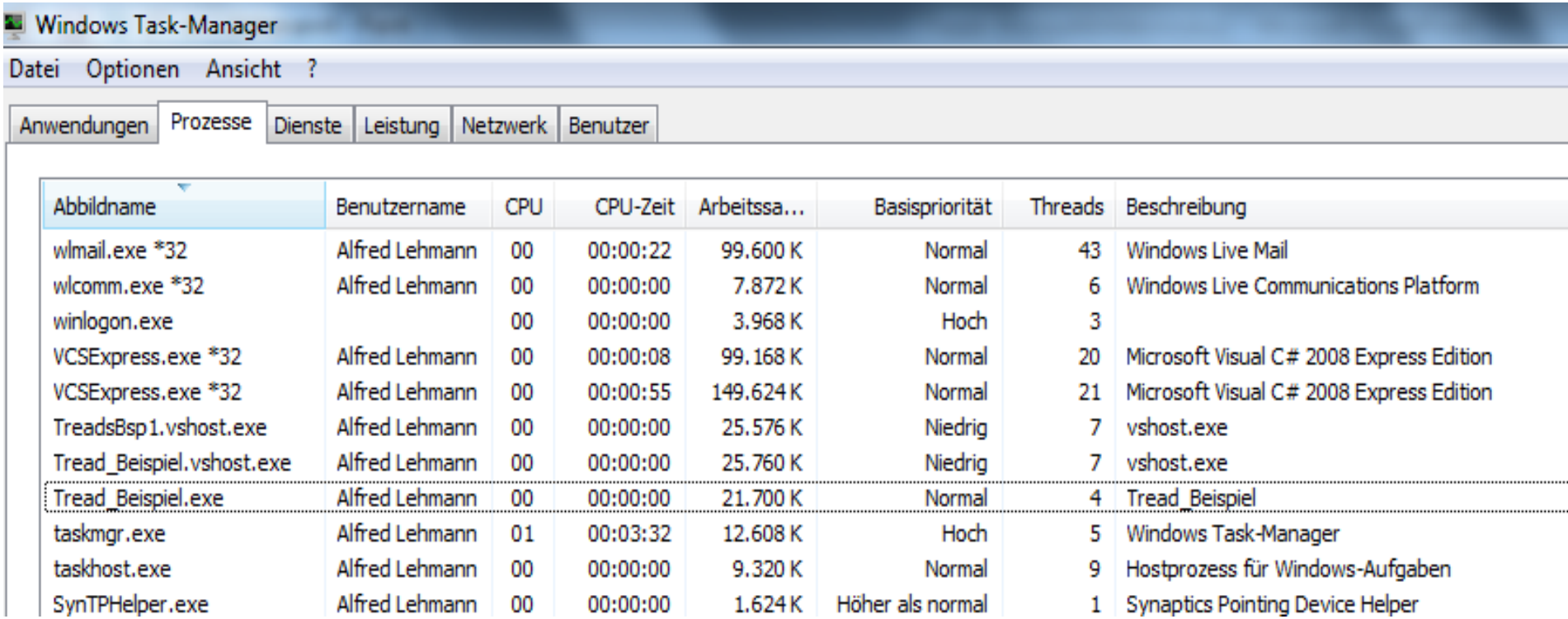
Multitasking vs. Multithreading

Prozesse und Threads stellen in Windows fundamentale Konzepte dar und tangieren viele Komponenten des Betriebssystems.

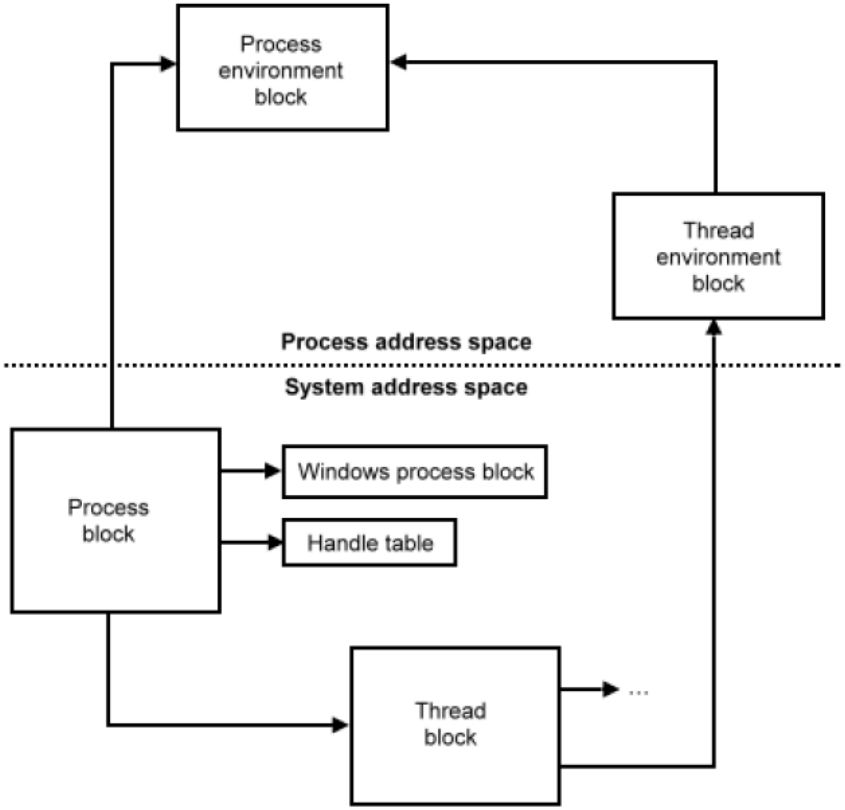
- **Multitasking** bedeutet, das mehrere Anwendungen Parallel ausgeführt werden können. Auf einem Prozessorkern kann immer nur ein Tasks ausgeführt werden, jedoch durch schnelles Wechseln der aktiven Anwendung auf dem Kern entsteht eine Scheinparallelität.
- **Multithreading** bedeutet, das mehrere Threads innerhalb eines Prozesses parallel ausgeführt werden können, damit ein Prozess mehrere Dinge gleichzeitig ausführen kann.
- Moderne Betriebssysteme verfügen dazu über einen **Prozessmanager**, der laufende Programme und Systemprozesse verwaltet.
- Die Grundlage bilden die Datenstrukturen, die Windows intern verwendet, um einen Prozess oder Thread zu verwalten.

Multitasking vs. Multithreading



Abbildname	Benutzername	CPU	CPU-Zeit	Arbeitssa...	Basispriorität	Threads	Beschreibung
wlmail.exe *32	Alfred Lehmann	00	00:00:22	99.600 K	Normal	43	Windows Live Mail
wlcomm.exe *32	Alfred Lehmann	00	00:00:00	7.872 K	Normal	6	Windows Live Communications Platform
winlogon.exe		00	00:00:00	3.968 K	Hoch	3	
VCSExpress.exe *32	Alfred Lehmann	00	00:00:08	99.168 K	Normal	20	Microsoft Visual C# 2008 Express Edition
VCSExpress.exe *32	Alfred Lehmann	00	00:00:55	149.624 K	Normal	21	Microsoft Visual C# 2008 Express Edition
TreadsBsp1.vshost.exe	Alfred Lehmann	00	00:00:00	25.576 K	Niedrig	7	vshost.exe
Tread_Beispiel.vshost.exe	Alfred Lehmann	00	00:00:00	25.760 K	Niedrig	7	vshost.exe
Tread_Beispiel.exe	Alfred Lehmann	00	00:00:00	21.700 K	Normal	4	Tread_Beispiel
taskmgr.exe	Alfred Lehmann	01	00:03:32	12.608 K	Hoch	5	Windows Task-Manager
taskhost.exe	Alfred Lehmann	00	00:00:00	9.320 K	Normal	9	Hostprozess für Windows-Aufgaben
SynTPHelper.exe	Alfred Lehmann	00	00:00:00	1.624 K	Höher als normal	1	Synaptics Pointing Device Helper

Multitasking vs. Multithreading

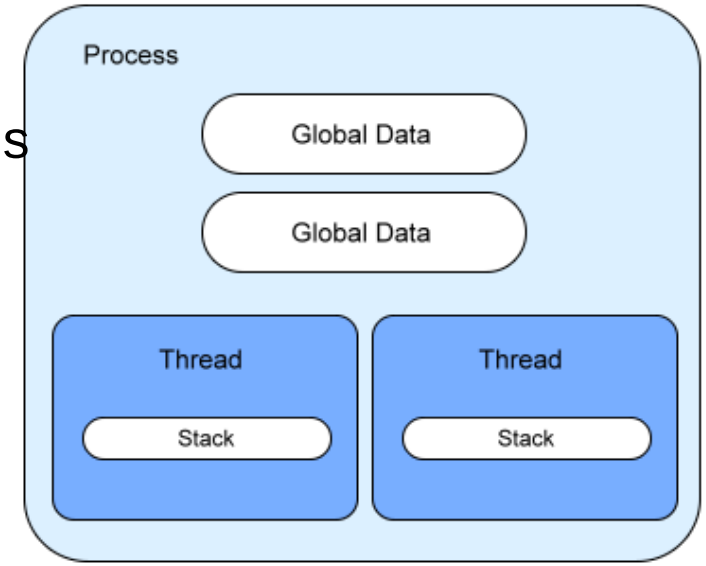


Quelle: http://www.codeplanet.eu/tutorials/csharp/64-multithreading-in-csharp.html

Definition – Thread:

- Ein Prozess besitzt in der Regel mindestens einen Thread, der den Code ausführt.

- Ein Thread bezeichnet einen Ausführungsstrang oder eine sequentielle Ausführungsreihenfolge in der Abarbeitung eines Programms.



Free Threading

- Ein Thread ist Teil eines Prozesses und teilt sich mit den anderen vorhandenen Threads (multithreading) des zugehörigen Prozesses eine Reihe von Betriebsmitteln, nämlich das Codesegment, das Datensegment und die verwendeten Dateideskriptoren.

- Ein Thread ist in Windows die kleinste ausführbare Einheit.

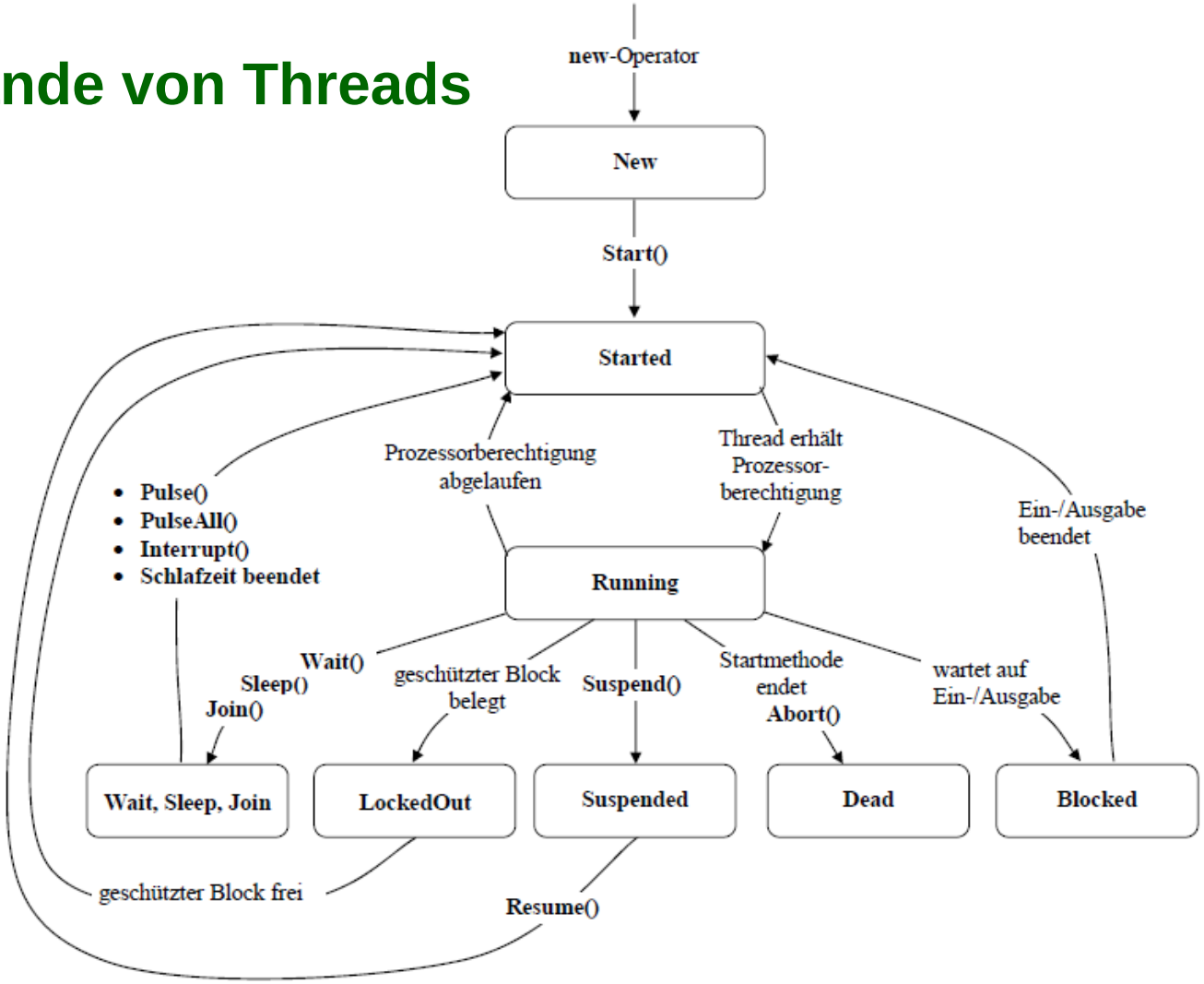
Vorteile – Threads:

- Durchführen zeitaufwendiger Operationen in einem parallelen Ausführungsstrang.
- Gute Skalierung auf Mehrkern-Prozessor-Systemen. Jeder Thread läuft auf einem Prozessorkern und nutzt die vorhandenen Ressourcen effizient aus.
- Nicht blockierende Kommunikation über ein Netzwerk, mit einem Webserver und mit einer Datenbank.
- Unterscheiden von Aufgaben nach unterschiedlicher Priorität. Ein Thread mit hoher Priorität übernimmt beispielsweise Aufgaben mit hoher Dringlichkeitsstufe, während ein Thread mit niedriger Priorität andere Aufgaben ausführt.
- Aufrechterhalten der Benutzeroberflächen-Reaktivität, während Hintergrundaufgaben bearbeitet werden.

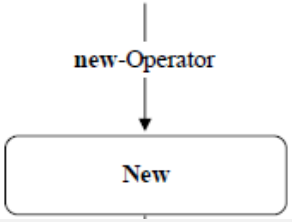
Nachteile – Threads:

- Das Steuern der Codeausführung mit vielen Threads ist sehr komplex und kann viele Fehler verursachen.
- Mehrere Threads müssen synchronisiert werden und die Fehlersuche (Debug) gestaltet sich schwieriger.
- Eine große Anzahl von Threads nimmt einen beträchtlichen Teil der CPU-Zeit in Anspruch.
- Das Betriebssystem muss sehr oft zwischen den Threads hin- und herschalten, was zu mehr Overhead führt.
- Wenn sich die meisten der aktuellen Threads in einem Prozess befinden, werden Threads in anderen Prozessen weniger häufig in den Ablaufplan aufgenommen.
- Das Betriebssystem nimmt Arbeitsspeicher für die Kontextinformationen in Anspruch, die für Prozesse, AppDomain-Objekte und Threads erforderlich sind. Deshalb ist die Anzahl der Prozesse, AppDomain-Objekte und Threads, die erstellt werden können, durch den verfügbaren Arbeitsspeicher begrenzt.
- Das Löschen von Threads erfordert Kenntnisse über mögliche Auswirkungen auf das Programm und deren Behandlung.

Zustände von Threads



Zustände von Threads



Threads erzeugen:

Thread t1 = new Thread(Incrementer);

Der Thread befindet sich danach im Zustand “unstarted (New)”
Incrementer ist hier ein Delegate auf einer Methode.

Die Methode “Incrementer”:

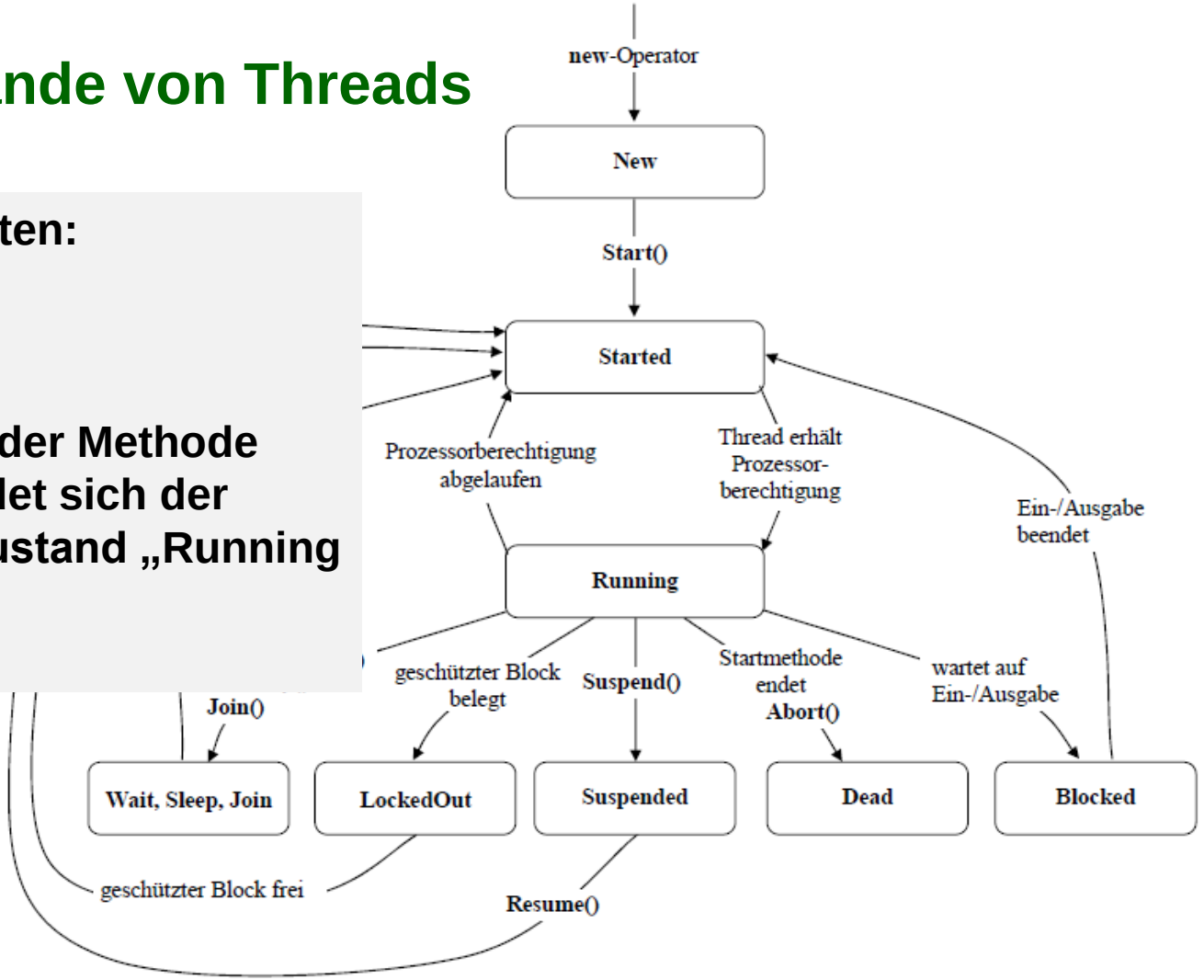
```
public static void Incrementer()
{
    for (int i = 0; i < 100; i++)
    {
        Console.WriteLine("Incrementer: {0}", i);
    }
}
```

Zustände von Threads

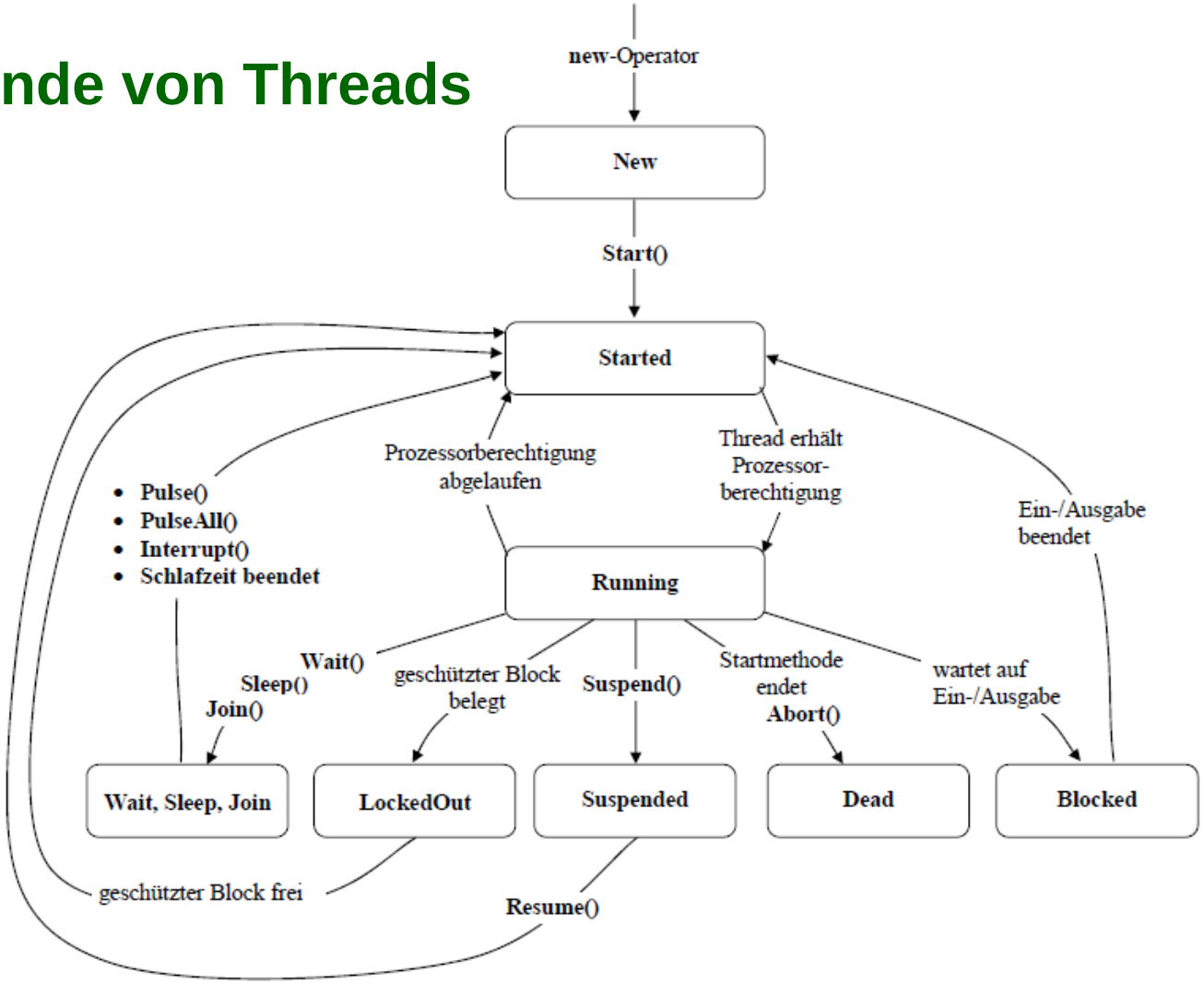
Threads starten:

t1.Start();

Nach Aufruf der Methode Start() befindet sich der Thread im Zustand „Running (Started)“.



Zustände von Threads

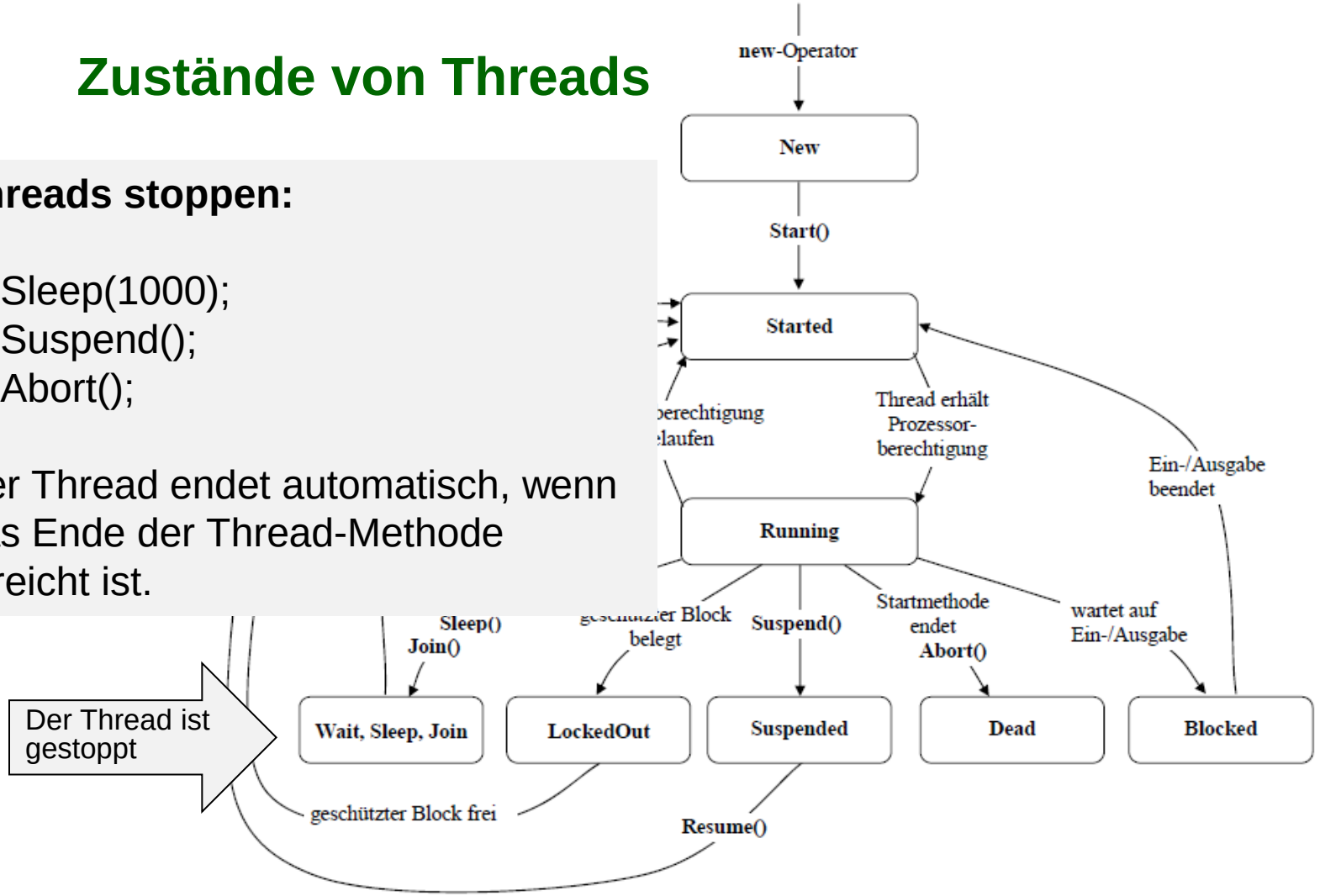


Zustände von Threads

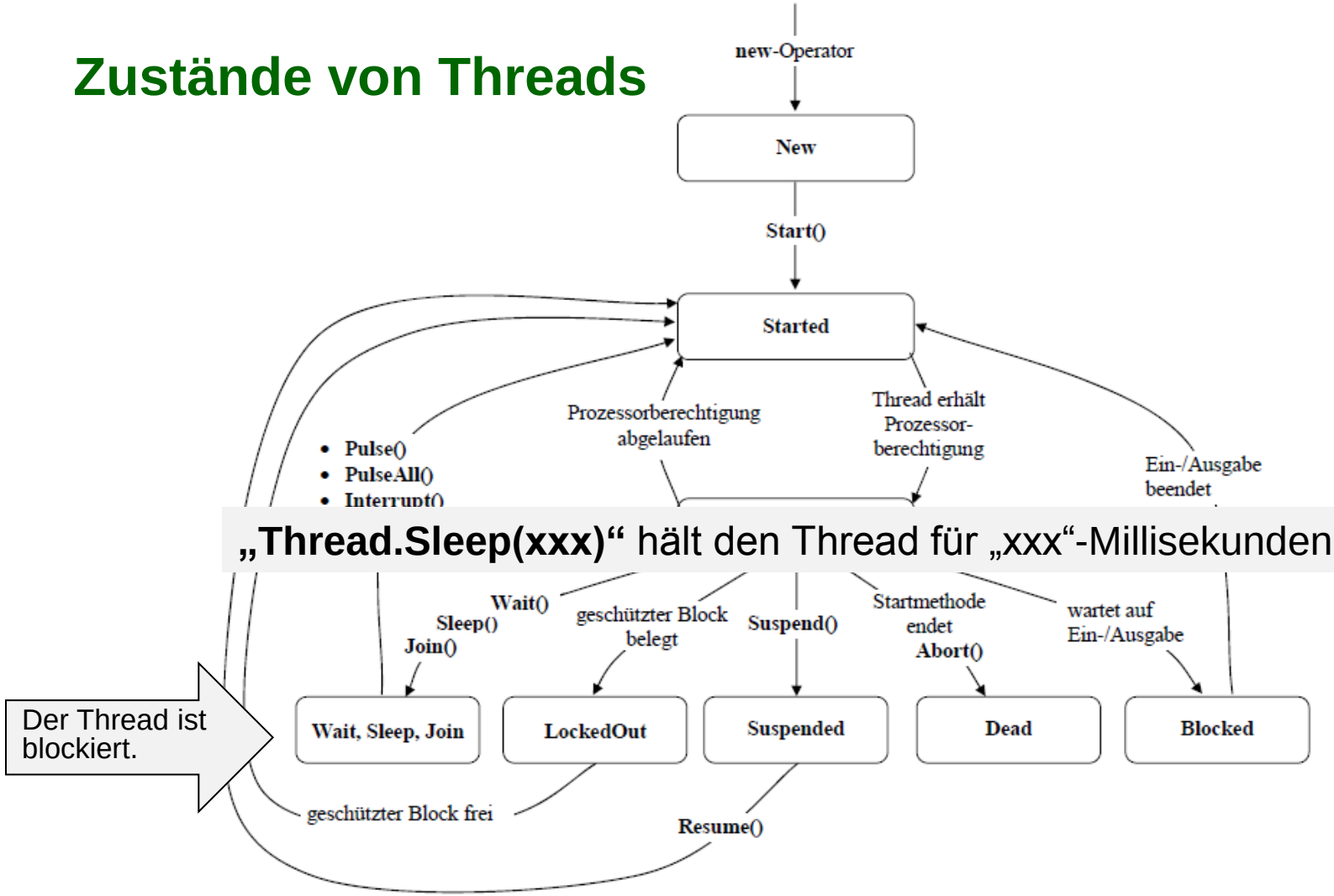
Threads stoppen:

t1.Sleep(1000);
t1.Suspend();
t1.Abort();

Der Thread endet automatisch, wenn das Ende der Thread-Methode erreicht ist.



Zustände von Threads



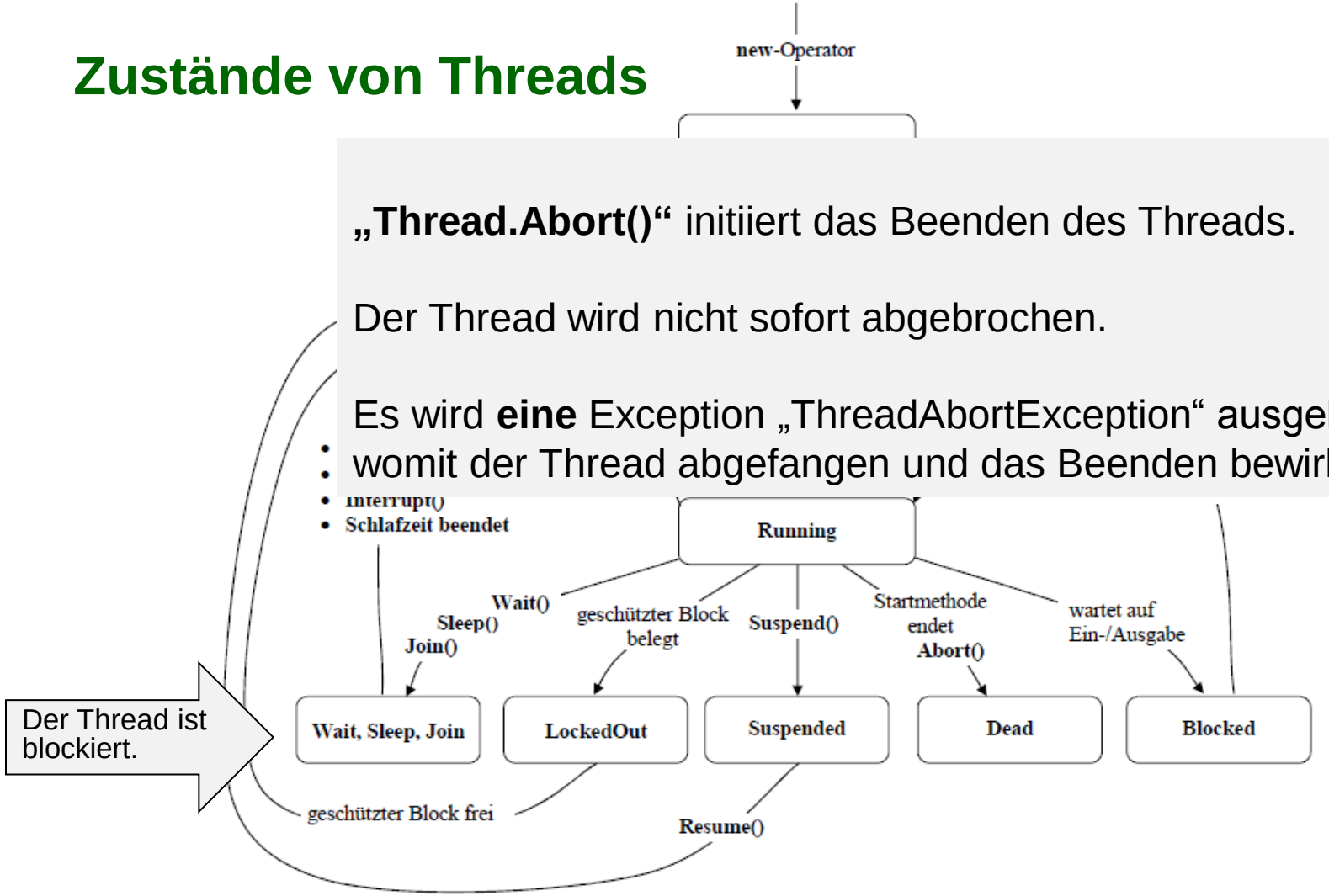
„Thread.Sleep(xxx)“ hält den Thread für „xxx“-Millisekunden an.

Zustände von Threads

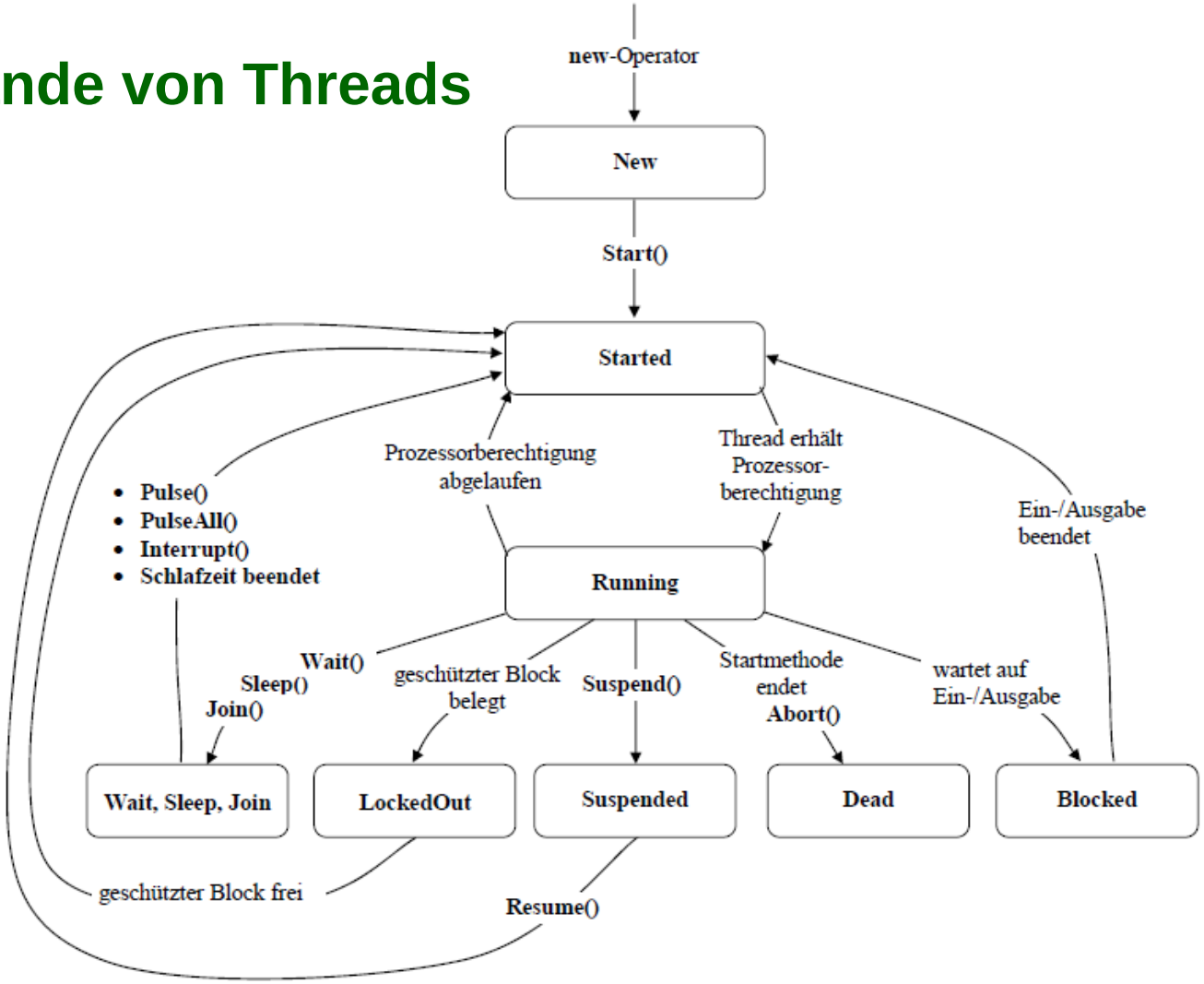
„Thread.Abort()“ initiiert das Beenden des Threads.

Der Thread wird nicht sofort abgebrochen.

Es wird **eine** Exception „ThreadAbortException“ ausgelöst, womit der Thread abgefangen und das Beenden bewirkt wird.



Zustände von Threads



Gute Seite für Threads

http://www.codeplanet.eu/tutorials/csharp/64-multithreading-in-csharp.html

Formulare: Control aus anderen Thread aktualisieren

Problem:

Zugriff aus einen Thread auf einer Control innerhalb des Formulars ist nicht möglich, da Controls nur über den eigenen UI-Thread heraus aktualisiert werden können.

Lösung:

Der *MethodInvoker* stellt einen einfachen Delegaten bereit, der zum Aufrufen einer Methode mit einer leeren Parameterliste verwendet wird.

Diesen Delegaten können Sie beim Aufrufen der *Invoke*-Methode eines Steuerelements verwenden.

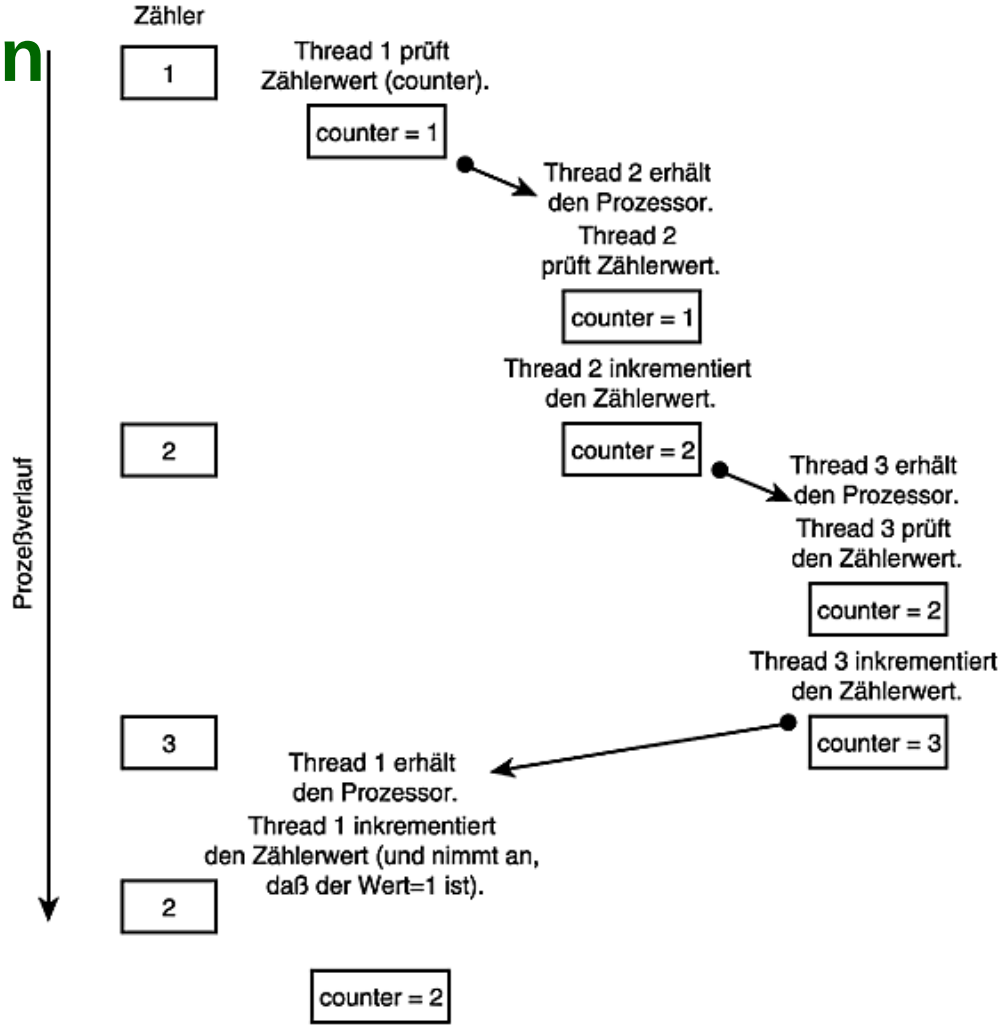
Siehe Beispiel:
ThreadForm1

```
MethodInvoker LabelUpdate = delegate
            {
                // Innerhalb dieses Blocks können Controls und Forms angesprochen
                // werden, neue Fenster erzeugt werden etc....
                label1.Text = MyName + ": " + i.ToString() + " Durchläufe absolviert";
            };
            Invoke(LabelUpdate);
```

Threads synchronisieren

Wenn es sich nicht vermeiden lässt, dass mehrere Threads gemeinsame Daten verwenden und dabei auch schreibend zugreifen, sind Maßnahmen zur Synchronisation der Zugriffe erforderlich.

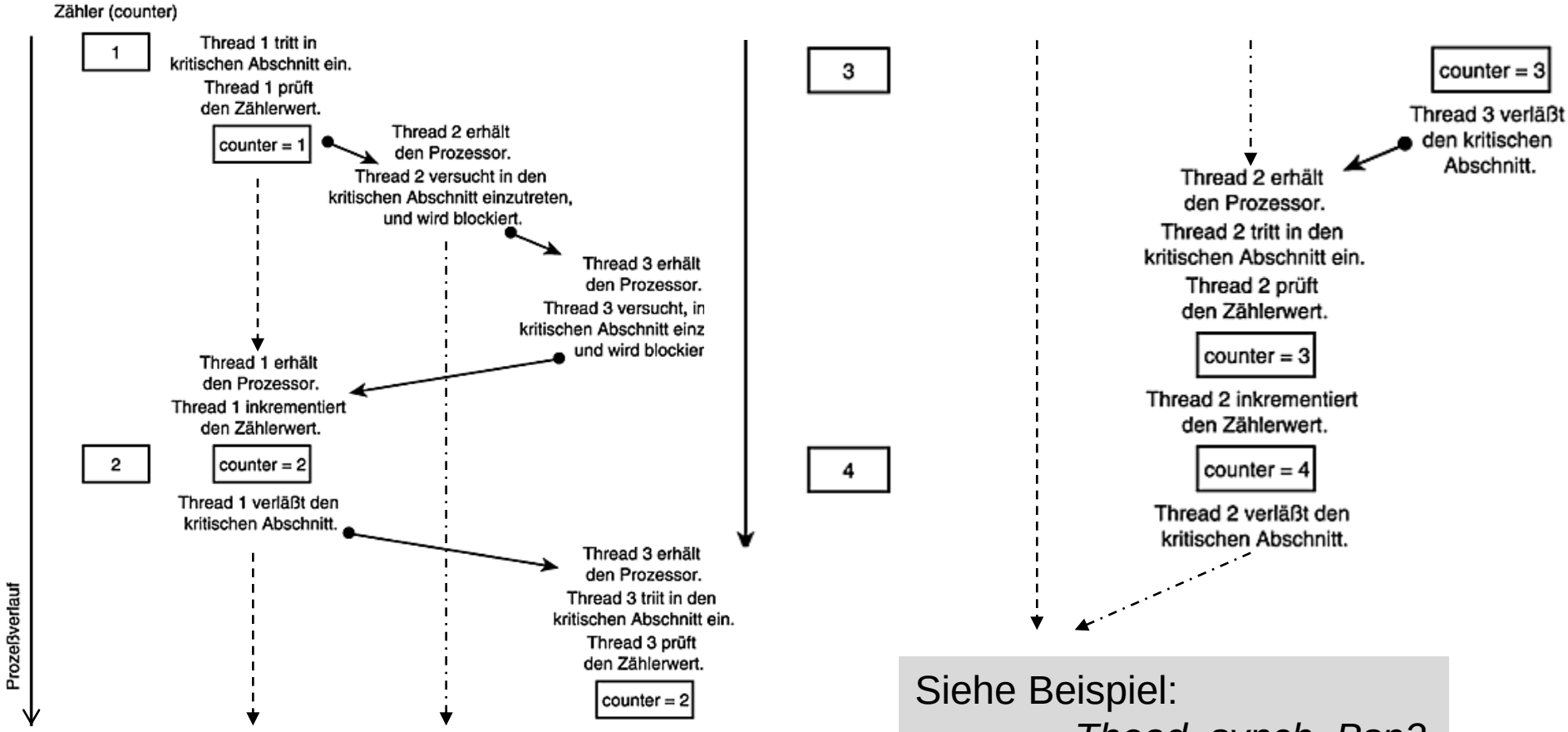
Siehe Beispiel:
Thead_synch_Bsp1



Threads – kritische Abschnitte

- Ein kritischer Abschnitt ist ein Mechanismus, der den Zugriff auf eine bestimmte Ressource auf einen einzelnen Thread innerhalb einer Anwendung beschränkt.
- Ein Thread tritt in den kritischen Abschnitt ein, bevor er mit der angegebenen gemeinsam genutzten Ressource arbeiten muß.
- Er verläßt den kritischen Abschnitt, nachdem der Zugriff auf die Ressource abgeschlossen ist.
- Wenn ein anderer Thread versucht, in den kritischen Abschnitt einzutreten, bevor der erste Thread den kritischen Abschnitt verlassen hat, wird der zweite Thread blockiert und erhält keine Prozessorzeit.
- Erst, wenn der erste Thread den kritischen Abschnitt verläßt, wird dem zweiten Thread den Eintritt ermöglicht..

Threads – kritische Abschnitte



Siehe Beispiel:
Thead_synch_Bsp2

Threads – Methoden der Klasse Monitor

Für den synchronisierten Zugriff von Threads auf geschützte Bereiche sorgt letztlich die Klasse **Monitor** aus dem Namensraum **System.Threading**.

- Monitor-Objekte synchronisieren den Zugriff auf einen Bereich des Codes.

- Die Klasse Monitor verwendet zum Setzen und Aufheben einer Sperre auf ein bestimmtes Objekt die Methoden
  - Monitor.Enter,
  - und Monitor.Exit.

- Sobald eine Sperre auf einen Codebereich gesetzt ist, können die Methoden
  - Monitor.Wait,
  - Monitor.Pulse und
  - Monitor.PulseAll
- verwenden werden.

Threads – Methoden der Klasse Monitor

Aktion	Beschreibungen
Enter, TryEnter	Erhält eine Sperre für ein Objekt. Diese Aktion kennzeichnet außerdem den Beginn eines kritischen Abschnitts. Ein anderer Thread kann nur dann auf den kritischen Abschnitt zugreifen, wenn er die Anweisungen im kritischen Abschnitt unter Verwendung eines anderen gesperrten Objekts ausführt.
Wait	Hebt die Sperre für ein Objekt auf, sodass anderen Threads das Sperren des Objekts und der Zugriff auf das Objekt ermöglicht wird. Der aufrufende Thread wartet, während ein anderer Thread auf das Objekt zugreift. Wartende Threads werden mithilfe von Impulssignalen über Änderungen am Objektzustand benachrichtigt.
Pulse (Signal), PulseAll	Sendet ein Signal an einen oder mehrere wartende Threads. Das Signal benachrichtigt einen wartenden Thread über eine Änderung am Zustand des gesperrten Objekts sowie über die bevorstehende Aufhebung der Sperre durch ihren Besitzer. Der wartende Thread wird in der Warteschlange für abgearbeitete Threads des Objekts platziert, sodass er schließlich die Sperre für das Objekt erhalten kann. Wenn der Thread die Sperre erhalten hat, kann er den neuen Zustand des Objekts überprüfen und ermitteln, ob der angeforderte Zustand erreicht wurde.
Exit	Hebt die Sperre für ein Objekt auf. Diese Aktion kennzeichnet außerdem das Ende eines kritischen Abschnitts, der durch das gesperrte Objekt geschützt ist.

Wait und Pulse

Ablauf an Hand eines Beispiels:

Thread A

```
1  lock(v) {
       ...
2      Monitor.Wait(v);5
       ...
   }
```

Thread B

```
3  lock(v) {
       ...
4      Monitor.Pulse(v);
       ...
   }6
```

1. A kommt zu *lock(v)* und erhält Eintritt, da kein anderer Thread im gesperrten Bereich ist.
2. A kommt zu *Wait*, legt sich schlafen und gibt die Sperre frei.
3. B kommt zu *lock(v)* und erhält Eintritt, da die Sperre frei ist.
4. B kommt zu *Pulse* und weckt damit A. Es kann (aber muß nicht) ein Thread-Wechsel stattfinden.
5. A versucht, die Sperre wieder zu erlangen, was aber nicht gelingt, da B sie noch hat.
6. B gibt am Ende des gesperrten Bereichs die Sperre frei; A kann nun wieder eintreten und weiterlaufen.

Threads – Lock-Anweisung

Statt der Verwendung der Monitorklasse kann auch die Anweisung ***„lock“*** verwendet werden.
Im Prinzip macht diese Anweisung nichts anderes als die Methoden Enter() und Exit() der Klasse Monitor.

Syntax:

```
lock(sperre)
{
      ...
}
```

wird vom Compiler umgesetzt in:

```
Monitor.Enter(sperre);
try
{
            ...
}
finally
{
            Monitor.Exit(sperre);
}
```

Siehe Beispiel:
Form_WaitPulse

Threads unterbrechen mit Join

- Man kann einen Thread dazu veranlassen, die Verarbeitung zu unterbrechen und darauf zu warten, dass ein anderer Thread seine Arbeit abgeschlossen hat.
- Dies wird als Vereinigen (engl. Join) des ersten mit dem zweiten Thread bezeichnet. Es ist, als würden Sie die Spitze des ersten Threads mit dem Ende des zweiten Threads verknüpfen, also »vereinigen«.
- Um einen Thread *t1* mit einem zweiten Thread *t2* zu vereinigen, schreiben Sie innerhalb der Methode von Thread *t1*: t2.Join();

Join:

```
namespace Threads_blockieren
{
   class Program
   {
      /// <summary>
      /// Thread-Methode zum Herunterladen von Dateien
      /// </summary>
      private static void Downloader(object args)
      {
         // Datei herunterladen (hier nur simuliert)
         Console.WriteLine("Lade " + args + " ...");
         Thread.Sleep(3000);
         Console.WriteLine(args + " ist heruntergeladen");
      }
```

```
      static void Main(string[] args)
      {
         Console.Title = "Threads blockieren";

         // Arbeits-Threads erzeugen und starten
         Thread thread1 = new Thread(Downloader);
         Thread thread2 = new Thread(Downloader);
         Thread thread3 = new Thread(Downloader);
         thread1.Start("Datei 1");
         thread2.Start("Datei 2");
         thread3.Start("Datei 3");

         Console.WriteLine("Warte auf das Ende der Arbeitsthreads ...");
         thread1.Join();
         thread2.Join();
         thread3.Join();
         Console.WriteLine("Verarbeite die Dateien ...");

         Console.WriteLine();
         Console.WriteLine("Beenden mit Return");
         Console.ReadLine();
      }
   }
}
```

Threads – ThreadPool

- Die Arbeit mit mehreren Threads lässt sich durch Threadpools wesentlich vereinfachen, denn die Laufzeitumgebung erzeugt eine bestimmte Anzahl von Threads, wenn sie gestartet wird.
- Sie können diese Threads nutzen und brauchen nicht eigens neue zu erzeugen, wenn Sie welche benötigen.
- Nach der Beendigung einer Threadmethode wird der frei gewordene Thread in den Pool zurückgeführt und steht anderen Aufgaben zur Verfügung.
- Angesprochen wird der Threadpool mit der gleichnamigen Klasse Threadpool. Mit deren statischer Methode QueueUserWorkItem wird der Threadpool aktiviert.
- Dabei wird der Methode ein Delegate vom Typ WaitCallback übergeben, das die Methode beschreibt, die mit dem Thread ausgeführt werden soll.

Siehe Beispiel:
ThreadPool

Threads – ThreadPool