

C# -Threads-

**Seminararbeit an der Universität Paderborn
Sommersemester 2002**

Christian Schneider

Inhaltsverzeichnis

1	Einleitung und Motivation	3
1.1	Was ist C# ?	3
1.2	Was sind Threads ?	4
2	Threads im .NET Framework	7
2.1	Anwendungsdomäne	7
2.2	Die <i>Thread</i> -Klasse	8
2.3	Zustände von Threads	9
2.4	Scheduling von Threads	11
2.5	Die <i>ThreadPool</i> -Klasse	12
3	Synchronisation von .NET Threads	13
3.1	Die <i>Monitor</i> -Klasse	14
3.2	Die <i>Mutex</i> -Klasse	16
3.3	Die <i>lock</i> -Anweisung	17
4	Fazit	18

1 Einleitung und Motivation

Die folgende Seminararbeit ist eine Ausarbeitung zum Thema „C#-Threads“ innerhalb des Blockseminars „C#“ von Prof. Dr. Uwe Kastens an der Universität Paderborn im Sommersemester 2002.

Diese Seminararbeit soll dem Leser einen Überblick über die Konzepte und die Implementierung von Threads, die das Microsoft .NET Framework bietet, verschaffen. Die Programmbeispiele in dieser Ausarbeitung sind in der neuentwickelten Programmiersprache C# angegeben.

1.1 Was ist C# ?

Eine der neusten Entwicklungen aus dem Hause Microsoft, welche seit Januar 2002 als 'final' Version erhältlich ist, trägt den Namen *.NET-Framework*. Das .NET-Framework besteht aus zwei Kernelementen. Zum einen aus der *Common Language Runtime* (CLR) und zum anderen der *Framework Class Library* (FCL).

Bei der *CLR* handelt es sich um eine Laufzeitumgebung für .NET Anwendungen. Ihre Aufgabe ist es .NET Anwendungen zur Laufzeit zu kompilieren und dann auszuführen. Diese Aufgabe übernimmt der *Just In Time Compiler* (JIT) der .NET Laufzeitumgebung. Die .NET Anwendungen liegen nicht, wie zum Beispiel Win32 Programme, in ausführbaren Maschinencode vor, sondern in einer prozessorunabhängigen Zwischensprache, der *Microsoft Intermediate Language* (MIL). Diese Zwischensprache wird erst zur Laufzeit von dem JIT in ausführbaren Maschinencode übersetzt und dann ausgeführt. Dies hat unter anderem den Vorteil, dass der Code der Anwendung speziell für den im System installierten Prozessor optimiert werden kann.

Weil .NET Anwendungen nicht direkt auf der Maschine laufen, sondern in einer *Umgebung*, spricht man auch von *Managed Code*. Dabei nimmt die Laufzeitumgebung dem Programmierer zusätzlich noch eine Vielzahl von Arbeiten ab, zum Beispiel die Speicherverwaltung. Durch die Laufzeitumgebung hat man eine Abstraktion von Betriebssystem und Prozessor erreicht. Muss man C/C++ Programme immer speziell für eine Prozessor-Familie und ein Betriebssystem kompilieren, laufen .NET Anwendungen auf allen Systemen, für die es eine entsprechende Implementierung des .NET Frameworks gibt. Die .NET Laufzeitumgebung ist, im Gegensatz zu Java, nicht auf eine Programmiersprache beschränkt, sondern unterstützt eine Vielzahl von aktuellen Programmiersprachen, unter anderem auch alle Programmiersprachen, die von Microsoft selbst stammen.

Um die Features des .NET-Frameworks im vollem Umfang nutzen zu können, hat man gleich eine neue, objektorientierte Programmiersprache entworfen und ihr den Namen *C#* gegeben. Die Sprache ist sehr stark an die Sprachen C++ und Java angelehnt. Wie auch in Java wird der Programmierer von der fehlerträchtigen Zeigerarithmetik abgeschottet. Ich werde in dieser Ausarbeitung nicht näher auf *C#* eingehen und verweise auf das Buch [2] von T. Archer und

auf das Buch [4] von P. Drayton.

Das zweite Kernelement ist die umfangreiche Klassenbibliothek (FCL), welche Bestandteil des .NET Frameworks ist. Die Klassenbibliothek, selbst zum größten Teil in C# geschrieben, enthält alle Funktionen, die ein Programmierer für die alltägliche Arbeit benötigt, und noch vieles darüber hinaus. Die Klassen der FCL sind thematisch in sogenannten Namensräumen (namespaces) geordnet. Das Augenmerk dieser Ausarbeitung liegt auf den Namensraum `System.Threading`. Dieser enthält alle notwendigen Klassen für die Erzeugung und Synchronisation von Threads.

1.2 Was sind Threads ?

Im Bereich der Betriebssysteme (BS) wird die Instanz eines Programms als *Prozess* bezeichnet. Dabei beschreibt ein Prozess die *sequentielle* Ausführung eines Programms in dem vom Betriebssystem zugeteilten Speicher, dem sogenannten *Adressraum* des Prozesses. Jeder Prozess besitzt seinen eigenen Adressraum und kein anderer Prozess darf auf diesen zugreifen, bzw. den Speicherinhalt in ihm modifizieren. Dies soll verhindern, dass fehlerhafte Programme die Speicherbereiche eines anderen Prozesses verändern. Solche fehlerhaften oder böswilligen Veränderungen können zum Absturz des anderen Prozesses führen oder was noch fataler wäre, wichtige Daten *unbemerkt* überschreiben. Die Fähigkeit eines Betriebssystems, jedem Prozess einen eigenen Adressraum zuzuordnen und diesen vor fremden Prozess-Zugriffen zu schützen, nennt man *Speicherschutz* (Memory-Protection). Unter dem Betriebssystem Linux wird zum Beispiel ein Prozess sofort mit dem Hinweis einer *Speicherschutzverletzung* (segmentation fault) beendet, wenn er versucht, auf einen fremden Adressraum zuzugreifen. Zusätzlich besitzt jeder Prozess einen sogenannten *Kontext*. Der Kontext beschreibt den veränderlichen Zustand eines Prozesses, der sich aus der Ausführungsposition und den Werten in den Prozessorregistern zusammensetzt. Die Fähigkeit eines Betriebssystems, mehrere Prozesse *nebenläufig* ausführen zu lassen macht den Gebrauch von Kontexten unverzichtbar. Nebenläufigkeit kann einmal die *verzahnte* oder die *parallele* Ausführung von Prozessen bedeuten. Bei der parallelen Ausführung werden die Prozesse *gleichzeitig* auf mehreren Prozessoren ausgeführt. Die Voraussetzung für diese Art der Ausführung ist natürlich ein Multiprozessorsystem. Steht jedoch nur ein Prozessor zur Verfügung, werden die Prozesse verzahnt ausgeführt, das heißt jeder Prozess bekommt abwechselnd den Prozessor für einen gewissen *Zeitabschnitt* (timeslice) zugeteilt. Die Prozesse laufen also zeitlich versetzt. Durch die schnelle Verarbeitungsgeschwindigkeit der heute verfügbaren Prozessoren und einem häufigen und schnellen Wechsel zwischen den Prozessen entsteht der Eindruck der parallelen Ausführung. Die nebenläufige Ausführung von Prozessen nennt man *Multitasking*. Alle modernen Betriebssysteme verfügen heute über diese Fähigkeit, wie zum Beispiel Linux, AmigaOS, Win9x und MacOS X. Ein Singletasking BS, also ein System, welches immer nur einen Prozess ausführen kann, ist zum Beispiel MS-DOS.

Damit ein Ausführungswechsel zwischen den Prozessen und deren spätere Weiterausführung überhaupt möglich ist, wird der Zustand des noch aktiven Pro-

zesses vor dem Wechsel im zugehörigen Kontext gesichert. Danach werden die Prozessorregister mit den Werten des zuvor gesicherten Kontextes des nachfolgenden Prozesses gefüllt. Die neue Ausführungsposition wird ebenfalls dem Kontext entnommen. Dieser Vorgang wird als *Kontextwechsel* bezeichnet.

Die Einheit des BS, die für die Zuteilung des Prozessors zu den Prozessen verantwortlich ist, wird *Scheduler* genannt. Die Hauptaufgabe des Schedulers ist es, die Menge der potentiell lauffähigen Prozesse zu bestimmen und abwechselnd jedem Prozess den Prozessor für einen bestimmten Zeitabschnitt zuzuteilen. Dieser Vorgang wird als *scheduling* bezeichnet. Auf die Gründe, warum ein Prozess vom Scheduler als nicht lauffähig eingestuft wird, wird noch eingegangen werden, wie auch auf die Bestimmung der Ausführungsreihenfolge der Prozesse, da diese stark von der verfolgten Strategie des Schedulers abhängt. Strategien können zum Beispiel sein, eine schnelle Reaktionszeit auf Ereignisse oder einen hohen Durchsatz an Prozessen zu erzielen. Weitere Informationen zu der Arbeitsweise eines Schedulers findet man in dem Buch [1] von Jürgen Nehmer und Peter Sturm.

Der Vorteil der nebenläufigen Ausführung von Prozessen liegt darin, dass mehrere Aufgaben (Tasks) *gleichzeitig* erledigt werden können. Es können zum Beispiel aufwändige Berechnungen durchgeführt werden und gleichzeitig eine Animation abgespielt werden. Der Datenaustausch zwischen zwei Programmen geht ebenfalls leichter von statten, da diese in einem Multitasking-BS nebenläufig ausgeführt werden und nicht nacheinander gestartet werden müssen. Besonders bei Multiprozessorsystemen kann eine Leistungssteigerung erzielt werden, wenn die Prozesse, die eine gemeinsame Aufgabe bearbeiten, auf mehreren Prozessoren zeitgleich ausgeführt werden. Diese Vorteile kompensieren den Nachteil des Aufwandes, der für einen Kontextwechsel notwendig ist, zumal die heutigen Prozessoren durch spezielle Befehle und Caches diesen Vorgang erheblich beschleunigen.

In einem Multitasking-BS besteht nun die Möglichkeit, mehrere Prozesse nebenläufig ausführen zu lassen, aber ein einzelner Prozess ist immer noch ein streng *sequentiell* auszuführendes Programm. Dieses Ausführungsmodell ist für die meisten Programme angemessen, aber nicht für alle! Benötigt man ein Programm, welches mehrere Aufgaben *gleichzeitig* erledigen kann, steht man vor einem großen Problem. Nehmen wir zum Beispiel einen Web-Browser. Mit einem Web-Browser kann man gleichzeitig mehrere Dateien herunterladen, während dabei der Benutzer von Homepage zu Homepage wandert. Dafür muss das Programm ständig die ankommenden Datenpakete der Dateien kontrollieren und sichern und dabei noch auf Eingaben des Benutzers warten und gegebenenfalls reagieren.

Wenn nur ein eigenständiger *Programmpfad* vorhanden ist, muss der Programmierer darauf achten, dass er den Prozessor 'gleichmäßig' auf die verschiedenen Aufgaben in seinem Programm verteilt. Wird der Abfrage und der Reaktion auf Benutzereingabe zu wenig Prozessorzeit geschenkt, besteht die Gefahr, dass Eingaben nur sehr schwerfällig oder gar nicht ausgeführt werden. Wird dieser Aufgabe hingegen zu viel Aufmerksamkeit geschenkt und der Benutzer macht keine Eingaben, wird wertvolle Prozessorleistung verschwendet, die bei anderen Aufgaben zu einer Erhöhung der Bearbeitungsdauer führen kann. Eine

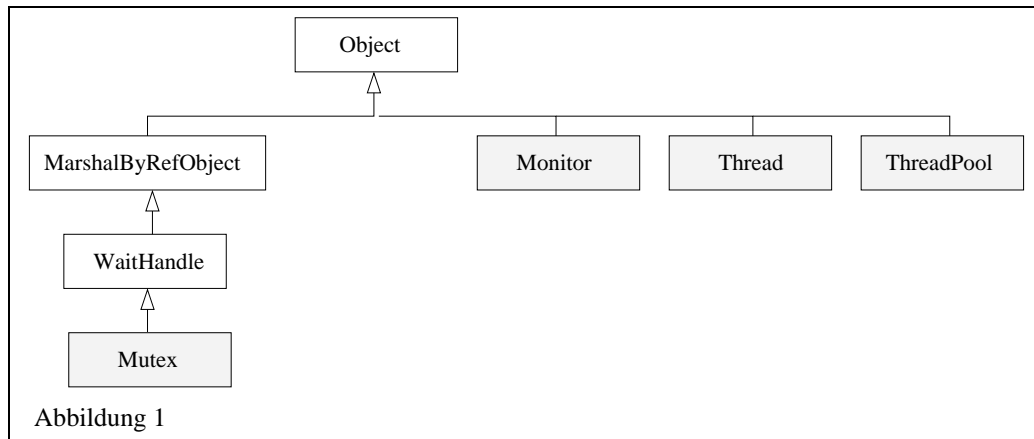
Lösung des Problems wäre die Implementierung eines eigenen Schedulers im Programm. Dies würde aber dazu führen, dass einfache Programme schnell an Komplexität zunehmen und dennoch nur ein unzulängliches Ergebnis liefern. Eine andere Möglichkeit wäre, das Programm in einzelne Prozesse aufzuteilen. Jede weitgehend unabhängige Aufgabe bekommt einen eigenen Prozess. Damit hätte man ein Problem gelöst, aber leider auch ein neues geschaffen, das Problem des Datenaustausches zwischen den einzelnen Prozessen. Man stelle sich nur vor, der Benutzer bricht das Herunterladen einer Datei ab. Dann müsste der Prozess der Benutzeroberfläche dem Prozess, der für das Herunterladen der Datei zuständig ist, dieses irgendwie mitteilen. Wie bereits erläutert, besitzt jeder Prozess einen eigenen geschützten Adressraum, auf den nur der zugehörige Prozess zugreifen darf. Dadurch wird die Kommunikation über gemeinsam genutzten Speicher ausgeschlossen. Es gibt zwar verschiedene Möglichkeiten der Inter-Prozess-Kommunikation, aber die Komplexität des Programms würde immer noch erheblich sein. Würde sich der Aufwand für einen Web-Browser vielleicht noch lohnen, kann er für einfache Programme schon zu hoch sein. Ideal wäre es, mehrere Prozesse zu haben, die sich einen Speicherbereich teilen, womit eine einfache Möglichkeit des Datenaustausches zwischen ihnen gegeben wäre. Diese Funktionalität wird von **Threads** realisiert. Solche Threads, oft auch als *Leichtgewichtsprozesse* (lightweight processes) bezeichnet, sind Prozesse, die parallel oder verzahnt in einem *gemeinsamen* Speicher (Adressraum) ablaufen. Da ein Prozess dadurch mehrere *unabhängige* Threads (Programmpfade) besitzen kann, die nebenläufig ausgeführt werden, spricht man auch von *Multithreading*. Ein Programm, welches mindestens zwei Threads einsetzt, verfügt dadurch über ein nebenläufiges Verarbeitungsmodell, was die Möglichkeit eröffnet, bestimmte Befehle asynchron ausführen lassen.

Wie so vieles, besitzen auch Threads einen großen Nachteil. Threads arbeiten auf den gleichen Daten, bzw. modifizieren die gleichen Objekte. Dies kann dazu führen, dass sich ein Objekt nach einem Kontextwechsel in einen inkonsistenten Zustand befindet, weil die Bearbeitung des Threads an dem Objekt noch nicht *vollständig* abgeschlossen ist. Beginnt daraufhin ein anderer Thread mit demselben Objekt zu arbeiten, welches sich noch in einen inkonsistenten Zustand befindet, ist das Ergebnis nicht mehr vorhersagbar. Dies macht es notwendig, die Zugriffe von Threads auf *gemeinsam* genutzte Objekte zu *synchronisieren*, damit immer dann, wenn ein neuer Thread mit der Bearbeitung eines Objektes beginnt, sich dessen Daten in einen konsistenten Zustand befinden. Ein Thread muss also warten, bis der andere mit seiner Bearbeitung der Daten fertig ist.

Die Synchronisation von gemeinsam genutzten Daten birgt auch eine große Fehlerquelle in sich. Durch die Synchronisation kann der Fall eintreten, dass Threads gegenseitig aufeinander warten, weil der jeweils andere Thread den Zugriff auf Daten besitzt, den der andere Thread fordert. Dies ist eine typische Verklemmungssituation, auch *Deadlock* genannt, aus der sich die Threads selbst nicht wieder befreien können. Desweiteren kann die Synchronisation auch erheblich die Performance der Anwendung negativ beeinflussen, weswegen man die Zugriffssynchronisation sparsam einsetzen sollte.

2 Threads im .NET Framework

Die .NET Klassenbibliothek besitzt bereits eine Fülle von Klassen zur Entwicklung von Anwendungen mit Multithreading. Untergebracht sind diese Klassen im Namensraum `System.Threading`. Um einen Überblick zu geben ist in Abbildung 1 ein Klassendiagramm abgebildet, welches die wichtigsten Klassen enthält.



Für die Erzeugung und Steuerung von Threads sind die Klassen *Thread* und *ThreadPool* da. Die Klasse *Thread* dient zur Erzeugung von zusätzlichen Threads neben dem initialen *Main-Thread*. Die *Thread* Klasse wird in Abschnitt 2.2 erläutert. Die *ThreadPool* Klasse dient nicht der direkten Erzeugung von Threads, vielmehr soll sie die Benutzung von Threads beschleunigen. Eine genaue Erklärung der *ThreadPool* Klasse befindet sich in Abschnitt 2.5.

Eine nicht zu unterschätzende Arbeit ist die Zugriffssynchronisation von gemeinsam genutzten Daten. In der FCL sind für diesen Zweck bereits Klassen implementiert, die dem Programmierer diese Arbeit so einfach wie möglich machen sollen. Zum einen ist dies die Klasse *Monitor* und desweiteren die Klasse *Mutex*. Auf diese Klassen werde ich erst in Kapitel 3 zu sprechen kommen.

Durch die Implementierung der Thread- und Synchronisationsmechanismen auf Klassen- und nicht auf Sprachebene, ist es möglich, diese in vielen vom .NET Framework unterstützten Programmiersprachen zu nutzen.

2.1 Anwendungsdomäne

Wie bereits in der Einleitung erläutert, versteht man unter einem Thread einen Prozess, der sich einen Adressraum mit anderen Prozessen teilt. Dadurch wird ein Thread auf Betriebssystemebene auf einen einzigen Prozess beschränkt. Dies hat den Nachteil, dass kein Thread eine Methode aufrufen kann, die zu einem Thread gehört, der in einem anderen Prozess läuft.

Im .NET Framework wird diese Restriktion durch das Konzept der *Anwendungsdomäne* aufgehoben. Eine Anwendungsdomäne bezeichnet die Umgebung, in der ein .NET Thread läuft. Dabei kann man eine Anwendungsdomäne eher als

einen *logischen* Prozess ansehen, welcher in einem *realen* Prozess des Betriebssystems läuft. Dadurch besteht die Fähigkeit, mehrere Anwendungsdomänen in einem *realen* Prozess laufen zu lassen, wodurch prozessübergreifende Methodenaufrufe möglich sind, da sie sich einen Adressraum teilen.

2.2 Die Thread-Klasse

Um einen Thread im .NET Framework zu erstellen ist die Klasse *Thread* erforderlich. Es wird ein Funktionszeiger, in C# ein sogenanntes *Delegate*, übergeben, den der Programmierer frei wählen kann, solange die Funktion die geforderte Signatur besitzt. Dadurch wird der Programmierer von der Notwendigkeit befreit, für den Einsatz von Threads eine zusätzliche Klasse zu implementieren. Dies ist zum Beispiel in Java erforderlich. Dort muss man, wenn man einen *vollständig* kontrollierbaren Thread verwenden möchte, von der Klasse **Thread** ableiten.

Das nachfolgende Listing verdeutlicht, wie einfach es ist, einen Thread mit der FCL in C# zu erzeugen.

```
using System;
using System.Threading;

class Foo
{
    public static void ThreadEntry()
    {
        Console.WriteLine("Hello Thread!");
    }

    public static void Main()
    {
        ThreadStart entry = new ThreadStart(ThreadEntry);
        Thread thread = new Thread(entry);
        thread.Start();
    }
}
```

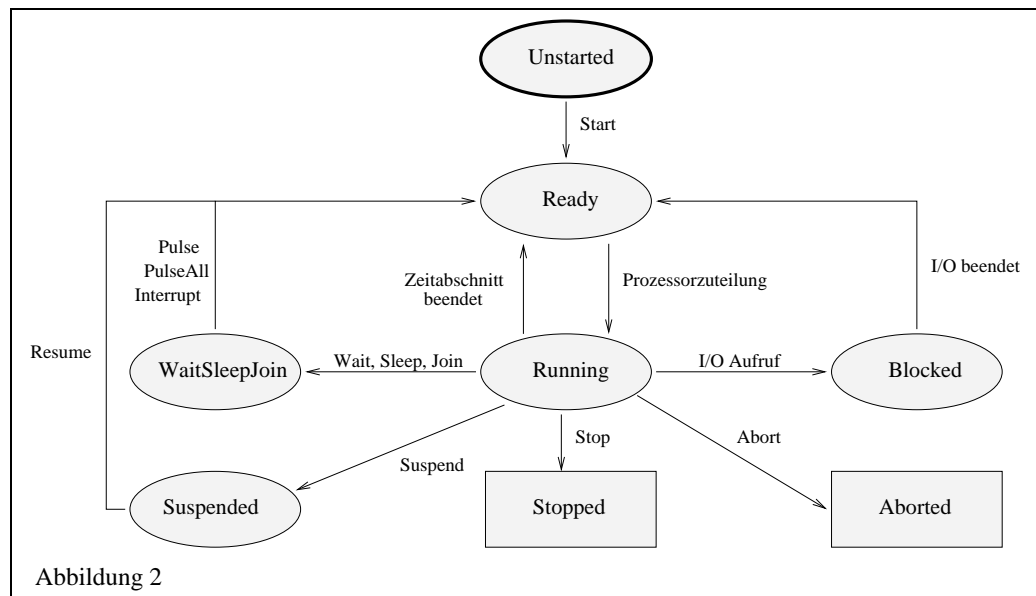
In der ersten Zeile der `Main()`-Methode wird zuerst ein `Delegate`-Objekt des Typs `ThreadStart` erzeugt. Dieses kapselt einen Funktionszeiger, in diesem Fall die Methode `ThreadEntry()`. Bei dieser Methode wird der neue Thread seine Ausführung beginnen. Danach erst wird das Thread-Objekt selbst erzeugt. Der Konstruktor der Thread-Klasse bekommt das zuvor erzeugte `Delegate`-Objekt als Parameter übergeben. Damit der Thread vom Scheduler des .NET Frameworks berücksichtigt wird, muss dieser noch gestartet werden. Dies geschieht mit der Methode `Start()` der Thread-Klasse. Das Verlassen der Einstiegs-methode beendet den Thread. Es gibt noch weitere Methoden einen Thread zu beenden, auf die ich aber erst im nächsten Abschnitt zu sprechen komme. Eines haben sie aber alle gemeinsam, ein Thread der seine Ausführung einmal

beendet hat, lässt sich nicht noch einmal starten, auch wenn man noch im Besitz eines gültigen Thread-Objektes ist.

2.3 Zustände von Threads

Ein .NET Thread kann sich während seiner Lebenszeit in verschiedenen *Zuständen* befinden. Ein Zustandswechsel kann einmal von der Laufzeitumgebung oder von der Anwendung selbst veranlasst werden.

Als Hilfe kann Abbildung 2 herangezogen werden, in der ein Automat abgebildet ist, der die *wichtigsten* Zustände und Übergänge eines Threads verdeutlicht.



Nachdem ein neues Thread-Objekt erzeugt worden ist, befindet sich der Thread im Zustand *Unstarted*. Befindet sich der Thread in diesem Zustand, wird der Thread noch nicht vom Scheduler des .NET Frameworks berücksichtigt. Deshalb muss der Thread noch gestartet werden. Dies geschieht mit der bereits bekannten Methode `Start()` der Thread-Klasse.

Danach wechselt der Thread in den Zustand *Ready* und ist damit ein potentieller Kandidat bei der Auswahl der Scheduler nach einem ausführbaren Thread. Wird daraufhin der Thread vom Scheduler dem Prozessor zugeteilt, geht dieser in den Zustand *Running* über.

Im *Running*-Zustand führt der Thread seine Anweisungen aus. Der Thread hat dafür aber nur eine begrenzte Zeit. Wird er innerhalb dieses Zeitabschnittes mit seiner Aufgabe fertig, beendet er also die Einstiegsmethode, geht der Thread in den *Stopped*-Zustand über. Damit befindet sich der Thread in einem Endzustand, den er nicht mehr verlassen kann. Schafft der Thread es nicht, seine Anweisungen innerhalb der gegebenen Zeit vollständig abzuarbeiten, wird dem Thread der Prozessor entzogen, und er wechselt wieder in den *Ready*-Zustand.

Außer diesen beiden Möglichkeiten für einen Thread, den *Running*-Zustand zu verlassen, gibt es noch weitere.

Eine davon ist, dass der Thread eine *blockierende* Methode aufruft, zum Beispiel durch Einlesen einer Datei. In diesem Fall muss der Thread darauf warten, bis die Datei vom Betriebssystem geladen worden ist und geht zu diesem Zwecke in den Zustand *Blocked*. Der Wechsel in den *Blocked*-Zustand ist deshalb sinnvoll, da die Wartezeit von anderen Threads für ihre weitere Ausführung genutzt werden kann. Darum muss ein blockierender Thread den Prozessor wieder abgeben. Erst wenn der blockierende Aufruf beendet worden ist, geht der Thread wieder in den Zustand *Ready*.

Eine weitere Möglichkeit ist der Wechsel in den *Suspended*-Zustand. Dieser Wechsel wird durch die Methode `Suspend()` der Thread Klasse ausgelöst. Dabei kann diese Methode von einem Thread für sich selbst oder für einen anderen Thread aufgerufen werden. Im *Suspended*-Zustand verweilt der Thread für eine unbestimmte Zeit, in der er von der Prozessorzuteilung ausgeschlossen ist, dadurch kann nur ein anderer Thread einen Zustandswechsel veranlassen. Hierfür gibt es die `Resume()`-Methode der Thread Klasse, die ein Wechsel eines anderen Threads vom *Suspended*- in den *Ready*-Zustand bewirkt.

Mit Ausführung der `Sleep()`-Methode kann sich ein Thread selbst für eine bestimmte oder unbestimmte Zeit von jeglicher Prozessorzuteilung ausschließen. Dies führt zu einem Wechsel in den *WaitSleepJoin*-Zustand. Ist die festgelegte Auszeit verstrichen, wechselt der Thread automatisch wieder in den *Ready*-Zustand. Hat sich der Thread für eine unbestimmte Zeit schlafen gelegt, kann dieser nur von einem anderen Thread wieder geweckt werden, zu diesem Zwecke stellt die Thread Klasse die `Interrupt()`-Methode zur Verfügung.

Manchmal ist es sinnvoll, einen Thread nicht für eine bestimmte Zeit, sondern auf ein bestimmtes Ereignis warten zu lassen. So ein Ereignis kann zum Beispiel ein Zustandswechsel eines Objektes sein. Dies ist besonders bei der Bedingungssynchronisation von Threads auf Objekte wichtig. Kann ein Thread mit dem augenblicklichen Zustand eines Objektes nichts anfangen, wartet er, bis ein anderer Thread den Zustand des Objektes geändert hat und dieses auch signalisiert. Mithilfe der `Wait()`-Methode der `Monitor`-Klasse geht ein Thread in den *WaitSleepJoin* Zustand und wartet dort auf eine Zustandsänderung des angegebenen Objektes. Um wartenden Threads eine Zustandsänderung ihres Objektes mitzuteilen, enthält die `Monitor` Klasse die `Pulse()`- und `PulseAll()`-Methoden. Damit kann man für den ersten oder allen auf das übergebene Objekt wartenden Threads ein Wechsel in den *Ready*-Zustand herbeiführen.

Die dritte Möglichkeit für einen Thread, in den *WaitSleepJoin*-Zustand zu wechseln, ist der Aufruf der `Join()`-Methode der Thread Klasse. Damit kann sich ein Thread mit einem anderen Thread synchronisieren. Der aufrufende Thread wartet auf die Beendigung eines zweiten Threads. Der Aufruf der Methode erfolgt synchron und kehrt damit erst zum Aufrufer zurück, wenn der andere Thread beendet worden ist.

Außer dem *Stopped*-Zustand gibt es noch einen weiteren Endzustand und zwar den *Aborted*-Zustand. Erzielt wird ein Wechsel in diesen Zustand durch Aufruf der `Abort()`-Methode. Dadurch kann ein Thread einen anderen Thread

vorzeitig beenden. Für einen Thread der sich in diesem Zustand befindet gilt ebenfalls, dass er sich nicht erneut starten lässt.

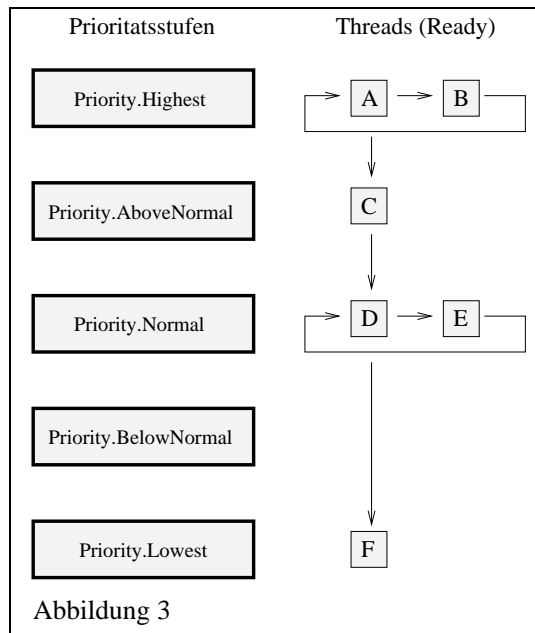
2.4 Scheduling von Threads

Das Scheduling ist, wie bereits geschrieben, für die Zuteilung des Prozessors, im Falle eines Einzelprozessorsystems, zu den Prozessen, bzw. Threads, zuständig. Dabei erhält jeder Thread den Prozessor für einen kurzen Zeitabschnitt. Nach dieser Zeit wird mit der Ausführung eines anderen Threads begonnen.

Wie aus der realen Arbeitswelt bekannt, gibt es Aufgaben, die besonders dringend erledigt werden müssen, zum Beispiel das Schreiben einer Seminararbeit, und solche, bei denen die Fertigstellung nicht so wichtig ist. Normalerweise ordnet man den Aufgaben dazu verschiedene Prioritäten zu. So wird auch mit .NET Threads verfahren. Hierfür besitzt die Thread Klasse die `Priority`-Eigenschaft, der man einen der fünf Werte `Highest`, `AboveNormal`, `Normal`, `BelowNormal`, `Lowest` zuweisen kann. Jeder neu erzeugte Thread besitzt zu Beginn die Priorität `Normal`.

Der Thread mit der höchsten Priorität wird bevorzugt behandelt, das heißt, der Thread wird solange ausgeführt, bis er seine Aufgabe beendet hat, den *Running*-Zustand verlässt oder ein Thread mit einer höheren Priorität in den *Ready*-Zustand wechselt. Der .NET Scheduler versucht also immer den Thread mit der höchsten Priorität am Laufen zu halten. Besitzen mehrere Threads die höchste Priorität, befinden sich somit auf der *gleichen* Prioritätsstufe, wird ihnen der Prozessor nach *Round Robin* Manier für einen Zeitabschnitt zugeteilt.

In der Abbildung 3 wird die Vorgehensweise des .NET Schedulers noch einmal verdeutlicht. Auf der linken Seite sind die fünf Prioritätsstufen dargestellt, die das .NET Framework anbietet. Auf der rechten Seite sieht man die Threads, die ausgeführt werden können, sie befinden sich im *Ready*-Zustand. Zuerst werden die Threads mit der höchsten Priorität ausgeführt, dies sind die Threads **A** und **B**. Erst wenn diese zwei Threads beendet oder blockiert sind, bekommt Thread **C** den Prozessor zugeteilt. Da Thread **F** die niedrigste Priorität von allen besitzt, bekommt er erst Rechenzeit, wenn alle anderen Threads beendet oder blockiert sind.



2.5 Die ThreadPool-Klasse

Die Erzeugung und Zerstörung von Threads ist eine zeitaufwändige Angelegenheit. Zur Beschleunigung des Vorganges gibt es die Klasse *ThreadPool*, von der die .NET Laufzeitumgebung selbst regen Gebrauch macht, um zum Beispiel Delegates schnell asynchron ausführen zu lassen. Ein ThreadPool ist sozusagen ein Vorratslager an Threads, die auf Aufgaben warten, die sie ausführen sollen. Diese Aufgaben werden in Form von Delegates übergeben, welche die Einstiegsmethoden für die Threads des ThreadPools kapseln. Ein ThreadPool enthält für die Bearbeitung der Aufgaben mehrere Threads, bis zu 25, die einmal erzeugt werden und erst beim Beenden der Anwendung wieder zerstört werden. Hat man dem ThreadPool eine Aufgabe übergeben, wird diese entweder sofort von einem Thread ausgeführt oder sobald einer frei wird. Zusätzlich nimmt der ThreadPool dem Programmierer die Verwaltung der Threads ab, da diese von der Laufzeitumgebung übernommen wird. Der Gebrauch eines ThreadPools ist besonders dann sinnvoll, wenn die Anwendung Threads nur für Aufgaben mit geringem Zeitaufwand einsetzt und somit die Threads die meiste Zeit im wartenden Zustand verbringen. Das nachfolgende Listing zeigt die Verwendung des ThreadPools.

```
using System;
using System.Threading;

class Foo
{
    public static void ThreadEntry(object state)
    {
        Console.WriteLine("Hello ThreadPool!");
    }
}
```

```

public static void Main()
{
    WaitCallback entry = new WaitCallback(ThreadEntry);
    ThreadPool.QueueUserWorkItem(entry);
}
}

```

Die erste Anweisung in der `Main()`-Methode erzeugt ein Delegate-Objekt vom Typ `WaitCallback`, welches die Methode kapselt, die von einem der Threads aus dem `ThreadPool` ausgeführt werden soll. Die zweite Anweisung übergibt dem `ThreadPool` das Delegate-Objekt, woraufhin sich der `ThreadPool` um die Ausführung der Methode durch einen seiner Threads kümmert. Ein explizites Erzeugen eines `ThreadPool`s ist nicht notwendig, das pro Prozess nur ein `ThreadPool` erlaubt ist. Die Erzeugung dieses `ThreadPool`s übernimmt die Laufzeitumgebung.

3 Synchronisation von .NET Threads

Threads kommunizieren durch gemeinsam genutzte Daten, bzw. Objekte. Um zu verhindern, dass mehrere Threads gleichzeitig ein Objekt bearbeiten, ist es notwendig, Zugriffe auf Objekte zu synchronisieren. Dies soll sicherstellen, dass immer nur ein Thread den Zugriff auf ein Objekt besitzt.

Eine Folge von Anweisungen, welche auf die Daten eines Objektes zugreifen, bzw. sie verändern, die auf keinen Fall durch einen Kontextwechsel unterbrochen werden darf, nennt man einen *kritischen Abschnitt* (critical section). Wird ein solcher kritischer Abschnitt dennoch durch einen Kontextwechsel unterbrochen, kann es zu einer Dateninkonsistenz des Objektes führen, wenn ein anderer Thread mit diesen Daten seine Arbeit aufnimmt. Um dies zu verhindern, wird der Zugriff auf einen kritischen Abschnitt synchronisiert. Es darf sich immer nur ein Thread innerhalb eines kritischen Abschnitts für ein Objekt aufhalten, das Betreten dieses Abschnitts ist für einen anderen Thread erst möglich, wenn der erste Thread den Abschnitt verlassen hat. Dies bezeichnet man als *gegenseitiger Ausschluss*. Um den gegenseitigen Ausschluss sicherzustellen, gibt es verschiedene Sprachkonstrukte. Ein Konstrukt ist das Monitor Konzept, welches die Idee verfolgt, Daten, die in einem kritischen Abschnitt bearbeitet werden, mit den darauf definierten Anweisungen zu einer Einheit zusammenzufassen. Dies kann zum Beispiel in Form einer Klasse implementiert sein. Die Klasse enthält die Daten, als auch die entsprechenden Anweisungen, welche sich in den Methoden der Klasse befinden. Zu einem Zeitpunkt darf sich immer nur ein Thread innerhalb einer der Methoden aufhalten. Wird ein Monitor von einem Thread benutzt, wird dieser für alle anderen Threads gesperrt. Threads, die den Monitor *betreten* möchten, werden solange blockiert, bis der Monitor wieder frei wird. Das Monitor-Konzept erhöht den Komfort der Synchronisation. Um den gegenseitigen Ausschluss zu realisieren, benutzt ein Monitor selbst *Sperren* oder *Semaphore*.

Eine Sperre besitzt zwei Zustände. Ist ein Thread im Besitz der Sperre, dann ist das Objekt gesperrt (locked) und der Thread darf den kritischen Abschnitt betreten. Beim Verlassen des kritischen Abschnittes gibt er die Sperre wieder frei (unlocked). Nach ihrem Zweck bezeichnet man deshalb auch die auf eine Sperre operierenden Funktionen *Sperren* (Lock) und *Freigeben* (Unlock).

Eine besondere Art der Sperre ist das sogenannte *Semaphore*. Ein Semaphore verfügt über eine *Zählvariable* mit der man den Eintritt von mehreren Threads in einen kritischen Abschnitt steuern kann. Dies ist zum Beispiel wünschenswert, wenn man von einer Resource mehrere zur Verfügung hat und den Zugriff auf diese synchronisieren möchte. Die entsprechenden Operationen für ein Semaphore lauten *P* (Passieren) und *V* (Verlassen). Die erste ermöglicht den Eintritt in einen kritischen Abschnitt und die zweite den Austritt aus diesen. Besitzt die Zählvariable den Anfangswert 1 kann nur ein Thread den kritischen Abschnitt betreten, in diesem Fall spricht man von einem *binären Semaphore*. Ein solches Semaphore hat die gleiche Funktionalität wie die zuvor beschriebene Sperre.

Ein weiteres Konzept ist die *Bedingungssynchronisation*. Dabei hängt der Zugriff auf ein Objekt nicht nur von der Sperre ab, sondern noch von einer Bedingung, die innerhalb des kritischen Abschnittes vom ausführenden Thread geprüft wird. Die Semantik der Bedingung liegt allein beim Programmierer. Falls die Bedingung nicht erfüllt ist, ist es wichtig, dass der wartende Thread die Sperre wieder frei gibt. Solange der Thread im Besitz der Sperre ist, hat kein anderer Thread die Möglichkeit, den Zustand des Objektes zu ändern. Die Folge wäre, dass die Bedingung nie erfüllt wird und man hätte dadurch eine klassische Verklemmungssituation. Der Thread muss also die Sperre freigeben und auf eine Änderung des Objektzustandes warten. Die Operationen für diese zwei Aufgaben lauten im allgemeinen *Warten* (Wait) und *Signalisieren* (Signal). Beim Benutzen der Wait-Operation gibt der Thread die Sperre des Objektes frei und wartet auf ein entsprechendes Signal. Hat ein wartender Thread ein Signal erhalten, bewirbt er sich um die Sperre für des Objekt, um dann erneut im kritischen Abschnitt die Bedingung prüfen zu können. Ist der Zustand eines Objektes von einem Thread verändert worden, kann er dies den wartenden Threads mit Hilfe der Signal-Operation mitteilen.

Um die Zugriffssynchronisation zu erleichtern, enthält die FCL Klassen, von denen ich zwei vorstellen werde. Sie unterscheiden sich vorallem in Aufwand und Flexibilität. Jedoch verfügt nur die Monitor-Klasse über die Funktionalität der Bedingungssynchronisation.

3.1 Die Monitor-Klasse

Auch die Klassenbibliothek des .NET Frameworks besitzt eine Monitor-Klasse. Suggestiert der Name der Klasse die zuvor beschriebenen Funktionalität eines allgemeinen Monitors, bietet die .NET Implementierung lediglich *statische* Methoden an, die dazu dienen sollen, die Funktionalität eines Monitors selbst zu implementieren.

Die Zugriffssynchronisation auf Daten eines Objektes erfolgt durch Setzen und Freigeben einer *Sperre* (lock). Jedes von der Klasse *Objekt* abgeleitete Objekt besitzt eine solche Sperre. Mithilfe der Methoden `Monitor.Enter()` und `Monitor.Exit()` der Klasse kann man einen kritischen Abschnitt betreten oder ihn wieder verlassen. Damit Erfüllen sie den gleichen Zweck der zuvor beschriebenen Operationen *Sperren* und *Freigeben*. Mit der Ersten wird ein Objekt gesperrt. Wenn das Sperren des Objektes erfolgreich war, betritt der Thread für dieses Objekt den geschützten Bereich. Konnte der Thread das Objekt nicht sperren, weil es schon gesperrt ist, wird der Thread solange blockiert, bis der andere Thread die Sperre wieder frei gibt. Dadurch kann kein anderer Thread den kritischen Abschnitt betreten, außer er will den Abschnitt mit einem anderen Objekt betreten. Die Freigabe der Sperre erfolgt mittels der `Monitor.Exit()` Methode. Der Bereich zwischen diesen beiden Methoden-Aufrufen beschreibt also den geschützten Bereich.

Wird die Methode `Monitor.Enter()` mehrmals von dem Thread aufgerufen, der schon im Besitz der Sperre ist, wird dieser nicht blockiert. Es wird allerdings die Anzahl der Aufrufe festgehalten und der Thread muss genauso oft die `Monitor.Exit()`-Methode aufrufen, um die Sperre letztendlich wieder freizugeben. Dies soll die frühzeitige Freigabe der Sperre bei rekursiven Aufrufen verhindern. Ein Sperre mit dieser Funktionalität nennt man ein *smartes* Semaphore und dazu in diesem Fall auch noch um ein binäres.

Das nachfolgende Listing verdeutlicht die Verwendung der beiden Methoden der Klasse.

```
class DataObject
{
    public void AddData(int value)
    {
        Monitor.Enter(this);
        int tmp = 0;
        while ((tmp = this.value) > 20)
            Monitor.Wait(this);
        tmp += value;
        this.value = tmp;
        Monitor.PulseAll(this);
        Monitor.Exit(this);
    }

    private int value;
}
```

Zusätzlich bietet die Monitor-Klasse noch Methoden an, um eine *bedingte* Synchronisation zu implementieren. Zu diesem Zwecke enthält sie drei Methoden, welche die zuvor beschriebenen Operationen, Wait und Signal, der Bedingungs-synchronisation abbilden.

Mit der `Wait()` Methode kann der Thread, der sich im Besitz der Sperre des

Objektes befindet, selbst in den blockierenden, bzw. wartenden Zustand versetzen. Zuvor aber gibt der Thread die Sperre des Objektes wieder frei. Für diesen Zweck referenziert jedes Objekt eine eigene Queue (waiting queue), in der alle auf dieses Objekt wartenden Threads enthalten sind. Diese Methode entspricht der Wait-Operation.

Um wartende Threads eine Änderung der Sperre des Objektes mitzuteilen stellt die Klasse die Methode `Pulse()` und `PulseAll()` zur Verfügung. Mit der ersten Methode wird dem ersten Thread in der *waiting* Queue eine Änderung der Sperre signalisiert. Der Thread bekommt aber nicht automatisch die Sperre für das Objekt, sondern seine Referenz wird aus der *waiting* Queue entfernt und in der *ready* Queue des Objektes aufgenommen. Wird nun tatsächlich die Sperre des Objektes freigegeben, bekommt der erste Thread aus der *ready* Queue diese und nimmt seine Arbeit auf. Die zweite Methode signalisiert, im Gegensatz zur ersten, allen wartenden Threads eine Änderung. Diese Methoden entsprechen der Signal-Operation. Beim Aufruf dieser Methoden wird noch nicht die Sperre des Objektes vom Thread freigegeben. Die Sperre muss explizit freigegeben werden. Dieses Variante bezeichnet man als *signal-and-continue*.

Die Methoden `Pulse()`, `PulseAll()` und `Wait()` dürfen nur innerhalb eines synchronisierten Abschnittes und nur von dem Besitzer der Sperre aufgerufen werden, andernfalls wird eine *Exception* ausgelöst.

3.2 Die Mutex-Klasse

Ein weitere Möglichkeit den Zugriff auf Abschnitte zu serialisieren, ist die Verwendung der *Mutex* Klasse. Der Begriff Mutex leitet sich von *mutually exclusion* ab. Dies bedeutet soviel wie „sich gegenseitig ausschließend“. Der gegenseitige Ausschluss wird dadurch realisiert, dass ein Mutex zu jedem gegebenen Zeitpunkt immer nur von einem einzigen Thread besitzt wird. Dadurch lässt sich ein Folge von Anweisungen, welche auf gemeinsam genutzte Daten zugreift, vor nebenläufigen Threadzugriffen schützen. Der Vorteil gegenüber der Monitor-Klasse liegt darin, dass Mutexe auch zur Synchronisation zwischen Prozessen eingesetzt werden können. Zu diesem Zwecke erzeugt man ein *benanntes* Mutex-Objekt. Wird bei der Erzeugung eines neuen Mutex-Objekt festgestellt, dass schon ein Mutex mit den selben Namen existiert, wird kein neues Objekt erzeugt, sondern man erhält eine Referenz auf das bereits vorhandene. Wie auch schon bei der Monitor-Implementierung des .NET Frameworks handelt es sich bei einem Mutex um ein *smartes binäres* Semaphore.

Das nachfolgende Listing zeigt die Anwendung eines Mutex-Objektes.

```
class DataObject
{
    Mutex mutex = new Mutex(false, "Muff");

    public void AddData(int value)
    {
        mutex.WaitOne();
    }
}
```

```

        int tmp = this.value;
        tmp += value;
        this.value = tmp;
        mutex.Close();
    }

    private int value;
}

```

Die Klasse *DataObjekt* besitzt ein Feld *mutex*. Der Parameter mit dem Wert *false* im Konstruktor der Klasse besagt, dass der Thread, der das Objekt erzeugt, nicht automatisch zum Besitzer des Mutex-Objektes wird. Betritt nun ein Thread die Methode `AddData()`, versucht er zuerst mittels der `WaitOne()`-Methode der Mutex Klasse zum Besitzer des Mutex-Objektes zu werden. Dies ist gleichbedeutend mit dem Versuch, das Objekt zu sperren. Wird das Mutex-Objekt zu diesem Zeitpunkt von keinen anderen Thread besessen, erhält er die geforderte Sperre und hat damit Zugriff auf die dann der Methode folgenden Anweisungen. Wird die Sperre schon von einem anderen Thread besessen, wird der fordernde Thread solange blockiert, bis die Sperre wieder frei ist. Zum Freigeben der Sperre, besitzt die Mutex-Klasse die Methode `Close()`. Der Bereich zwischen diesen beiden Methoden wird geschützt.

3.3 Die lock-Anweisung

Ein weitere Möglichkeit, den Zugriff auf Daten zu serialisieren ist die `lock`-Anweisung. Die `lock`-Anweisung ist Bestandteil der C# Sprache und besitzt die selbe Funktionalität wie die zwei Methoden `Monitor.Enter()` und `Monitor.Exit()` der `Monitor` Klasse.

lock Syntax: `lock(Ausdruck) {Anweisungen}`

Das nachfolgende Listing zeigt die Verwendung der `lock`-Anweisung.

```

class DataObject
{
    public void AddData(int value)
    {
        lock(this)
        {
            int tmp = this.value;
            tmp += value;
            this.value = tmp;
        }
    }

    private int value;
}

```

Die Klammern der lock-Anweisung kennzeichnen den geschützten Bereich. Dadurch ist eine *explizite* Freigabe der Sperre nicht notwendig, was besonders nützlich ist, wenn Anweisungen innerhalb des kritischen Abschnittes **Exceptions** auslösen können. Die Laufzeitumgebung stellt sicher, dass die Sperre des Objektes auf jeden Fall wieder freigegeben wird.

4 Fazit

Das .NET Framework von Microsoft ist sicherlich eine interessante Entwicklung, die man gerade im Bereich der Entwicklung von Web-Applikationen nicht außeracht lassen sollte. Besonders in diesem Bereich ist man auf multithreaded Anwendungen angewiesen, die schnell und sicher laufen.

Im allgemeinen sollte man Threads erst dann verwenden, wenn es die Anwendung wirklich erfordert. Die notwendigen Maßnahmen und Kenntnisse der Zugriffssynchronisation bei der Entwicklung von Anwendungen mit multithreading sind nicht zu unterschätzen. Besonders der übermäßige Einsatz derer führt schnell zu Verklemmungssituationen, die oft nur schwer zu lokalisieren sind. Darüber hinaus kann die Laufzeit durch Zugriffssynchronisationen erheblich beeinträchtigt werden. Desweiteren sollte bei der Verwendung der FCL bedacht werden, dass diese nicht Thread-sicher ist. Die Folge daraus ist, dass die Anwendung bei der Verwendung von Komponenten daraus den Zugriff selbst synchronisieren muss.

Besteht die Notwendigkeit, Aufgaben in einer Anwendung asynchron ausführen zu lassen, sollte als erstes über den Einsatz der ThreadPool-Klasse nachgedacht werden. Die ThreadPool-Klasse eignet sich besonders für unkritische, nicht rechenintensive oder einmalige Aufgaben. Erst wenn es zwingend erforderlich ist, mehr Kontrolle über die Ausführung zu haben, sollte die Thread-Klasse verwendet werden.

Die erste Wahl bei der Zugriffssynchronisation sollte die lock-Anweisung sein. Obwohl sie nicht den Funktionsumfang wie die Monitor-Klasse besitzt, werden Laufzeitfehler durch eine falsche Verwendung ausgeschlossen, bzw. schon bei der statischen Analyse des Programms erkannt. Erst wenn eine Bedingungs-synchronisation nötig ist, sollte auf die Monitor-Klasse zurückgegriffen werden. Bleibt zum Schluss noch die Mutex-Klasse. Ein Mutex Objekt sollte nur der Synchronisation zwischen Anwendungen dienen und nicht innerhalb einer. Dies ist unter anderem aus Performance Gründen sinnvoll.

Literatur

- [1] Jürgen Nehmer und Peter Sturm *Systemsoftware -Grundlagen moderner Betriebssysteme-*, dpunkt.verlag, 2001.
- [2] Tom Archer *Inside C# Objektorientiertes Programmieren der nächsten Generation mit C#*, .net Fachbibliothek, Microsoft Press, 2001.
- [3] Microsoft *Win32 Programmer's Reference Volume 2*, Microsoft Press, 1993.

Literatur

- [4] P. Drayton, B. Albahari, T. Neward *C# in a Nutshell*, O'Reily, 2002.