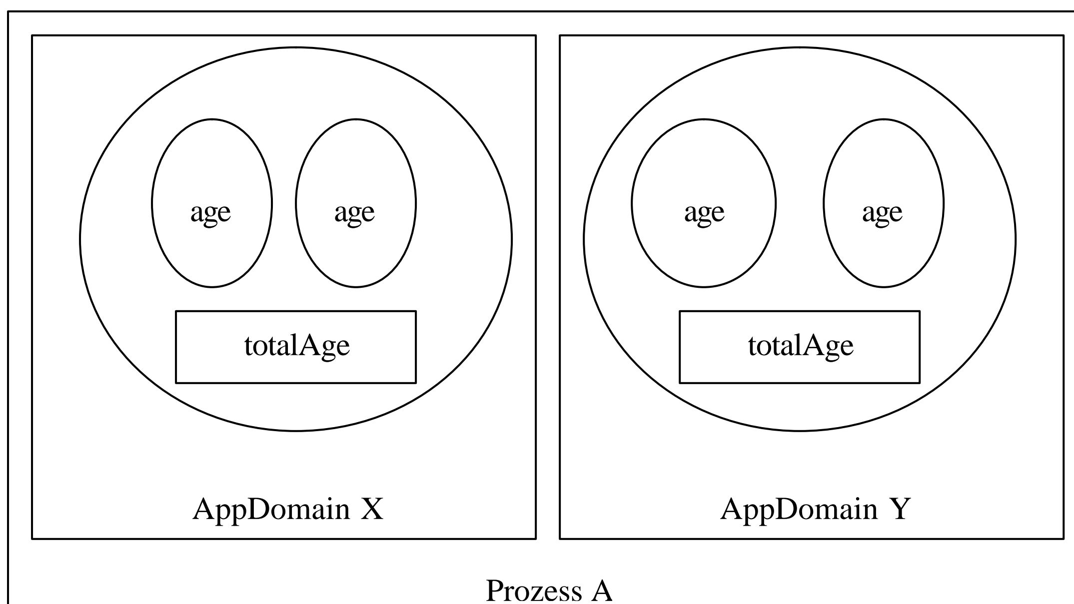
Ganz grob betrachtet, ist ein Prozess nichts weiter als eine laufende Applikation. D.h.: jedes Programm somit auch jede .NET-Applikation verwendet ihren eigenen Prozess. Startet man z.B.: Visual Studio und Internet-Explorer, so kann man im Task-Manager zwei Prozesse erkenne. Ein solcher Prozess verwendet mindestens einen bzw. mehrere Applikation Domains (***AppDomain***). Applikation-Domains können als eine Art Prozess gesehen werden, mit dem kleinen Unterschied, das diese weniger Ressourcen verbrauchen und somit auch als „leichtgewichtiger" Prozess bezeichnet werden. Weiters sind sie sicherer und vielseitiger, indem sie eine Anwendung fehlertoleranter machen. Um das ganze etwas besser zu veranschaulichen muss man sich folgende Situation vorstellen. Man erstellt in einem zweiten AppDomain ein Objekt. Dieses Objekt stützt ab und eliminiert folglich auch den dazugehörigen AppDomain, aber **NICHT** das gesamte Programm! Dies ist besonders in Echtzeitsystemen bzw. Betriebssystemen oder auch bei Applikationen welche viel mit Plug-In's arbeiten wichtig.

Jeder Prozess hat am Anfang eine Standard-AppDomain und kommt im Normalfall auch damit aus. Falls man mit dieser einen Domain nicht auskommen sollte, so können beliebig viele AppDomains hinzugefügt werden. Wird in einer Applikation ein Klassenbibliothek verwendet, welche von einem anderen Programmierer stammt, und man vertraut dieser nicht, ist es am Besten diese Klasse in einer eigenen Domain zu isolieren. Denn falls nun eine Methode aus der Klassenbibliothek fehlerhaft ist und abstürzt, ist nur eine Domain betroffen und nicht das gesamte Programm. z.B.: Beim IIS (Internet Information Server) liegt es nahe für jedes Plug-In eine eigene Domain zu verwenden.



Typen und Objekte existieren nur in einem AppDomain. Falls andere AppDomains einen Typen, welcher bereits in einer anderen AppDomain existiert, verwenden wollen, wird dieser kopiert und in den eigenen AppDomain gestellt.

Wie bei den Prozessen, existiert bei den AppDomains auch ein Zugehörigkeitsschema. Alle Ressourcen (Module, Assemblies, Typen,…), welche benötigt werden, werden solange im Speicher gehalten, solange wie ein AppDomain existiert. Deshalb kann man nur ein Modul oder ein Assemblie aus dem Speicher laden, indem man das entsprechende AppDomain beendet.




```
public class Person {
{
   Static int totalAge;
   int age;
}
```

3.2.Unterschied zwischen AppDomain und Thread

Ein AppDomain ist dem Thread sehr ähnlich, muss aber unterschieden werden. Ein AppDomain existiert für die Dauer der Applikation, ein Thread hingegen existiert nur für eine bestimmte Zeit innerhalb eines AppDomains. Ein Thread kann feststellen, in welcher AppDomain dieser erstellt und aufgerufen wurde, wobei innerhalb eines AppDomain wieder beliebig viele Threads laufen können. Weiters wird ein AppDomain von der Execution-Engine erzeugt.

3.3.Verwendung:

Eine AppDomain kann mittels der Methode *CreateDomain()* erzeugt werden. Sobald ein Domain-Objekt erzeugt wurde, kann mit Hilfe von *CreateInstance()* Interfaces, Klasseninstancen, usw. erzeugt werden.

Signatur:

```
public ObjectHandle CreateInstance(
   string assemblyName,
   string typeName,
   bool ignoreCase,
   BindingFlags bindingAttr,
   Binder binder,
   object[] args,
   CultureInfo culture,
   object[] activationAttributes,
   Evidence securityAttributes
);
```

Diese Signatur kann wie folgt verwendet werden:

```
…
AppDomain aDomain = AppDomain.CreateDomain("My Domain");
…
ObjectHandle objHandle = aDomain.CreateInstance(
   "ProgCSharp",                    // der Assembly Name
   "ProgCSharp.Shape",              // Typ-Name mit Namensraum
   False,                           // Groß-Kleinschreibung ign.
   System.Reflection.BindingFlags.CreateInstance,
   null,                            // Binder
   new object[] {3,5},              // Argumente
   null,                            // Culture
   null,                            // Aktivierungsattribute
   null                             // Sicherheitsattribure
);
…
```

Der erste Parameter „ProgCSharp“ ist der Name der Assembly. „ProgCSharp.Shape“ ist der Klassenname, wobei der Klassenname durch den Namensraum voll qualifiziert sein. Ein Binder ist ein Objekt, das das dynamische Binden einer Assembly während der Laufzeit erlaubt. Standardmäßig wird ein Standardbinder verwendet und der Parameter kann somit als null übergeben werden. Die Bindingsflags helfen dabei, das Verhalten des Binders genau abzustimmen.

```
using System;

public class MyApp {
```

```
   public static int main(string[] argv) {
      // create Domain
      AppDomain child = AppDomain.CreateDomain("childapp");
      // execute
      int r = child.ExecuteAssembly("yourapp.exe",null,argv);
      //unload domain
      AppDomain.Unload(child);
      // return result
      Return r;
   }
}
…
```

Das oben angeführte Beispiel startet eine externe EXE-Datei in einem separaten AppDomain. Falls das externe Programm einen Fehler hat und abstürzen sollte, so ist das eigentliche Hauptprogramm nicht davon betroffen.

3.4.AppDomain Events:

Wie bei vielen Objekten in .Net besitzt auch der Typ AppDomain Events welche es erlauben viele interessante Tätigkeiten zur Laufzeit durchzuführen.

Event Name	EventArg Properties	Description
AssemblyLoad	Assembly LoadedAssembly	Assembly has just been successfully loaded
AssemblyResolve	string Name	Assembly reference cannot be resolved
TypeResolve	string Name	Type reference cannot be resolved
RessourceResolve	string Name	Ressource reference cannot be resolved
DomainUnload	None	Domain is about to be unloaded
ProcessExit	None	Process is about to shut down
UnhandledException	bool is Terminating, object ExceptionObject	Exception escaped thread-specific handlers

Das Event DomainUnload wird aufgerufen, wenn ein AppDomain beendet wird. UnhandledException wird nur dann aufgerufen, wenn im Programm / AppDomain eine Exception auftritt und nicht behandelt wird, was jedoch auf eine unsaubere Art der Programmierung hinweist. Falls eine Exc eption geworfen wird uns es befindet sich kein Exception-Handler auf dem Stack, so verwendet die CLR ihren eigenen ExceptionHandler. Dieser Exception-Handler ist die letzte Möglichkeit für das Programm den Fehler „entsprechend“ zu korrigieren und z.B.: eine Fehlermeldung auszugeben.

3.5.Code Management:

Wie bereits schon erwähnt, besitzt jeder AppDomain eine eigene private Kopie von den statischen Daten eines Typs. Von unterschiedlichen Faktoren abhängig, benötigt jedoch dieser AppDomein eine bzw. keine private Kopie vom Ausführungscode des Typen.
Kurz gesagt, AppDomains können die Arbeitsweise des JIT-Compilers beeinflussen und sogar auch festlegen wie dieser arbeiten soll. Wenn alle AppDomains innerhalb eines Processes nur Maschinencode beinhalten, wird der Arbeitsaufwand verringert, jedoch der Zugriff auf statische Felder eines Typen wird möglicherweise etwas langsamer sein. Hingegen, wenn die CLR den Maschinencode für jeden AppDomain erstellt, ist der Zugriff auf die statischen Felder sehr schnell, aber der Arbeitsaufwand ist dementsprechend auch sehr aufwendig. Wie man hier sieht, besitzt der Programmierer die Möglichkeit festzulegen, wie man JIT-compiled-code verwaltet.

SingleDomain legt fest, dass jeder Prozess nur einen AppDomain besitzt, und somit wird jeder Code separat JIT-compiled. Der Zugriff auf die statischen Felder ist entsprechend schnell und es besteht kein zusätzlicher Arbeitsaufwand, da sowieso nur ein AppDomain existiert.

MultiDomain besagt, dass ein Prozess mehrere AppDomains beinhalten kann und somit jeder AppDomain eine eigene Kopie von den Typen besitzt. Weiters wird versucht die Speichernutzung so optimal wie möglich zu gestallten, das sich jedoch auf die Zugriffsgeschwindigkeit auf statische Felder auswirkt.

C# SourceCode

```
class Bob {
   static int x = 0;
   static int y = 0;
   static void Method() {
      x+=10;
      y-=10;
   }
}
```

LoaderOptimization.SingleDomain

```
push ebp
move sbp, esp
   add dword ptr ds:[3e5110h],
   0Ah
   Add dword ptr ds:[3e5114h],
   0FFFFFFF6h
move ebp, esp
pop ebp
ret
```

LoaderOptimization.MultiDomain

```
push ebp
move sbp, esp
   move ecx,588h
   call GetCurrentDomainData; loead ptr to domain-block from TLS
                            ; into eax
```

```
    move edx, eax
    add dword ptr [etx], 0Ah
    add dword ptr [edx+4], 0FFFFFFF6h
move ebp, esp
pop ebp
ret
```

z.B.: alle Arbeitsprozess in ASP.Net arbeiten mit der MultiDomainHost-Option.

4. Literaturliste

[BoS03] Don Box, Chris Sells: Essential .NET, The Common Language Runtime, Addison-Wesley 2003

[Bur02] Kevin Burton: .NET Common Language Runtime Unleashed, Sams Publishing 2002

[SNS03] Dave Stutz, Ted Neward, Geoff Shilling: Shared Source CLI Essentials. O'Reilly 2003

[JL02] Jesse Liberty: Programming C# 2nd Edition, O'Reilly & Associates Inc. 2002

[Komp02] Arne Schäpers,Rudolf Huttary,Dieter Bremes: C# Windows- und Web-Programmierung mit Visual Stuido .NET, Markt+Technik Verlag 2002