HSLU Hochschule Luzern

I.BA_CSA
C# in Action

Teil 2
Parallele und Verteilte Programmierung

Roger Diehl, Luzern

Inhalt

Übungen

Quellen und Links

- `https://www.albahari.com/threading/` Für die Themen und Beispiele aus Kapitel 1 und 2 wurde die Dokumentation von Joseph Albahari als Basis verwendet. In dieser Dokumentation sind weitere Themen und Vertiefungen zu C# Threads zu finden.
- `https://codeplanet.eu/tutorials/csharp/4-tcp-ip-socket-programmierung-in-csharp.html` Dieses Tutorial ist eine gute Ergänzung und Vertiefung zum Kapitel 3. Es enthält weitere interessante Themen, wie Formulare im Internet senden (HTTP - Hypertext Transfer Protocol).

1. C# Threads

1.1. Der Thread Lebenszyklus

Da von der Common Language Runtime (CLR) mehrere Threads gestartet werden können, kann es sein, dass ein Thread mit einem oder mehreren anderen Threads interagiert. So kann z.B. die Ausgabe eines Threads als Eingabe für einen weiteren Thread verwendet werden. Müssen die Threads dadurch aufeinander warten, sind sie blockiert. Andererseits ist es auch möglich, dass ein laufender Thread, von der CLR unterbrochen wird. Das kann passieren, wenn der Scheduler der CLR auf einen anderen Thread umschalten will. Ein Thread kann nun während seiner Laufzeit folgende Lebenszyklen durchlaufen:

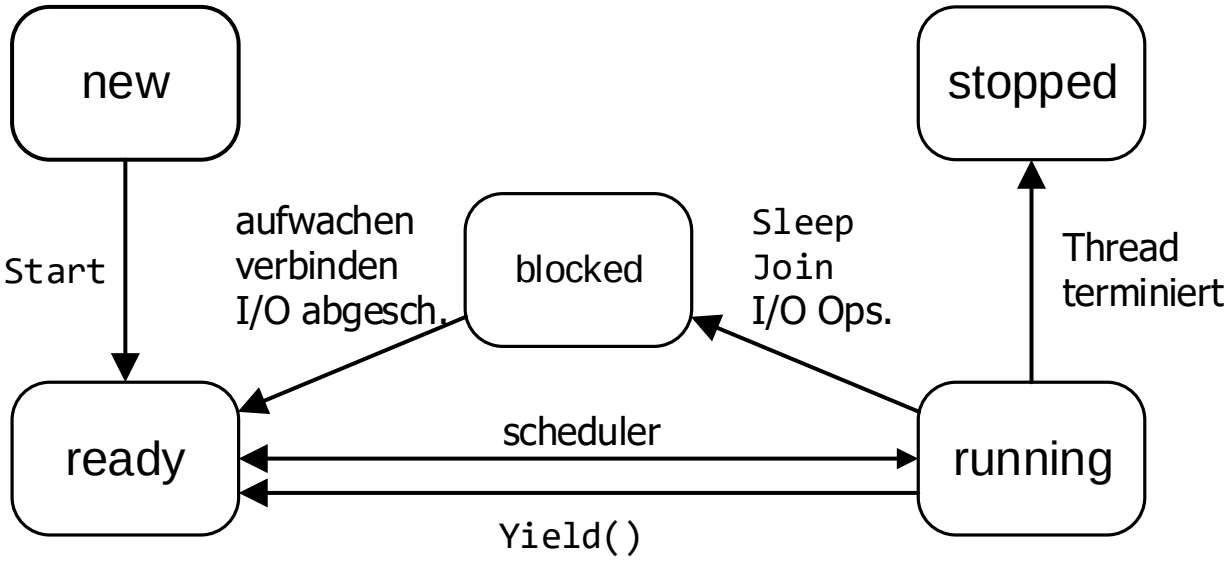

Abbildung 1: Der Thread Lebenszyklus

Legende

new	Das Thread-Objekt ist erzeugt, aber noch nicht gestartet.
ready	Der Thread ist gestartet, lokal Speicher (Stack) ist zugeteilt, er wartet nur noch auf die Zuweisung des Prozessors.
running	Der Thread führt seine Anweisungen auf dem Prozessor aus.
blocked	Der Thread muss warten, bis eine Bedingung erfüllt wird. Dies sind: Warten auf den Ablauf einer gewissen Zeit oder Ende eines anderen Threads. Ein Thread kann auch in den Zustand «blocked» gelangen, wenn eine Betriebssystemroutine aufgerufen wird. Das geschieht beispielsweise bei File-Operationen. Sollte von einem Stream gelesen werden und sind noch keine Daten vorhanden, so kann das Betriebssystem den Thread in den «blocked» Zustand führen.
stopped	Der Thread existiert nicht mehr. Das Thread-Objekt jedoch schon und kann, falls es referenziert ist, benutzt werden. Andernfalls wird es vom Garbage Collector entfernt.

Ein kleiner Thread Überblick

Die C# Implementierung von Thread ist im Namespace `System.Threading` enthalten. Der Namespace muss importiert (`using System.Threading`) werden.

Für die Erzeugen eines Threads muss die Thread-Klasse instanziiert werden. Dem Konstruktor ist die Methode zu übergeben, die nebenläufig ausgeführt werden soll.

```
class ThreadDemo1 {
    static void Main() {
        Thread t = new Thread(WriteY);
        t.Start();
        for (int i = 0; i < 1000; i++) Console.Write("x");
    }
    static void WriteY() {
        for (int i = 0; i < 1000; i++) Console.Write("y");
    }
}
```

Codeskizze 1: Projektmappe «Threads Erzeugen» / Projekt ThreadDemo1

Jeder Thread hat sein eigenes Stack, sodass lokale Variablen darauf abgelegt werden können:

```
class ThreadDemo2 {
    static void Main() {
        new Thread(Go).Start();
        ThreadDemo2.Go();
    }
    static void Go() {
        for (int i = 0; i < 10; i++) {
            Console.Write('x');
        }
    }
}
```

Codeskizze 2: Projektmappe «Threads Erzeugen» / Projekt ThreadDemo2

Thread können auch ein gemeinsames Stack, d.h. gemeinsame Ressourcen, miteinander teilen:

```
class ThreadDemo3 {
    private bool done = false;
    static void Main() {
        ThreadDemo3 tt = new ThreadDemo3();
        new Thread(tt.Go).Start();
        tt.Go();
    }
    void Go() {
        if (!done) {
            //Thread.Yield();
            done = true;
            Console.WriteLine("Done");
        }
    }
}
```

Codeskizze 3: Projektmappe «Threads Erzeugen» / Projekt ThreadDemo3

Frage: Was könnte in `ThreadDemo3` passieren?

1.2. Erzeugen und starten von Threads

Im vorhergehenden Arbeitsblatt wurden Threads in der verkürzten Schreibeweise erstellt und gestartet. Üblicherweise wird mit Hilfe der Methode `public delegate void ThreadStart()` angegeben, bei welcher Methode der Thread zu starten hat.

```
class ThreadDemo4 {

    static void Main() {
        Thread t = new Thread(new ThreadStart(Go));
        t.Start();
        Go();
    }
    static void Go() {
        Console.WriteLine("hello!");
    }
}
```

Codeskizze 4: Projektmappe «Threads Erzeugen» / Projekt ThreadDemo4

Selbstverständlich lassen sich auch Argumente an eine als Thread auszuführende Methode übergeben. Dazu wird die Methode `public delegate void ParameterizedThreadStart(object obj)` benutzt:

```
class ThreadDemo5 {

    static void Main() {
        Thread t = new Thread(new ParameterizedThreadStart(Go));
        t.Start(true);
        Go(false);
    }
    static void Go(object upperCase) {
        bool upper = (bool)upperCase;
        Console.WriteLine(upper ? "HELLO!" : "hello!");
    }
}
```

Codeskizze 5: Projektmappe «Threads Erzeugen» / Projekt ThreadDemo5

Auch hier gibt es die Möglichkeit der verkürzten Schreibeweise:

```
class ThreadDemo6 {

    static void Main() {
        Thread t = new Thread(Go);
        t.Start(true);
        Go(false);
    }
    static void Go(object upperCase) {
        bool upper = (bool)upperCase;
        Console.WriteLine(upper ? "HELLO!" : "hello!");
    }
}
```

Codeskizze 6: Projektmappe «Threads Erzeugen» / Projekt ThreadDemo6

Ein Thread soll Zahlen von 1 bis n addieren und die Summe am Schluss ausgeben. `AdditionTask` ist eine gewöhnliche C#-Klasse. Sie implementiert eine Methode `Add` mit der geforderten Funktionalität.

```
class AdditionTask {

    private readonly int n;
    private readonly string id;

    public AdditionTask(string id, int n) {
        this.id = id;
        this.n = n;
    }
    public void Add() {
        long sum = 0;
        for (int i = 0; i <= n; i++) {
            sum += i;
        }
        Console.WriteLine("{0}: SUM ({1}) -> {2}", id, n, sum);
    }
}
```

Codeskizze 7: Projektmappe «Threads Erzeugen» / Projekt AdditionTask

Um die Klasse `AdditionTask` zu demonstrieren, werden drei Thread-Objekte erzeugt und gestartet.

```
class ThreadDemo7 {

    static void Main() {
        // Tasks erzeugen...
        AdditionTask work1 = new AdditionTask("A", 100_000_000);
        AdditionTask work2 = new AdditionTask("B", 100_000);
        AdditionTask work3 = new AdditionTask("C", 100);
        // Threads erzeugen...
        Thread t1 = new Thread(new ThreadStart(work1.Add));
        Thread t2 = new Thread(new ThreadStart(work2.Add));
        Thread t3 = new Thread(new ThreadStart(work3.Add));
        // Threads starten...
        t1.Start();
        t2.Start();
        t3.Start();
    }
}
```

Codeskizze 8: Projektmappe «Threads Erzeugen» / Projekt ThreadDemo7

Experiment:

Starten Sie das Programm `ThreadDemo7` und beantworten Sie folgende Fragen.

Fragen:

- Was stellen Sie beim Ausführen der drei Threads fest?
- Was stellen Sie bei der Ausführung fest, nachdem Sie die folgende Zeile vor dem Thread-Start eingefügt haben?
  `t2.Priority = ThreadPriority.Lowest;`
- Wie erklären Sie sich das Programmverhalten?

1.3. Beenden von Threads

Der Thread Lebenszyklus besagt, dass mit dem Ende der Thread-Methode auch der Thread beendet wird. Ist in der Methode eine Endlosschleife programmiert, würde dies dann ein theoretisch nie endender Thread bilden.

Allgemein ist ein Thread beendet, wenn eine der folgenden Bedingungen zutrifft:

- Die Thread-Methode wird ohne Fehler beendet.
- In der Thread-Methode tritt eine Ausnahme (`Exception`) auf, welche die Methode beendet.
- Der Thread wird von aussen abgebrochen.

Das Beispiel mit dem Thread, der Zahlen von 1 bis n addiert und am Schluss die Summe ausgibt, hat gezeigt wie eine `Add` Methode ohne Fehler beendet werden kann.

Tritt in der Thread-Methode eine Ausnahme auf, so kann vermutlich nicht mehr ordentlich weitergerechnet werden und die Beendung des Threads ist eine zwingende Folge davon. Damit nicht die ganze Applikation mit einer unschönen Meldung beendet wird, sollte eine Ausnahme Behandlung durchgeführt werden.

```
class ThreadBeenden1 {

    static void Main() {
        try {
            new Thread(Go).Start();
        }
        catch (Exception ex) {
            Console.WriteLine("Exception! " + ex.Message);
        }
        // mache was anderes
        for (int i = 1; i <= 10; i++) {
            Console.WriteLine("for loop {0}", i);
            Thread.Sleep(200);
        }
    }
    static void Go() {
        throw null;    // NullReferenceException wird ausgelöst
        Console.WriteLine("uups!");
    }
}
```

Codeskizze 9: Projektmappe «Threads Beenden» / Projekt ThreadBeenden1

Experiment:

Starten Sie das Programm `ThreadBeenden1` und beantworten Sie folgende Fragen.

Fragen:

- Was stellen Sie beim Ausführen des Threads fest?
- Wie erklären Sie sich das Programmverhalten?
- Wie müsste das Exception Handling durchgeführt werden?

Jeder Thread, bzw. dessen Methode, ist für das Exception Handling selber verantwortlich. Nur so können Threads individuell funktionieren und stören sich bei Ausnahmen nicht gegenseitig.

```
class ThreadBeenden2 {

    static void Main() {
        new Thread(GoX).Start();
        new Thread(GoY).Start();
        Console.WriteLine("finished!");
    }
    static void GoX() {
        try {
            for (int i = 0; i < 10; i++) {
                Console.Write("X");
                if (i == 2) {
                    throw null; // NullReferenceException wird ausgelöst
                }
            }
        }
        catch (Exception ex) {
            Console.WriteLine("Exception! {0}", ex.Message);
        }
    }
    static void GoY() {
        try {
            for (int i = 0; i < 10; i++) {
                Console.Write("Y");
            }
        }
        catch (Exception ex) {
            Console.WriteLine("Exception! {0}", ex.Message);
        }
    }
}
```

Codeskizze 10: Projektmappe «Threads Beenden» / Projekt ThreadBeenden2

Experiment:

Starten Sie das Programm `ThreadBeenden2` und beantworten Sie folgende Fragen.

Fragen:

- Was stellen Sie beim Ausführen des Threads fest?
- Wie erklären Sie sich das Programmverhalten?

Experiment:

Versetzen Sie die `try/catch` Anweisung der Methode `GoX` **in** die `for` Schlaufe.

Frage:

- Was passiert, wenn Sie die `try/catch` Anweisung der Methode `GoX` in die `for` Schlaufe versetzen?

Die dritte Möglichkeit, den Thread von aussen zu beenden, soll das folgende Beispiel zeigen. Das Terminieren des Threads wird mit der Methode `Abort` durchgeführt. Der Aufruf löst die Ausnahme `ThreadAbortException` aus. Damit ist es möglich, die Methode ordnungsgemäss zu beenden.

```
class ThreadStop3 {

    static void Main() {
        Thread t = new Thread(delegate () {
            try {
                Console.WriteLine("Thread starts.");
                while (true) ; // Endlosschleife
            }
            catch (ThreadAbortException) {
                Console.WriteLine("ThreadAbortException.");
                for (int i = 0; i < 100_000_000; i++) ;
            }
            finally {
                Console.WriteLine("Thread ends.");
            }
        });
        Console.WriteLine("1. " + t.ThreadState);
        t.Start();
        Thread.Sleep(1000);
        Console.WriteLine("2. " + t.ThreadState);
        t.Abort();
        Console.WriteLine("3. " + t.ThreadState);
        t.Join();
        Console.WriteLine("4. " + t.ThreadState);
    }
}
```

Codeskizze 11: Projektmappe «Threads Beenden» / Projekt ThreadBeenden3

Experiment:

Starten Sie das Programm `ThreadStop3` und beobachten Sie die Terminalausgaben.

Fragen:

- Was stellen Sie beim Ausführen des Threads fest?
- Was schliessen Sie aus dem Programmverhalten?

Achtung: `Abort` ist in .NET Core nicht implementiert. Allerdings gibt `Abort` in einer .NET Core Anwendung keinen Syntax Fehler. Erst in der Ausführung wird eine `PlatformNotSupportedException` geworfen.

Thread Lebenszyklus mit `AbortRequested`

Der zusätzliche Thread Zustand `AbortRequested` führt dazu, dass ein Thread nun während seiner Laufzeit folgende Lebenszyklen durchlaufen kann:

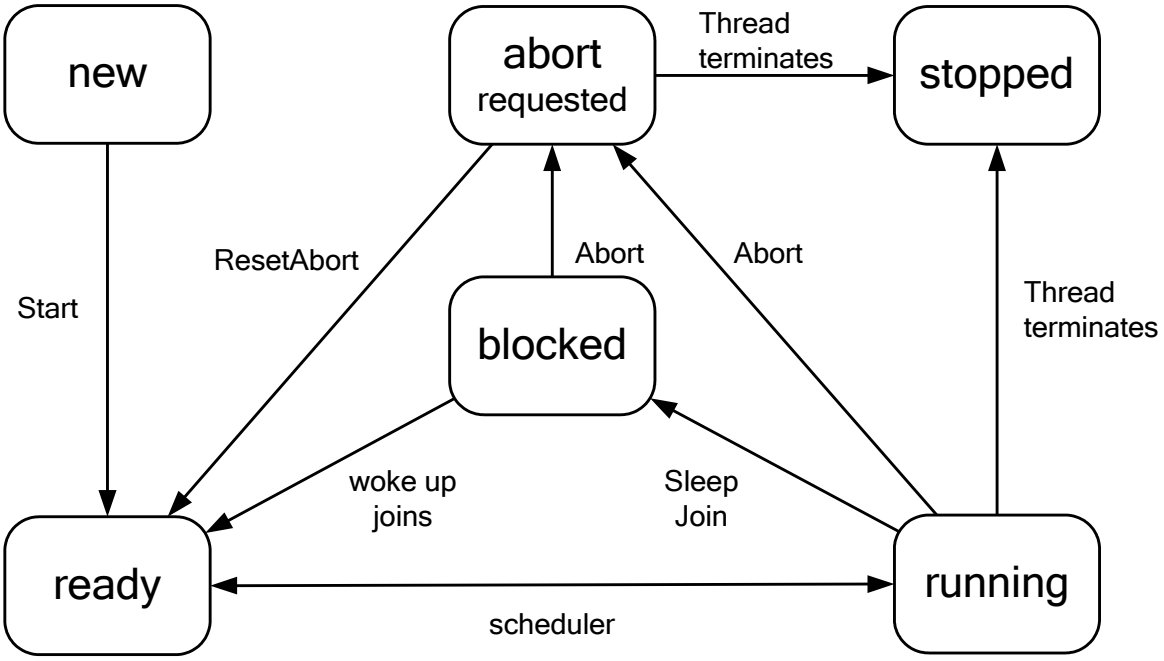

Abbildung 2: Der Thread Lebenszyklus mit Abort Zustand

Somit ist es möglich einen «abort requested» Thread wieder in den ausführungsbereiten Zustand zu versetzen:

```
class ThreadStop4 {

    static void Main() {
        Thread t = new Thread(Work);
        t.Start();
        Thread.Sleep(1000); t.Abort();
        Thread.Sleep(1000); t.Abort();
        Thread.Sleep(1000); t.Abort();
    }
    static void Work() {
        while (true) {
            try {
                while (true) ; // Endlosschleife
            }
            catch (ThreadAbortException) {
                Thread.ResetAbort();
            }
            Console.WriteLine("I will not die!");
        }
    }
}
```

Codeskizze 12: Projektmappe «Threads Beenden» / Projekt ThreadBeenden4

2. Synchronisation

Wenn innerhalb von parallel ablaufenden Threads gemeinsame Ressourcen benutzt werden, zum Beispiel Heaps, serielle Schnittstellen, Dateien, Fenster, etc., kann es passieren, dass gleichzeitig auf ein und dieselbe Ressource zugegriffen wird. An diesen Stellen ist Vorsicht geboten, weil die Inhalte dieser Ressource nach dem Zugriff eventuell falsche oder nicht erwartete Werte enthalten können. Um zu verhindern, dass eine gemeinsam genutzte Ressource durch den Zugriff mehrerer Threads beschädigt wird, muss man Konstrukte zur Thread Synchronisierung in den Code einfügen. Falls man dies nicht tut, können folgende Situationen auftreten:

Race Conditions

Ein kritischer Wettlauf, auch Wettlaufsituation (engl. Race Condition oder Race Hazard) ist in der Programmierung eine Konstellation, in denen das Ergebnis einer Operation vom zeitlichen Verhalten bestimmter Einzeloperationen abhängt.

Nehmen wir beispielsweise an, es gibt zwei Threads - einer ist dafür zuständig, eine Datei zu öffnen, und der andere, soll in diese Datei schreiben. Man muss nun dafür sorgen, dass der erste Thread rechtzeitig die Datei öffnet, nämlich noch bevor der zweite Thread aktiv wird und Daten in die Datei schreiben will. Die beiden Threads können nicht unabhängig voneinander laufen, man muss sicherstellen, dass der erste Thread fertig ist, bevor der zweite Thread beginnt.

Deadlocks

Der Deadlock oder auch tödliche Umklammerung bezeichnet in der Informatik einen Zustand, bei dem ein oder mehrere Prozesse auf Betriebsmittel warten, die dem Prozess selbst oder einem anderen beteiligten Prozess zugeteilt sind. Es warten zwei oder mehrere Threads aufeinander, und keiner kann sich aus dieser Situation befreien.

Nehmen wir an, wir haben zwei Threads, `Thread1` und `Thread2`. `Thread1` sperrt ein Objekt vom Typ `Student` und versucht dann, den Zugriff auf eine Datenbankzeile zu sperren, damit er alleine darauf zugreifen kann. Dabei stellt sich heraus, dass `Thread2` diese Zeile bereits gesperrt hat.

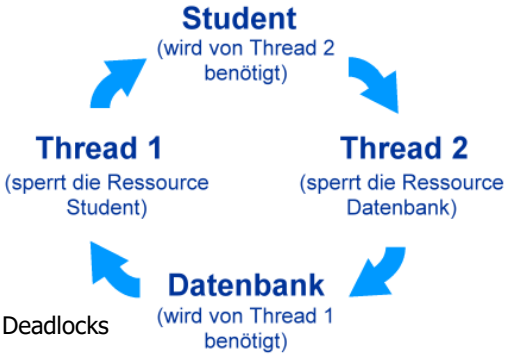

Abbildung 3: Beispiel eines Deadlocks

Eine gute Lösung für den gegenseitigen Ausschluss muss vier Bedingungen erfüllen:

1. In einem kritischen Abschnitt darf sich zu jedem Zeitpunkt höchstens immer nur ein Thread befinden.
2. Es dürfen keine Annahmen über die zugrunde liegende Hardware (Clock, CPU-Anzahl etc.) gemacht werden.
3. Ein Thread darf andere Threads nicht blockieren, ausser er ist in einem kritischen Bereich.
4. Es muss sichergestellt sein, dass ein Thread nicht unendlich lange warten muss, bis er in den kritischen Bereich eintreten kann.

[Andrew S. Tanenbaum 1994: Moderne Betriebssysteme]

2.1. Einfache Synchronisation

Die Thread-Synchronisierung umfasst Techniken, die sicherstellen sollen, dass mehrere Threads ihre Zugriffe auf gemeinsam benutzte Ressourcen koordinieren. Diese Koordination kann auf unterschiedliche Weise erreicht werden.

Blockieren

Idee: Die Threads stimmen sich zeitlich ab.

Im folgenden Beispiel teilen sich der Main-Thread und der Thread `t` die gemeinsam Ressource `long sum`. Thread `summieren` führt die Addition durch, der Main-Thread gibt das Resultat aus.

```
class SimpleBlocking {

    static void Main() {
        long sum = 0;
        bool fertig = false;
        Thread summieren = new Thread(delegate () {
            for (int i = 0; i <= 1_000_000_000; i++) {
                sum += i;
            }
            fertig = true;
        });
        summieren.Start();
        // Variante 1
        while (!fertig);
        // Variante 2
        //Thread.Sleep(10);
        // Variante 3
        //t.Join();
        Console.WriteLine("Summe = {0}", sum);
    }
}
```

Codeskizze 13: Projektmappe «Synchronisation» / Projekt SimpleBlocking

Experiment:

Starten Sie das Programm `SimpleBlocking` mit jeweils einer Variante der einfachen Blockierung und beantworten Sie die folgenden Fragen.

Fragen:

- Funktionieren alle drei Varianten?
- In welchen Zuständen befinden sich die Threads in den drei Varianten?
- Bewerten Sie die drei Varianten. Welche ist die beste Variante? Warum?

lock-Konstrukt

Idee: Die Threads reservieren einen Codebereich für sich.

Zur Synchronisation nebenläufiger Threads wurde das Konzept des Monitors implementiert. C. A. R. Hoare hat dieses Konzept im Aufsatz „Communicating Sequential Processes“ von 1978 erstmals veröffentlicht. Ein Monitor ist die Kapselung eines kritischen Bereichs (also eines Programmteils, der nur von jeweils einem Thread zurzeit durchlaufen werden darf) mit Hilfe einer automatisch verwalteten Sperre. Diese Sperre wird beim Betreten des Monitors gesetzt und beim Verlassen wieder zurückgenommen. Ist sie beim Eintritt in den Monitor bereits von einem anderen Thread gesetzt, muss der aktuelle Thread warten, bis der Konkurrent die Sperre freigegeben und den Monitor verlassen hat.

C# bietet hierfür die Anweisung `lock`, die sicherstellt, dass immer nur ein Thread zu einem gewissen Zeitpunkt einen Codebereich betreten kann. Das `lock` Konstrukt von C# entspricht dem `synchronized` in Java. Allerdings können mit dem `lock` Konstrukt nur Codeblöcke geschützt werden und nicht ganze Methoden.

Im folgenden Beispiel teilen sich zwei Threads (`t1` und `t2`) die gemeinsame Ressource `long sum`. Die kritischen Codebereiche werden durch das gemeinsame Objekt `locker` geschützt.

```
class LockingConstruct1 {

    private static readonly object locker = new object();

    static void Main() {
        long sum = 0;
        Thread t1 = new Thread(delegate () {
            lock (locker) {
                for (int i = 0; i <= 1_000_000; i++) {
                    sum += i;
                }
            }
        });
        Thread t2 = new Thread(delegate () {
            lock (locker) {
                Console.WriteLine("Summe = {0}", sum);
            }
        });
        t1.Start();
        t2.Start();
    }
}
```

Codeskizze 14: Projektmappe «Synchronisation» / Projekt LockingConstruct1

Experiment:

Starten Sie das Programm `LockingConstruct1` und beantworten Sie die folgenden Fragen.

Fragen:

- Warum wird die Summe richtig berechnet?
- Können Sie sich eine Situation vorstellen, wo die Summe nicht richtig berechnet wird?

Thread Lebenszyklus mit lock-Konstrukt

Mit dieser Art der Synchronisation erhält der Thread Lebenszyklus einen zusätzlichen Zustand, den Object lock-pool. Jedes Objekt hat genau einen Object lock-pool (Abbildung 4).

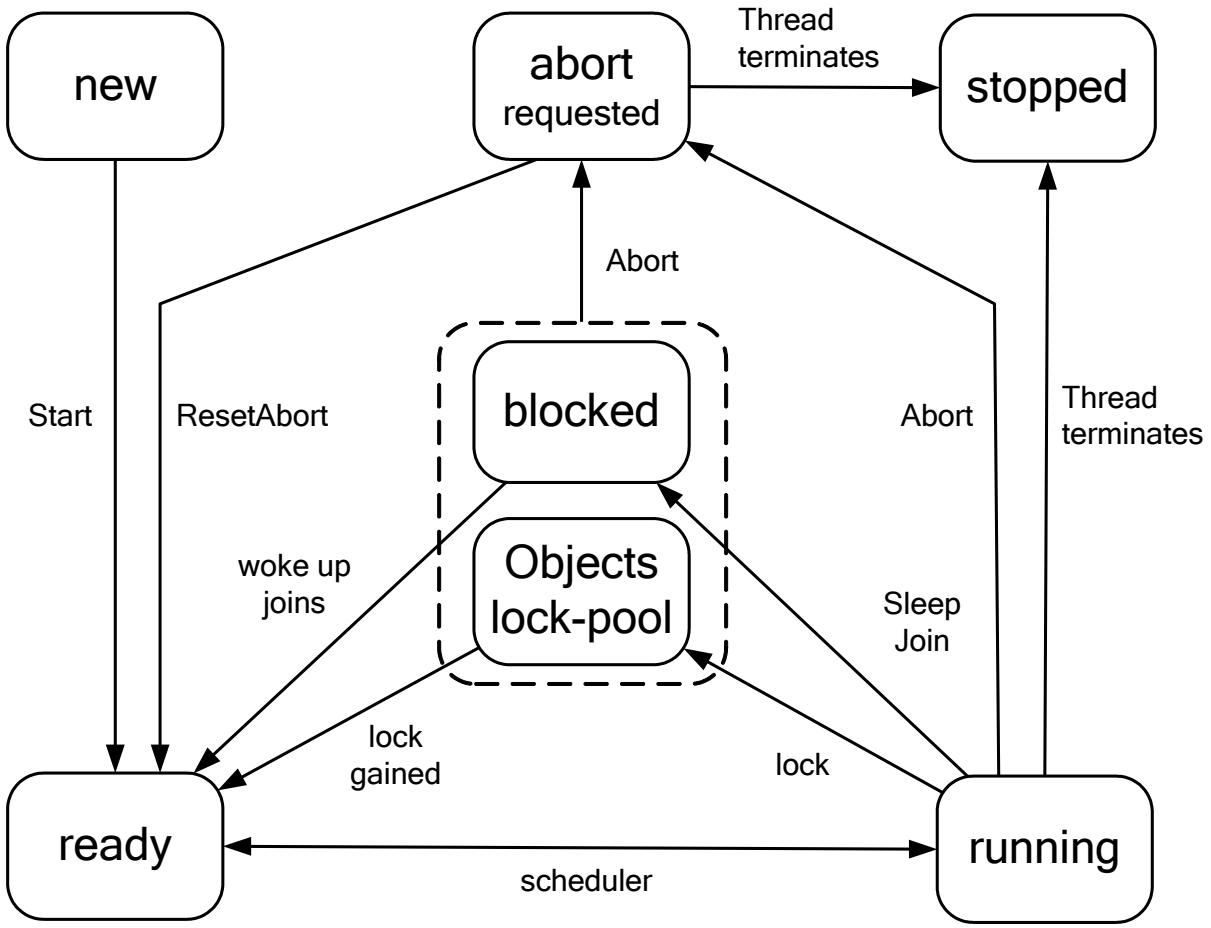

Abbildung 4: Der Thread Lebenszyklus mit lock-Konstrukt

Das `lock` Konstrukt von C# ist eine syntaktische Kurzform und entspricht folgendem Code:

```
private static readonly object locker = new object();
...
Monitor.Enter(locker);
try {
   // kritischer Bereich
}
finally {
    Monitor.Exit(locker);
}
```

Codeskizze 15: die Klasse `Monitor` setzt das Lock Konstrukt um

Die `Monitor` Klasse stammt aus dem Namespace `System.Threading`. Ein Beispiel zur `Monitor` Klasse finden Sie im Programm `LockingConstruct2`.

Übung 1: Gemeinsamer Zähler

Im folgenden Programm wird auf einen für alle Threads gemeinsamer Zähler (Klasse `Counter`) referenziert. Ein Thread gibt in der `Go` Methode seine ID und den Zählerstand mit der Methode `NextNumber` aus und wartet dann 50 msec.

```
class Counter {
    private int count = 0;
    public int NextNumber() {
        count++;
        return count;
    }
}

class DemoCounter {
    private readonly string id;
    private readonly Counter counter;

    public DemoCounter(string id, Counter counter) {
        this.id = id;
        this.counter = counter;
    }
    void Go() {
        for (int i = 0; i < 10; i++) {
            Console.WriteLine(id + counter.NextNumber());
            Thread.Sleep(50);
        }
    }
    static void Main() {
        Counter counter = new Counter();
        DemoCounter ct1 = new DemoCounter("T1: ", counter);
        DemoCounter ct2 = new DemoCounter("T2: ", counter);
        DemoCounter ct3 = new DemoCounter("T3: ", counter);
        new Thread(ct1.Go).Start();
        new Thread(ct2.Go).Start();
        new Thread(ct3.Go).Start();
    }
}
```

Codeskizze 16: Projektmappe «Thread Uebungen» / Projekt Counter

Startet man das Programm so wird in den meisten Fällen der `Counter` eine eindeutige Nummer liefern. Allerdings könnte eine Ausgabe auch so aussehen:

```
T2: 9
T3: 10
T2: 12 <---
T1: 12 <---
T3: 13
T1: 15 <---
T2: 15 <---
T3: 16
```

Beim Zählerstand 12, bzw. 15 ist offensichtlich bei der Ausgabe etwas passiert, was nicht gewollt war.

Fragen:

- Wie erklären Sie sich das Programmverhalten?
- Wo genau im Code tritt das Problem auf?
- Wie müsste eine korrekte Lösung aussehen?

2.2. Synchronisierungsereignisse

WaitHandle

Idee: Die Threads warten an einem inaktiven Ereignisobjekt, bis dieses aktiv (frei) geschaltet wird.

Es gibt zwei Arten von Synchronisierungsereignissen: `AutoResetEvent` und `ManualResetEvent`. Der einzige Unterschied zwischen den beiden besteht darin, dass `AutoResetEvent` automatisch von signalisiert (aktiv) zu nicht signalisiert (inaktiv) geändert wird, wenn das Ereignis einen Thread aktiviert. Umgekehrt ist es mit `ManualResetEvent` möglich, eine beliebige Anzahl von Threads über den signalisierten Zustand zu aktivieren, und das Ereignis wird nur in den nicht signalisierten Zustand zurückgesetzt, wenn seine `Reset` Methode aufgerufen wird.

Anmerkung: Dieses C# Konstrukt entspricht nicht dem `wait` in Java.

Im folgenden Beispiel wird ein inaktives `EventWaitHandle` vom Typ `AutoResetEvent` erzeugt. Ein Thread ruft in der `Waiter` Methode am `EventWaitHandle` die `WaitOne` Methode auf. Damit wartet der Thread, bis das `EventWaitHandle` Objekt mit Hilfe der `Set` Methode aktiviert, und damit der Thread aus dem Wartezustand befreit wird.

```
class BasicWaitHandle {

    private static readonly EventWaitHandle wh = new AutoResetEvent(false);

    static void Main() {
        new Thread(Waiter).Start();
        Thread.Sleep(1000);
        wh.Set();
    }
    static void Waiter() {
        Console.WriteLine("Waiting...");
        wh.WaitOne();
        Console.WriteLine("...Notified");
    }
}
```

Codeskizze 17: Projektmappe «Synchronisation» / Projekt WaitHandles

Experiment:

Starten Sie das Programm `BasicWaitHandle` und beantworten Sie die folgenden Fragen.

Fragen:

- In welchem Zustand (an welchem Ort des Thread-Lebenszyklus) befindet sich ein Thread, der an der Methode `WaitOne` wartet?
- Was passiert, wenn Sie einen zweiten Thread starten?
- Wie erklären Sie sich das Verhalten des zweiten Threads?
- Gibt es eine Möglichkeit, dass der zweite Thread sich gleich wie der erste verhält?

Übung 2: Pferderennen

Latch

Ein praktischer Synchronisationsmechanismus ist das Latch. Latches sperren so lange, bis sie einmal ausgelöst werden und danach sind sie frei passierbar. Sie sollen nun ein Latch in C# schreiben und testen. Dazu ist ein Interface `ISynch` gegeben, das implementiert werden soll.

```
interface ISynch {
    void Acquire();
    void Release();
}
```

Codeskizze 18: Projektmappe «Thread Uebungen» / Projekt Latch

Für den Test veranstalten wir ein kleines Pferderennen. Dazu gibt es Pferde...

```
class RaceHorse {

    private readonly ISynch startSignal;
    private readonly int nr;

    public RaceHorse(int nr, ISynch startSignal) {
        this.nr = nr;
        this.startSignal = startSignal;
    }
    public void Run() {
        Console.WriteLine("Rennpferd {0} geht in die Startbox.", nr);
        startSignal.Acquire();
        Console.WriteLine("Rennpferd {0} läuft los.", nr);
        Thread.Sleep(new Random().Next(3000));
        Console.WriteLine("Rennpferd {0} ist im Ziel.", nr);
    }
}
```

Codeskizze 19: Projektmappe «Thread Uebungen» / Projekt Latch

...und eine Rennbahn.

```
class Turf {

    static void Main() {
        ISynch startBox = new Latch();
        for (int i = 1; i <= 5; i++) {
            new Thread(new RaceHorse(i, startBox).Run).Start();
        }
        Console.WriteLine("Start...");
        startBox.Release();
    }
}
```

Codeskizze 20: Projektmappe «Thread Uebungen» / Projekt Latch

Aufgabe:

Damit alle Pferde fair gestartet werden, kommen diese in eine Starterbox. Sobald diese geöffnet wird, sollen die Pferde loslaufen. Implementieren Sie mit diesen Vorgaben die Klasse `Latch`.

Frage: Ist das Rennen wirklich gerecht?

Wait und Pulse

Idee: Die Threads warten an einem Monitorobjekt, bis dieses einen Impuls erhält, um einen oder mehrere Threads frei zu schalten.

Wenn der Zugang zu einem kritischen Abschnitt von bestimmten Bedingungen oder Zuständen abhängt, so reicht das Konzept der einfachen Synchronisation mit `lock` allein nicht aus. Mit Hilfe der Klasse `Monitor` lässt sich dieser Zugang steuern. Um einen Thread auf eine Bedingung oder Zustand warten zu lassen stellt C# die `Monitor` Methoden `Wait` und `Pulse` zur Verfügung. Das zu übergebende Argument `Object obj` stellt das Monitorobjekt dar.

```
public static bool Wait(Object obj)
public static bool Wait(Object obj, int millisecondsTimeout)
public static bool Wait(Object obj, TimeSpan timeout)

public static void Pulse(Object obj)
public static void PulseAll(Object obj)
```

Codeskizze 21: Varianten von `Wait` und `Pulse`

Die Methoden `Wait` und `Pulse` werden wie folgt aufgerufen. Wobei das Monitorobjekt `obj` ein beliebiges Objekt sein kann.

```
Monitor.Wait(obj);
Monitor.Pulse(obj);
```

Codeskizze 22: Klasse `Monitor` mit den Methoden `Wait` und `Pulse`

Wichtig: Sowohl `Wait` als auch `Pulse` dürfen nur innerhalb eines `lock` Abschnitts aufgerufen werden, dabei muss das Monitorobjekt den Lock-pool stellen.

Bei Aufruf von `Wait` wird der aufrufende Thread in einen Wartezustand versetzt, und gleichzeitig wird der Lock auf diesen Abschnitt freigegeben. Es führen genau drei Wege aus dem Warte-Zustand wieder heraus:

- Ein anderer Thread signalisiert den Zustandswechsel mittels `Pulse` bzw. `PulseAll`.
- Die angegebene Zeit (Timeout) ist abgelaufen.
- Ein anderer Thread ruft die Methode `Abort` des wartenden Threads auf.

Dann wartet der Thread eventuell noch einmal, bis er den Lock für den Abschnitt hat und wird dann wieder «ready», d.h. bereit zur erneuten Ausführung. `Pulse` weckt genau einen Thread im «wait» Zustand auf. Falls mehrere Threads warten, ist nicht vorhersehbar oder bestimmbar, welcher Thread aufgeweckt wird. `PulseAll` weckt alle im Lock-pool wartenden Threads auf. Der Rückgabewert der `Wait` Methode ist in diesem Fall `true`.

Die Variante von `Wait` mit Timeout ist das «timed wait», bei dem eine Zeit angegeben wird, wie lange maximal auf das Eintreten einer Bedingung gewartet werden soll. Falls die Zeit verstrichen ist, terminiert die Methode, als ob ein `Pulse` stattgefunden hätte. Das heisst, auch der Lock wird dann wieder von diesem Thread gehalten. Der Rückgabewert der `Wait` Methode ist in diesem Fall `false`.

Achtung: Es sollte die Bedingung, die zum Wartezustand geführt hat, erneut getestet werden, um festzustellen, ob nur die Zeit abgelaufen ist oder die Bedingung tatsächlich erfüllt ist.

Ähnlich wie beim «blocked» Zustand, kann ein Thread im Wartezustand von einem anderen Thread unterbrochen werden, in dem er die `Abort` Methode des wartenden Threads aufruft.

Wait und Pulse (Fortsetzung)

Dieses C# Konstrukt entspricht dem `wait` und `notify` in Java.

Tabelle 1: Wait und Pulse im Vergleich zu Java

C#	Java

`Object x = new Object();`
`lock (x) {`
`  try {`
`  Monitor.Wait(x);`
`  } catch (ThreadInterruptedException) {}`
`  // weitere Aktionen...`
`}`
	`Object x = new Object();`
	`synchronized(x) {`
	`  try {`
	`  x.wait();`
	`  } catch (InterruptedException e) {}`
	`  // weitere Aktionen...`
	`}`

`// aufwecken...`
`lock (x) {`
`  Monitor.Pulse(x);`
`}`
	`// aufwecken...`
	`synchronized(x) {`
	`  x.notify();`
	`}`

Im folgenden Beispiel wird vom Main-Thread ein Thread erzeugt und gestartet. Dieser ruft in der `Waiter` Methode die `Wait` Methode auf. Damit wartet der Thread, bis der Main-Thread die Pulse `Pulse` Methode aufruft und damit der Thread aus dem Wartezustand befreit wird. Falls jedoch das Timeout Argument kleiner ist als die Sleep Time des Main Thread, wird der Thread ebenfalls aus dem Wartezustand befreit. Den Lock auf das `synch` Objekt erhält er aber erst, wenn dieses durch den Main Thread freigeben wurde.

```
class WaitAndPulse {

    private static readonly object synch = new object();

    static void Main() {
        new Thread(Waiter).Start();
        Thread.Sleep(1000);
        lock (synch) {
            Monitor.Pulse(synch);
        }
    }
    static void Waiter() {
        lock (synch) {
            Console.WriteLine("Waiting...");
            if (Monitor.Wait(synch, 1000)) {
                Console.WriteLine("...Notified");
            }
            else {
                Console.WriteLine("Timeout");
            }
        }
    }
}
```

Codeskizze 23: Projektmappe «Synchronisation» / Projekt WaitAndPulse

Achtung: `Pulse` wird nicht gespeichert. Dies bedeutet, wird `Pulse` vor dem `Wait` aufgerufen, wird der Thread ewig warten.

Thread Lebenszyklus mit Warteliste

Zusätzlich zu dem bereits erwähnten lock-Pool, der einem Objekt zugeordnet ist, besitzt ein Objekt auch noch eine Warteliste, einen Object wait-pool (Abbildung 5). Dabei handelt es sich um eine (möglicherweise leere) Menge von Threads, die vom Scheduler unterbrochen wurden und auf ein Ereignis warten, um fortgesetzt werden zu können.

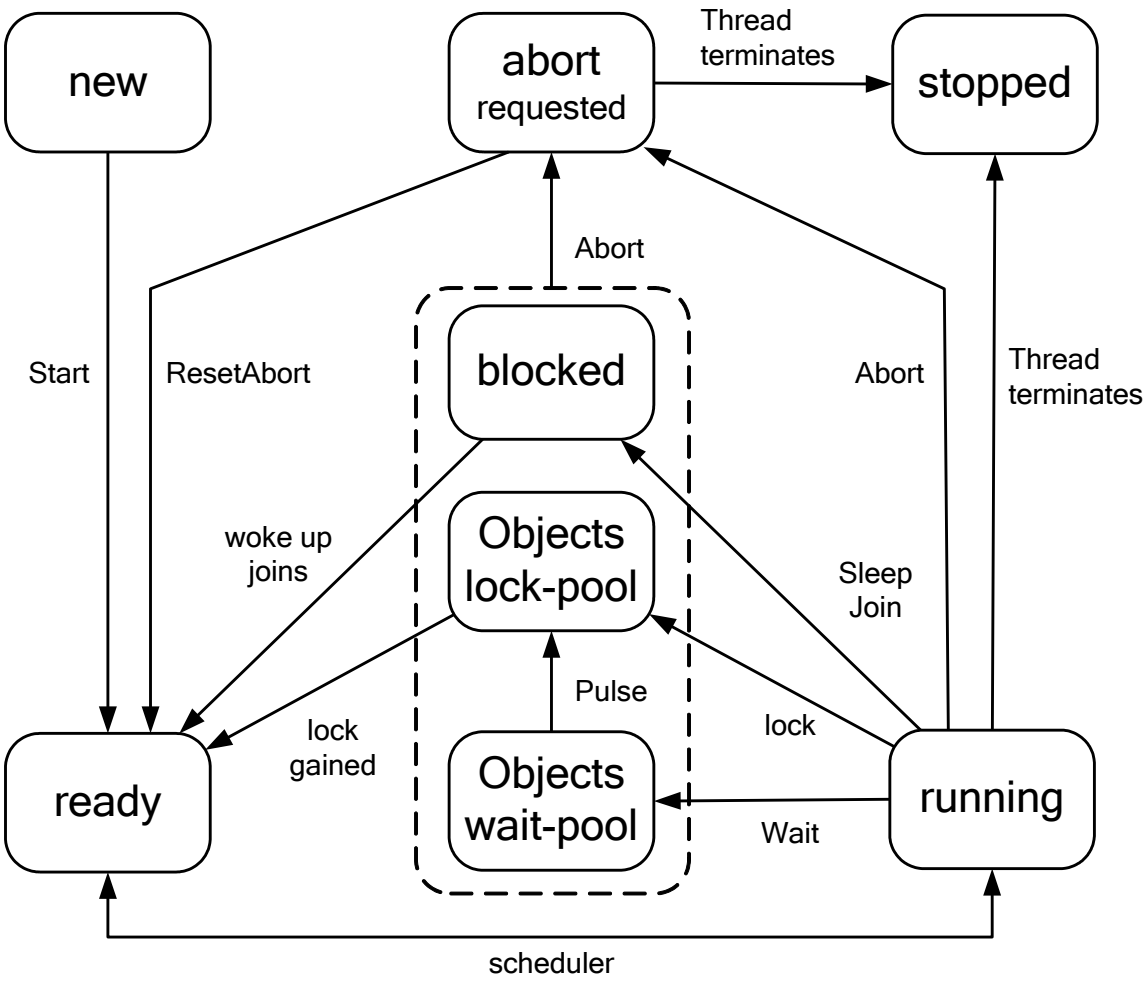

Abbildung 5: Der Thread Lebenszyklus mit wait-Pool

Wichtig: Der Objects lock-pool und der Objects wait-pool müssen zum gleichen Objekt gehören. Ist dies nicht der Fall führt dies während der Laufzeit zu einem Fehler:

`System.Threading.SynchronizationLockException`

Dieser Fehler besagt, dass der `Wait` aufrufende Thread nicht im Besitzt des Locks von dem Objekt ist, dessen wait-pool er benutzen möchte. Es muss sichergestellt sein, dass die beiden Objekt pools zum selben Objekt gehören.

```
Object synch = new Object();
lock (synch) {
    Monitor.Wait(synch);
}
```

Codeskizze 24: das Objekt `synch` als Monitor für den Lock- und Wait-Pool

Übung 3: Wait-Pool

Man möchte den Wait-Pool eines Objektes testen mit folgenden Klassen.

```
class MyTask {

    private readonly object synch;

    public MyTask(object synch) {
        this.synch = synch;
    }
    public void Run() {
        Console.WriteLine("warten...");
        lock (synch) {
            Monitor.Wait(this);
        }
        Console.WriteLine("...aufgewacht");
    }
}

class DemoWaitPool {

    static void Main() {
        Object synch = new Object();
        MyTask myTask = new MyTask(synch);
        new Thread(myTask.Run).Start();
        Thread.Sleep(1000);
        Monitor.Pulse(synch);
    }
}
```

Codeskizze 25: Projektmappe «Thread Uebungen» / Projekt WaitPool

Aufgabe:

Starten Sie das Programm `DemoWaitPool` und beantworten Sie die folgenden Fragen.

Fragen:

- Was passiert bei der Ausführung von `DemoWaitPool`?
- Wie erklären Sie sich das Verhalten der Klassen?
- Welche Korrekturen sind nötig, damit das Programm korrekt abläuft?

Übung 4: Nested Monitor

Gegeben ist die Klasse `NotifyingQueue`. Es ist eine Warteschlange, die zwei Methoden besitzt, um ein Objekt abzulegen, bzw. herzunehmen. Die Warteschlange ist als Ringbuffer implementiert. Ein Thread, der die Methode `Dequeue` aufruft, muss warten, falls die `NotifyingQueue` leer ist.

```
class NotifyingQueue<T> {

    private const int SIZE = 10;
    private T[] queue = new T[SIZE];
    private int head = 0;
    private int tail = 0;
    public void Enqueue(T item) {
        lock (this) {
            head++;
            head %= SIZE;
            queue[head] = item;
            Monitor.Pulse(this);
        }
    }
    public T Dequeue() {
        lock (this) {
            if (head == tail) {
                Monitor.Wait(this);
            }
            tail++;
            tail %= SIZE;
            return queue[tail];
        }
    }
}
```

Codeskizze 25: Projektmappe «Thread Uebungen» / Projekt NestedMonitor

Frage:

- Was muss an der Klasse `NotifyingQueue` verbessert werden?

Übung 4: Nested Monitor (Fortsetzung)

Zusätzlich ist die Klasse `BlackHole` gegeben. In einem schwarzen Loch kann etwas verschwinden (`Put`) oder etwas herauskommen (`Get`).

```
class BlackHole {

    private readonly NotifyingQueue<string> queue = new();

    public void Put(string thing) {
        lock (this) {
            queue.Enqueue(thing);
        }
    }
    public String Get() {
        lock (this) {
            return queue.Dequeue();
        }
    }
}
```

Codeskizze 26: Projektmappe «Thread Uebungen» / Projekt NestedMonitor

Zugegeben, es ist nicht erwiesen, dass irgendetwas, das in einem schwarzen Loch verschwunden ist auch tatsächlich wieder herauskommt. Nun, Sie sollen das beweisen mit folgendem Programm:

```
class DemoBlackHole {

    public static readonly BlackHole blackhole = new BlackHole();

    public static void Main() {
        Thread t1 = new Thread(delegate() {
            Console.WriteLine(blackhole.Get().ToString());
        });
        Thread t2 = new Thread(delegate() {
            blackhole.Put("Sonne, Licht, irgendetwas...");
        });
        Console.WriteLine("Wir starten die Untersuchung...");
        t1.Start();
        t2.Start();
        t1.Join();
        t2.Join();
        Console.WriteLine("Was ist passiert...");
    }
}
```

Codeskizze 27: Projektmappe «Thread Uebungen» / Projekt NestedMonitor

Aufgabe:

Starten Sie das Programm `DemoBlackHole` und beantworten Sie die folgenden Fragen.

Fragen:

- Was stellen Sie beim Ausführen von `DemoBlackHole` fest?
- Wie erklären Sie sich das Programmverhalten?
- Wie müsste eine korrekte Lösung aussehen?

Übung 5: Lost Signals

Objekte der Klasse `Worker` können bei einem Objekt der Klasse `Operator` eine bestimmte Anzahl (`MAXCOUNT`) Aufgaben abholen. Bevor die Aufgaben zur Abholung bereit stehen muss der `Operator` diese initialisieren. Ist die Initialisierung abgeschlossen, werden `MAXCOUNT` am `ISynch` Signal wartenden `Worker` (hier vier) befreit.

```
interface ISynch {

    void Acquire();
    void Release();
}
```

Codeskizze 28: Projektmappe «Thread Uebungen» / Projekt LostSignals

Hat ein `Worker` seine Aufgabe abgeschlossen, so informiert er mit Aufruf der Methode `Done` den `Operator`. Sobald alle ausstehenden Aufgaben gelöst sind, initialisiert der `Operator` eine neue Anzahl Aufgaben.

```
class Operator {

    public const int MAXCOUNT = 4;
    private int counter = 0;
    private readonly ISynch signal;
    private readonly Random rnd;
    private readonly EventWaitHandle next = new AutoResetEvent(false);

    public Operator(ISynch signal) {
        this.signal = signal;
        rnd = new Random();
    }
    private void Init() {
        lock (this) {
            counter = MAXCOUNT;
            Console.WriteLine("Es wurden {0} Operationen vorbereitet...",
                counter);
        }
    }
    public int Operation() {
        lock (this) {
            return rnd.Next(1000);
        }
    }
    public void Done(string id) {
        lock (this) {
            counter++;
            Console.WriteLine("{0}. Operation wurde durch {1} beendet.",
                counter, id);
            if (counter == MAXCOUNT) {
                next.Set();
            }
        }
    }

//... weiterer Code siehe nächste Seite
```

Codeskizze 29: Projektmappe «Thread Uebungen» / Projekt LostSignals

Übung 5: Lost Signals (Fortsetzung)

```
    public void Do() {
        while (true) {
            this.Init();
            lock (this) {
                for (; counter > 0; counter--) {
                    signal.Release();
                }
            }
            next.WaitOne();
        }
    }
}
```

Codeskizze 30: Projektmappe «Thread Uebungen» / Projekt LostSignals

Ein `Worker` wartet am `ISynch` Signal bis ein Release vom `Operator` ausgeführt wird. Dann wird eine Aufgabe geholt (hier Aufruf der Methode `Operation`) und ausgeführt. Ist die Aufgabe beendet, wird der `Operator` mit der Methode `Done` benachrichtigt.

```
class Worker {

    private readonly ISynch signal;
    private readonly Operator op;
    private readonly string id;

    public Worker(ISynch signal, Operator op, string id) {
        this.signal = signal;
        this.op = op;
        this.id = id;
    }
    public void Do() {
        while (true) {
            signal.Acquire();
            Console.WriteLine("{0} released...", id);
            Thread.Sleep(op.Operation());
            op.Done(id);
        }
    }
}
```

Codeskizze 31: Projektmappe «Thread Uebungen» / Projekt LostSignals

Aufgabe:

Schreiben Sie eine Klasse `AWait`, welche das Interface `ISynch` implementiert.

Anmerkung: Das oben beschriebene Verfahren ähnelt der Scatter/Gather Verarbeitung. Ein Verfahren zur Berechnung eines komplexen Problems in mehreren Schritten. In der Scatter Phase werden dabei Teilprobleme an einzelne Prozesse (workers) zugeteilt, die daraufhin unabhängig voneinander Zwischenlösungen berechnen. In der Gather Phase werden diese Zwischenlösungen eingesammelt und zu einer Gesamt-Zwischenlösung zusammengefasst, die dann in einem weiteren Scatter/Gather Zyklus erneut in Teilprobleme zerlegt wird. Dies wird solange wiederholt, bis das Endergebnis vorliegt.

Semaphore

Ein allgemeineres Konzept für die Synchronisation sind die so genannten Semaphoren (Signalmasten, Leuchttürme). Die Signalmasten zeigen ähnlich wie bei der Eisenbahn an, ob in eine Strecke (hier jetzt ein kritischer Bereich) eingefahren werden darf, oder ob ein Zug (hier ein Thread oder Prozess) halten muss. Im Gegensatz zum Mutex können sie nicht nur die Zustände 'verschlossen' und 'frei' einnehmen, sondern beliebig viele (abzählbar viele). Semaphoren wurden zuerst entwickelt, um das Problem der «lost signals» zu lösen.

Das Semaphor Konzept wurde 1968 von E.W. Dijkstra eingeführt. Es gibt zwei Operationen `V' (Abkürzung für holländisch Verhogen = Erhöhen) und `P' (für Proberen = Probieren) auf das Semaphor `sema` angewendet:

`sema.P()`: entspricht dem Eintritt (Passieren) in einen synchronisierten Bereich, wobei mitgezählt wird, der wievielte Eintritt es ist.

`sema.V()`: entspricht dem Verlassen (Freigeben) eines synchronisierten Bereiches, wobei ebenfalls mitgezählt wird, wie oft der Bereich verlassen wird.

Im folgenden Beispiel wird ein Semaphor erzeugt, das beim Start der Applikation die Parameter «Available» und «Capacity» initialisiert. Das heisst, der Zähler wird auf drei initialisiert und maximal drei Thread können den kritischen Abschnitt betreten.

```
class SemaphoreDemo {

    private static readonly Semaphore sema = new Semaphore(3, 3);

    static void Main() {
        for (int i = 0; i < 5; i++) {
            new Thread(Go).Start(i);
        }
    }
    static void Go(object number) {
        for (int i = 0; i < 5; i++) {
            Console.WriteLine("{0}. Thread waits.", number);
            sema.WaitOne();
            Console.WriteLine("{0}. Thread is in critical section", number);
            Thread.Sleep(1000); // Only 3 threads can get here at once
            sema.Release();
            Console.WriteLine("{0}. Thread leaves.", number);
        }
    }
}
```

Codeskizze 32: Projektmappe «Synchronisation» / Projekt Semaphore

Experiment:

Starten Sie das Programm `SemaphoreDemo` und beantworten Sie folgende Fragen.

Fragen:

- Welche Abhängigkeit besteht zwischen den Argumenten «Available» und «Capacity» des Konstruktors der Klasse `Semaphore`?
- Was passiert, wenn Sie die Initialisierung des Semaphors ändern?
- Wie sieht die Lösung für die Übung «Lost Signals» mit der Klasse `Semaphore` aus?

Mutex

Mutex (Abk. für engl. Mutual Exclusion, «wechselseitiger Ausschluss») bezeichnet Verfahren, mit denen verhindert wird, dass nebenläufige Prozesse bzw. Threads gleichzeitig auf Daten zugreifen und so unter Umständen inkonsistente Zustände herbeiführen. Das Mutex ist keineswegs identisch mit einem binären Semaphor, da Semaphore von anderen Aktivitätsträgern freigegeben werden dürfen, d.h. der Thread, der einen Mutex besitzt, muss diesen auch wieder freigeben. Wenn ein Thread beendet wird, während er einen Mutex besitzt, wird der Mutex abgebrochen. Der Zustand vom Mutex wird auf signalisiert festgelegt, und der Besitz geht auf den nächsten Thread in der Warteschlange über.

Wichtig: C# bietet die `Mutex` Klasse an. Sie ist eigentlich eine redundante Implementation zum `lock` Konstrukt. Aber das Mutex bietet die Möglichkeit das Sperren von Codebereichen in Threads Prozess übergreifend einsetzen zu können.

Im folgenden Beispiel wird ein Mutex erzeugt, welches beim Start der Applikation während 5 Sekunden geprüft wird, ob es frei geschaltet ist. Ist dies nicht der Fall wird die Applikation wieder beendet.

```
class MutexDemo {

    // Use a name unique to the application (eg include your company URL)
    private static readonly Mutex mutex =
        new Mutex(false, @"Global\OneAtATimeDemo");

    static void Main() {
        // Wait 5 seconds if contended – in case another instance
        // of the program is in the process of shutting down.
        if (!mutex.WaitOne(TimeSpan.FromSeconds(5), false)) {
            Console.WriteLine("Another instance of the app is running.");
            return;
        }
        try {
            Console.WriteLine("Running - press Enter to exit");
            Console.ReadLine();
        }
        finally {
            mutex.ReleaseMutex();
        }
    }
}
```

Codeskizze 33: Projektmappe «Synchronisation» / Projekt Mutex

Experiment:

Starten Sie das Programm `MutexDemo` und beantworten Sie folgende Fragen.

Fragen:

- Was passiert in diesem Programm?
- Wo kommen in diesem Programm Threads und wo Prozesse vor?
- Erklären Sie anhand dieses Programms was «Prozess übergreifend» bedeutet.
- In welchem Zustand befindet sich ein Thread, der an der Methode `WaitOne` wartet?
- Wie sieht der Code aus, um das lock-Konstrukt mit der Klasse `Mutex` zu ersetzen?

Übung 6: Prozessübergreifende Synchronisation

Gegeben sind die folgenden drei Programme. Ein Zahlengenerator (`RandomCounter`), ein «Addierer» (`AddNumbers`), der die Zahlenreihe zusammenzählt und einen «Sucher» (`FindMinMaxNumber`), das den minimalen und maximalen Wert der Zahlenreihe ermittelt. Die Zahlenreihe steht im File `daten.txt`.

```
class RandomCounter {
    static void Main() {
        Random rnd = new Random();
        Console.WriteLine("Random counter");
        using (StreamWriter sw = new StreamWriter("daten.txt")) {
            int n = 1000 + rnd.Next(5000);
            for (int i = 0; i < n; i++) {
                int value = -25_000 + rnd.Next(50_000);
                sw.WriteLine(value);
            }
        }
    }
}

class AddNumbers {
    static void Main() {
        int sum = 0;
        Console.WriteLine("Add numbers");
        using (StreamReader sr = new StreamReader("daten.txt")) {
            string line;
            while ((line = sr.ReadLine()) != null) {
                sum += Int32.Parse(line);
            }
        }
    }
}

class FindMinMaxNumber {
    static void Main() {
        int min = 0;
        int max = 0;
        Console.WriteLine("Find min./max. number");
        using (StreamReader sr = new StreamReader("daten.txt")) {
            string line;
            while ((line = sr.ReadLine()) != null) {
                int num = Int32.Parse(line);
                min = min > num ? num : min;
                max = max < num ? num : max;
            }
        }
    }
}
```

Codeskizze 31: Projektmappe «Thread Uebungen» / Projekt ProcessSynch

Aufgabe:

Alle drei Programme sollen so schnell wie möglich und so parallel wie möglich ausgeführt werden. Gleichzeitiges Lesen und Schreiben führen zu Konflikten. Ergänzen Sie eine prozessübergreifende Synchronisation. Ein Konstrukt für die prozessübergreifende Synchronisation kennen Sie, den `Mutex`. Aber auch das `Semaphor` und der `EventWaitHandle` kann prozessübergreifend verwendet werden.

3. Socket Kommunikation

In diesem Teil wird die Kommunikation zwischen .NET Applikationen mit Hilfe von TCP/IP Sockets besprochen. Das Transmission Control Protocol (TCP) und das Internet Protocol (IP) sind nur zwei der Protokolle. Es sind die am häufigsten verwendeten Protokolle und so bekam die Familie ihren Namen. Welche Protokolle vom .NET Framework unterstützt werden, kann man in der .NET Reference nachlesen und zwar in der Dokumentation der Klasse `System.Net.Sockets.Socket`.

Da die C#-Netzwerk-Programmierung eigenständige und nicht zwingend grafikorientierte Anwendungen hervorbringt, bietet C# eine umfangreiche Bibliothek zur sequentiellen Ein- und Ausgabe an. Die dafür verwendeten Klassen realisieren das aus anderen Sprachen bekannte Konzept der *Streams.*

3.1. Ein- und Ausgabe-Streams

Streams dienen generell dazu, drei elementare Operationen ausführen zu können:

- Dateninformationen müssen in einen Stream geschrieben werden. Nach welchem Muster das geschieht, wird durch den Typ des Streams vorgegeben.
- Aus dem Datenstrom muss gelesen werden, ansonsten könnte man die Daten nicht weiterverarbeiten. Das Ziel kann unterschiedlich sein: Die Bytes können Variablen oder Arrays zugewiesen werden, sie könnten aber auch in einer Datenbank landen und zur Ausgabe an einem Peripheriegerät wie dem Drucker oder dem Monitor dienen.
- Nicht immer ist es erforderlich, den Datenstrom vom ersten bis zum letzten Byte auszuwerten. Manchmal reicht es aus, erst ab einer bestimmten Position zu lesen. Man spricht dann vom wahlfreien Zugriff.

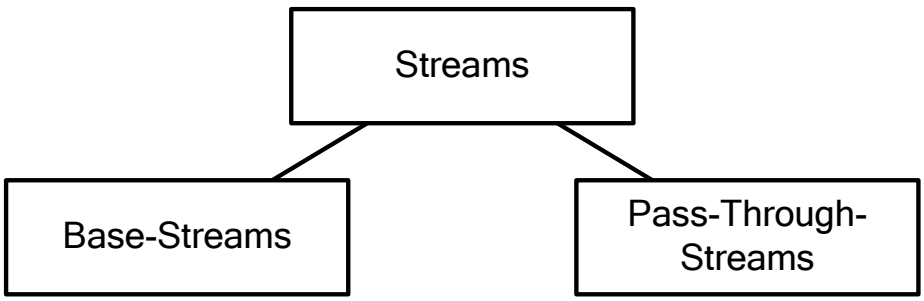

Abbildung 6: Arten von Streams

Base-Streams, die direkt aus einem Strom Daten lesen oder in diesen hineinschreiben. Diese Vorgänge können z.B. in Dateien, im Hauptspeicher oder in einer Netzwerkverbindung enden.

Pass-Through-Streams ergänzen einen Base-Stream um spezielle Funktionalitäten. So können manche Streams verschlüsselt oder im Hauptspeicher gepuffert werden. Pass-Through-Streams lassen sich hintereinander in Reihe schalten, um so die Fähigkeiten eines Base-Streams zu erweitern. Auf diese Weise lassen sich sogar individuelle Streams konstruieren.

Die Stream-Klassen implementieren sequentielle Ein-/Ausgabe auf verschiedenen Datenquellen und -senken. Die I/O-Art kann sein

- zeichen- bzw. textorientiert (`StreamReader,StreamWriter,StringReader,StringWriter`): mit Wandlung zwischen interner Binärdarstellung und externer Textdarstellung. Grundlage ist die byteorientierte Ein- und Ausgabe mit den Klassen `TextReader` und `TextWriter`.
- binär (`BinaryReader,BinaryWriter,` Unterklassen von `Stream`): ohne Wandlung der Binärdarstellung.

Stream Architektur

Die .NET-Stream-Architektur konzentriert sich auf drei Konzepte (ähnlich wie in Java): Adapter, Dekorator und Sicherungsspeicher* (Abbildung 7).

Adapter formen Daten aus Programmen um. Daten können Text, bzw. Strings, elementare Datentypen und XML sein. Die Implementierung der Stream Adapter wird mit Hilfe des Entwurfsmusters Adapter umgesetzt (https://de.wikipedia.org/wiki/Adapter_(Entwurfsmuster)).

Dekorator fügen neue Eigenschaften zu dem Stream hinzufügen. Dekoratoren können in Reihe geschaltet werden, so dass mehrere Eigenschaften in einem Stream vereint sind. Die Implementierung der Stream Dekoratoren wird mit Hilfe des Entwurfsmusters Dekorator umgesetzt (https://de.wikipedia.org/wiki/Decorator)).

Ein **Sicherungsspeicher*** ist ein Speichermedium, wie etwa ein Datenträger oder Arbeitsspeicher. Jeder einzelne Sicherungsspeicher implementiert seinen eigenen Datenstrom als Implementierung der Stream-Klasse. Jeder Datenstromtyp liest und schreibt Bytes in einen bzw. aus einem angegebenen Sicherungsspeicher.

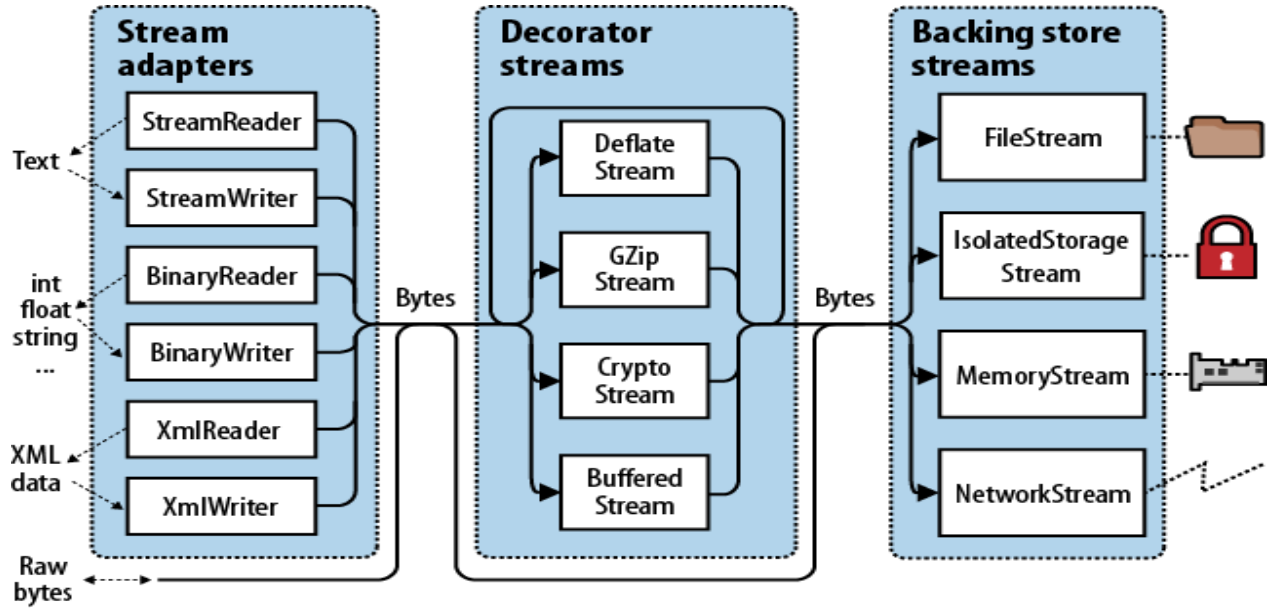

Abbildung 7: Stream Architektur

*Namensgebung, bzw. Übersetzung von Microsoft

Alle Klassen zur Datenein- und -ausgabe befinden sich im Namespace `System.IO`. Um sie zu verwenden, die entsprechende Import-Anweisung an den Anfang eines Programms gestellt werden:

```
using System.IO;
```

Auswahl von Input Streams

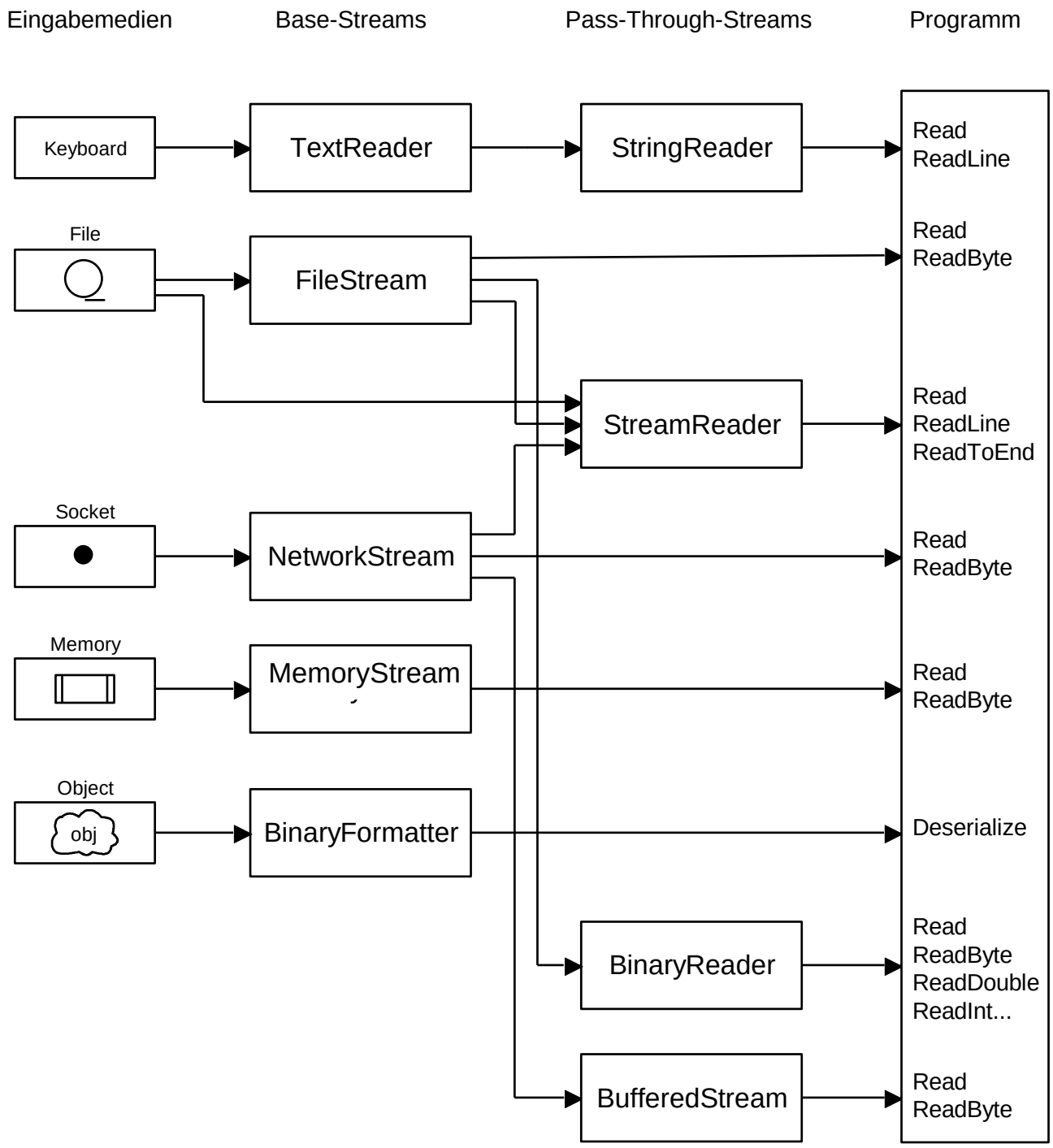

Abbildung 8: Stream Architektur der Input Streams

Auswahl von Output Streams

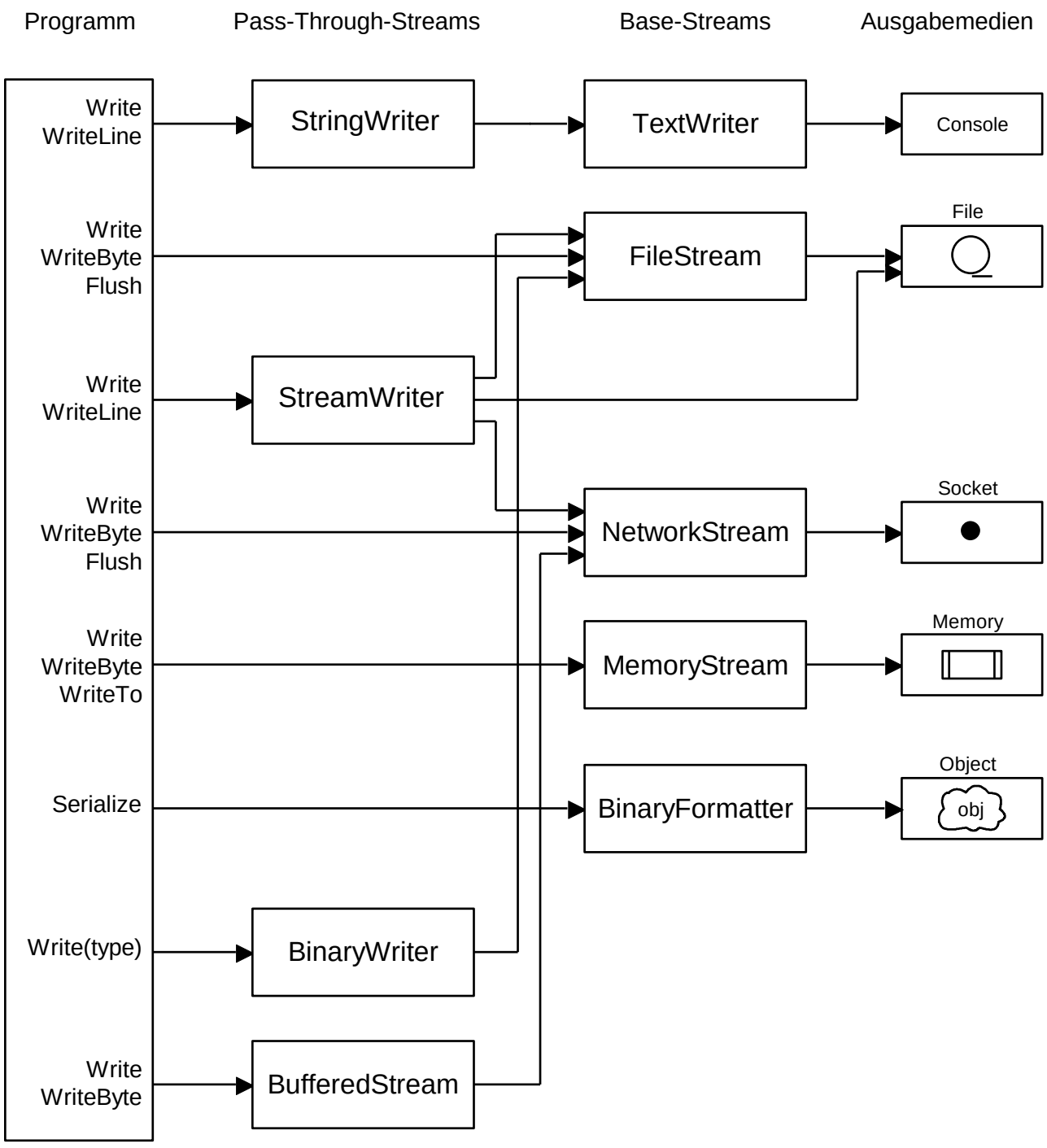

Abbildung 9: Stream Architektur der Output Streams

Stream Beispiele

Schreiben in eine Datei mit StreamWriter und implizitem FileStream:

```
class FileWrite {
    public static void DoIt() {
        try {
            using (StreamWriter sw = new StreamWriter("daten.txt")) {
                string[] text = { "Titel", "Köln", "4711" };
                for (int i = 0; i < text.Length; i++)
                    sw.WriteLine(text[i]);
            }
            Console.WriteLine("Datei geschrieben.");
        }
        catch (Exception e) {
            Console.WriteLine(e);
        }
    }
}
```

Codeskizze 32: Projektmappe «Streams» / Projekt FileReadWrite

Lesen aus einer Datei mit StreamReader und implizitem FileStream:

```
class FileRead {
    public static void DoIt() {
        try {
            using (StreamReader sr = new StreamReader("daten.txt")) {
                string line;
                while ((line = sr.ReadLine()) != null) {
                    Console.WriteLine(line);
                }
            }
            Console.WriteLine("Datei gelesen.");
        }
        catch (Exception e) {
            Console.WriteLine(e.ToString());
        }
    }
}
```

Codeskizze 33: Projektmappe «Streams» / Projekt FileReadWrite

Schreiben und Lesen in/aus einen/m Netzwerk-TCP-Socket:

```
class TcpReadWrite {
    public static void Main() {
        using (TcpClient client = new TcpClient("sbb.ch", 80)) {
            StreamWriter outStream = new StreamWriter(client.GetStream());
            StreamReader inStream = new StreamReader(client.GetStream());
            outStream.WriteLine("GET / HTTP1.1");
            outStream.WriteLine();
            outStream.Flush();
            String line;
            while ((line = inStream.ReadLine()) != null) {
                Console.WriteLine(line);
            }
        }
    }
}
```

Codeskizze 34: Projektmappe «Streams» / Projekt TcpReadWrite

3.2. Socket-Kommunikation

Die Socket-Kommunikation kann allgemein als Interprozesskommunikation bezeichnet werden. Zwei oder mehr Prozesse (z.B. Applikationen) kommunizieren miteinander. Die Prozesse können auf demselben oder verschiedenen Computern oder auch anderen Maschinen laufen. Damit zwei Prozesse sich verstehen, müssen sie dieselbe Sprache sprechen, also gleiche Kommunikationsprotokolle verwenden. Die heute am häufigsten verwendeten Protokolle auf Netzwerkebene stammen aus der Protokollfamilie TCP/IP.

Socket Prinzip

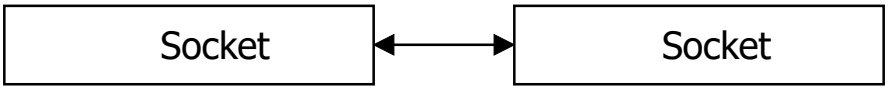

Abbildung 10: Ende-zu-Ende Kommunikation

Sockets stellen eine Verbindung zwischen zwei Prozessen dar (Abbildung 10). Sie entsprechen etwa den Filehandles, die für den Zugriff auf Dateien angelegt und verwaltet werden müssen. Der Benutzer braucht sich nicht um technische Details der Datenübertragung zu kümmern. Er kann mit anderen Benutzern kommunizieren, die Details des Protokolls sind für ihn nicht sichtbar. Er kann einfach lesen und schreiben.

Was kann ein Socket:

1. Verbindung zu einem entfernten Prozess aufbauen
2. Daten senden
3. Daten empfangen
4. Verbindung beenden
5. Einen Port (Applikation) binden
6. An einem Port auf Verbindungswunsch hören
7. Verbindungswunsch am Port akzeptieren

Der `System.Net.Sockets` Namespace stellt eine verwaltete Implementierung der Windows Sockets-Schnittstelle (Winsock) bereit. Es bietet eine umfangreiche Socket Funktionalität, wie TCP-Sockets, Datagram-Sockets und Multicast-Sockets. Für vernetzte Anwendungen (z.B. für eigene Protokolle oder allgemeiner vernetzte Anwendungen), bietet C# die Klasse `Socket` als Abstraktion von standardmässigen Socket-Programmiertechniken. Die Sockets werden direkt mit `NetworkStreams` verknüpft. Damit können dann alle auf Streams aufbauenden Klassen und Methoden genutzt werden.

Einsatz der Sockets

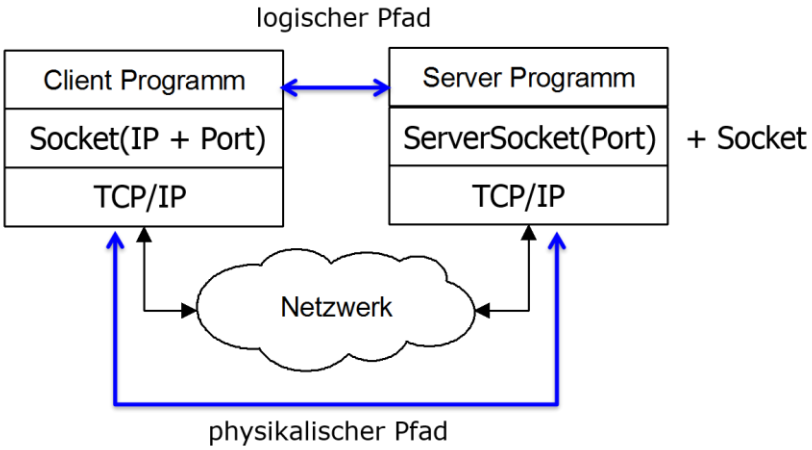

Abbildung 11: Logischer und physikalischer Datenfluss

Das folgende Beispiel benutzt den `TcpClient`-Konstruktor und überprüft, welche Ports beim übergebenen Host noch frei sind. Falls kein `TcpClient` Objekt erstellt werden kann, ist der angegebene Host nicht erreichbar oder unter dem entsprechenden Port kein Dienst aktiv.

```
public class PortScan {

    public static void Main(string[] args) {
        TcpClient theClient;
        string host = "localhost";
        Console.WriteLine("Port Scanner");
        for (int port = 1; port <= 65535; port++) {
            try {
                theClient = new TcpClient(host, port);
                Console.WriteLine("Port {0} des Hosts {1} kennt TCP.",
                    port, host);
            }
            catch (Exception e) {
                Console.WriteLine("Port {0} des Hosts {1}: {2}",
                    port, host, e.Message);
            }
        }
    }
}
```

Codeskizze 35: Projektmappe «Socket Grundlagen» / Projekt PortScan

Experiment:

Starten Sie das Programm `PortScan` und beantworten Sie folgende Fragen.

Fragen:

- Was stellen Sie fest beim Ausführen des Programms?
- Wie erklären Sie sich das Programmverhalten?
- Wie können Sie das Programmverhalten verbessern?

Socket Eigenschaften

Das folgende Beispiel zeigt einige Eigenschaften, die Informationen über die Socket-Verbindung liefern.

```
public class SocketInfo {

    public static void Main(string[] args) {
        Console.WriteLine("Socket Information");
        for (int i = 0; i < args.Length; i++) {
            try {
                TcpClient theClient = new TcpClient(args[i], 80);
                Socket theSocket = theClient.Client;
                Console.WriteLine("Der lokale Host ({0}) ist verbunden mit",
                    theSocket.LocalEndPoint);
                Console.WriteLine("dem entfernten Host {0} ({1})", args[i],
                    theSocket.RemoteEndPoint);
                Console.WriteLine("ueber einen {0} Socket.",
                    theSocket.ProtocolType);
                theClient.Close();
            }
            catch (Exception e) {
                Console.Error.WriteLine(e.Message);
            }
        }
    }
}
```

Codeskizze 36: Projektmappe «Socket Grundlagen» / Projekt SocketInfo

Experiment:

Starten Sie das Programm `SocketInfo` und beantworten Sie folgende Fragen.

Fragen:

- Was stellen Sie beim lokalen Host und entfernten Host fest, wenn Sie dieses Programm ausführen?
- Was folgern Sie aus der obigen Feststellung?

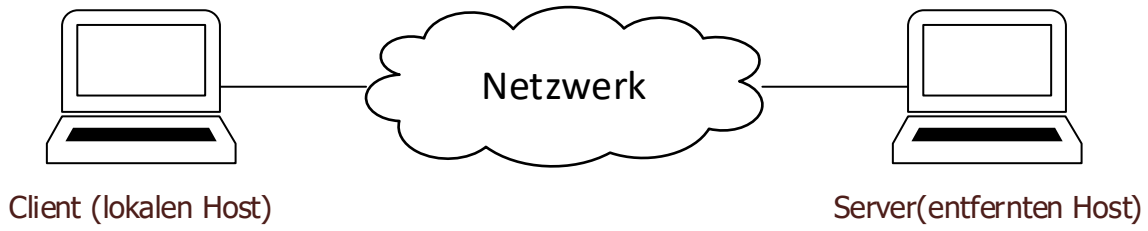

Abbildung 12: Host Verbindung

Server Socket

Ein `TcpListener` Objekt wird für einen bestimmten Port erzeugt und gestartet. Die Portnummer bestimmt die Art der Serveranwendung (https://www.iana.org/). Anschliessend wird die Methode `AcceptTcpClient` aufgerufen, um auf einen eingehenden Verbindungswunsch zu warten. Die Methode `AcceptTcpClient` ist blockierend, d.h. es wird solange gewartet bis sich ein Client bei der Serveranwendung anmeldet. Ist der Verbindungsaufbau erfolgreich, liefert diese Methode ein `Socket` Objekt, das zur Kommunikation mit der wie bei Client Anwendung verwendet wird. Anschliessend steht der `TcpListener` für einen weiteren Verbindungsaufbau zur Verfügung.

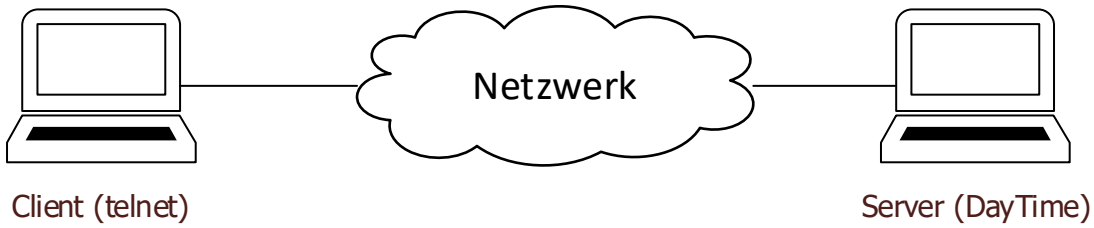

Abbildung 13: Client / Server Verbindung

Das folgende Server-Programm nimmt auf Port 13 einen Verbindungswunsch entgegen und sendet dem Client einen String mit Datum und Zeit.

```
public class SimpleDayTimeServer {

    public static void Main() {
        TcpListener listen = new TcpListener(13);
        listen.Start();
        while (true) {
            Console.WriteLine("Warte auf Verbindung auf Port {0}...",
                listen.LocalEndpoint);
            TcpClient client = listen.AcceptTcpClient();
            Console.WriteLine("Verbindung zu {0}",
                client.Client.RemoteEndPoint);
            TextWriter tw = new StreamWriter(client.GetStream());
            tw.Write(DateTime.Now.ToString());
            tw.Flush();
            client.Close();
        }
    }
}
```

Codeskizze 37: Projektmappe «Socket Grundlagen» / Projekt SimpleDayTimeServer

Experiment:

Starten Sie das Programm `SimpleDayTimeServer` auf dem PC. Beantworten Sie folgende Fragen.

Fragen:

- Wie müssen Sie das `telnet` Programm starten, um mit dem `SimpleDayTimeServer` Verbindung aufzubauen und die Zeitangabe zu erhalten? `telnet` ist ein Dienst den Sie unter Windows aktivieren müssen.
- Was ist eine grosse Einschränkung dieses `SimpleDayTimeServer` Servers?

Übung 7: Socket Implementationen

Der Raspberry Pi mit dem Explorer 700 dient als Day Time Server. Auf dem Laptop oder PC läuft das `telnet` Programm, womit wir die Zeit auf dem Raspberry Pi abfragen.

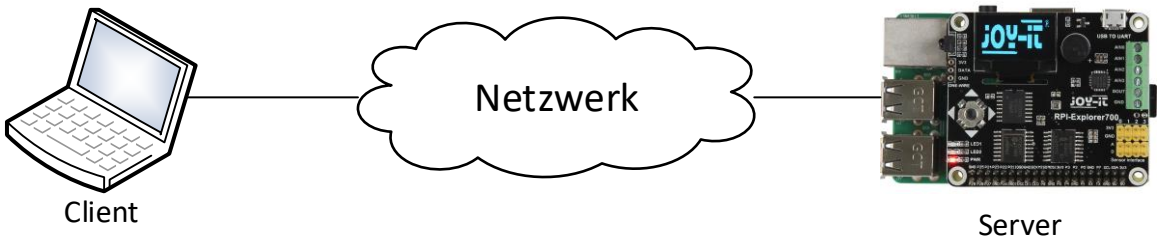

Abbildung 14: PC / Raspberry Pi Verbindung

Aufgabe A:

Beheben Sie die Einschränkung des `SimpleDayTimeServer`. Lassen Sie das Programm als `DayTimeServer` auf dem Raspberry Pi laufen und beantworten Sie folgende Fragen.

Fragen:

- Was stellen Sie beim Ausführen von `DayTimeServer` fest?
- Wie erklären Sie sich das Programmverhalten?
- Wie müsste eine korrekte Lösung aussehen?

Aufgabe B:

Erweitern Sie das Programm `DayTimeServer` auf dem Raspberry Pi mit dem Aufruf des Buzzers, dass er bei korrektem Verbindungsaufbau ertönt. Die `Explorer700Library` ist bereits im Übungsprojekt eingebunden. Verlinken Sie die Library mit folgender Angabe im Projekt XML.

```
<ItemGroup>
    <ProjectReference Include="..\Explorer700Library\Explorer700Library.csproj"/>
</ItemGroup>
```

Codeskizze 38: Referenz zu Explorer700 Library Projektmappe «Socket Uebungen» / Projekt SimpleDayTimeServer

Den Buzzer binden Sie wie folgt in Ihren Code ein. Das Argument der Methode `Beep` ist in Millisekunden.

```
using Explorer700Library;
//...
Explorer700 exp = new Explorer700();
exp.Buzzer.Beep(500);
```

Codeskizze 39: Buzzer Aufruf

Aufgabe C:

Erweitern Sie das Programm `DayTimeServer` mit der Möglichkeit, dass der Client (`telnet`) einen Integer Wert mitgeben kann, der die Beep Dauer des Buzzers steuert. Die Umwandlung von String zu Integer finden Sie in dieser Dokumentation:

`https://docs.microsoft.com/en-us/dotnet/api/system.int32.parse?view=net-8.0`

3.3. Client-Server

Das Client-Server-Modell beschreibt eine Möglichkeit, Aufgaben und Dienstleistungen innerhalb eines Netzwerkes zu verteilen. Dabei bietet ein Server (Dienstleister) auf bekannten Ports Dienste an. Die Clients (Dienstnehmer) stellen eine Verbindung zu diesen Ports her und nehmen entsprechend dem Protokoll den angebotenen Dienst in Anspruch.

Modell

Übertragung des Client/Server Modells auf Rechner in einem Netzwerk:

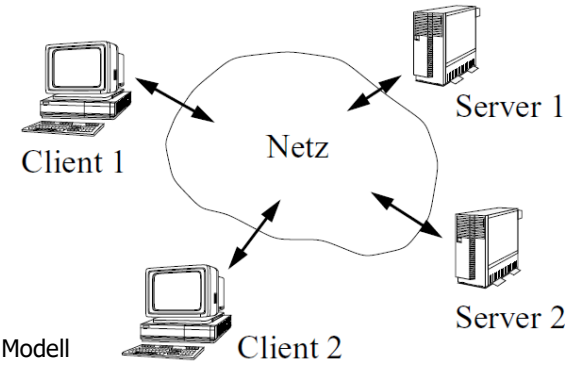

Abbildung 15: Client/Server Modell

Dieses Modell wird typischerweise in folgenden Fällen eingesetzt.

- Die Anzahl der Clients ist nicht oder nur schwer zu bestimmen. Zu unterschiedlichen Zeiten können unterschiedlich viele Clients auf einen Server zugreifen.
- Die Aufgaben sind eventuell nicht unabhängig voneinander, z.B. werden einige Details nur nachgefragt, wenn deren Information für den Client wichtig ist.
- Der Umfang der nachgefragten Leistungen variiert oder ist nicht genau bestimmbar, z.B. eine Web Seite oder ein ganzer Teil Baum von Web Seiten wird nachgefragt.
- Ein Client nimmt zum Teil die Leistungen mehrerer Server in Anspruch, bis die Aufgabe gelöst ist, z.B. Suchmaschine, Web Server, FTP Server.

Das Client/Server Modell eignet sich besonders gut, da

- Populär wegen des eingängigen Modells. Es entspricht den Geschäftsvorgängen in unserer Dienstleistungsgesellschaft. Es ist ein gewohntes Muster, besitzt eine intuitive Struktur und gute Überschaubarkeit.
- Hohe Effizienz durch spezialisierte «Dienstleister» mit grosszügiger Ausstattung (CPU-Leistung, Speicherkapazität usw.) und bestückt mit spezieller Software (Datenbank etc.)
- Kosteneffektivität durch bessere Auslastung wertvoller Ressourcen bei «Compute Server». Clients brauchen oft kurzfristig Spitzenleistung, kann aber die Ressourcen nicht dauerhaft auslasten.
- Passend für viele Kooperationsbeziehungen, z.B. Client erbittet Auskunft von einem spezialisierten Service oder gefährdete Clients geben wertvolle Daten in Obhut des hoch gesicherten Servers (gegen Missbrauch, Verlust, Diebstahl usw.)

Typischerweise werden PCs als Clients eingesetzt. Andere, leistungsfähigere Rechner werden als Server eingesetzt, um zentrale Dienste (z.B. File- oder Speicherserver) oder gemeinsam benutzte Betriebsmittel zur Verfügung zu stellen. Im Allgemeinen müssen sich aber Server und Client Prozesse nicht auf dedizierten Rechnern befinden, d.h. Rechnern die nur für eine bestimmte Aufgabe da sind («dedicated to service»).

Client-Server Konzept

Das Client/Server Konzept definiert die Bezeichnung der Anwendungskomponenten Client (deutsch: Kunde, Dienstnutzer; nutzt Dienste) oder Server (deutsch: Bediener, Anbieter, Dienstleister, englisch: to serve; bietet einen Dienst an) und den zeitlichen Ablauf einer Interaktion zwischen Client und Server (Abbildung 16).

Ein Client kann einen Dienst bei dem Server anfordern.

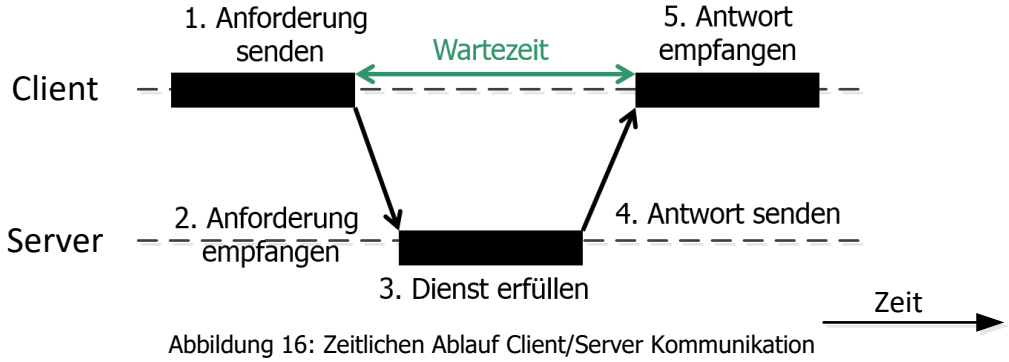

Abbildung 16: Zeitlichen Ablauf Client/Server Kommunikation

Ein Server ist ein Programm (Prozess), das mit einem anderen Programm (Prozess), dem Client kommuniziert, um ihm Zugang zu einem Dienst zu verschaffen. Hierbei muss abgrenzend beachtet werden, dass es sich beim «Server» um eine Rolle handelt, nicht um einen Computer an sich. Ein Computer kann nämlich ein Server und Client zugleich sein (Peer-to-Peer).

Server müssen ggf. zur Durchführung eines Dienstes die Dienstleistungen anderer Server in Anspruch nehmen (Abbildung 17), z.B. Zeitserver für Log-Funktionen oder Authentifizierungsserver. Authentifizierung und Zeit sind nichttriviale Dienste, die nicht nur vom Fileserver benötigt werden!

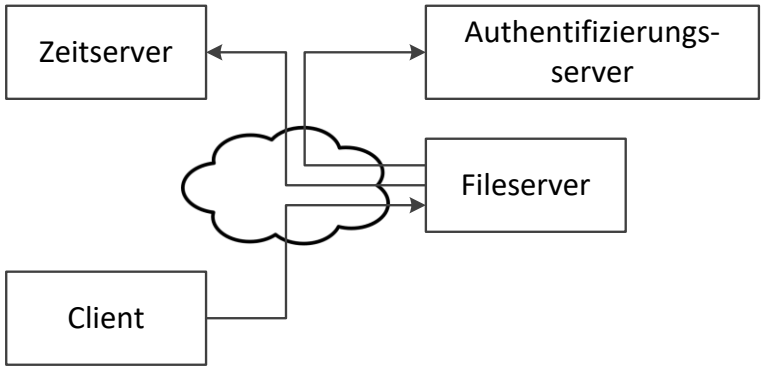

Abbildung 17: Durchführung eines Dienstes (Beispiel File Server)

Der Fileserver hat prinzipiell die Rolle eines Servers, zwischenzeitlich jedoch auch die Rolle eines Clients.

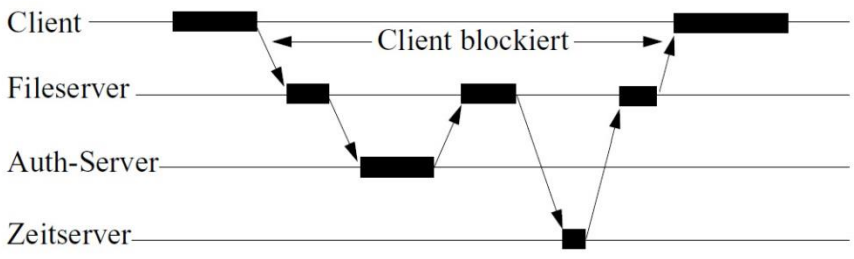

Abbildung 18: Zeitliche Abfolge bei Durchführung eines Dienstes (Beispiel File Server)

Iterativer Server

Beim Client/Server Modell können viele Clients «gleichzeitig» bei einem Server eine Anfrage für einen Dienst machen. Wobei «gleichzeitig» nicht unbedingt zum gleichen Zeitpunkt bedeutet. Der Server überprüft in bestimmten Zeitabständen jeweils, ob von den einzelnen Clients Anfragen angekommen sind, und bearbeitet die Aufträge gegebenenfalls.

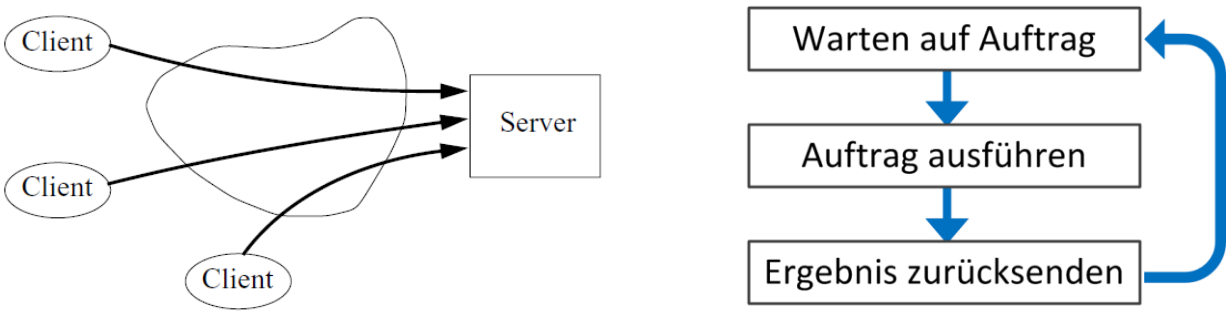

Abbildung 19: Zugriff und Zeitliche Abfolge bei iterativem Server

Iterative Server bearbeiten nur einen Auftrag pro Zeitpunkt. Sie werden oft auch als «single threaded» Server bezeichnet. Ein iterativer Server ist einfach zu realisieren und bei trivialen Diensten (Dienste mit kurzer Bearbeitungszeit) sinnvoll. Bei eintreffenden Anfragen während der Auftragsbearbeitung kann wie folgt vorgegangen werden: abweisen, puffern oder einfach ignorieren der Anfrage.

Das folgende Server-Programm nimmt am Port 7070 einen Auftrag entgegen und sendet dem Client drei Zeilen mit der aktuellen Zeit des Servers, der IP Adresse des Clients und der Servers.

```
public class IterativerServer {
    public static void Main() {
        TcpListener listen = new TcpListener(IPAddress.Any, 7070);
        listen.Start();
        while (true) {
            Console.WriteLine("Warte auf Verbindung auf Port {0}...",
                listen.LocalEndpoint);
            using (TcpClient client = listen.AcceptTcpClient()) {
                Console.WriteLine("Verbindung zu {0}",
                    client.Client.RemoteEndPoint);
                StreamWriter sw = new StreamWriter(client.GetStream());
                sw.WriteLine("Zur Zeit: {0}", DateTime.Now.ToString());
                sw.WriteLine("bist Du: {0}", client.Client.RemoteEndPoint);
                sw.WriteLine("bin ich: {0}", client.Client.LocalEndPoint);
                sw.Flush();
            }
        }
    }
}
```

Codeskizze 39: Projektmappe «ClientServer» / Projekt IterativerServer

Experiment:

Starten Sie das Programm `IterativerServer` auf dem PC und dem Raspberry Pi.

Nebenläufiger Server

In allen Fällen, in denen der Auftrag eines Clients lange dauern kann, ist ein nebenläufiger (multithreaded) Server einem iterativen Server vorzuziehen. Der nebenläufige Server bearbeitet die Aufträge zum gleichen Zeitpunkt. Diese Form ist ideal bei Mehrprozessormaschinen (physische Parallelität). Kann aber auch bei Monoprozessor-Systemen mit preemptivem Multitasking Betriebssystem eingesetzt werden.

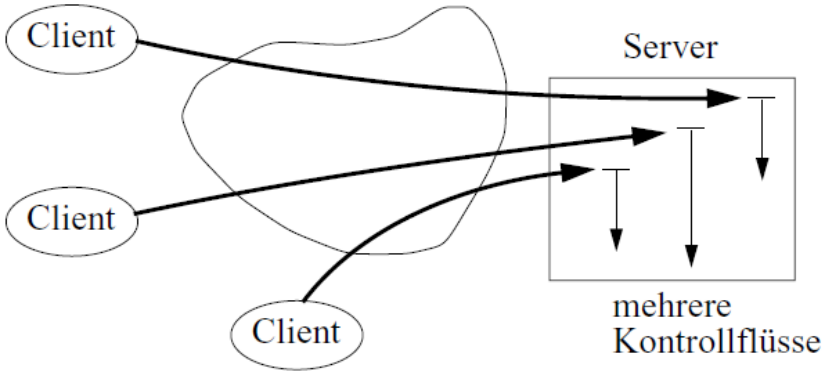

Abbildung 20: Zugriff bei nebenläufigem Server

Für jeden Auftrag gründet der Master, d.h. der Listener Teil des Servers, einen neuen Slave-Prozess und wartet dann auf einen neuen Auftrag.

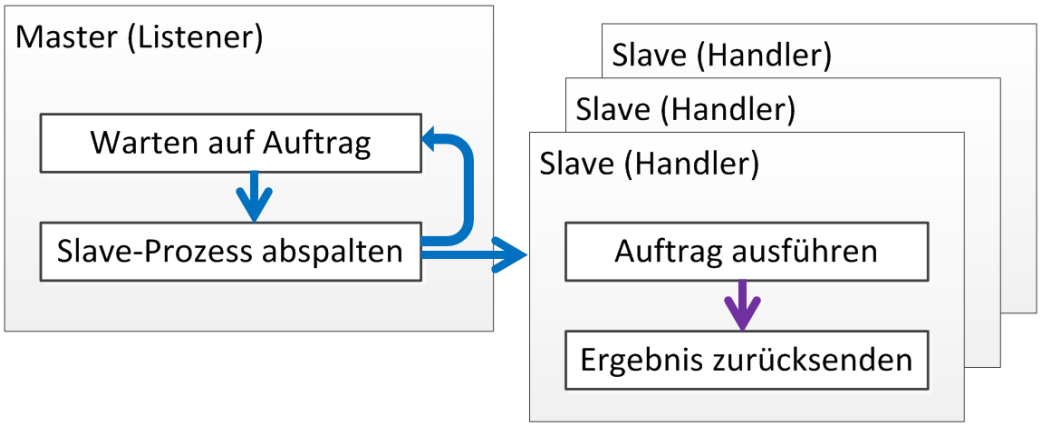

Abbildung 21: Zeitliche Abfolge bei nebenläufigem Server

Der neu gegründete Slave (Handler) übernimmt den Auftrag. Der Client kommuniziert dann ggf. direkt mit dem Handler (z.B. über dynamisch eingerichteten Kanal bzw. Port). Handler sind ggf. Leichtgewichtsprozesse (Threads) und terminieren nachdem sie das Ergebnis an den Client gesendet haben.

Die Anzahl gleichzeitiger Slaves sollte begrenzt und auf die zu Verfügung gestellten Systemressourcen abgestimmt werden. Dieses Verfahren nennt man «Process preallocation» und ist feste Anzahl statischer Slave Prozesse. Zur effizienten Nutzung der Systemressourcen kommt noch der Wegfall der Erzeugungskosten des Prozesses, bzw. Threads.

Nebenläufiger Server (Fortsetzung)

Das folgende Server-Programm nimmt auf Port 7777 einen Echo Auftrag entgegen, erzeugt einen Handler und übergibt diesem die Client Verbindung. Der Handler sendet dem Client jedes Zeichen wieder zurück, das er von diesem empfängt.

```
public class EchoServer {
    public static void Main() {
        IPAddress ipAddress = Dns.GetHostEntry("localhost").AddressList[0];
        TcpListener listen = new TcpListener(ipAddress, 7777);
        listen.Start();
        while (true) {
            Console.WriteLine("Warte auf Verbindung auf Port {0}...",
                listen.LocalEndpoint);
            TcpClient client = listen.AcceptTcpClient();
            Console.WriteLine("Echo Handler für {0} erzeugen und starten",
                client.Client.RemoteEndPoint);
            EchoHandler handler = new EchoHandler(client);
            new Thread(handler.DoEcho).Start();
        }
    }
}
```

Codeskizze 40: Projektmappe «ClientServer» / Projekt NebenlaeufigerServer

```
public class EchoHandler {

    private TcpClient client;

    public EchoHandler(TcpClient client) {
        this.client = client;
    }
    public void DoEcho() {
        StreamReader sr = new StreamReader(client.GetStream());
        StreamWriter sw = new StreamWriter(client.GetStream());
        try {
            while (client.Connected) {
                Char ch = (char)sr.Read();
                sw.Write(ch);
                sw.Flush();
            }
        }
        catch (Exception ex) {
            Console.WriteLine(ex.Message);
        }
        finally {
            client.Close();
        }
    }
}
```

Codeskizze 41: Projektmappe «ClientServer» / Projekt NebenlaeufigerServer

Experiment:

Starten Sie den `EchoServer` auf dem PC und dem Raspberry Pi.

Übung 8: Einfacher HTTP Fileserver

Als Vorstufe zum Testat Projekt sollen Sie einen sehr einfachen HTTP Fileserver entwickeln. Fragt ein Client den Server an, so sendet der Server eine zum Voraus bestimmte Datei mit dem Hypertext Transfer Protocol (siehe Anhang HTTP).

Aufgabe:

Implementieren Sie den einfachen HTTP Fileserver als nebenläufiger Server. Nehmen Sie als Datei eine einfache ASCII Textdatei.

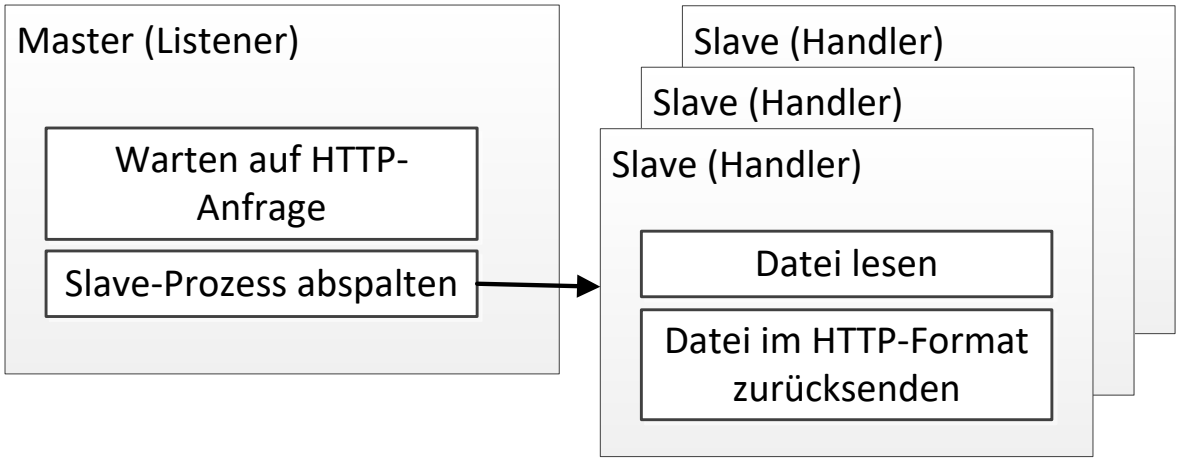

Abbildung 21: Zeitliche Abfolge beim HTTP Fileserver

Experiment:

Lassen Sie den `HttpServer` auf dem Raspberry Pi laufen und rufen Sie die einfache ASCII Textdatei mit verschiedenen Web Browsern (https://de.wikipedia.org/wiki/Webbrowser) auf. Beantworten Sie untenstehenden Fragen.

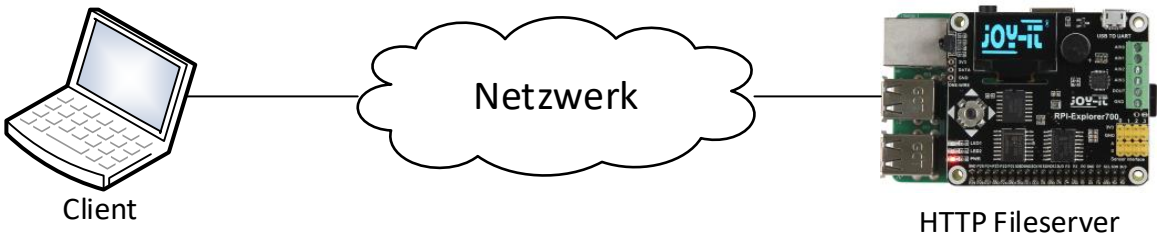

Abbildung 22: PC / Raspberry Pi Verbindung

Fragen:

- Welche HTTP Kommandos werden von Ihrem HTTP File Server akzeptiert?
- Welchen Port müssen Sie zum Start des HTTP Fileservers angeben?
- Wie muss ein Web Browser den HTTP Fileserver adressieren, damit er das vordefinierte File erhält?
- Wie können Sie den HTTP Fileserver mit dem Telnet testen?

Übung 9: Client/Server mit Raspberry PI

Es wird eine ausführliche Aufgabenstellung separat abgegeben.

Kern der Aufgabe wird die Kommunikation über TCP-Sockets zwischen einem Client Programm und dem Raspberry PI sein.

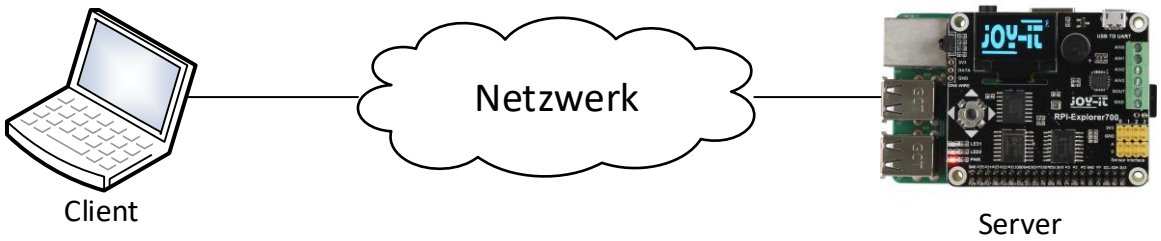

Abbildung 23: PC / Raspberry Pi Verbindung