

ZUSAMMENFASSUNG

Lernziele

Programmierung von parallelen und nebenläufigen Systemen

- Grundlagen der nebenläufigen und parallelen Programmierung (Threads, Synchronisation, Korrektheits- und Fairnessbedingungen, Thread Pools, asynchrone Programmierung, Speichermodelle) kennen und verstehen.
- Entwurf und Implementation von nebenläufigen und parallelen Programmen in modernen verbreiteten Technologien (z.B. .NET C#, Java) umsetzen können.
- Datenstrukturen, Algorithmen und Design Patterns zur effizienten Parallelisierung (Lock-Free/Wait-Free Data Structures, Recursive Parallel, Producer/Consumer u.a.) kennen und anwenden können.
- Weitergehende Concurrency-Modelle und Technologien (Actors/CSP, Software Transactional Memory, Cluster-Parallelisierung mit MPI, GPU-Parallelisierung) kennen und einsetzen können.

Lerninhalte

Multi-Threading und Synchronisation

- Einführung in die nebenläufige/parallele Programmierung und zugrundeliegenden Systemarchitekturen
- Multi-Threading mit .NET und/oder Java
- Kritische Abschnitte und Synchronisationsmechanismen
- Monitor-Konzept und deren Umsetzung in gängigen Sprachen
- Spezifische Synchronisationsprimitiven (Semaphore, Reader-Writer Locks, Latches, Barrieren u.a.)
- Korrektheits- und Fairnessbedingungen; Problematik von Race Conditions, Deadlocks und Starvations

Thread Pools und effiziente Parallelisierung

- Thread Pools: Mechanismus, Eignung und Limitationen
- Task- und Daten-Parallelität
- Parallele Algorithmen (Sortierung, Suchen etc.)
- Asynchrone Programmierung
- GUI und Nebenläufigkeit
- Entwurfsmuster der Nebenläufigkeit (Producer/Consumer, Concurrent Pipelines, Reader/Writer, Recursive Parallel)
- Speichermodelle: Atomanität, Sichtbarkeit und Optimierung
- Lock-Free & Wait-Free Datenstrukturen

Fortgeschrittene Nebenläufigkeitsmodelle

- Verteilte Parallelisierung mit Actors/CSP
- Cluster Computing mit MPI
- Software Transactional Memory
- Vektorparallelisierung; GPU / Coprozessor-Parallelisierung

EINFÜHRUNG	7
MULTI-THREADING GRUNDLAGEN	8
PARALLELITÄT DES BETRIEBSSYSTEMS	8
<i>Speicherressourcen</i>	8
<i>Thread-Implementationen</i>	8
<i>Thread Scheduling</i>	8
<i>Prozessor Multiplexing</i>	9
<i>Kontextwechsel</i>	9
<i>Multi-Tasking</i>	9
<i>Thread Zustände</i>	9
GRUNDLAGEN DER JAVA THREADS	9
<i>Ausführungsmodell</i>	9
<i>Implementierung</i>	10
<i>Join</i>	11
<i>Passivierung</i>	12
<i>InterruptedException</i>	12
<i>Weitere Thread Methoden</i>	12
MONITOR SYNCHRONISATION	13
THREAD SYNCHRONISATION.....	13
<i>Gemeinsame Ressourcen</i>	13
<i>Kritische Abschnitte</i>	14
<i>Gegenseitiger Ausschluss</i>	14
<i>Java Monitor Konzept</i>	16
<i>Wait & Signal</i>	17
<i>Fallstricke beim Java Monitor</i>	18
SPEZIFISCHE SYNCHRONISATIONSPRIMITIVEN	20
LETZTE VORLESGUNGSWOCHE.....	20
<i>Quiz</i>	20
<i>Wann reicht ein Single Notify?</i>	20
<i>Monitor Diskussion</i>	20
SEMAPHOR	20
<i>Nachbau mit einem Monitor</i>	21
<i>Arten von Semaphoren</i>	21
<i>Faire Semaphore</i>	21
<i>Anwendung</i>	21
<i>In anderen Systemen</i>	22
<i>Diskussion</i>	22
LOCK & CONDITION	22
<i>Primitten</i>	22
<i>Buffer mit Lock & Conditions</i>	22
READ-WRITE LOCK	23
<i>Verwendung</i>	23
<i>Beispiel</i>	23
<i>Read-Write Lock mit Conditions</i>	23
ZWISCHENSTAND	23
<i>Beispiel «Autorennen»</i>	24

<i>Synchronisationspunkt</i>	24
COUNT DOWN LATCH	24
<i>Beispiel</i>	24
<i>Details</i>	24
CYCLIC BARRIER	25
<i>Latch versus Barriere</i>	25
PHASER	26
RENDEZ-VOUS	26
EXCHANGER	26
<i>Beispiel</i>	26
GEFAHREN DER NEBENLÄUFIGKEIT	27
UNTERSCHIEDE	27
<i>Java Monitor</i> ↔ <i>Lock & Conditions</i>	27
<i>Semaphore</i> ↔ <i>CountDownLatch</i>	27
<i>CountDownLatch</i> ↔ <i>CyclicBarrier</i>	27
FEHLER DER NEBENLÄUFIGKEIT	27
<i>Race Conditions</i>	27
<i>Deadlocks</i>	29
<i>Starvation</i>	31
PARALLELITÄT KORREKTHEITSKRITERIEN	32
THREAD POOLS	33
THREAD-POOL	33
<i>Konzept und Funktionsweise</i>	33
<i>Vorteile</i>	33
<i>Einschränkungen</i>	33
<i>Fork & Join Pool</i>	34
ASYNCHRONE PROGRAMMIERUNG	37
<i>Unnötige Synchronität</i>	37
<i>Asynchroner Aufruf</i>	37
<i>Moderne Asynchronität (Java 8)</i>	38
<i>Ende des asynchronen Aufrufs</i>	38
<i>Continuation</i>	38
TASK CONTINUATIONS	38
<i>Ausführung der Continuation</i>	38
<i>Continuation-Style Programming</i>	38
<i>Multi Continuation</i>	38
TASK PARALLEL LIBRARY	39
THREADING IN .NET	39
<i>.NET Threads</i>	39
<i>C# Lambdas</i>	39
<i>Monitor in .NET</i>	39
<i>:NET Synchronisationsprimitiven</i>	39
TASK PARALLEL LIBRARY	40
<i>Task Parallelität (Task Parallelization)</i>	40
<i>Datenparallelität (Data Parallelization)</i>	40
ASYNCHRONE PROGRAMMIERUNG MIT TPL	43
<i>Task Continuations</i>	43
HSR PARALLEL CHECKER	43

JAVA AWT/SWING/SWT.....	44
<i>Swing/AWT und Multi-Threading</i>	44
<i>Swing – Dispatching an UI Thread</i>	44
<i>Swing Background Worker</i>	45
<i>Andere Java GUI Frameworks</i>	46
.NET WPF/WINFORMS	46

C# ASYNC/AWAIT **46**

RÜCKGABETYPEN	47
SPEZIELLE REGELN	47
AUSFÜHRUNGSMODELL	47
<i>Mechanismus & Methodenaufruf</i>	47
VERSCHIEDENE AUSFÜHRUNGEN	48
<i>Fall 1 – Kein UI Thread</i>	48
<i>Fall 2 – UI Thread</i>	48
<i>async/await Sequenz</i>	48
HÄUFIGE FEHLER.....	49
<i>Sonderheit</i>	49
DESIGN-KRITIK.....	49
EMPFEHLUNGEN	49

MEMORY MODELS **50**

QUIZ AUS DER LETZTEN VORLESUNG	50
EINSTIEGSBEISPIEL.....	50
<i>Ursachen für Probleme</i>	50
JAVA MEMORY MODEL	50
<i>Java Atomicity Garantien</i>	51
<i>Java Visibility Garantien</i>	52
<i>Java Ordering Garantien</i>	53
<i>Java Volatile Keyword</i>	53
<i>Atomare Operationen</i>	53
<i>Atomic Klassen</i>	54
<i>Optimistische Synchronisation</i>	54
<i>ABA Problem</i>	54
<i>Java 8 Update mit Lambda</i>	54
LOCK-FREIE DATENSTRUKTUREN.....	55
.NET MEMORY MODEL	56
<i>.NET Volatile Read / Write Fences</i>	56
<i>.NET Volatile Beispiel</i>	56

ACTOR MODEL **57**

QUIZ AUS VORHERGEHENDEM KAPITEL	57
MOTIVATION.....	57
ACTOR MODELL UND CSP	57
<i>Historisches</i>	57
<i>CSP – Communicating Sequential Processes (CSP)</i>	58
<i>Vorteile – Actor und CSP</i>	58
<i>Actor Implementierungen</i>	59
AKKA.....	59

Konzept.....	59
Empfangsverhalten.....	59
Beispiele für Actor Anwendungen	60
Patterns	60
Verteilbarkeit	61
Actor Hierarchies.....	61
Senden.....	62
Laufzeitsystem.....	62
Supervision	63
System Shutdown.....	63
Schwächen	63
GPU PARALLELISIERUNG 1	64
QUIZ – LETZTES KAPITEL.....	64
GPU CO-PROZESSOR ARCHITEKTUR.....	64
CPU versus GPU.....	64
GPU Aufbau	64
SIMD	64
GPU Parallelisierungspotential.....	64
GPUs vs. CPUs.....	64
Vergleich GPUs	65
NUMA Modell.....	65
CUDA PROGRAMMIERMODELL	65
CUDA Kernel.....	65
CUDA Threads.....	65
CUDA Blocks.....	66
CUDA Ausführungsmodell.....	66
CUDA Thread Pool Abstraktion	66
Datenaufteilung.....	66
CUDA Ausführung	67
Device Query.....	68
Wahl der Launch Configuration.....	68
GPU PARALLELISIERUNG 2	69
QUIZ KONFIGURATION	69
SPEICHERMODELL	69
3D Thread Hierarchie	69
Beispiel C Limitation	69
Matrix Multiplikation	70
CUDA Speicherstufen	71
Schnelle Matrix Multiplikation.....	71
SYNCHRONISATION	72
CUDA Barriere	72
ERWEITERUNG DER MATRIX-MULTIPLIKATION MIT SHARED MEMORY	72
WARPS	73
DIVERGENZ	73
COALESCING	74
PERFORMANCE EMPFEHLUNGEN	74
CLUSTER PARALLELISIERUNG	75
QUIZ VOM LETZTEN KAPITEL.....	75

HPC CLUSTER ARCHITEKTUR	75
<i>Stufen der Parallelisierung</i>	75
<i>Motivation</i>	75
<i>Computer Cluster</i>	75
JOBs UND TASKS, INPUT / OUTPUT	76
<i>Job Manager</i>	76
<i>Einfacher HPC Job starten</i>	76
MPI (MESSAGE PASSING INTERFACE)	76
<i>Erstes Programm</i>	76
<i>MPI Programmausführung</i>	77
<i>Single Program Multiple Data</i>	77
<i>Monte Carlo Pi Simulation</i>	79
REACTIVE PROGRAMMING	80
QUIZ – VORHERIGES KAPITEL	80
PROGRAMM-DATENFLÜSSE	80
<i>Mainstream Datenfluss-Modelle</i>	80
<i>LINQ = Pull-Mechanismus</i>	81
REACTIVE PROGRAMMING (PUSH-MECHANISMUS)	81
<i>Rx.NET (Reactive Extensions)</i>	81
<i>Buffer-Varianten</i>	82
<i>Rx und Concurrency</i>	82
<i>RX mit LINQ kombinieren</i>	83
<i>Vorgefertigte Observables</i>	83
FAZIT	84
<i>Vorteile</i>	84
<i>Nachteile</i>	84
SOFTWARE TRANSACTIONAL MEMORY	85
MOTIVATION	85
<i>Historisches</i>	85
<i>Software Transactions</i>	85
<i>ACI Transaktionen</i>	85
KONZEPT	85
<i>STM versus Locking</i>	85
<i>Nested Transactions</i>	86
<i>Keine Races, keine Deadlocks</i>	86
<i>Transaktionsaufführung</i>	86
<i>Typische Probleme</i>	86
<i>Warten auf Bedingungen</i>	86
<i>Nested Transactions</i>	86
<i>Baldiger Abbruch</i>	87
PRAKТИSCHE STM	87
<i>Hardware Support</i>	87
<i>STM auf Java</i>	87
<i>Zugriffe in Transaktionen</i>	88
<i>Transaktionstheorie</i>	88
DISKUSSION	88

Einführung

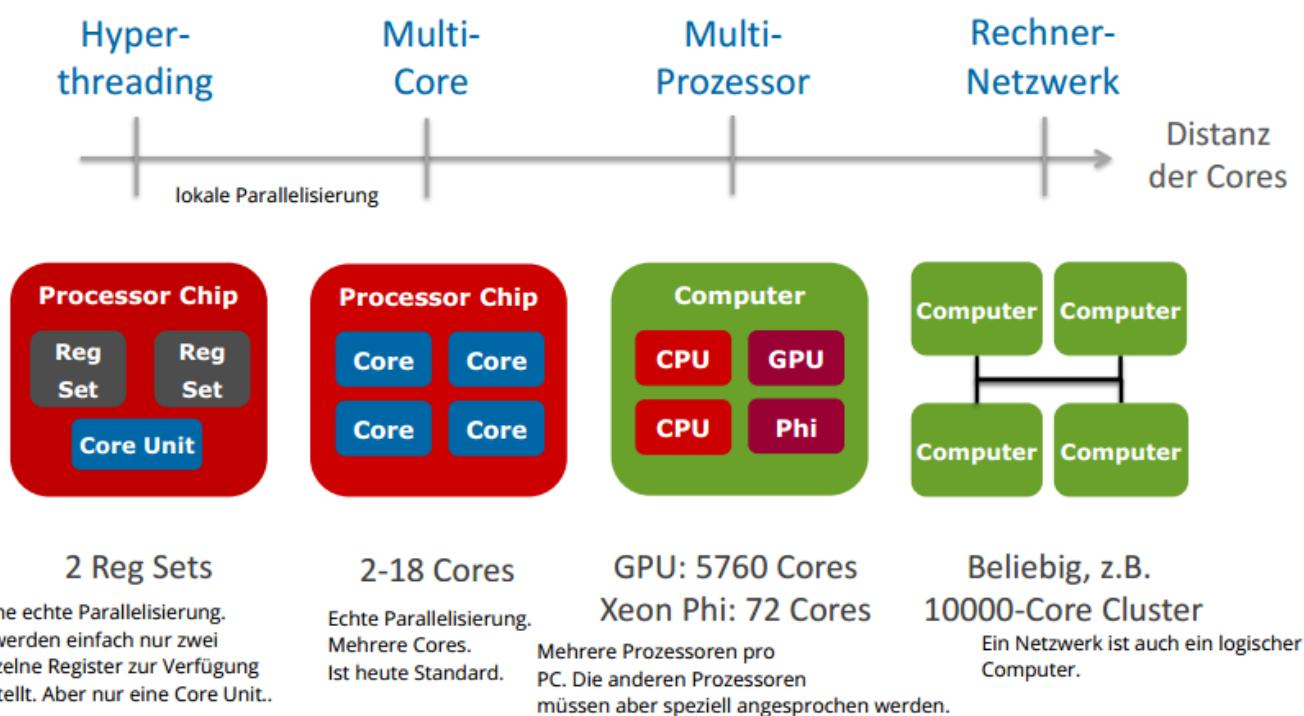
Wieso eigentlich Parallel Programmierung?

Es führt zu einer Performance-Steigerung, da wir mehrere Prozessoren sowie die Netzwerkverteilung nutzen können. Zudem entspricht es der natürlichen Modellierung. Es gibt vieles das in einer asynchronen Ausführung stattfinden (wie Druckjob, Downloads, etc.). Wie auch ein Server mit mehreren Klienten-Sitzungen.

«*The Free Lunch is Over*»

Bis 2003 hat circa alle 2 Jahre eine Verdoppelung der Prozessor-Taktraten stattgefunden. Damit wurden die Programme schneller ohne Code-Änderung. Seit 2003 stagnieren die Taktraten, dafür Hyperthreading und mehr Prozessorkerne. Wir müssen also die Programme so schreiben, dass Sie parallelisiert werden können.

Stufen der Parallelisierung



Parallelität (Parallelism)

Ist die Zerlegung eines Ablaufs in mehrere Teilabläufe, welche gleichzeitig auf mehreren Prozessoren laufen. Das Ziel sind schnellere Programme.

Nebenläufigkeit (Concurrency)

Gleichzeitig oder verzahnt ausführbare Abläufe, welche auf gemeinsame Ressourcen zugreifen. Das Ziel ist es hier einfachere Programme zu haben.

Dasselbe Prinzip gilt für mehrere Threads/Prozesse, die interagieren.

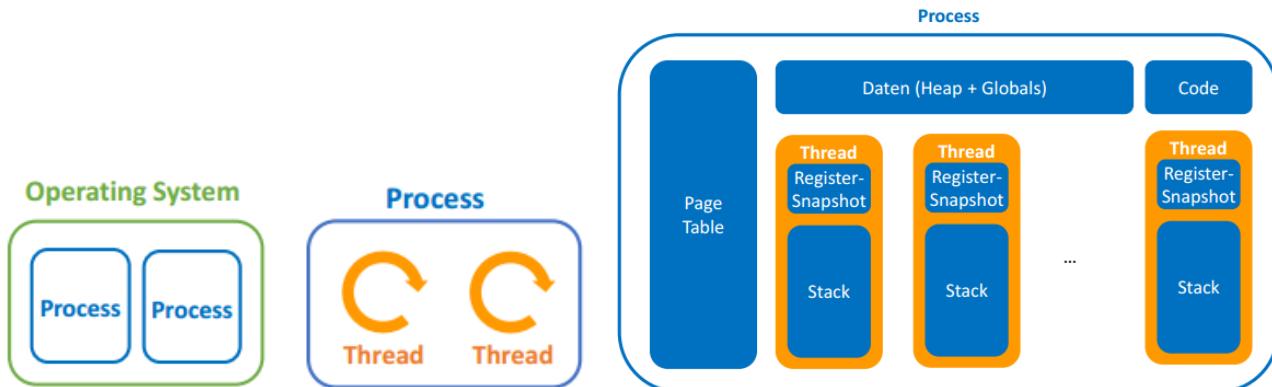
Multi-Threading Grundlagen

Parallelität des Betriebssystems

Es gibt dabei zwei verschiedene Arten von Parallelitätsabläufen.

Ein **Prozess** (Schwergewichtsprozess) ist eine parallel laufende Programm-Instanz im System. Pro Prozess ist ein eigener Adressraum vorhanden.

Ein **Thread** (Leichtgewichtsprozess) ist eine parallele Ablaufsequenz innerhalb eines Programms. Sie teilen sich den gleichen Prozess innerhalb des Prozesses.



Speicherressourcen

Prozesse sowie jeder Thread brauchen auch Speicherressourcen. So muss zum Beispiel jeder Thread einen Stack abbilden.

Thread-Implementationen

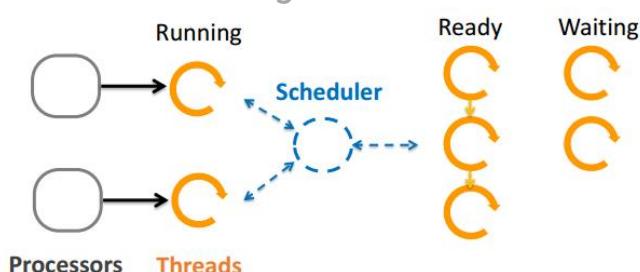
Es wird bei der Implementation zwischen zwei Arten von Threads unterschieden.

Der User-Level Thread ist im Prozess implementiert (Programm-Laufzeitsystem). Keine echte Parallelität durch mehrere Prozessoren.

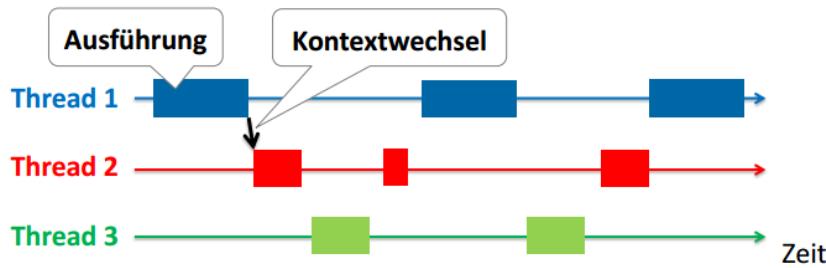
Der Kernel-Level Thread ist im Kernel implementiert (Multi-Core Ausnutzung). Der Kontextwechsel vom Prozess findet per SW-Interrupt (Kernel-Mode) statt.

Die Java-Threads der JVM werden eins zu eins auf Kernel-Threads abgebildet. Heutzutage wird in der Regel Kernel-Level Threading verwendet.

Thread Scheduling



Es kommt zum Prozessor Sharing, da wir im Normalfall mehr Thread als Prozessoren ausführen. Bei einer Wartebedingung wird der Prozessor also an anderen bereiten Thread freigegeben.



Gleichzeitig findet ebenfalls eine verzahnte Ausführung statt. Der Prozessor führt Instruktionen von mehreren Threads in abwechslungsweisen Teilsequenzen aus. Somit entsteht eine Illusion der Parallelität mehrerer Threads auch bei nur einem Prozessor (Quasiparallelität).

Kontextwechsel

Synchron (Warten auf Bedingung)

Thread wartet auf Bedingung. Reiht sich als wartend ein und gibt dann den Prozessor frei.

Asynchron (Zeitablauf)

Nach einer gewissen Zeit soll der Thread den Prozessor freigeben. Somit wird verhindert, dass ein Thread dauerhaft den Prozessor belegt.

Multi-Tasking

Kooperativ

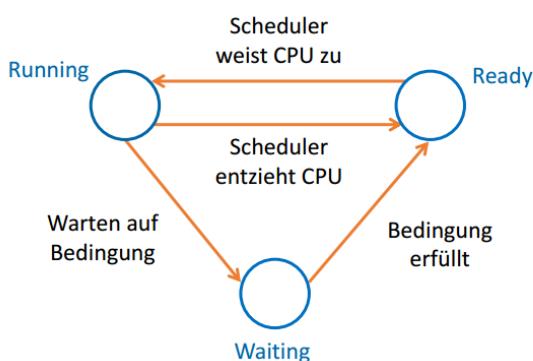
Threads müssen explizit beim Scheduler in Abständen Kontextwechsel synchron initiieren. Der Scheduler kann den laufenden Thread nicht unterbrechen.

Preemptiv

Scheduler kann per Timer-Interrupt den laufenden Thread asynchron unterbrechen. Es findet Time-Sliced Scheduling statt. Jeder Thread besitzt den Prozessor für einen maximalen Zeitintervall.

Heutzutage in der Regel nur *preemptiv*.

Thread Zustände



Grundlagen der Java Threads

Ausführungsmodell

JVM Thread Modell

Java ist ein **Single Process System**. Die Java Virtual Machine (JVM) ist ein Prozess im Betriebssystem. Die JVM erzeugt beim Aufstarten den Main-Thread, welcher die statische Methode main() aufruft. Der Programmierer kann weitere Threads starten. Subsysteme /Laufzeitsysteme starten auch eigene Threads (z.B. Garbage Collector).

JVM Terminierung

Die JVM läuft, solange Thread laufen. Ausnahme sind dabei die Threads, welche als Daemon markiert werden. Die JVM warten nämlich nicht auf Daemon Thread (z.B. Garbage Collector). Daemon Threads brechen beim JVM Ende unkontrolliert ab. Eine Terminierung findet auch bei System.exit()/Runtime.exit() statt. Dies führt zu einer direkten Terminierung der JVM, was unsauber ist. Alle Threads werden unkontrolliert abgebrochen.

Implementierung

Erzeugen eines Threads

```
class SimpleThread extends Thread {  
    @Override  
    public void run() {  
        // thread behavior  
    }  
}
```

Thread implementieren

```
Thread myThread = new SimpleThread();  
myThread.start();
```

Thread instanzieren

Thread starten

Start und Ende eines Threads

Ein richtiger Thread wird erst bei start() erzeugt. Start() führt die run()-Methode des Objektes aus.

Ein Thread endet beim Verlassen von run() und somit am Ende der Methode. Ein Return Statement kann mitgegeben werden. Zudem können unbehandelte Exceptions auftreten. Bei einer unbehandelten Exception laufen alle anderen Threads aber weiter.

Multi-Thread Beispiel

```
class SimpleThread extends Thread {  
    private String label;  
    private int nofIt;  
  
    public SimpleThread(String label, int nofIt) {  
        this.label = label; this.nofIt = nofIt;  
    }  
  
    public class MultiThreadTest {  
        public static void main(String[] args) {  
            new SimpleThread("A", 10).start();  
            new SimpleThread("B", 10).start();  
            System.out.println("main finished");  
        }  
    }  
}
```

```
@Override public void run() {  
    for (int i = 0; i < nofIt; i++) {  
        System.out.println(i + " " + label);  
    }  
}
```



Pro Thread (run-Methode) wird eine Liste von 0A,1A und so weiter ausgegeben. Wenn mit run() aufgerufen, dann passiert nichts nebenläufiges. Es werden dann keine Threads erstellt. Somit werden zuerst alle A's, dann alle B's und am Schluss die Ausgabe ausgegeben.

Mit start() sind A und B für sich geordnet. Es könnte aber auch sein dass alle A und dann alle B's kommen. Die Ausgabe von Main kann auch irgendwann auftauchen. Die Reihenfolge kann sich daher von Ausführung zu Ausführung ändern → Nicht-Determinismus.

Nicht-Determinismus

Threads laufen ohne Vorkehrung beliebig verzahnt oder parallel. Viele JVMs realisieren einzelne System Outputs ohne Verzahnung/Thread-Fehler – aber es ist nicht spezifiziert.

Runnable Implementierung

Es eine Alternativ zur Vererbung von Thread. Dies wenn andere Basisklasse benötigt wird (Single-Inheritance). Es findet eine Schnittstellenimplementierung des Interfaces statt.

```
class SimpleLogic implements Runnable {
    @Override
    public void run() {
        // thread behavior
    }
}
```

Das Thread-Objekt muss dann irgendwann erstellt werden.
Thread myThread = new Thread(new SimpleLogic());

Wieder genau gleich. Es darf aber nur start und nicht run aufgerufen werden.
myThread.start();

Beispiel

Das Runnable Interface könnte man auch mit Lambdas benutzen.

```
class SimpleLogic implements Runnable {
    private String label;
    private int nofIt;

    public SimpleLogic(String label, int nofIt) {
        this.label = label;
        this.nofIt = nofIt;
    }

    @Override public void run() {
        for (int i = 0; i < nofIt; i++) {
            System.out.println(i + " " + label);
        }
    }
}

public class MultiThreadTest {
    public static void main(String[] args) {
        new Thread(new SimpleLogic("A", 10)).start();
        new Thread(new SimpleLogic("B", 10)).start();
        System.out.println("main finished");
    }
}
```

Kürzer mit Java Lambda

Es ist eine Ad-Hoc Implementierung einer Funktionsschnittstelle (1 abstrakte Methode). In diesem Fall ein Runnable Interface mit run(). Die Syntax ist (Parameterliste) => {Methodenrumpf}.

```
Thread myThread = new Thread(() -> {
    // thread behavior Direkt die Implementierung der Run-Methode.
});

myThread.start();
```

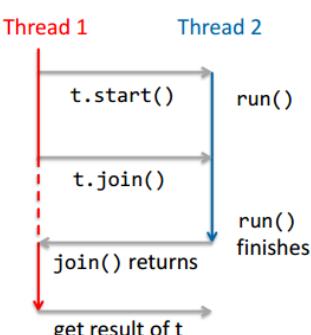
Beispiel

```
Was soll im Larma gemacht werden.
new Thread(() -> simpleLogic("A", 10)).start();
new Thread(() -> simpleLogic("B", 10)).start();
...
}

void simpleLogic(String label, int nofIt) {
    for (int i = 0; i < nofIt; i++) {
        System.out.println(i + " " + label);
    }
}
```

Diese Variante ist komfortabler. Man braucht keinen Konstruktor um Parameter an die Run-Methode weiterzugeben. Grundsätzlich ist der gesamte Code um einiges einfacher.

Join



Ein Warten auf die Beendigung eines Threads. T.join() blockiert solange, bis t terminiert (Warten bis jemand anders fertig wird). Nach Join gilt !t.isAlive(). Der Zugriff auf die Variable von t ist erst nach dem Join sicher (z.B. für Resultate des Threads. Main Finished (Beispiel) soll somit erst kommen, wenn beide wirklich fertig sind. Beide Threads müssen dazu im Main gejoint werden.

Beispiel

```
Thread threadA = new SimpleThread("A", 100); ...
Thread threadB = new SimpleThread("B", 100); threadA.join();
System.out.println("Threads start");           threadB.join();
threadA.start();                           System.out.println("Threads joined");
threadB.start();
```

Passivierung

Dies findet mit statischen Methoden der Thread-Klasse statt.

- **Thread.sleep(milliseconds)**
 - o Der Laufende Thread geht in den Wartezustand. Nach Zeitablauf wird er wieder Ready.
Muss dann aber zuerst wieder zugeordnet werden.
 - o Er kann nur sich selbst schlafen legen und nicht die anderen. Mit Sleep sind die Verzahnungen meist besser.
- **Thread.yield()**
 - o Der Laufende Thread gibt den Prozessor frei. Er wird aber direkt wieder ready.
 - o Es provoziert mehr Thread-Wechsel und daher sollte eine stärkere Verzahnung resultieren (ist aber nicht immer so).

Beispiel

```
for (int i = 0; i < 100; i++) {
  System.out.println(i);
  try {
    Thread.sleep(100);
  } catch (InterruptedException e) { }
}
```

Deklarierte Exception
von sleep()

InterruptedException

Ist eine mögliche Exception bei blockierenden Aufrufen von sleep(), join() etc. Threads können auch von aussen unterbrochen werden mit myThread.interrupt().

Kooperatives Caneling sollte nur verwendet werden, wenn die Cancel-Policy des Threads bekannt ist. Oft wird die Exception zum Aufbrechen von Blockaden missbraucht. Meist hinterlässt dies inkonsistente Zwischenzustände oder die Exception wird ignoriert und es blockiert weiter.

Weitere Thread Methoden

- Static Thread **currentThread()**
 - o Gerade ausführende Thread-Instanz
- Void **setDaemon(boolean on)**
 - o Thread als «Daemon» markieren (default ist false)
- Thread Name einstellen
 - o String **getName()**, void **setName(String name)**
 - o Thread(String name), Thread(Runnable r, String name)
 - o Default: "Thread-0", "Thread-1" etc.
 - o Grundsätzlich nur für diagnostische Ausgaben von Vorteil

Monitor Synchronisation

Thread.currentThread().join() ➔ Es wird herausgefunden, wer er gerade selbst ist. Wir rufen nun einen join auf uns selber auf und somit kommt es zu einem Stehenbleiben (ewig warten), da er ja auf sich selber wartet, bis er beendet wird. Dieser Fall wird aber nie eintreten.

Thread Synchronisation

Ohne Vorkehrung laufen Threads beliebig verzahnt oder parallel. Oft muss aber die Nebenläufigkeit der Threads beschränkt werden. Die Synchronisation ist die Einschränkung der Nebenläufigkeit.

Fälle der Synchronisation

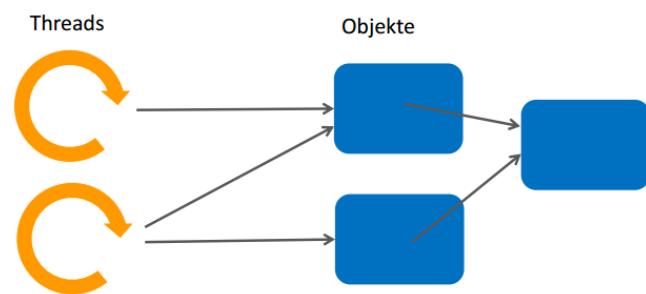
Gegenseitiger Ausschluss

Nur einer darf die Kaffeemaschine auf einmal benutzen. Oder nur eine Vorlesung zur gleichen Zeit im gleichen Raum.

Warten auf Bedingungen

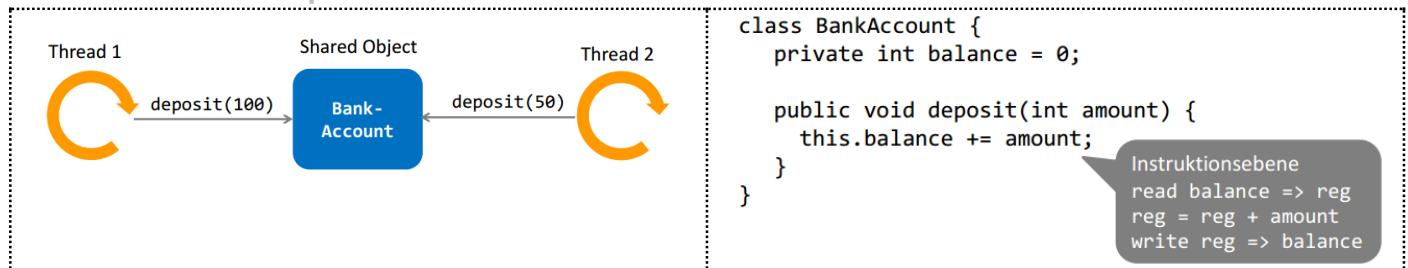
Post erst abholen, wenn Sie angeliefert worden ist. Oder am Telefon erst reden, wenn die Gegenseite abgenommen hat.

Gemeinsame Ressourcen



Threads teilen sich einen Adressraum und damit den Heap. Threads können daher gleichzeitig auf dasselbe zugreifen. Wie die Instanzvariablen, Statische Variablen, Elemente in einem Array oder lokale Variablen (in Java 8 nicht mehr). Die Zugriffe müssten daher genügend synchronisiert werden, sonst kommt es zu unkontrollierten oder falschen Interaktionen zwischen Threads (Race Conditions).

Race Condition Beispiel



Thread 1	Balance	Thread 2
read balance => reg (reg == 0)	0	
	0	read balance => reg (reg == 0)
reg = reg + amount (reg == 100)	0	
	0	reg = reg + amount (reg == 50)
write reg => balance	100	
	50	write reg => balance

Folgende Tabelle zeigt, dass auch etwas anders resultieren kann. Und dies obwohl wir eigentlich 150 erwartet haben. (Wir haben ja gesamthaft 150 eingezahlt.)

Kritische Abschnitte

Deposit() aus dem vorgehenden Beispiel ist der kritische Abschnitt (CriticalSection). Dieser sollte nur von einem Thread zur gleichen Zeit ausführbar sein. Es braucht daher einen gegenseitigen Ausschluss (Mutual Exclusion).

```
public void deposit(int amount) {
    enter critical section
    this.balance += amount;
    exit critical section
}
```

Wie implementiert man den gegenseitigen Ausschluss?

Naiver Ansatz

```
class BankAccount {
    private int balance = 0;
    private boolean locked = false;

    public void deposit(int amount) {
        while (locked) { } // busy waiting loop
        locked = true;    // enter critical section
        this.balance += amount;
        locked = false;   // exit critical section
    }
}
```

```
public void deposit(int amount) {
    while (locked) { }
    locked = true;
    this.balance += amount;
    locked = false;
}
```

Schlaufe und Zuweisung sind nicht atomar

Thread 1	locked	Thread 2
Read locked in loop (locked == false)	false	Read locked in loop (locked == false)
Write locked = true	true	Write locked = true
Critical section		Critical section

Bei dieser Implementation kommt es aber genau zum gleichen Fehler. Die Sachen sind nicht zusammengeschweißt und können aufgeteilt werden. Es findet kein gegenseitiger Ausschluss statt.

Gegenseitiger Ausschluss

Die korrekte Implementierung ist nicht trivial. Algorithmen wie Dekker, Peterson oder Lamport's Bakery. Es könnten auch Atomare Compare-and-Set (CAS) Instruktionen verwendet werden. Zudem muss auch die Weak Memory Consistency beachtet werden (Sehe ich Änderungen anderer CPU's oder nicht, Memory Fences). Des Weiteren stehen noch Effizienzfragen im Raum. Busy Waiting zu teuer für lange Wartezeiten oder 1 CPU und die Warteschlange zu teuer für sehr kurzes Warten bei Multi-GPU. Daher sollten in jedem Fall die vorgefertigten Synchronisationsmechanismen verwendet werden.

Java Synchronized Methoden

```
class BankAccount {
    private int balance = 0;

    public synchronized void deposit(int amount) {
        this.balance += amount;
    }
}
```

Kritischer Abschnitt

Für Methoden steht der Modifier «synchronized» zur Verfügung. Der Body der Methode ist ein kritischer Abschnitt und wird unter gegenseitigen Ausschluss ausgeführt.

Funktionsweise von Synchronized

Jedes Objekt hat einen Lock (Monitor-Lock). Maximal ein Thread kann denselben Lock haben. Der synchronized Block belegt den Lock des Objektes. Bei Eintritt wird Lock besetzt – sonst warten, bis frei. Bei Austritt wird der Lock wieder freigegeben.

Beispiel Monitor Lock

```
class BankAccount {
    private int balance = 0;

    public synchronized void deposit(int amount) {
        this.balance += amount;
    }

    public synchronized boolean withdraw(int amount) {
        if (amount > this.balance) {
            this.balance -= amount;
            return true;
        } else {
            return false;
        }
    }
}
```

Nur ein Thread kann eine der synchronized Methoden in derselben Instanz zur gleichen Zeit ausführen

Im gegenseitigen Ausschluss
Nur eine der beiden Methoden kann gleichzeitig ausgeführt werden. Gilt nur für das gleiche Objekt.

Synchronized Statement Block

Mit synchronized (object) {statements} kann explizit angegeben werden, welche Instanz geblockt werden soll.

```
class BankAccount {
    private int balance = 0;

    public void deposit(int amount) {
        synchronized(this) {
            this.balance += amount;
        }
        System.out.println("Deposit done");
    }
}
```

Monitor-Lock auf this
Kritischer Abschnitt

```
public class Test {
    synchronized void f() { ... }
    static synchronized void g() { ... }
}
```

Object Lock
Class Lock

Bei Arten sind gleichwertig und es ist egal wie man es verwendet.

Äquivalente Verwendungen

```
public class Test {
    void f() {
        synchronized(this) { ... }
    }
    static void g() {
        synchronized(Test.class) { ... }
    }
}
```

Object Lock
Class Lock

Exit aus Synchronized Block

Der Lock wird bei jedem Exit freigegeben. Ein Exit findet am Ende des Blocks statt, bei einem Return Statement oder bei einer Unbehandelten Exception.

Rekursive Locks

```
synchronized void limitedDeposit(int amount) {
    if (amount + balance <= limit) {
        deposit(amount);
    }
}

synchronized void deposit(int amount) { ... }
```

Nested Lock

Der gleiche Thread kann durch geschachtelte Aufrufe denselben Monitor-Lock mehrfach beziehen. Der Lock wird erst beim letzten Release freigegeben werden. Ich muss aber mitzählen wie tief ich bereits bin.

Naiver Ansatz – Warten auf Bedingung

```
class BankAccount {
    private int balance = 0;

    public synchronized void withdraw(int amount)
        throws InterruptedException {
        while (amount > this.balance) {
            Thread.sleep(1);
        }
        this.balance -= amount;
    }

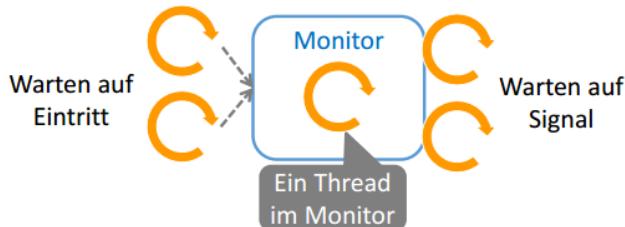
    public synchronized void deposit(int amount) {
        this.balance += amount;
    }
}
```

Keine Lösung!

Der Monitor-Lock sperrt das Objekt vor weiterem Zugriff. Um aber Geld einzahlen zu können, dürfte das Objekt nicht gesperrt sein. Sleep() und yield() geben den Monitor-Lock nicht frei. Der andere Thread kann Bedingung gar nie erfüllen. Ein Pollen in Zeitabständen ist ineffizient. Viele unnötige Versuche (Kontextwechsel und Rechenzeit). Zudem hohe Verzögerung, bis die erfüllte Bedingung erkannt wird.



Wieso funktioniert das nicht?



Dieses Konzept stammt von Brinch Hansen und Hoare. Das Objekt hat einen internen gegenseitigen Ausschluss. Nur ein Thread operiert zur gleichen Zeit im Monitor. Non-private Methoden sind alle synchronized, gearbeitet wird mit privaten Variablen. Mit dem Wait & Signal Mechanismus können Threads um Monitor auf die Bedingung warten. Zudem können Threads die Bedingungen signalisieren und wartende Threads aufwecken.

Beispiel

```
class BankAccount {
    private int balance = 0;

    public synchronized void withdraw(int amount)
        throws InterruptedException {
        while (amount > balance) {
            wait(); ————— Warte auf Bedingung
        }
        balance -= amount;
    }

    public synchronized void deposit(int amount) {
        balance += amount;
        notifyAll(); ————— Wecke alle im Monitor
        wartende Threads
    }
}
```

Wie funktioniert das?

Wait() gibt den Monitor-Lock temporär frei. Damit kann ein anderer Thread die Bedingung im Monitor erfüllen.

```
public synchronized void withdraw(int amount)
    throws InterruptedException {
    while (amount > this.balance) {
        wait();
    }
    this.balance -= amount;
}
```

1. In Warteraum gehen
2. Monitor freigeben
3. (Inaktiv, bis zum Wecksignal)
4. Monitor neu beziehen

Wecksignal

Es wird vom siganlsieren einer Bedingung im Monitor gesprochen. Notify() weckt einen beliebigen wartenden Thread im Monitoir. notifyAll() weckt alle im Monitor wartende Threads. Kein Effekt, falls kein Thread wartet.

```
public synchronized void deposit(int amount) {
    this.balance += amount;
    notifyAll(); —————
    } ————— 1. Weckt alle Threads in wait()
              2. Behält Monitor und macht weiter
```

Analyse des Beispiels

```
class BankAccount {
    private int balance = 0;

    public synchronized void withdraw(int amount)
        throws InterruptedException {
        while (amount > balance) {
            wait();
        }
        balance -= amount;
    }

    public synchronized void deposit(int amount) {
        balance += amount;
        notifyAll();
    }
}
```



Wieso wird hier notifyAll() verwendet?
Wieso ist das Warten in einer Schlaufe?

Es wird notifyAll verwendet, da auch mehrere wartende Threads möglich sind.

Das Warten ist in einer Schlafe, da sonst die Rechenzeit wieder abgegeben wird.

Pauschales Wait & Signal

Es gibt keine Unterscheidung zwischen verschiedenen semantischen Bedingungen. Wartende müssen selber schauen, ob sie ein Signal interessiert (ihre Bedingung erfüllt ist).

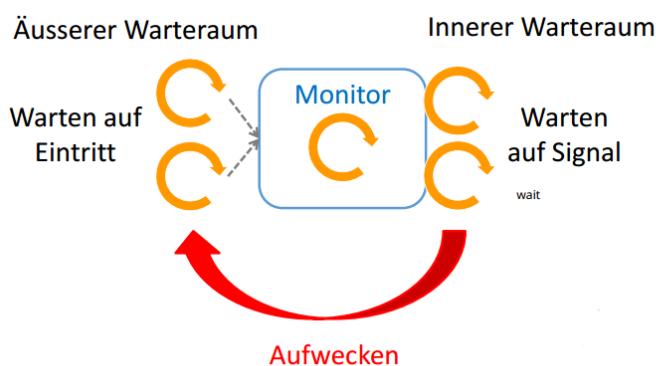
Java Monitor - Details

Wait(), notify() und notifyAll() nur in einem synchronized Block verwendbar. Sonst eine IllegalMonitorStateException.

Bei wait() mit rekursiven Locks werden alle gehaltenen Locks auf dem Objekt temporär freigegeben. Aber nur auf dem Objekt und nicht auf anderen Objekten.

Gründe zum Aufwachen aus dem wait()

- Notify(), notifyAll()
- InterruptedException
- Spurious Wakeup
 - o Fälschliches Aufwecken in vereinzelten Betriebssystemen
 - o Schlechtes Design → OS Implementierung vereinfacht, dafür Benutzung kompliziert.

Java Monitor Funktionsweise

Wenn ich aufgeweckt werde, muss ich natürlich wieder neu anstehen. Er bekommt nicht direkt den Monitor.

Signal and Continue

Der signalisierende Thread behält den Monitor. Nach notify()/notifyAll() läuft er im Monitor weiter. Der Aufgeweckte Thread kommt nicht direkt in den Monitor. Der Signalisierende Thread ist ja noch drin. Daher der Kompromiss. Der kommt in den äusseren Warteraum. Der aufgeweckte Thread muss also neu um den Monitor-Eintritt kämpfen wie alle anderen eintrittswilligen Threads.



Falsch

```
class BoundedBuffer<T> {  
    private Queue<T> queue = new LinkedList<>();  
    int limit; // initialize in constructor  
  
    public synchronized void put(T x) throws InterruptedException {  
        if (queue.size() == limit) {  
            wait(); // await non-full  
        }  
        queue.add(x);  
        notify(); // signal non-empty  
    }  
  
    public synchronized T get() throws InterruptedException {  
        if (queue.size() == 0) {  
            wait(); // await non-empty  
        }  
        T x = queue.remove();  
        notify(); // signal non-full  
        return x;  
    }  
}
```



Was macht dieser Code?
Sehen Sie die Fehler?

Falle 1 – Wait mit if

Aufgeweckter Thread muss neu um Montor-Eintritt kämpfen. Anderer Thread kann vor ihm in den Monitor eintreten (ihn überholen) und seine Bedingung invalidieren. Eventuell kommt es auch zu einem Spurious Wakeup. Lösung → Eine Wartebeindung in der Schlaufe testen.

```
while (!condition) {  
    wait();  
}
```

Falle 2 – Single Notify

Mehrere semantische Warebedingungen «nicht leer», «nicht voll». Es können sich so Bedingung für beide Threads anstauen. Notify() weckt irgendein Thread im Warteraum auf. Der Aufgeweckte Thread wartet evtl. auf andere Bedingung. Es kommt dann kein Thread dran, der weitermachen kann.

⇒ notifyAll() verwenden.

Korrekte Lösung

```

class BoundedBuffer<T> {
    private Queue<T> queue = new LinkedList<T>();
    int limit; // initialize in constructor

    public synchronized void put(T x) throws InterruptedException {
        while (queue.size() == limit) {
            wait(); // await non-full
        }
        queue.add(x);
        notifyAll(); // signal non-empty
    }

    public synchronized T get() throws InterruptedException {
        while (queue.size() == 0) {
            wait(); // await non-empty
        }
        T x = queue.remove();
        notifyAll(); // signal non-full
        return x;
    }
}

```



Monitor Effizienz Problem

Bei mehreren Warebedingungen weckt notifyAll() prophylaktisch immer alle auf. Alle treten nacheinander in den Monitor ein. Für einen ist die Bedingung erfüllt, alle anderen rufen wieder wait() auf. Es kommt zu vielen Kontext-Wechsel und hohen Synchronisationskosten.

Monitor Fairness Probleme

- **Java** – Keine garantie FIFO-Warteschlange
 - o Notify() weckt irgendein Thread im Warterraum
 - o Einige Threads können so nie dran kommen.
 - o Auch ein Grund für notifyAll().
- **Signal-and-Continue**
 - o Aufgeweckte Threads kommen in den äusseren Warterraum
 - o Können kontinuierlich überholt werden
 - o Auch bei FIFO-Warteschlange (.NET) nicht garantiert fair.

Spezifische Synchronisationsprimitiven

Letzte Vorlesungswoche

Quiz

```

Drehkreuz

public class Turnstile {
    private boolean open = false;

    public synchronized void pass()
        throws InterruptedException
        while (!open) { wait(); }
        open = false;
    }

    public synchronized void open() {
        open = true;
        notify(); // or notifyAll() ?
    }
}

```

Braucht es ein Notify? Braucht es die wait()-Schlaufe?

Ja ein Single Notify reicht aus, da wir nicht mehrere Bedingungen haben. NotifyAll() haben wir damals eingesetzt, da mehrere Bedingungen vorhanden waren.

Wann reicht ein Single Notify?

1. Nur eine semantische Bedingung (Uniform Waiters)
 - a. Bedingung interessiert jeden wartenden Thread
 2. Bedingung gilt jeweils nur für einen (One-In/One-Out)
 - a. Nur ein einziger wartender Thread kann weitermachen
- ⇒ Fairness-Problem in Java: Weckt beliebigen Thread. Dies ist der Grund für notifyAll() – garantiert aber auch keine Fairness.

Monitor Diskussion

Vorteile Sehr mächtiges Konzept, jede Synchronisation damit realisierbar, OO Einbettung (Synchronisation in Objekte gekapselt).

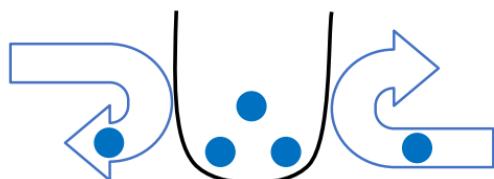
Nachteil Nicht für alles optimal, muss Logik selber implementieren, Effizienzprobleme (notifyAll() bei mehreren Conditions), Fairnessprobleme (Signal and Continue, kein FIFO)

Semaphor

```

Semaphore s = new Semaphore(3);
- Initialisierung mit 3 freien Ressourcen

```



s.acquire()
– Bezieht freie Ressource
– Wartet, wenn keine verfügbar

s.release()
– Gibt Ressource frei
– Benachrichtigt Wartende

Das Konzept kommt von E.W. Dijkstra.
Die Semaphor kommt vom Eisenbahn-Signal. Dabei handelt es sich um eine Vergabe einer beschränkten Anzahl freier Ressourcen. (Freie Zugstrecke für einen, Freie Parkplätze für mehrere, Kuchenstücke für mehrere/einen).

Es ist ein Objekt mit Zähler. Der Zähler ist die Anzahl noch freier Ressourcen.

Methode **acquire()** bezieht die freie Ressource. Wartet, falls keine verfügbar ist (Zähler <= 0). Ansonsten wird der Zähler dekrementiert.

Methode **release()** gibt die Ressource frei und inkrementiert den Zähler.

```
public class Semaphore {
    private int value;

    public Semaphore(int initial) {
        value = initial;
    }

    public synchronized void acquire() throws InterruptedException {
        while (value <= 0) { wait(); }
        value--;
    }

    public synchronized void release() {
        value++;
        notify();
    }
}
```

Hier nicht garantiert fair

Arten von Semaphoren

Allgemeine Semaphore

Zähler zwischen 0 bis N, mit new Semaphore(N), Bis zu N Threads können gleichzeitig akquiriert haben, Für Quotas, Service Throttling etc.

Binäre Semaphore

Zähler nur 0 oder 1, mit new Semaphore(1), für einen gegenseitigen Ausschluss (1=offen, 0=geschlossen).

In Java kann der Zähler auch negativ initialisiert werden. Nicht so in anderen Systemen (Windows, .NET).

Faire Semaphore

Wird mit new Semaphore(N, true) erstellt. Es benutzt die FIFO-Warteschlange für eine Fairness. Diese Variante ist aber langsamer als die unfaire Variante. Default ist unfair.

Anwendung

Semaphore mit synchronized

Zeit: 4 Sekunden

```
class BoundedBuffer<T> {
    private Queue<T> queue = new LinkedList<>();
    private Semaphore upperLimit = new Semaphore(CAPACITY, true);
    private Semaphore lowerLimit = new Semaphore(0, true);

    public void put(T item) throws InterruptedException {
        upperLimit.acquire(); Reduziere Kapazität
        synchronized (queue) { queue.add(item); }
        lowerLimit.release(); Erhöhe Inhalt
    }

    public T get() throws InterruptedException {
        T item;
        lowerLimit.acquire(); Reduziere Inhalt
        synchronized(queue) { item = queue.remove(); }
        upperLimit.release(); Erhöhe Kapazität
        return item;
    }
}
```

Nur mit Semaphore

Zeit: 1.2 Sekunden (mit/oder ohne fairen Semaphoren). Diese Variante ist schneller da keine Schleifen benötigt werden und kein pauschales Aufwecken stattfindet.

```
class BoundedBuffer<T> {
    private Queue<T> queue = new LinkedList<>();
    private Semaphore upperLimit = new Semaphore(Capacity, true);
    private Semaphore lowerLimit = new Semaphore(0, true);
    private Semaphore mutex = new Semaphore(1, true);

    public void put(T item) throws InterruptedException {
        upperLimit.acquire();
        mutex.acquire(); queue.add(item); mutex.release();
        lowerLimit.release();
    }

    public T get() throws InterruptedException {
        lowerLimit.acquire();
        mutex.acquire(); T item = queue.remove(); mutex.release();
        upperLimit.release();
        return item;
    }
}
```

Mit fair immer noch 1.2 Sekunden.

Evtl. noch in try-finally setzen

Synonyme Operationen

acquire → decrement → wait → P ("probeer")
release → increment → signal → V ("verhoog")

Es gibt es auch im Betriebssystem für die Inter-Prozess Synchronisation (Betriebssysteme 1 und 2). In Java dient es der Intra-Prozess Synchronisation. Benutzt intern keine Betriebssystem-Semaphore da zu teuer, sondern eine übliche Thread-Synchronisation über den Heap.

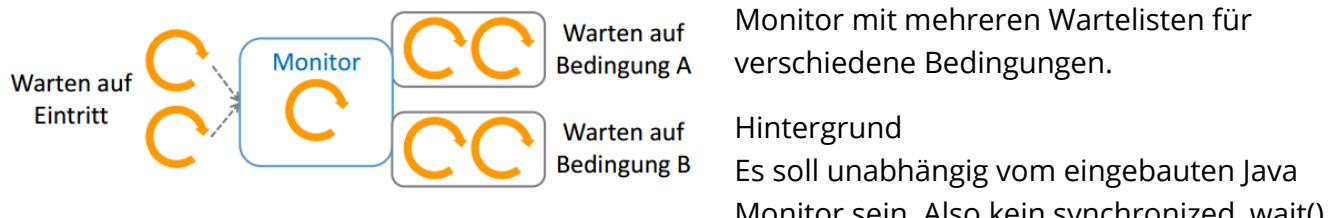
Es ist auch möglich mehr als eine Ressource anzufordern und freizugeben.

```
acquire(int permits)
• Wartet, solange Zähler < permits ist
• Zähler -= permits
release(int permits)
• Zähler += permits
```

Diskussion

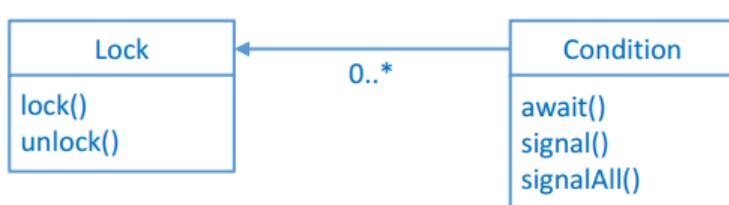
Es ist sehr mächtig. Beliebige Synchronisation implementierbar. Aber relativ low-level. Wir wollen beim Buffer die ineffiziente notifyAll() vermeiden. Zudem ist eine Signalisierung der spezifischen Bedingungen gewünscht. Nicht leer ⇔ lowerLimit > 0 und Nicht voll ⇔ upperLimit > 0

Lock & Condition



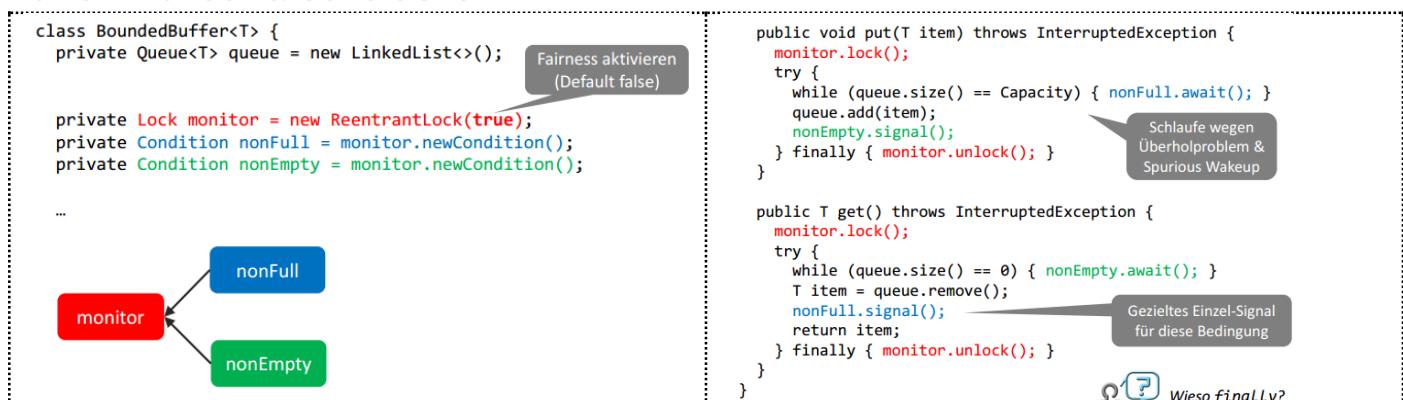
notify(), notifyAll(). Sondern spezifische Synchronisationsprimitiven über die API. Diese kann benutzt werden, um eigenen Monitor zu bauen. → Package java.util.concurrent.locks.

Primitiven



Das Lock-Objekt ist eine Sperre für den Eintritt in den Monitor (Äussere Warteliste). Das Condition-Objekt ist «Wait & Signal» für bestimmte Bedingungen (also die Innere Warteliste). Mehrere Conditions sind pro Lock (daher Monitor) möglich.

Buffer mit Lock & Conditions



Read-Write Lock

Parallel	Read	Write
Read	Ja	Nein
Write	Nein	Nein

Ein gegenseitiger Ausschluss ist unnötig streng für rein lesende Abschnitte. Read-Write erlauben parallele Lese-Zugriffe (Reader), beim Schreib-Zugriff findet aber ein gegenseitiger Ausschluss statt (Writer).

Verwendung

```

Fairer Lock

ReadWriteLock rwLock = new ReentrantReadWriteLock(true);

rwLock.readLock().lock();           Shared Lock
// read-only accesses
rwLock.readLock().unlock();

rwLock.writeLock().lock();          Exclusive Lock
// write (and read) accesses
rwLock.writeLock().unlock();

```

Falls ein schreibender Zugriff in Abschnitt involviert ist, muss der Write Lock verwendet werden.

Beispiel

```

class NameDatabase {
    private Collection<String> names = new HashSet<>();
    private ReadWriteLock rwLock = new ReentrantReadWriteLock(true);

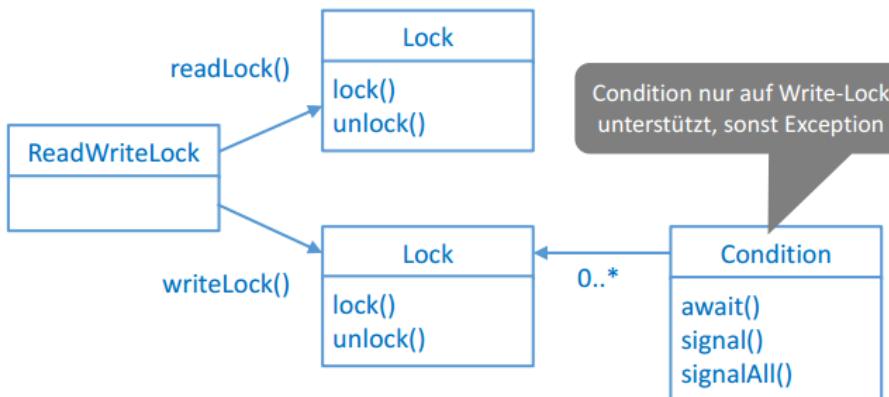
    public Collection<String> find(String pattern) {
        rwLock.readLock().lock();
        try {
            for (String name : names) {
                if (name.matches(pattern)) { return true; }
            }
            return false;
        } finally {
            rwLock.readLock().unlock();
        }
    }

    public void put(String name) {
        rwLock.writeLock().lock();
        try {
            names.add(name);
        } finally {
            rwLock.writeLock().unlock();
        }
    }
}

```

HSR

Read-Write Lock mit Conditions



Macht es Sinn, die Condition nur auf Write-Locks zu gestatten?
Beim Read macht man eigentlich gar keine Modifikation. Daher ja. Es gibt aber auch Fälle, wo der Reader auf etwas wartet. Daher nicht unbedingt immer.

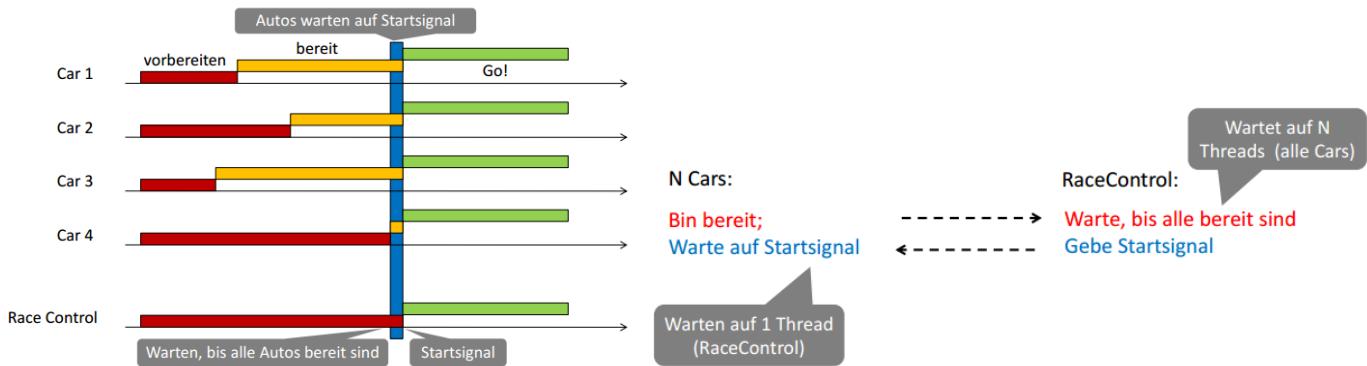
Zwischenstand

Bisherigen Synchronisationsprimitiven

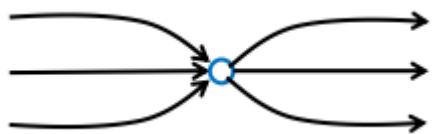
Schutz von Shared Ressourcen bei Multi-Threading (Monitor, Semaphore, Lock & Condition, Read-Write Lock)

Weitere Synchronisationsprimitiven

Zeitlicher Synchronisationspunkt von mehreren Threads (CountDownLatch, CyclicBarrier, Phaser)



Synchronisationspunkt



Anzahl von Threads warten auf eine Bedingung. Nach der Erfüllung der Bedingung laufen alle weiter. Beispiele: Autos warten auf Startsignal, Player warten alle aufeinander.

Count Down Latch

Eine Synchronisationsprimitive mit Count Down Zähler. Threads können warten, bis Zähler ≤ 0 ist. Await() wartet das der Count Down 0 ist. Threads können auch runterzählen, mit countDown() wird der Zähler um 1 dekrementiert. Es gibt Ähnlichkeiten zur Semaphore. Diese blockiert bei ≤ 0 , während der Count Down Latch bei >0 blockiert.

Beispiel

Bespiel mit CountDown Latch

```
CountDownLatch carsReady = new CountDownLatch(N);
CountDownLatch startSignal = new CountDownLatch(1);

N Cars: carsReady.countDown(); -----> Einer gibt Signal
RaceControl: startSignal.await(); <----- carsReady.await();
```

Analoge Lösung mit Semaphoren

```
Semaphore readyCars = new Semaphore(0);
Semaphore startAllows = new Semaphore(0);

N Cars: readyCars.release(); -----> RaceControl: readyCars.acquire(N);
RaceControl: startAllows.acquire(); <----- startAllows.release(N); Erlaube N Cars
```

Details

Latches synchronisieren die Anzahl von Threads auf die Anzahl von Ereignissen. Ein Ereignis ist eine Count Down Operation, eine Synchronisation, ist ein Warten bis der Count Down Zähler ≤ 0 wird.

Auslösen der Ereignisse findet durch beliebige Thread statt, an einer beliebigen Stelle. Der countDown() blockiert nie.

Latches sind nur einmalig verwendbar. Der Count Down Zähler lässt sich nicht wieder hochsetzen. Neue Synchronisation == Neue Latch Instanz.

Cyclic Barrier

Treffpunkt für eine fixe Anzahl von Threads. Die Threads warten, bis alle angekommen sind. Sie warten am Treffpunkt mit await(). Die Anzahl treffender Threads muss vorgegeben sein. Await() blockiert, bis so viele Threads await() aufgerufen haben wie es sollten. Der Cyclic Barrier ist wiederverwendbar. Therads können sich in mehreren Runden bei der gleichen Barriere synchronisieren.

Beispiel mit Cyclic Barrier

```
CyclicBarrier raceStart = new CyclicBarrier(N);
N Cars:
raceStart.await();
```

Autos fahren direkt los,
sobald alle da sind

Anzahl sich
treffender Threads

Brauche kein Race Control mehr

Wiederholte Barriere

```
CyclicBarrier gameRound = new CyclicBarrier(N);
```

N Players:

```
while (true) {
  gameRound.await();
  // play concurrently with others
}
```

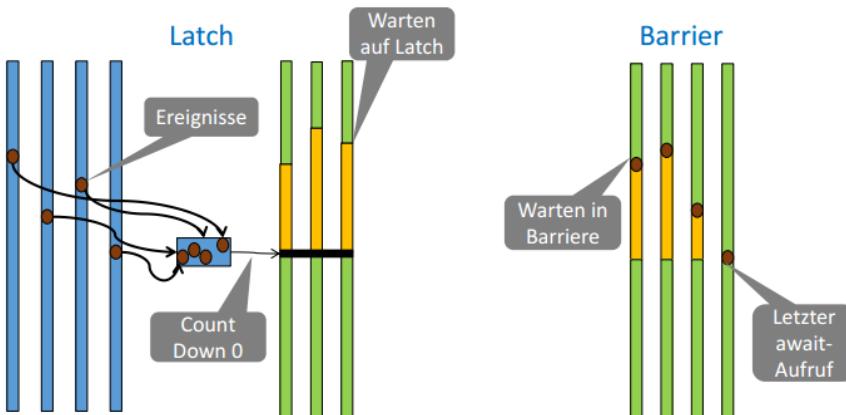
Barriere schliesst sich
automatisch für nächste Runde



Die Anzahl Teilnehmer wird beim Konstruktor festgelegt und ist nicht mehr änderbar. Mit int getParties() lässt sich die Zahl abrufen. Das Passieren der Barriere geschieht mit int await(). Als Rückgabe wert folgt die Anzahl noch fehlender Threads bei der Barriere. Wenn gleich 0 wird die Barriere geöffnet und die Wartenden aufgeweckt.

«Broken Barrier» Problem: Exception in await(), z.B. InterruptedException. Es sind dann alle betroffen und es kommt zu einer BrokenBarrierException.

Latch versus Barriere



Phaser

Verallgemeinerter Cyclic Barrier. Mit `arriveAndAwaitAdvance()` kann die Barriere passiert werden. Ein Teilnehmer kann sicher später anmelden bzw. abmelden. Mit `register()` bzw. `arriveAndDeregister()`. Dies wird erst bei der nächsten Warterunde wirksam.

Player-Thread:

```
phaser.register();
while (...) {
    phaser.arriveAndAwaitAdvance();
    playRound();
}
phaser.arriveAndDeregister();
```

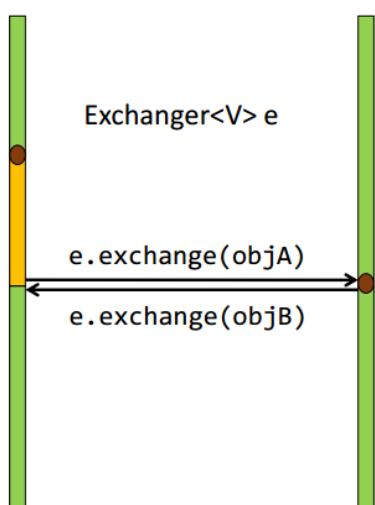
```
Phaser phaser = new Phaser(0);
```

Anfangs keine Player

Rendez-Vous

Ist eine Barriere mit Informationsaustausch, bei welchem nur 2 Parteien involviert sind. Zwei Threads treffen sich und tauschen Objekte aus. Ohne Austausch mit **new CyclickBarrier(2)** und mit Austausch **Exchanger.exchange(something)**.

Exchanger



`V exchange(V x)` blockiert, bis ein anderer Thread auch `exchange()` aufruft. Es liefert das Argument `x` des jeweils anderen Threads.

Beispiel

```
Exchanger<Integer> exchanger = new Exchanger<>();
for (int k = 0; k < 2; k++) {
    new Thread(() -> {
        for (int in = 0; in < 5; in++) {
            try {
                int out = exchanger.exchange(in);
                System.out.println(
                    Thread.currentThread().getName() + " got " + out);
            } catch (InterruptedException e) { }
        }
    }).start();
}
```

Gefahren der Nebenläufigkeit

Nebenläufige Programmierung birgt Risiko neuer Arten von Programmierfehler. Fehler, die es bei Single-Threading so nicht gibt. Sie können sporadisch oder selten auftreten und sind daher schwierig durch Tests zu finden.

Unterschiede

Java Monitor ⇔ Lock & Conditions

In L&C hat man mehrere Warteräume und kann damit gezielter Warten. Zudem ist technisch eine Fairness mit einer FIFO-Queue implementiert (ist aber nicht garantiert).

Semaphore ⇔ CountDownLatch

Der CountDownLatch kann nur einmal verwendet werden (nicht reziklierbar). Beide zählen herunter. Bei der Semaphore ist die Zahl der Durchlässe fix. Die Fairness bei der Semaphore möglich und beim CountDownLatch immer automatisch gegeben.

CountDownLatch ⇔ CyclicBarrier

Die CyclicBarrier ist reziklierbar

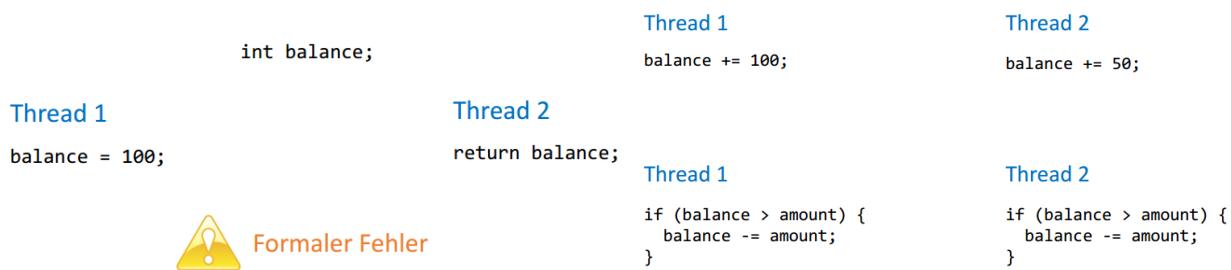
Fehler der Nebenläufigkeit

Race Conditions

Mehrere Threads greifen auf gemeinsame Ressourcen ohne genügende Synchronisation zu. So sind falsche Resultate oder Verhalten möglich. Je nach Thread-Verzahnung und zeitlicher Ausführung. Ursache ist oft ein Data Race, aber nicht immer!

Data Race

Ein unsynchronisierter Zugriff auf den gleichen Speicher (dieselbe Variable oder Array-Element). Es ist mindestens ein schreibender Zugriff (Read-Write, Write-Read, Write-Write) vorhanden.



Race Condition ohne Date Race

```
class BankAccount {
    int balance = 0;
    synchronized int getBalance() { return balance; }
    synchronized void setBalance(int x) { balance = x; }
}
```

Mehrere Threads führen folgendes aus

```
account.setBalance(account.getBalance() + 100)
```

In diesem Falle sind Lost Updates möglich, da beide 0 + 100 machen und somit nur 100 und nicht 200 dabei herauskommt. (also kein atomare Inkrementieren).

Die Critical Sections sind dabei nicht geschützt. Die Low-Level Data Races sind zwar mit Synchronisation eliminiert, aber es sind nicht genügend grosse synchronisierte Blöcke.

	Race Condition	Keine Race Condition
Data Race	Fehlerhaftes Programmverhalten	Programm verhält sich zwar korrekt, dennoch formal falsch
Kein Data Race	Fehlerhaftes Programmverhalten	Richtig

Einfach alles synchronisieren?

Nein, das hilft nichts. Race Conditions sind auch mit Synchronisation möglich. Weitere Nebenläufigkeitsfehler werden später noch gezeigt. Zudem entstehen Synchronisationskosten. Die Synchronisation ist relativ teuer. Zum Beispiel verhindert die Cache Invalidierung die Optimierung.

Verzichtbare Fälle der Synchronisation

In folgenden Fällen kann man auf eine Synchronisation verzichten.

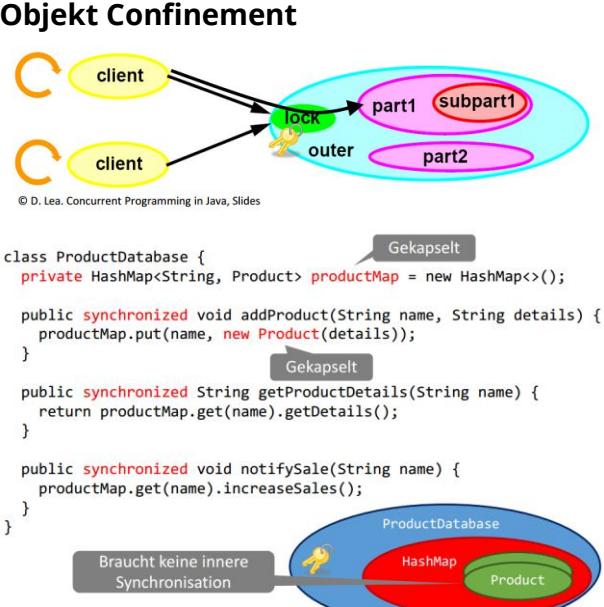
- Immutability (Unveränderlichkeit)
 - o Objekte mit nur lesendem Zugriff
- Confinement (Einsperrung)
 - o Objekt gehört nur einem Thread zu einer Zeit.

Immutable Objects

Dabei sind alle Instanzvariablen alle final (Primitive Datentypen oder Referenz auf wiederum Immutable Objekte). Die Methoden haben nur einen lesenden Zugriff. Der Konstruktor initialisiert die Instanzvariablen. Nach dem Konstruktor kann das Objekt ohne Synchronisation von Threads verwendet werden.

Confinement

Die Struktur garantiert, dass das Objekt nur durch einen Thread zur gleichen Zeit zugegriffen wird. Entweder durch Thread Confinement (Objekt nur über Referenzen von einem Thread erreichbar) oder über Object Confinement (Objekt in anderem bereits synchronisierten Objekt eingekapselt).

Thread Confinement	Objekt Confinement
<pre>void service() { new Thread(() -> { OutputStream output = new FileOutputStream("..."); try { doService(output); } finally { output.close(); } }).start(); } void doService(OutputStream s) { // s must not be passed to other threads }</pre> <p>Kein anderer Thread darf s zugreifen</p> <p>Service Thread</p> <p>Stream</p> <p>Objekt nur vom Thread erreichbar</p>	<p>© D. Lea. Concurrent Programming in Java, Slides</p> <p>Gekapselt</p> <p>Braucht keine innere Synchronisation</p> <p>ProductDatabase</p> <p>HashMap</p> <p>Product</p>  <pre>class ProductDatabase { private HashMap<String, Product> productMap = new HashMap<>(); public synchronized void addProduct(String name, String details) { productMap.put(name, new Product(details)); } public synchronized String getProductDetails(String name) { return productMap.get(name).getDetails(); } public synchronized void notifySale(String name) { productMap.get(name).increaseSales(); } }</pre>

Version	Beispiele	Thread-sicher
Alte Java 1.0 Collections	Vector, Stack, Hashtable	JA
Moderne Collections (java.util, Java > 1.0)	HashSet, TreeSet, ArrayList, LinkedList, HashMap, TreeMap	NEIN 
Concurrent Collections (java.util.concurrent)	ConcurrentHashMap, ConcurrentLinkedQueue, CopyOnWriteArrayList, ...	JA

Klassen / Methoden, welche intern synchronisiert sind. Also keine Race Conditions innerhalb des Codes und der kritische Abschnitt nur pro Methode erfüllt. Es gibt aber keinen kritischen Abschnitt über mehrere Methodenaufrufe und daher sind andere Nebenläufigkeitsfehler möglich.

Die Thread-Safety bei den Java Collections sieht so aus (Bild oben). Die Modernen Collections sind nicht thread-Sicher, da die Synchronisation oft nicht nötig ist und so unnötige Kosten entstehen. Zudem ist die Synchronisation meist ungenügend (Die Elemente und die Iteration der Elemente) ist nicht synchronisiert.

Concurrent Collections

Sind die effizienten Thread-sicheren Collections, welche für eine starke Nebenläufigkeit geeignet sind. Sie haben aber schwach konsistente Iteratoren. Ich sehe nebenläufige Updates bei der Iteration vielleicht nicht.

Achtung – Verstecktes Multi-Threading

- Finalizers (Laufen über separaten Finalizer-Thread)
- Timers (Handler durch separatem Thread ausgeführt, ausser GUI)
- Externe Libaries und Frameworks

Daher Gefahr von Race Conditions.

Deadlocks

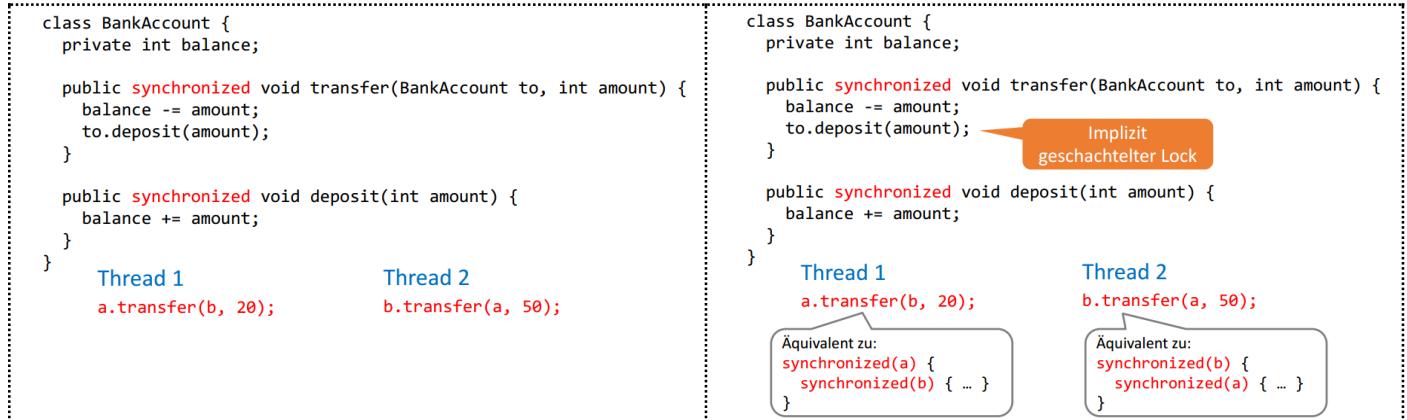
«Gegenseitiges Aussperren von Threads»

Definition

Einige Threads sperren sich gegenseitig so, dass keinen von denen weitermachen kann. Ein Programm mit einem potentiellen Deadlock ist inkorrekt, da dieses Blockieren plötzlich stattfinden kann.

Deadlock – Nested Locks

Thread 1 <pre>synchronized(listA) { synchronized(listB) { listB.addAll(listA); } }</pre>	Thread 2 <pre>synchronized(listB) { synchronized(listA) { listA.addAll(listB); } }</pre>	Thread 1 <pre>synchronized(listA) { synchronized(listB) { listB.addAll(listA); } }</pre>	Thread 2 <pre>synchronized(listB) { synchronized(listA) { listA.addAll(listB); } }</pre>															
 Welches Problem kann hier auftreten? Deadlockgefahr (Stillstand, da Sie sich gegenseitig aussperren.)		<table border="1"> <thead> <tr> <th>Thread 1</th> <th>Thread 2</th> <th>Gesperrte Objekte</th> </tr> </thead> <tbody> <tr> <td>synchronized(listA)</td> <td></td> <td>listA</td> </tr> <tr> <td></td> <td>synchronized(listB)</td> <td>listA, listB</td> </tr> <tr> <td>synchronized(listB) => blockiert</td> <td></td> <td>listA, listB</td> </tr> <tr> <td></td> <td>synchronized(listA) => blockiert</td> <td>listA, listB</td> </tr> </tbody> </table> <p> Beide Threads haben sich gegenseitig ausgesperrt</p>		Thread 1	Thread 2	Gesperrte Objekte	synchronized(listA)		listA		synchronized(listB)	listA, listB	synchronized(listB) => blockiert		listA, listB		synchronized(listA) => blockiert	listA, listB
Thread 1	Thread 2	Gesperrte Objekte																
synchronized(listA)		listA																
	synchronized(listB)	listA, listB																
synchronized(listB) => blockiert		listA, listB																
	synchronized(listA) => blockiert	listA, listB																



Spezialfall Livelocks

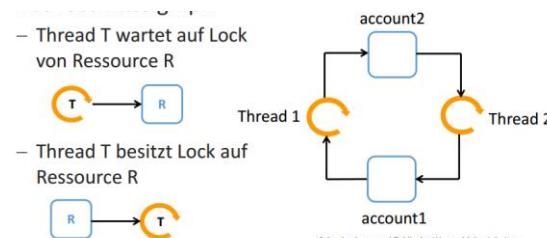
Thread 1 Thread 2

```
b = false;           a = false;
while (!a) { sleep(1); }   while (!b) { sleep(1); }
```

Threads haben sich gegenseitig permanent blockiert, führen aber noch Warteinstruktionen aus und verbrauchen somit CPU während dem Deadlock.

Erkennung

Dies lässt sich mit einem Betriebsmittelgraph erkennen.



Deadlock \Leftrightarrow Zyklus im Betriebsmittelgraph

Voraussetzung

Alle der folgenden vier Bedingungen müssen zutreffen

- Geschachtelte Locks
- Zyklische Warteabhängigkeiten
- Gegenseitiger Ausschluss (Locks)
- Sperren ohne Timeout/Abbruch

Vermeidung

Lineare Ordnung der Ressourcen einführen

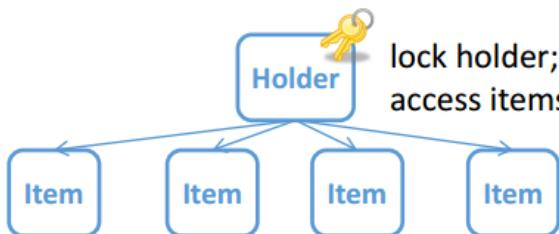
Also nur geschachtelt in aufsteigender Reihenfolge sperren. Zum Beispiel die Konten mit aufsteigender Nummer. Dies eliminiert zyklische Warteabhängigkeiten.

Lock [1] -----> Lock [3] --> Lock [4]



Grobgranulare Locks wählen

Dies wenn lineare Ordnung nicht möglich oder sinnvoll ist. Zum Beispiel die gesamte Bank sperren bei einem Kontenzugriff. Es eliminiert die Schachtelung von Locks.



Starvation

«Kontinuierliche Fortschrittsbehinderung von Threads wegen Fairness-Probleme»

Ein Thread kriegt nie die Chance, auf eine Ressource zuzugreifen. Dies obwohl die Ressource immer wieder frei wird (kein Deadlock oder Livelock). Andere Threads können ihn aber dauernd überholen und Ressource wegschnappen. Im folgenden Beispiel ist Starvation möglich:

```

do {
    success = account.withdraw(100);
} while (!success);
  
```

Starvation ist ein Liveness/Fairnessproblem.

Vermeidung

Faire Synchronisationskonstrukte

Länger wartende Threads mit erfüllter Bedingung haben Vortritt. Dies kann zum Beispiel durch das Einschalten der Fairness bei der Java Semaphore, Lock & Condition sowie Read-Write Lock erreicht werden. Der Java Monitor hat aber ein Fairness-Problem. Er ist sehr starvation-anfällig, vor allem bei vielen Threads.

Thread Prioritäten

Die Prioritäten lassen sich mit `myThread.setPriority(priority)` setzen.

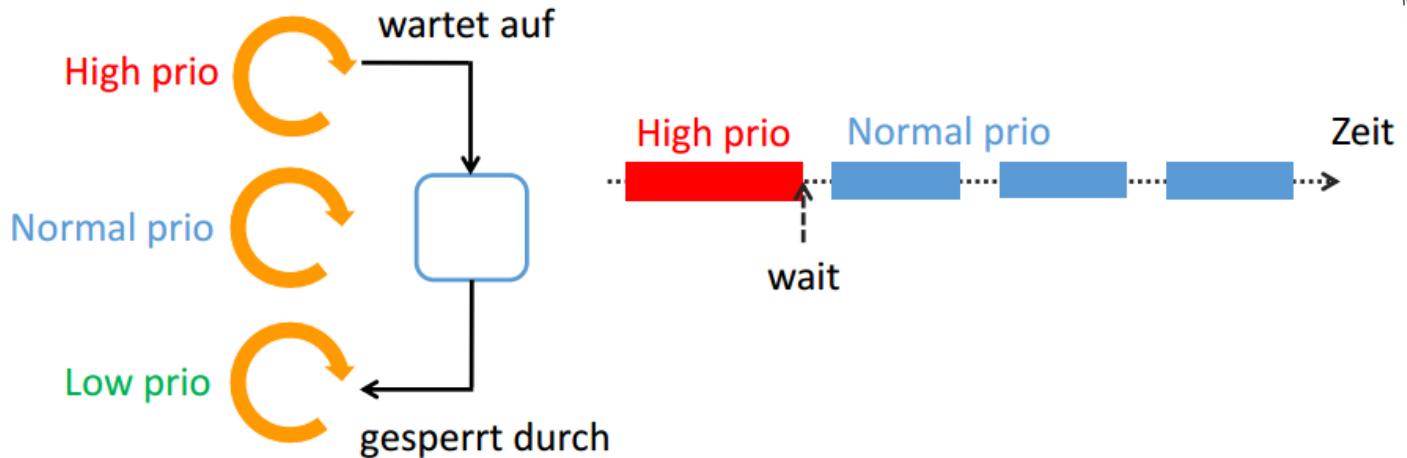
1: MIN_PRIORITY, 5: NORM_PRIORITY, 10: HIGH_PRIORITY

Das Scheduling ist vom Betriebssystem abhängig. Meist haben hochprioritäre Threads immer Vorrang. Preemption findet unter Threads mit gleicher Priorität statt. Zu beachten ist, dass Windows nur 6 Prioritäten hat.

Starvation ist mit Thread Prioritäten dennoch möglich.

Priority Inversion

Ein Hoch prioritärer Thread wartet auf Bedingung von tief prioritärem Thread. Normale Threds können dazwischen laufen. Tief und hoch prioritärer Thread werden verhungern.



Parallelität Korrektheitskriterien

Keine Race Conditions (Safety)

Kritische Abschnitte auf gemeinsame Ressourcen sind genügend synchronisiert.

Keine Deadlocks (Safety)

Threads können sich nicht gegenseitig für unbeschränkte Zeit sperren.

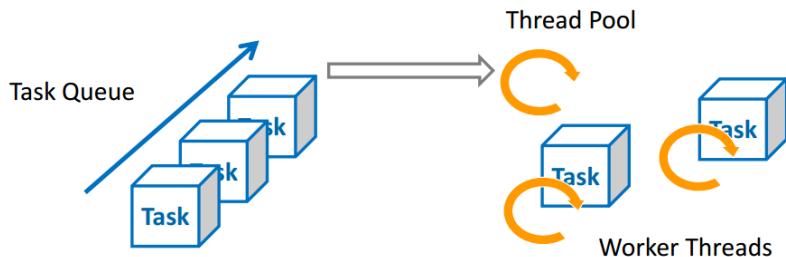
Keine Starvation (Liveness)

Wenn ein Thread auf eine Bedingung wartet, soll er nach einer bestimmten Zeit forschreiten können, sofern die Bedingung genügend oft erfüllt wird.

Thread Pools

Thread-Pool

Konzept und Funktionsweise



Task Queue

Tasks implementieren potentiell parallele Arbeitspakete. Auszuführende Tasks werden in Warteschlange eingereiht.

Thread Pool

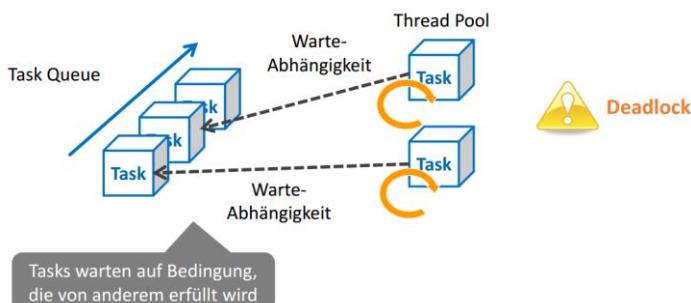
Eine beschränkte Anzahl von Worker-Threads. Sie holen Tasks aus der Warteschlange und führen Sie aus.

Vorteile

- **Beschränkte Anzahl von Threads**
 - o Denn zu viele Threads verlangsamen das System oder überschreiten den verfügbaren Speicher.
- **Recycling der Threads**
 - o Spare Thread-Erzeugung und Freigabe
- **Höhere Abstraktion**
 - o Trennung der Task-Beschreibung (Problem Space) von der Task-Ausführung (Machine Space).
- Die **Anzahl von Threads pro System sind konfigurierbar** → #Worker Threds = #Prozessoren + #I/O Aufrufe.

Somit entsteht ein neuer «Free Lunch» wenn man Programme mit Tasks modelliert laufen diese automatisch schneller auf parallelen Maschinen und ermöglichen eine Ausschöpfung der Parallelität ohne hohe Thread-Kosten.

Einschränkungen



Die Tasks dürfen nicht aufeinander warten. Denn der Task muss zu Ende laufen, bevor der Worker Thread anderen Tasks ausführen kann. Die Ausnahme sind hier die geschachtelten Tasks (Sub-Tasks).

Fork & Join Pool

Der Fork-Join Pool (seit Java 7 &) unterstützt rekursive Aufgaben und hat eine effiziente Implementierung. Die einfachen Executors (seit Java 5) haben keine rekursiven Aufgaben und sind nicht besonders optimiert.

Lancierung

```
Moderner Thread Pool
```

```
ForkJoinPool threadPool = new ForkJoinPool();
```

```
Task in Thread Pool einreihen
```

```
Future<Integer> future = threadPool.submit(() -> {
    int value = ...;
    // long calculation
    return value;
});
```

```
Tasks können Rückgabe haben
```

Future

Konzept

Das Future repräsentiert ein zukünftiges Resultat. Es ist ein Proxy auf ein Resultat, dass evtl. noch nicht bekannt ist, weil die Berechnung noch läuft. Es muss das Ende der Berechnung abgewartet werden, bevor das Resultat zurückgegeben wird.

Verwendung

```
Blockiert nicht, lanciert Task ohne zu Warten
```

```
Future<T> future = threadPool.submit(...);
```

```
...
```

```
T result = future.get();
```

```
Blockiert, bis Task beendet ist
```

Details

Fehler Propagierung

Wenn der Task mit einer unbehandelten Exception beendet wird, lässt sich mit get() dann die Exception liefern. Die ursprüngliche Exception ist darin geschachtelt (Cause).

Task Abbruch

Dieser findet mit cancel(boolean mayInterruptIfRunning) statt. Dabei wird der Tasks aus der Warteschlange herausgenommen. Dies geschieht kooperativ (Bricht den laufenden Task nicht einfach ab). Optional gibt es einen Interrupt auf den Worker-Thread bei laufendem Task.

Fire and Forget

Tasks starten, ohne später das Resultat abzuhören. Der Submitter interessiert sich dabei nicht für das Resultat/Task-Ende. Zum Beispiel für einen Task ohne Resultat. Die unbehandelten Exceptions im Task werden ignoriert.

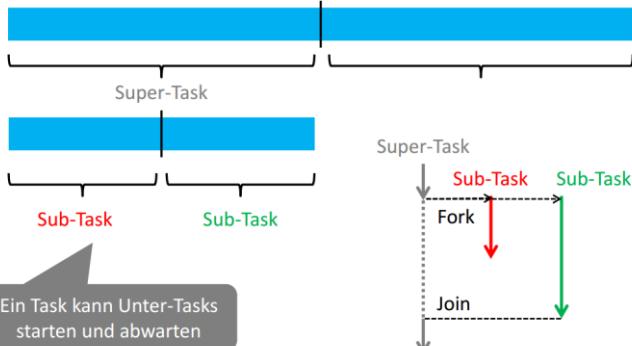
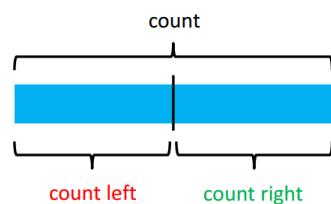
```
threadPool.submit(() -> {
    // Task implementation
});
```

Beispiele**Zählen von Primzahlen (Sequentiell = Ausgangslage)**

Wie viele Primzahlen gibt es zwischen 2 und N?

```
int counter = 0;
for (int number = 2; number < N; number++) {
    if (isPrime(number)) { counter++; }
}
```

Mit nur 2 Mal parallel ist natürlich das Optimum noch nicht erreicht. Dafür weitere Parallelisierungsschritte ist daher die Rekursion geeignet.

**Paralleles Zählen**

```
Future<Integer> left =
  threadPool.submit(() -> count(leftPart));
Future<Integer> right =
  threadPool.submit(() -> count(rightPart));
result = left.get() + right.get();
```

```
class CountTask extends RecursiveTask<Integer> {
  // Constructor

  @Override
  protected Integer compute() {
    // if no or single element => return result
    // split into two parts
    CountTask left = new CountTask(leftPart);
    CountTask right = new CountTask(rightPart);
    left.fork(); right.fork();
    return right.join() + left.join();
  }
}
```

Rekursive Tasks

Diese Tasks können Untertasks starten und abwarten. Sie sind verboten in den alten Java Thread Pools, da es dort zu Deadlocks kommt.

Die Tasks erben von `RecursiveTask<T>` und haben daher folgende Methoden.

- `T compute()` – Task Implementierung
- `fork()` – Starte als Sub-Task in einem anderen Task
- `T join()` – Warte auf Task-Ende und frage Resultat ab
- `T invoke()` – Ein Sub-Tasks starten und abwarten
- `invokeAll()` – Mehrere Sub-Tasks starten und abwarten

Falls es kein Rückgabewert hat, kann man auch `RecursiveAction` verwenden. Dann mit `void` anstatt `T`.

Ausführung*Expliziter Thread-Pool*

```
ForkJoinPool threadPool = new ForkJoinPool();

int result = threadPool.invoke(new CountTask());
```

blockiert

Standard Pool (Java 8)

– `ForkJoinPool.commonPool()`

```
int result = new CountTask().invoke();
```

Konkreter Ausbau

```
class CountTask extends RecursiveTask<Integer> {
    private final int lower, upper;

    public CountTask(int lower, int upper) {
        this.lower = lower; this.upper = upper;
    }

    protected Integer compute() {
        if (lower == upper) { return 0; }
        if (lower + 1 == upper) { return isPrime(lower) ? 1 : 0; }
        int middle = (lower + upper) / 2;
        CountTask left = new CountTask(lower, middle);
        CountTask right = new CountTask(middle, upper);
        left.fork(); right.fork();
        return right.join() + left.join();
    }
}
```

Aber Achtung. Keine Über-Parallelisierung. Sonst wird es wieder langsam.

```
protected Integer compute() {
    if (upper - lower > THRESHOLD) {
        // parallel count
        int middle = (lower + upper) / 2;
        CountTask left = new CountTask(lower, middle);
        CountTask right = new CountTask(middle, upper);
        left.fork(); right.fork();
        return right.join() + left.join();
    } else {
        // sequential count
        int count = 0;
        for (int number = lower; number < upper; number++) {
            if (isPrime(number)) { count++; }
        }
        return count;
    }
}
```

Tuning mit Schwellwert durch Programmierer

Reihenfolge

Schneller

```
task1.fork();
task2.fork();
task2.join();
task1.join();
```

Langsamer

```
task1.fork();
task2.fork();
task1.join();
```



Alternative:
invokeAll(task1, task2);

Internals

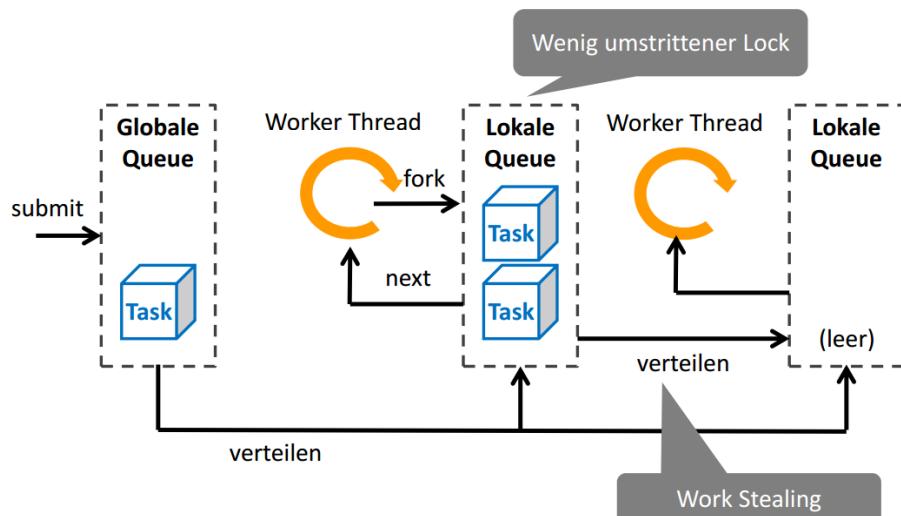
Die Fire-And-Forget laufen evtl. nicht zu Ende, denn die Worker Threads laufen als Daemon-Threads.

Der automatische Parallelitätsgrad

Standardmäßig entspricht die Anzahl der Worker Threads im Pool der Anzahl der Prozessoren.

Nachher findet dynamisches Hinzufügen und Wegnehmen von Threads statt. Das Ziel ist es immer genügend aktiv/laufende Worker-Threads zu haben, dies ist jedoch nicht garantiert.

Der Common Pool verhindert Engpässe durch zu viele Thread Pool. Der Parallelitätsgrad ist zum Teil tiefer als die Anzahl der Prozessoren.



die Schachtelung der fork/joins.

Asynchrone Programmierung

Unnötige Synchronität

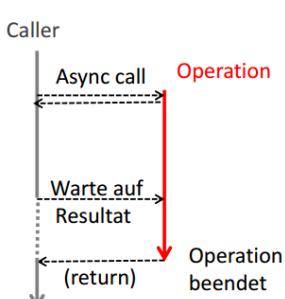
Meisten sind vielfach unnötige blockierende Methodenaufrufe vorhanden. Zum Beispiel bei langlaufenden Rechnungen oder I/O Aufrufen.

```
long result = longOperation();
//other work
process(result);
```

Aufrufer unnötig blockiert

Asynchroner Aufruf

Der Aufrufer soll während der Operation weiterarbeiten können.

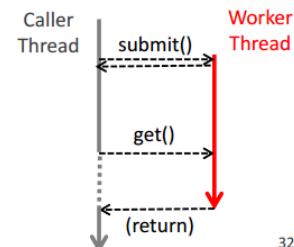


Klassisch würde die Operation in einen Thread oder einen Thread Pool auslagert werden.

```
Future<Long> future =
    threadPool.submit(() -> longOperation());
//other work
process(future.get());
```

Resultat über Future

Asynchrone Ausführung



32

Da wird die asynchrone Aufgabe im Standard Pool gestartet. `runAsync()`, falls es kein Rückgabetyp gibt.

```
CompletableFuture<Long> future =
    CompletableFuture.supplyAsync(() -> longOperation());
//other work
process(future.get());
```

Ende des asynchronen Aufrufs

Caller-zentrisch (Pull)

Der Caller wartet auf das Task Ende und holt sich das Resultat. Z.B. das Future abfragen.

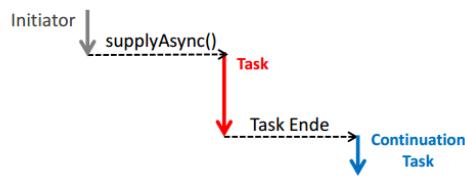
Callee-zentrisch (Push)

Asynchrone Operation informiert direkt über das Resultat. Z.B. Completion Callback (Continuation, Promise).

Continuation

Dabei wird eine Folgeaufgabe an eine asynchrone Aufgabe angehängt. Die Ausführung findet statt, sobald der vorgängige Task fertig ist.

```
CompletableFuture<Long> future =
    CompletableFuture.supplyAsync(() -> longOperation());
...
future.thenAccept(result -> System.out.println(result));
```



Task Continuations

Ausführung der Continuation

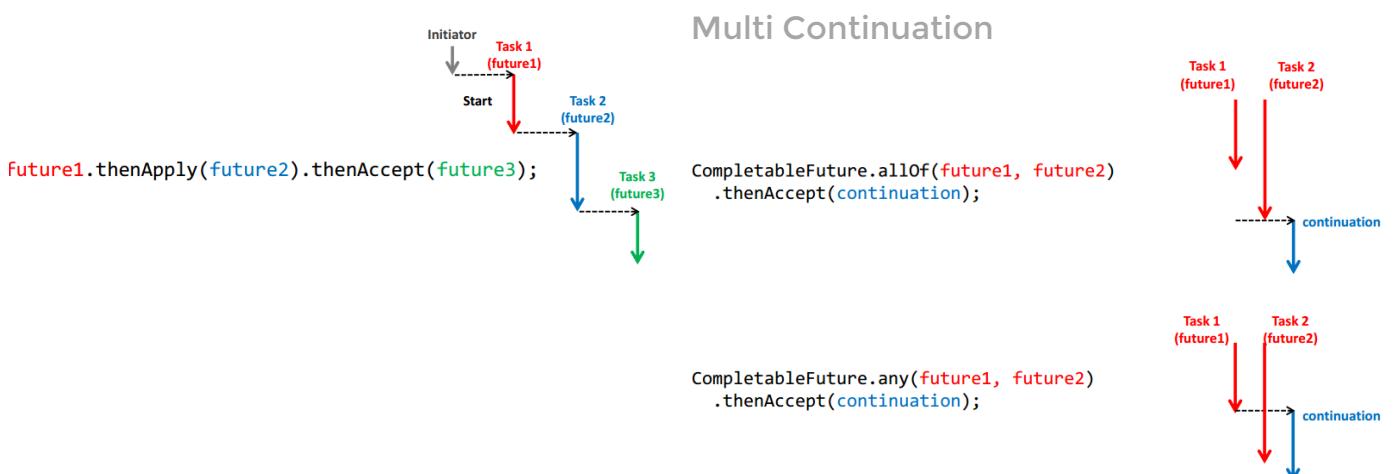
Findet durch einen beliebigen Thread statt. Initiator, wenn das Future bereits ein Resultat hat.

thenAccept() für Handler ohne Rückgabe
thenApply() für Funktion mit Rückgabe

⇒ Continuation läuft potentiell in einem anderen Thread. Daher die Synchronisation beachten.

Continuation-Style Programming

Die Tasks verketten via deren CompletableFuture.



Task Parallel Library

Wir wechseln hier von Java zu .NET. Dabei kommt die Sprache C# ins Spiel mit dem .NET Framework. Als IDE werden wir Visual Studio verwenden. Die .NET Task Parallel Library ist ein moderner Work-Stealing Thread Pool.

Threading in .NET

.NET Threads

Es gibt keine Vererbung. Dies findet über Delegate bei Konstruktor statt. Eine Exception im Thread führt zu einem Abbruch des Programms.

```
Thread myThread = new Thread(() => {
    for (int i = 0; i < 100; i++) {
        Console.WriteLine("MyThread step {0}", i);
    }
});
myThread.Start();
...
myThread.Join();
```

```
bool finished = false;
Thread myThread = new Thread(() => {
    ...
    finished = true;
});
```



Prädestiniert für Data Races
sogar auf lokalen Variablen

C# Lambdas

Die Syntax ist fast genau gleich wie in Java 8. Das Lambda kann dabei auf die umgebende Variable zugreifen. Auch schreibend.

- (Parameterliste) => { Statement Sequence }
- (Parameterliste) => Expression

Monitor in .NET

```
class BankAccount {
    private decimal balance;
    private object syncObject = new object();

    public void Withdraw(decimal amount) {
        lock(syncObject) {
            while (amount > balance) {
                Monitor.Wait(syncObject);
            }
            balance -= amount;
        }
    }

    public void Deposit(decimal amount) {
        lock(syncObject) {
            balance += amount;
            Monitor.PulseAll(syncObject);
        }
    }
}
```

Analog zu
synchronized Statement

Monitor auf
Hilfsobjekt als
Best Practice

Schlaufe auch
notwendig

Analog zu
notifyAll()

Implementiert ist es als eine FIFO Warteschlange. Das Pulse informiert den längst Wartenden.

Das Wait findet in einer Schleife statt. Es kommt da zwar zu keinem Spurious Wakeup, das Problem des Überholens (Singal and Continue) besteht aber weiterhin.

Das PulseAll() funktioniert analog zum Java Monitor und wird bei mehreren Bedingungen oder Threads eingesetzt.

Die Synchronisation nimmt man am besten mit einem Hilfsobjekt vor. Dies ist aber nicht zwingend nötig. Es gibt Gründe dafür und dagegen.

:NET Synchronisationsprimitiven

Im Gegensatz zu Java fehlt in .NET das Fairness-Flag und es gibt keine Lock & Conditions. Zusätzlich gibt es aber ReadWriteLockSlim für einen UpgradeableRead/Write, die Semaphoren sind auch auf OS-Stufe nutzbar, eine Mutex (binäre Semaphore auf OS-Stufe) ist vorhanden und es gibt spezielle Manual/AutoResetEvents. Grundsätzlich sind aber die Collections nicht Thread-safe. Ausser jene aus System.Collections.Concurrent.

Task Parallel Library

Es ist ein Work Stealing Thread Pool und damit einer der modernsten Thread Pools seit .NET 4. Dabei gibt es verschiedene Abstraktionsstufen:

- Task Parallelization → Explizite Tasks starten und warten
- Data Parallelization → Parallelle Statements und Queries
- Asynchrone Programmierung → mit Continuation Style

Task Parallelität (Task Parallelization)

Start and Wait

```
Task task = Task.Run(() => {
    // task implementation
});
// perform other activity
task.Wait();
```

Blockiert, bis Task beendet ist

Task mit einer Rückgabe

Rückgabetyp

```
Task<int> task = Task.Run(() => {
    int total = ... // some calculation
    return total;
});
...
Console.WriteLine(task.Result);
```

Blockiert bis Task Ende und liefert dann Resultat

```
for (int i = 0; i < 100; i++) {
    Task.Run(() => {
        // use i as input
        Console.WriteLine("Working with " + i);
    });
}
```

Im linken Beispiel treten Data Races auf. Denn es kommt darauf an, wann die Tasks ausgeführt werden. Normal werden alle Tasks so ausgeführt, dass i bereits bei 100 ist und somit überall 100 ausgegeben wird. Wenn man etwas warten, ist die Verzahnung etwas besser aber sicher immer noch nicht sehr gut.

Geschachtelte Threads

Die Tasks können auch Sub-Tasks starten und/oder abwarten. Dafür ist kein spezieller ForkJoinTask

nötig. Thread Injection

Der TPL fügt zur Laufzeit weitere neue Worker Threads hinzu. Und zwar nach dem Hill Climbing Algorithmus. (Er misst den Durchsatz und variiert dabei mit der Anzahl der Worker Threads). Es kann

somit zu keinem Deadlock bei den Task-Abhängigkeiten kommen (es kommen ja immer weitere Threads dazu). Es ist aber ineffizienz, da es nicht dafür gemacht wurde. Wird

ThreadPool.SetMaxThreads() eingesetzt sind Deadlocks natürlich wieder möglich.

Datenparallelität (Data Parallelization)

Es gibt zwei verschiedene Arten von Datenparallelität in TPL. Einerseits die „Parallel Statements“ und andererseits den „Parallel Loop“. Parallel Statements sind unabhängige Statements und Parallel Loops sind unabhängige Schlaufen-Bodies.

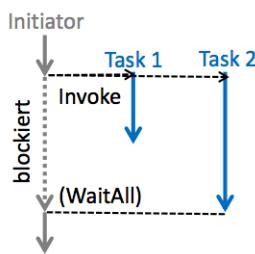
```
void MergeSort(l, r) {
    long m = (l + r)/2;
    MergeSort(l, m);
    MergeSort(m, r);
    Merge(l, m, r);
}
```

```
void Convert(IList list) {
    foreach (File file in list) {
        Convert(file);
    }
}
```

Parallele Statements

Dabei werden eine Menge von Statements potentiell parallel ausgeführt. Sie werden als Tasks gestartet und setzen am Ende des Tasks eine Barriere voraus, damit gewartet werden kann.

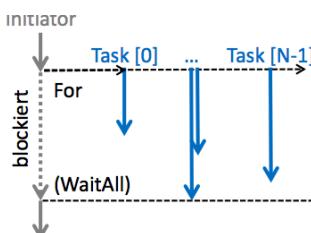
```
Parallel.Invoke(  
    () => MergeSort(l, m),  
    () => MergeSort(m, r)  
)
```



Parallele Loops

Schlaufen-Bodies die potentiell parallel ausführbar sind. Dabei findet eine Gruppierung der Bodies in den Tasks statt. Am Ende findet eine Barriere dieser Tasks statt.

```
Parallel.ForEach(list,  
    file => Convert(file)  
)
```



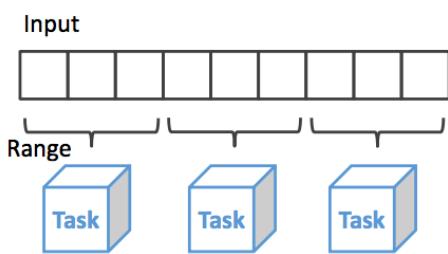
Neben dem Parallel ForEach gibt es auch das Parallel For.

```
for (i = 0; i < array.Length; i++) {  
    DoComputation(array[i]);  
}
```

Falls Iterationen unabhängig sind

```
Parallel.For(0, array.Length,  
    i => DoComputation(array[i]))  
)
```

Partitionierung



Bei Schlaufen mit vielen sehr kurzen Bodies ist es ineffizient jede Iteration als parallelen Tasks auszuführen. Die TPL gruppiert automatisch mehrere Bodies zu einem Task.

Die Aufteilung findet gemäss den verfügbaren Worker Threads statt. Die Partitionierungen sind je nachdem unterschiedlich.



Eine explizite Partitionierung ist ebenfalls möglich. Dies hat den Vorteil, dass weniger Body-Delegates nötig sind, es muss aber eine künstliche Unterschlaufe hinzugefügt werden.

```
Range Partitioner  
Parallel.ForEach(Partitioner.Create(0, array.Length),  
    (range, _) => {  
        for (int i = range.Item1; i < range.Item2; i++) {  
            DoCalculation(array[i]);  
        }  
    });
```

Inklusive untere Grenze

Exklusive obere Grenze

Parallele Programmierung ParProg

Aber Achtung. Dabei können schnell fehlerhafte Programme entstehen, welche Race Conditions enthalten. Dies zeigt untenstehendes Beispiel sehr gut.

```
long totalWords = 0;
Parallel.ForEach(list, file => {
    totalWords += CountWords(file);
});
```

Race Condition



```
long totalWords = 0;
Parallel.ForEach(list, file => {
    int subTotal = CountWords(file);
    lock(someLockObj) {
        totalWords += subTotal;
    }
});
```

Parallel LINQ (PLINQ)

Es ist die Parallelisierung von Language-Integrated Query. LINQ selbst ist eine SQL-analoge Anfragesprache auf dem Objektmodell. PLINQ dient zur parallelen Verarbeitung von Collection Operationen. Es ist das Pendant zum Java 8 Stream API.

LINQ Beispiele mit Methoden (sequentiell)

ISBN von allen Büchern zum Titel Concurrency

```
bookCollection.
    Where(book => book.Title.Contains("Concurrency"))
    Select(book => book.ISBN)
```

Jede Zahl einer Liste auf bool abbilden (true ⇔ Primzahl)

```
inputList.
    Select(number => IsPrime(number))
```

Eingebettete C# LINQ Syntax (sequentiell)

ISBN von allen Büchern zum Titel Concurrency

```
from book in bookCollection
    where book.Title.Contains("Concurrency")
    select book.ISBN
```

Jede Zahl einer Liste auf bool abbilden (true ⇔ Primzahl)

```
from number in inputList
select IsPrime(number)
```

Parallele LINQ Beispiele

ISBN von allen Büchern zum Titel Concurrency

```
from book in bookCollection.AsParallel()
    where book.Title.Contains("Concurrency")
    select book.ISBN
```

Beliebige Reihenfolge als Resultat

Jede Zahl einer Liste auf bool abbilden (true ⇔ Primzahl)

```
from number in inputList.AsParallel().AsOrdered()
select IsPrime(number)
```

Reihenfolge erhalten (Performance-Nachteil)

Zum Vergleich die Java 8 Stream API

ISBN von allen Büchern zum Titel Concurrency

```
bookCollection.parallelStream().unordered().
    filter(book -> book.getTitle().contains("Concurrency")).
    map(book -> book.getISBN());
```

Explizit unordered erlauben

Jede Zahl einer Liste auf bool abbilden (true ⇔ Primzahl)

```
inputList.parallelStream().
    map(number -> isPrime(number));
```

Reihenfolge bleibt bei Liste ohne Weiteres erhalten

LINQ ist durch funktionales/deskriptives Programmierparadigma inspiriert und müsste eigentlich frei von Seiteneffekten sein. Seiteneffekte per Kriterium (Where, Select) ist aber möglich, da eine beliebige verzahnte/parallele Ausführung möglich ist. Daher Race Conditions, Deadlocks etc.

```
from book in bookCollection.AsParallel()
    where cache.Contains(book)
    select book;
```

Versteckter Seiteneffekt?

cachedItems.add(book)

Parallele Programmierung ParProg

Wie im folgenden Beispiel kann man PLINQ auswerten und in einer Pipeline die Resultate **fortlaufend** verarbeiten. Die Analogie im Java Stream 8 API ist `stream.forEach(b ->....)`.

```
ParallelQuery<Book> query =
    from book in bookCollection.AsParallel()
    where book.Title.Contains("Concurrency")
    select book;

query.ForAll(b => {
    if (Interesting (b)) {
        Read(b);
    }
});
```

PLINQ nebenläufig zur Iteration auswerten

Java 8 Stream API Analogon:
`stream.forEach(b -> ...);`

Asynchrone Programmierung mit TPL

Dies Tasks dazu können in TPL lanciert werden.

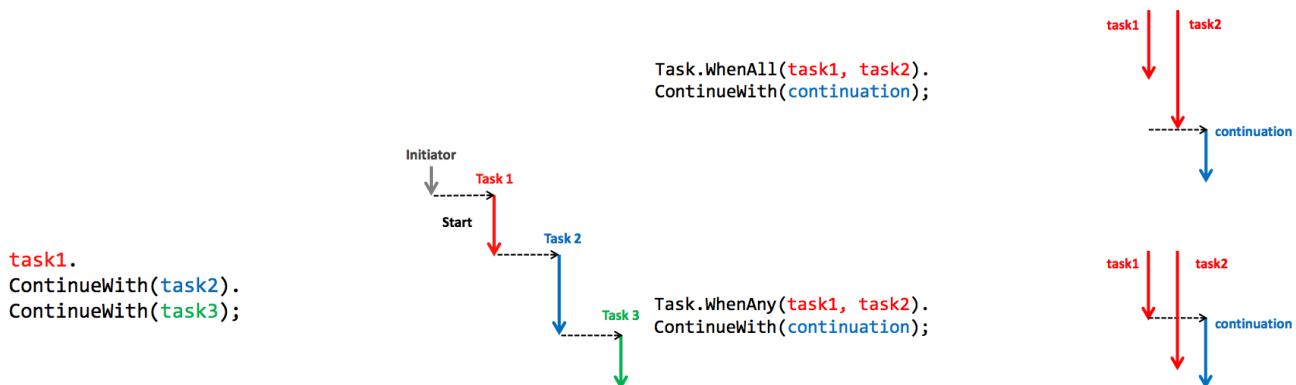
Task repräsentiert asynchronen Aufruf

```
...
var task = Task.Run(
    LongOperation
);
// perform other work
int result = task.Result;
```

Zugriff ähnlich wie mit Future

Task Continuations

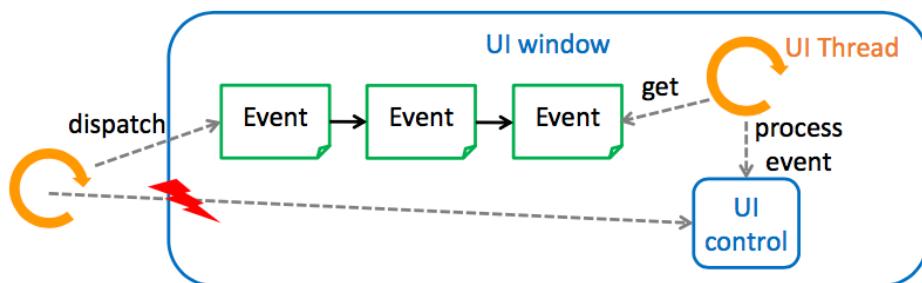
Damit kann ein Task definiert werden, der automatisch nach dem Ende eines anderen startet. Wie in Java bei den CompletableFutures. Zudem ist auch hier Multi-Continuation möglich.



HSR Parallel Checker

Es ist ein Programm zur Erkennung von Nebenläufigkeitsfehlern in C#. Dazu zählen Data Races und Deadlocks. Es findet eine statische Analyse in Visual Studio während dem Coding statt. Es gibt schnelle und präzise Warnungen, dafür werden eventuell nicht alle Fehler angezeigt. Weitere Infos und eine Installationsanleitung ist auf der Projektwebseite vorhanden.

<http://parallel-checker.com>



Die GUI Frameworks erlauben nur Single-Threading. Es darf nur ein spezielle UI-Thread auf die UI-Komponenten zugreifen. Der UI Thread holt mittels einem Loop die Ereignisse (Events) aus einer Queue und verarbeitet diese im Anschluss auf die UI Controls. Der Grund dafür das diese Frameworks nur im Single Thread Modell verfügbar, ist da sonst jede Komponente synchronisiert werden muss. Ein solches Locking an allen Komponenten und Methoden ist relativ teuer (Synchronisationskosten) und es gibt vermehrtes Deadlock-Risiko. Vor allem bei geschachtelten zyklischen Aufrufen im MVC Umfeld.

Java AWT/Swing/SWT

Der Java UI Thread wird auch Event Dispatching Thread (EDT) genannt. Er arbeitet eine Event Queue ab. Diese Events beschreiben ein Neuzeichnen/Kopfdurck etc. Es findet ein Run to Completion pro Event-Behandlung statt. Es führt die Ereignis-Code `paint()` und `update()` aus, welche überschreibbar sind.

GUI Implikationen

Es sollten also keine langen Operationen in den UI Event stattfinden. Sonst blockiert das UI. Dies betrifft Listener, `paint()` und `update()`. Zudem darf kein Zugriff auf die UI Elemente durch fremde Threads möglich sein. Sonst kommt es zu Race Conditions. In AWT/Swing bleibt dies eventuell unentdeckt. In SWT, Android und :NET gibt es eine „Invalid Thread Access“ Exceptions.

Swing/AWT und Multi-Threading

Swing ist nicht Thread-Safe. Es findet keine saubere Behandlung statt, wenn anderer Thread auf UI-Komponenten zugreift und kann daher irgendeine Inkonsistenz oder Exception verursachen. Aufgrund des Thread Confinement darf aber eigentlich nur der UI Thread auf die UI Komponenten zugreifen.

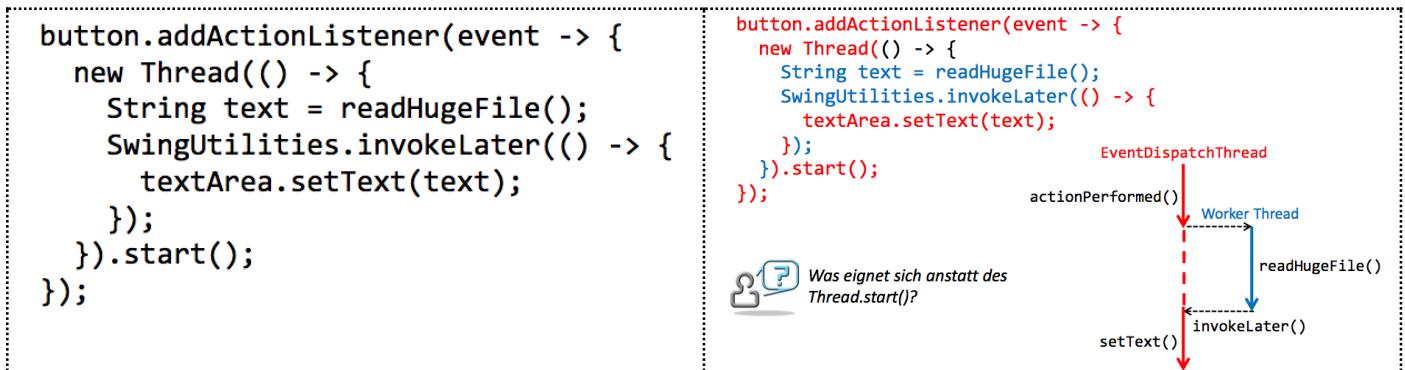
Swing - Dispatching an UI Thread

Man delegiert hier den Komponentenzugriff an den UI Thread und reiht die Aufgabe in die Event Queue ein. Der UI Thread führt die Ausgabe später dann aus.

Für diesen Prozess wird die Klasse `SwingUtilities` benutzt. **Static void invokeLater(Runnable doRun)** ist asynchron, während **static void invokeAndWait(Runnable doRun)** synchron ist.

UI Dispatching Beispiel

Der UI Thread startet hier einen neuen, separaten Thread, wo das File geladen wird. Im UI Thread wird dann wieder der setText ausgeführt.



Sauberer Swing QUI Setup

Dazu müsste pack() und setVisible() in einem UI Thread ausgeführt werden.

```
public static void main(String args[]) {
    final JFrame frame = new JFrame("My Frame");
    JButton button = new JButton();
    button.setText("Click");
    frame.getContentPane().add(button, ...);
    frame.setSize(200, 100);
    // ...
    SwingUtilities.invokeLater(() -> {
        frame.pack();
        frame.setVisible(true);
    });
}
```

Swing Background Worker

Ist eine Hilfsklasse für Hintergrundarbeiten (wo nicht so umständlich mit Lambdas gearbeitet werden muss. Die zeitaufwendige Operation **doInBackground()** wird als Task im Thread Pool abgearbeitet (separater Thread), während die UI-Zugriffe **done()** durch den EventDispatchThread geführt werden.

```
public abstract class SwingWorker<Result, Temp> {
    protected abstract Result doInBackground();
    protected void done();
}
```

läuft in separatem Thread
(keine UI-Zugriffe)

Durch UI Thread
(UI Zugriffe)

Beispiel

```
class BackgroundCalculator extends SwingWorker<Integer, Void> {
    @Override
    public Integer doInBackground() {
        return longComputation();
    }
    @Override
    protected void done() {
        try {
            Integer result = get();
            label.setText("Result: " + result);
        } catch (InterruptedException | ExecutionException e) {
            ...
        }
    }
    ...
}
```

Resultat von
doInBackground()

Resultat von
Background

Keine Zwischen-
Resultate

Separater Thread

UI Thread

new BackgroundCalculator().execute();

Diese werfen eine Exception bei einem falschen Thread-Zugriff auf das GUI. In SWT gibt es ein Pendant zu SwingUtilites, welches **widget.getDisplay().asnycExec(Runnable)** heisst.

In Android gibt es verschiedene Dispatch-Möglichkeiten. Einerseits post(Runnable) auf einer UI-Komponente aufrufen, AsyncTask implementieren und benutzen oder Activity.runOnUiThread(Runnable).

.NET WPF/WinForms

Gleiches Prinzip wie bei Java mit einem Single Thread Apartment (Confinement). Der UI Thread ist der Aufrufer von Application.Run(), welches bei WPF implizit der Main-Thread ist. Ein UI Dispatching kann bei WPF durch **control.Dispatcher.BeginInvoke(delegate)** erreicht werden, unter WinForm ist dies mit **control.BeginInvoke(delegate)** möglich. BeginInvoke ist asynchron, Invoke ist synchron.

Beispiel (analog zu Java)

```
public class MainWindow : Window {
    private void startCalculationButton_Click(...) {
        calculationResultLabel.Content = "(computing)";
        int number;
        if (int.TryParse(numberTextBox.Text, out number)) {
            Task.Factory.StartNew(() => {
                int result = LongCalculation(number);
                Dispatcher.BeginInvoke(new ThreadStart(() => {
                    resultLabel.Content = result;
                }));
            });
        }
    }
}
```

Unleserlich

Zerstückelung in mehrere UI-Events statt.

Klassisch führt dies zu einer Zerstückelung der Logik. Es kommt zu einer von Dispatches zwischen UI Thread und den fremden Threads. Die Hilfsklasse BackgroundWorker kann da auch eingesetzt werden.

Leserlicher Code ist mit C# async/await möglich. Die Logik ist in einem Guss (also eine Methode). Hinter den Kulissen findet dann die

C# async/await

Async Methode

```
public async Task<int> LongOperationAsync() { ... }
```

```
...
Task<int> task = LongOperationAsync();
OtherWork();
int result = await task;
```

Warte auf Beendigung
der Async Methode

Schlüsselwort **async** (Methode)

Der Aufrufer ist nicht zwingend während der gesamten Ausführung der async Methode blockiert.

Schlüsselwort **await** (Tasks)

Wartet auf das Ende eines TPL Tasks. Ein Resultat des Tasks wird geliefert, falls eine Rückgabe vorhanden ist.

Rückgabetypen

Void „Fire-and-Forget“

Task Keine Rückgabe, erlaubt das Warten auf ein Ende

Task<T> Für Methode mit einem Rückgabetypr T

Es gibt keine ref oder out – Parameter.

Spezielle Regeln

Async Methode

Die Methode muss await enthalten. Ansonsten kommt es zu einer Compiler Warnung.

Await Anweisung

Diese Anweisung ist nur in einer async Methode erlaubt. Sonst kommt es ebenfalls zu einem Compiler-Fehler.

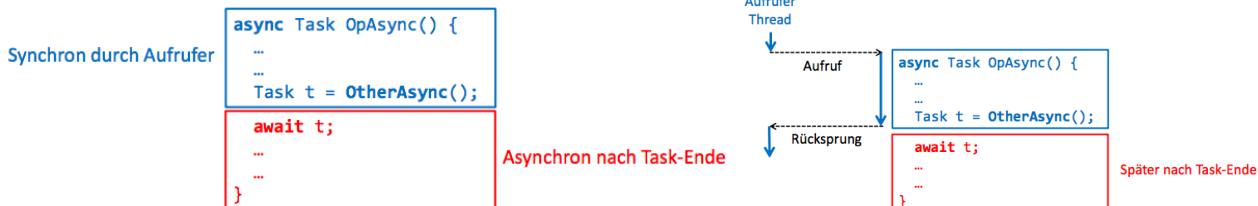
Der Grund für diese Regeln, ist das spezielle Ausführungsmodell.

Ausführungsmodell

Die Async-Methode läuft teilweise synchron, teilweise asynchron. Genauer gesagt für der Aufrufer die Methode solange synchron aus, bis ein blockierendes await anliegt. Danach läuft die Methode asynchron.

```
async Task<int> GetSiteLengthAsync(string url) {
    HttpClient client = new HttpClient();
    Task<string> task = client.GetStringAsync(url);
    string site1 = await task;
    return site1.Length;
}
```

Mechanismus & Methodenaufruf



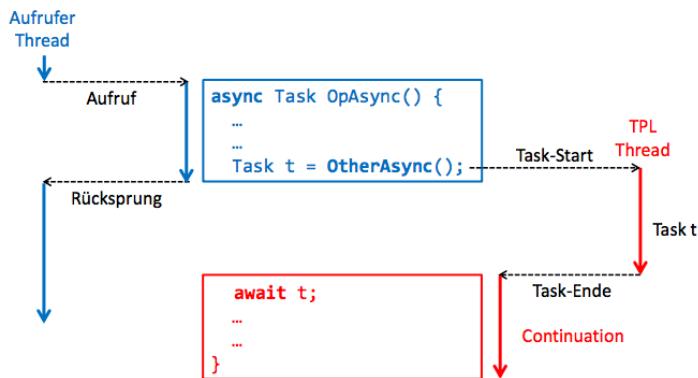
Der Compiler zerlegt die Methode in Abschnitte. Der erste Abschnitt vor dem Await (Synchron durch den Aufrufer) und der Abschnitt nach dem Await, welcher später nach dem Task-Ende läuft (Continuation).

Parallele Programmierung ParProg

Verschiedene Ausführungen

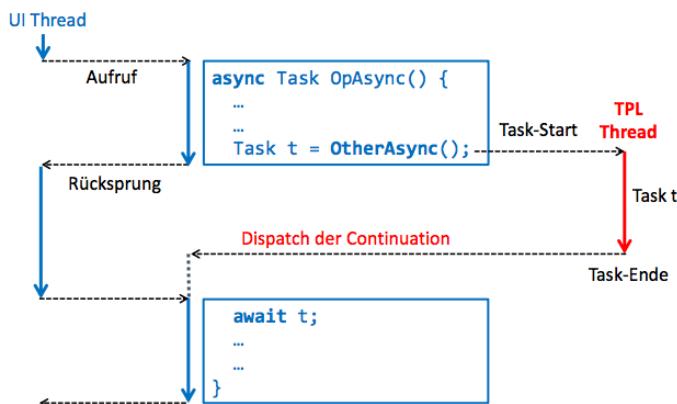
Fall 1 – Kein UI Thread

Der Aufrufer ist ein “normaler” Thread. Dies ist der meiste Fall (Console, TPL, etc.). Der Abschnitt wird durch den Thread des erwartenden Tasks ausgeführt.



Fall 2 – UI Thread

Der Aufrufer ist ein UI-Thread. Der Abschnitt wird an das UI dispatchet. Dies wird dann als Event vom UI-Thread ausgeführt.

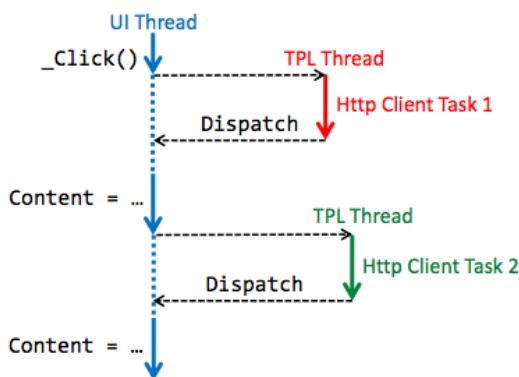


async/await Sequenz

```

async void startDownload_Click(...) {
    HttpClient client = new HttpClient();
    foreach (var url in collection) {
        var data = await client.GetStringAsync(url);
        textArea.Content += data;
    }
}
  
```

API bietet diverse
async Methoden



Häufige Fehler

Async Methoden ist nicht per se asynchron. Dies führt dazu das der Aufrufer während der gesamten Ausführung blockiert wird. Als Workaround kann man die teure Operation explizit als Task ausführen.

Fehler

```
public async Task<bool> IsPrimeAsync(long number) {
    for (long i = 2; i <= Math.Sqrt(number); i++) {
        if (number % i == 0) { return false; }
    }
    return true;
}
```

Läuft vollständig synchron

Workaround

```
public async Task<bool> IsPrimeAsync(long number) {
    return await Task.Run(() => {
        for (long i = 2; i <= Math.Sqrt(number); i++) {
            if (number % i == 0) { return false; }
        }
        return true;
    });
}
```

Sonderheit

Es kommt zu einem Thread-Wechsel innerhalb des Methodenaufrufs. Daher muss man die partielle Nebenläufigkeit unbedingt beachten. **Ausgabe vom Code:** BEFORE 10 ... AFTER 14

```
public async Task DownloadAsync() {
    Console.WriteLine("BEFORE " + Thread.CurrentThread.ManagedThreadId);
    HttpClient client = new HttpClient();
    Task<string> task = client.GetStringAsync("...");  

    string result = await task;
    Console.WriteLine("AFTER " + Thread.CurrentThread.ManagedThreadId);
}
```

Design-Kritik

Für die eine Async-Methode gibt es zwei Aufruffälle, die zu beachten sind.

Viraler Effekt: Der `async/await`-Aufrufer wird auch `async`. Die führt zu unnötig komplexen Methodensignaturen mit Tasks und es entstehen Kosten wegen der Methodenersterzung.

Zudem sollte die synchrone/asynchrone Verwendung meist orthogonal sein. Der „Caller“ sollte entscheiden und nicht der „Callee“.

Empfehlungen

Primär ist der Einsatz im UI Layer sinnvoll. Man muss sich aber den Zerstückelungs-Mechanismus verinnerlichen. Sowieso ist der UI-Layer meist der Top-Layer der Architektur.

Im darunterliegenden Layer ist es weniger empfohlen. Es ist problematisch wenn die Methoden ad-hoc `async` werden (Nebenläufigkeit/Verzahnung kommt ins Spiel). Zudem kann der Aufrufer sowieso leicht zu asynchron wechseln mit `Task.Run()`.

Memory Models

Quiz aus der letzten Vorlesung

```
async void startDownload_Click(...) {
    HttpClient client = new HttpClient();
    foreach (var url in collection) {
        var data = await client.GetStringAsync(url);
        textArea.Content += data;
    }
}
```

InvalidOperationException
Collection was modified

Wie kann ein solches Problem auftreten?

Wenn man dem laufenden Download einen URL hinzufügen möchte, kommt es zu einer Exception. Da die Collection verändert wird,

während darauf gearbeitet wird → Die Liste vom foreach wird im Hintergrund verändert.

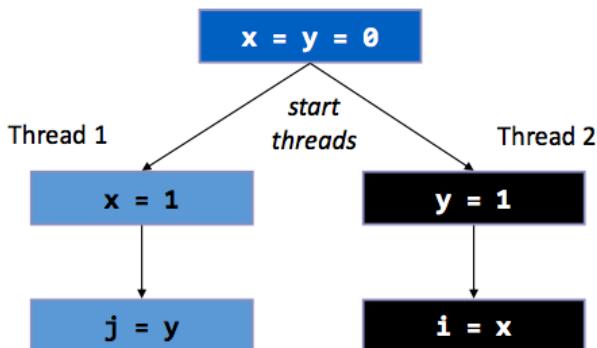
Quick Fix

Bei jedem Download die Liste (Daten) zwischenspeichern. Also eine Kopie der Liste anlegen. Oder man können den Button deaktivieren, sodass weitere Events nicht mehr möglich sind.

Lock-Freie Programmierung

Ist eine korrekte nebenläufige Interaktion ohne Locks, welche die Garantien des Speichermodells nutzt. Das Ziel dabei ist eine effiziente Synchronisation.

Einstiegsbeispiel



Das rechte Programm kann vier verschiedene Werte aufweisen (1,1) (0,1) (1,0) und sogar (0,0). Dies aufgrund von verschiedenen Thread-Verzahnung. Aus Sicht des schwarzen Threads ist gar eine Unordnung möglich, da keine Abhängigkeit zwischen den Statements besteht. Somit ist eben der Fall 0,0 auch möglich.

Ursachen für Probleme

Weak Consistency

Die Speicherzugriffe werden in verschiedenen Reihenfolgen auf verschiedenen Threads gesehen (Sie können beliebig umrangiert werden). Ausnahme bildet natürlich die Synchronisationen und Speicherbarrieren.

Optimierungen

Der Compiler, das Laufzeitsystem und die CPU nehmen Optimierungen vor. Daher werden Instruktionen umgeordnet oder gar wegoptimiert.

Abschließend muss gesagt werden, dass keine sequentielle Konsistenz bei der Nebenläufigkeit besteht.

Java Memory Model

Minimale spezifizierte Garantien

- Atomicity (Unteilbarkeit)
- Visibility (Sichtbarkeit)
- Ordering (Reihenfolge)

Der Zugriff auf eine Variable (Lesen/Schreiben) ist atomar für

- Primitive Datentypen von bis 32 Bit (byte, int, short, float, char)
- Objekt-Referenzen
- Long und Double nur mit dem volatile Keyword atomar

Atomar heisst das von 12 zum Beispiele die 12 als Ganzes geschrieben wird und nicht aufgeteilt wird.

Aber Achtung. Unteilbarkeit heisst nicht nicht Sichtbarkeit. Nach Write sieht ein anderer Thread evtl. noch den alten Wert. Der Wert ist aber immer gültig. Entweder Alten oder Neuen (nur nicht nur ein Teil davon geschrieben).

Analysebeispiele

x = Nein
o = Ja

1. long l = -1; x
2. double d = 3.14; x
3. int i = 1; x da eigentlich int x = 0 und daher erst nachher x = 1 geschrieben wird
4. String s = "first"; x ==> wegen gleichem Grund von 3.
5. boolean b = true; x
- ...
6. i = (int)l; x x und schreibe sind sowieso nicht atomar.
7. l = 42; x
8. i = 7; o (nur die Zuweisung)
9. ++i; x (Lesen und Schreiben)
10. s = "second"; o (ja String wird schon vorher in Stringpool abgelegt)
11. d = d * i; x, da zweimal gelesen und einmal geschrieben.
12. s = null; o
13. b = false; o



Welche sind atomar?

Sichtbarkeitsproblematik

```
class Worker extends Thread {
    private boolean doRun = true;

    public void run() {
        while (doRun) {
            ...
        }
    }

    public void stopRequest() {
        doRun = false;
    }
}
```

Hier ist nicht klar, zu welchem Zeitpunkt er die Änderung mitkriegt.

Wie bereits erwähnt sehe ich die Änderungen eines anderen Threads eventuell nicht oder viel später. Dies hat mit Optimierungen zu tun, welche vorgenommen werden (z.B. hält VM Variablenwert im Register). Dies führt in diesem Fall zu einer Endlossscheife.

```
void run() {
    load doRun to reg1;
    while (reg1) {
        ...
    }
}
```



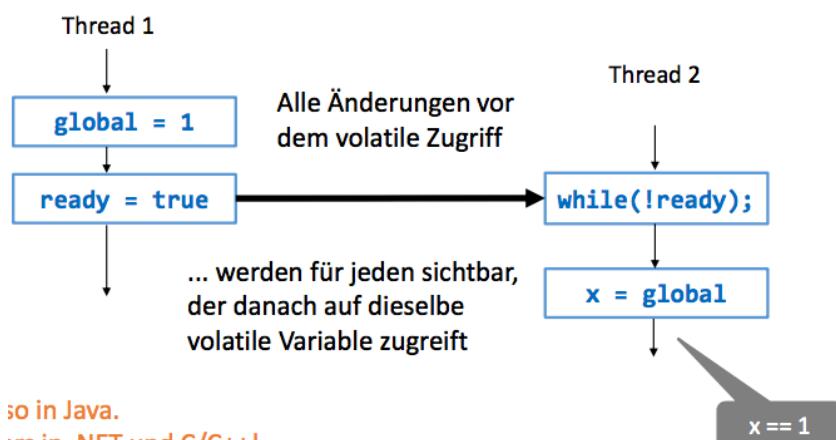
Endlosschlaufe

Die **Sichtbarkeit** (Man bekommt die Änderung mit) ist garantiert bei

- Locks Releases & Acquire
 - o Die Änderungen vor dem Release werden bei Acquire sichtbar.
- Volatile Variable
 - o Der Zugriff macht Änderungen anderer Zugreifer sichtbar.
- Initialisierung von final Variablen
 - o Nach dem Ende des Konstruktors
- Thread-Start und Join
 - o Ebenso bei Task Start und Ende

Sichtbarkeit mit Volatile in Java

```
int global = 0;
volatile boolean ready = false;
```



Alle Änderungen vor dem volatile Zugriff werden für jeden sichtbar, der danach auf dieselbe volatile Variable zugreift. Dies ist nur in Java so. In .NET, C und C++ ist dieses Verhalten nicht zu entdecken.

Analysebeispiele

1. <code>int x = 0; Semaphore s = new Semaphore(0);</code>	<code>Thread 1</code>	<code>x = 1;</code>	<code>Thread 2</code>	<code>s.acquire();</code>	ok
				<code>// x == 1?</code>	
2. <code>int x = 1;</code>			<code>Thread 2</code>	<code>// x == 1?</code>	nicht sichtbar da kein keyword
3. <code>final int x = 1;</code>			<code>Thread 2</code>	<code>// x == 1?</code>	ok, sofern nicht im Konstruktor
4. <code>volatile boolean b; int x;</code>	<code>Thread 1</code>	<code>x = 1;</code>	<code>Thread 2</code>	<code>while (b) { }</code>	ok, Folie von vorher
		<code>b = false;</code>		<code>// x == 1?</code>	



Wo ist die Sichtbarkeit von `x==1` garantiert?

Java Ordering Garantien

Innerhalb eines Threads kommt die „As-If-Serial“ Semantik statt. Das sequentielle Verhalten innerhalb des Threads bleibt also.

Zwischen den Threads bleibt die Reihenfolge nur erhalten für Synchronisationsbefehle und Zugriffe auf volatile Variablen.

Keine Umordnung findet über die Synchronisation oder die volatile Zugriffe hinweg statt (Memory Barriers / Memory Fences).

Analysebeispiele

1. int x, y; Ausgangslage x = 0; y = 1;	Optimierung? y = 1; x = 0	ja
2. volatile int x; int y; Ausgangslage x = 0; y = 1;	Optimierung? y = 1; x = 0	nein
3. int x, y; Ausgangslage x = 0; y = x;	Optimierung? y = x; x = 0	nein
4. int x, y; Ausgangslage x = 0; synchronized(this) { y = 1; }	Optimierung? y = 1; x = 0	nein, wie 2.



Welche Umordnungen sind möglich?

Java Volatile Keyword

Atomicity

Macht atomares Lesen und Schreiben auch für long und double möglich. Aber Achtung, andere Operationen wie i++ etc. sind nicht atomar.

Visibility

Die Änderungen werden an andere Zugreifende propagiert. Aber Achtung. Es findet kein Sperren wie bei den Locks statt.

Reordering

Keine Umordnung durch den Compiler / Laufzeitsystem / CPU. Nicht volatile Variablen werden evtl. umgeordnet.

Atomare Operationen

Diese haben kein Blockieren oder Warten auf Locks. Sie sind komplexer als nur atomare Lesen und Schreiben. Weitere Effizienz kann mit atomaren Instruktionen des Prozessors erreicht werden. Atomare Operationen garantieren auch Visibility und Ordering.

Spin-Lock**Fehlversuch**

```
public class SpinLock {
    private volatile boolean locked = false;

    public void acquire() {
        while (locked) {
            Thread.yield();
        }
        locked = true;
    }

    public void release() {
        locked = false;
    }
}
```



Nicht korrekt ohne atomares
Lesen & Schreiben

Müsste atomar sein

Korrekt mit atomaren Operationen

```
public class SpinLock {
    private AtomicBoolean locked = new AtomicBoolean(false);

    public void acquire() {
        while (locked.getAndSet(true)) {
            Thread.yield();
        }
    }

    public void release() {
        locked.set(false);
    }
}
```

Initialwert false

Lese alten Wert und setze
neuen Wert atomar
(Rückgabe = gelesener Wert)

Setze false und
mache sichtbar

Atomare Compare and Set

boolean compareAndSet
(boolean expect, boolean update)

- Setzt update, wenn alter Wert gleich expect ist (atomar)
- Retourniert true bei erfolgreichem Update

```
if (current == expect) {
    current = update;
    return true;
} else {
    return false;
}
```

atomar

Atomic Klassen

Es gibt Klassen für Boolean, Integer, Long und Referenzen. Und sogar auch für Array-Elemente.

Darauf sind diverse atomare Operationen wie addAndGet(), getAndAdd() möglich. Ab Java 8 sogar mit Lambda.

```
AtomicInteger counter = new AtomicInteger(0);
```

```
counter.updateAndGet(x -> x + 1);
```

Optimistische Synchronisation

Dies ist nötig, da zwischen einem if und dem ersten Statement noch der Thread gewechselt werden kann. Siehe Testatübung. Es wird aber nicht garantiert, dass niemand anders dazwischen geschrieben hat sondern nur, dass der Wert derselbe ist wie vorher (siehe ABA Problem).

```
do {
    oldValue = var.get();
    newValue = calculateChanges(oldValue);
} while (!var.compareAndSet(oldValue, newValue));
```

Lese aktuellen Wert

Schreibe, falls gelesener
Wert immer noch aktuell ist

ABA Problem

Ein anderer Thread kann nicht unbemerkt dazwischen schreiben. ABA ist nicht immer ein Problem (dies ist ganz abhängig von der Anwendung). Es gilt es aber zu beachten.

Thread 1

```
var.get();  
-> A
```

Variable

A

Thread 2

```
var.set(B);
```

Java 8 Update mit Lambda

```
var.updateAndGet(old -> calculateChanges(old))
```



Hinter den Kulissen

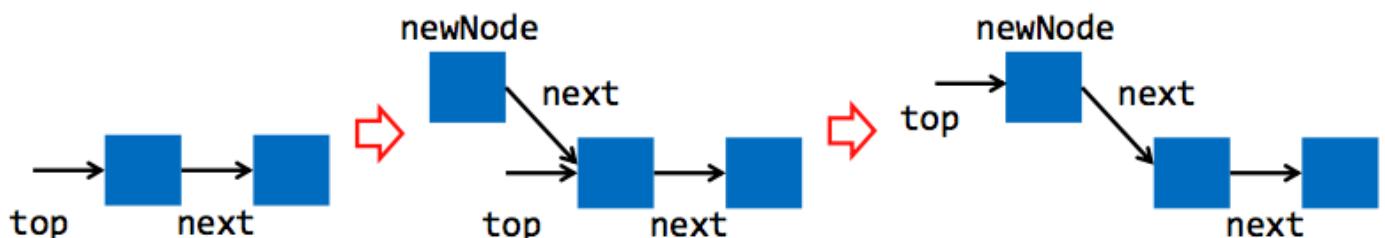
```
var.compareAndSet(A, C);
```

Originaler Wert
wieder vorhanden

ABA ist nicht immer ein Problem
(abhängig von Anwendung)

```
do {
    oldValue = var.get();
    newValue = calculateChanges(oldValue);
} while (!var.compareAndSet(oldValue, newValue));
```

```
AtomicReference<Node<T>> top = new AtomicReference<>();  
...  
void push(T value) {  
    Node<T> newNode = new Node<>(value);  
    Node<T> current;  
    do {  
        current = top.get();  
        newNode.setNext(current);  
    } while (!top.compareAndSet(current, newNode));  
}
```



Folgende vorgefertigte Lock-Freie Datenstrukturen sind in Java vorhanden:

- ConcurrentLinkedQueue<V>
- ConcurrentLinkedDeque<V>
- ConcurrentSkipListSet<V>
- ConcurrentHashMap<K, V>
- ConcurrentSkipListMap<K, V>

.NET Memory Model

Grundsätzlich ist die Unterstützung von .NET ähnlich wie in Java. Es gibt aber folgende Unterschiede zum Java Memory Model

- Bei der Atomarität: Long/Double nicht mit volatile atomar.
- Bei der Visibility: Nicht definiert. Wird implizit durch die Ordnung hergestellt.
- Beim Ordering: Volatile ist nur ein partieller Fence.(MemoryBarrier nötig)

Die atomaren Instruktionen sind in der Klasse Interlocked vorhanden. Atomic-Datentypen gibt es so in .NET aber nicht.

.NET Volatile Read / Write Fences

Volatile Read

Funktioniert nach der Acquire Semantik. Bleibt vor den nachfolgenden Zugriffen.

Volatile Write

Funktioniert nach der Release Semantik. Bleibt nach den vorherigen Zugriffen.



.NET Volatile Beispiel

```
volatile bool a = false, b = false;
```

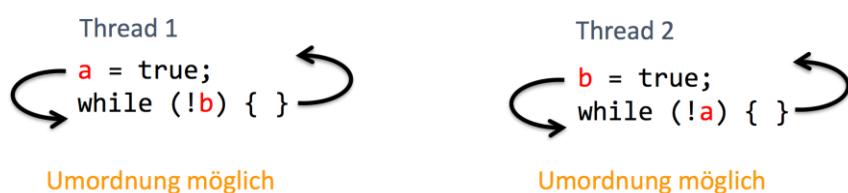
Thread 1

```
a = true;
while (!b) { }
```

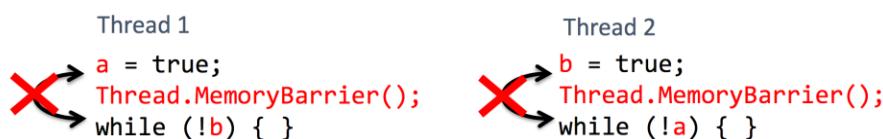
Thread 2

```
b = true;
while (!a) { }
```

Das Rendez-Vous funktioniert in diesem Fall nicht. Bei .NET reicht volatile nicht aus um das Umordnen zu unterbinden. Daher sind noch Umordnungen möglich. Somit ist das Programm inkorrekt.



Für eine Reparatur mit Full Sences muss Thread.MemoryBarrier() dazwischen eingesetzt werden um das Reordering zu unterbinden. Volatile verhindert zumdem Optimierungen im Register. Im ganzen nun eine garantierter Sichtbarkeit und keine Data Races.



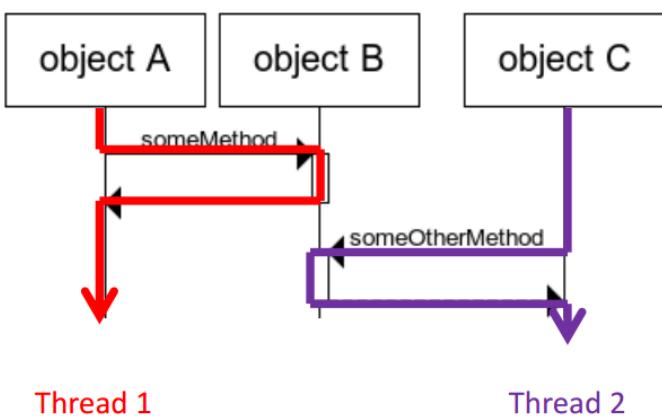
Actor Model

Quiz aus vorhergehendem Kapitel

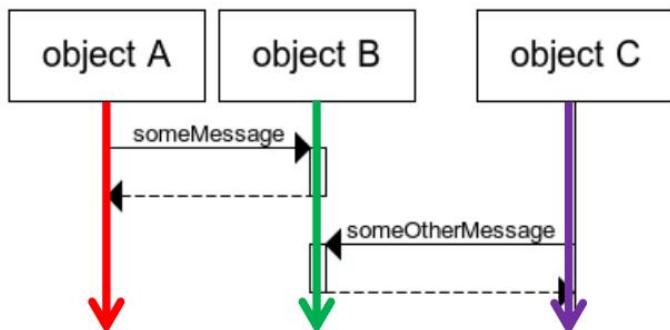
```
class LazyCreation {
    private volatile LargeObject instance;

    public LargeObject get() {
        if (instance == null) {
            synchronized (this) {
                if (instance == null) {
                    instance = new LargeObject();
                }
            }
        }
        return instance;
    }
}
```

 Wieso ist Double Checked Locking ohne volatile falsch?



Actor Modell und CSP



Historisches

Simula I

War die Geburt von OO in Norwegen. Hatten schon aktive Objekte (ohne Kommunikation, nur Prozeduraufrufe). Später mit Simula 67 leider nicht mehr (nur noch passive Objekte).

Actor Model

Theorie begründet von C. Hewitt in 1973. Gemäss der damaligen Definition umfasst ein Actor Processing, Storage und Communication. Prinzipiell gleiches Modell wie Actors. Nur hat der Actor keine Channels. Senden ist dort immer asynchron, keine garantierte Reihenfolge des Empfangs.

Hier kommt die Antwort auf die Frage hin.

Motivation

Herkömmliche Programmiersprachen sind nicht für Nebenläufigkeit entworfen (sind optimiert für sequentielle Ausführung, Nebenläufigkeit als Second Class Feature). Zudem ist dort der Speicher per Default nicht Thread-Safe. Korrekte nebenläufige Programme zu schreiben ist daher besonders schwierig.

Die Threads operieren auf Modell von passiven Objekten. Dies hat zur Konsequenz, dass nur ein beschränkter Grad an Nebenläufigkeit möglich ist. (Concurrency sehr selektiv, Maschinenorientierte Concurrency, viele unnötige blockierende Aufrufe). Des Weiteren besteht eine sehr grosse Fehleranfälligkeit (Race Conditions immanent). Zu Letzt ist es sehr schlecht verteilbar (Shared Memory Modell).

Das Actor Model ist ein substantiell anderes Programmierkonzept.

Aktive Objekte

Objekte haben ein nebenläufiges Innenleben

Kommunikation

Objekte senden und empfangen Nachrichten

Kein Shared Memory

Nur Austausch von Nachrichten über Kanäle

Modell

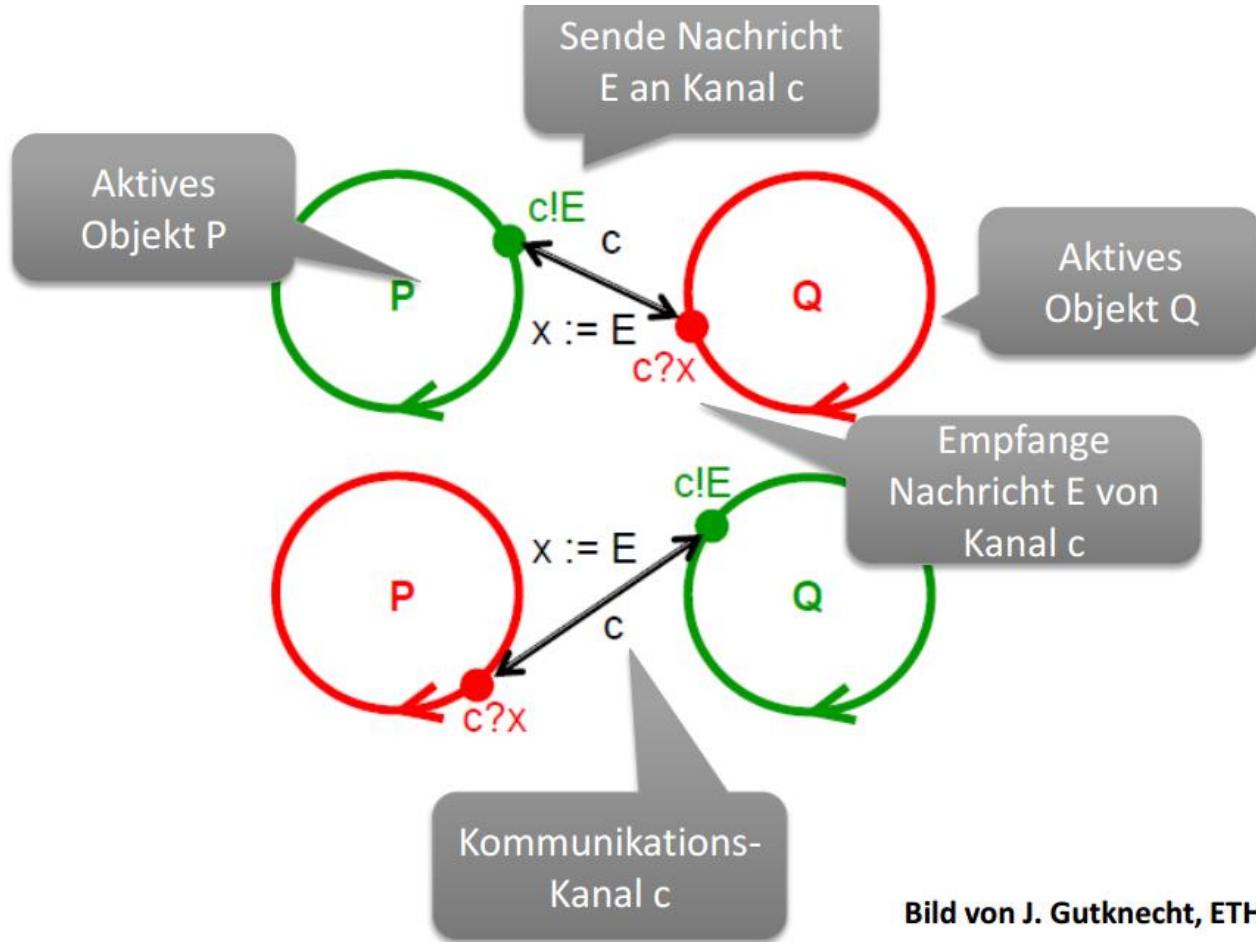
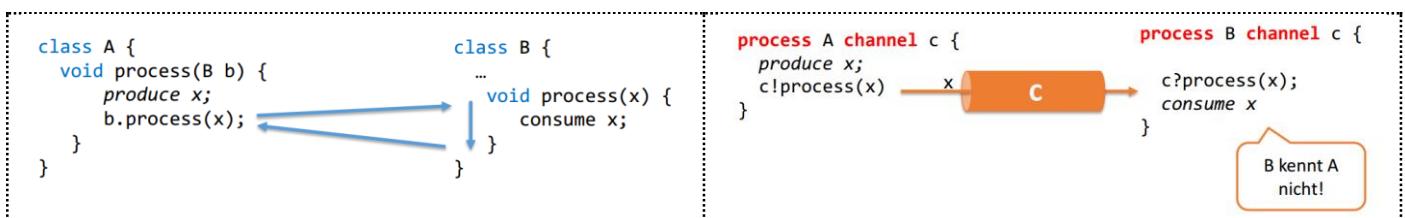


Bild von J. Gutknecht, ETH Zürich

Klassisch versus CSP



Vorteile – Actor und CSP

- Inhärente Nebenläufigkeit
 - o Alle Objekte (Actors) laufen nebenläufig
 - o Maschine kann Grad an Nebenläufigkeit ausnutzen
- Keine Race Conditions
 - o Kein Shared Memory
 - o Nachrichtenaustausch synchronisiert implizit
- Gute Verteilbarkeit
 - o Kein Shared Memory
 - o Nachrichtenaustausch für Netz prädestiniert

Parallele Programmierung ParProg

Actor Implementierungen

Erlang

Entwickelt von Joe Armstrong bei Ericsson für Telekommunikation (Telefonie-Switches). Eigenes Laufzeitsystem, Lightweight Threads.

Akka

Für die JVM (Java, Scala, JRuby, etc.), Meta-Programmiermodell auf «herkömmlicher Sprache»

Java Communicating Sequential Processes

JCSP von U Kent, veraltet.

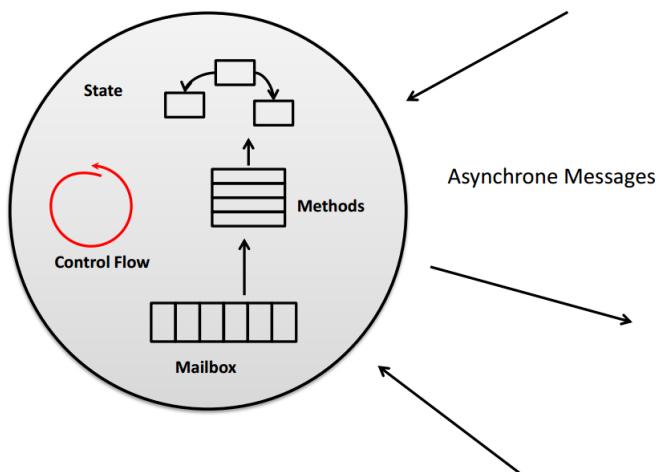
.NET

Concurrency and Coordination Runtime CCR (low-level), Orleans (Microsoft Research, für Xbox eingesetzt), Akka.NET Portierung des Akka-Sourcecodes nach C#.

Akka

Es ist implementiert in Scala als zusätzliches Java-API. Zur Entwicklung nebenläufiger, verteilter, fehlertoleranter und event-basierter Systeme. Neben Erlang wohl das verbreitetste Actor-Framework.

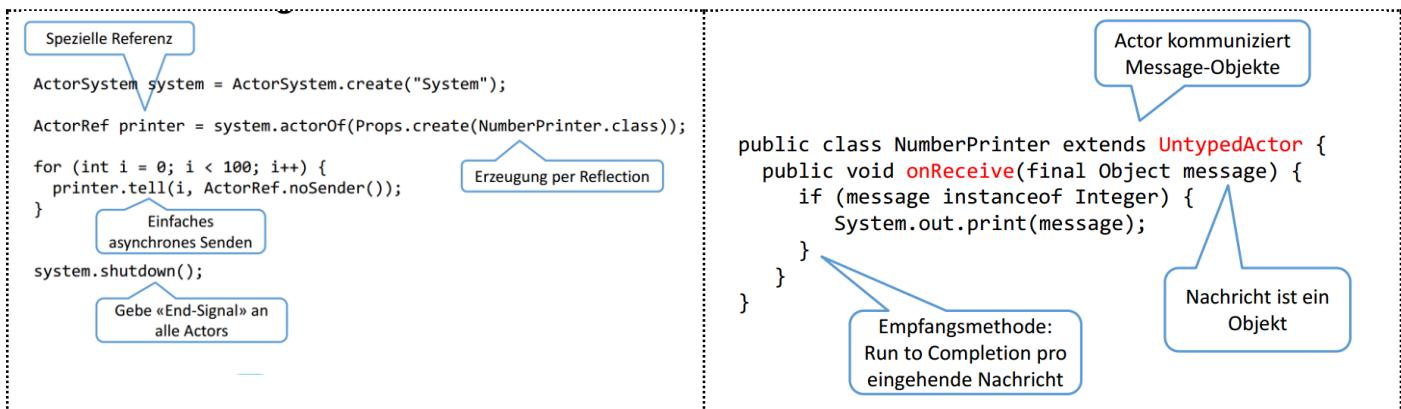
Konzept



Actor sind aktive Objekte und laufen konzeptionell nebenläufig zueinander. Es ist ein privater Zustand, aber aufpassen, dass per Java Referenzen kein Shared State entsteht. Pro Actor gibt es eine Mailbox, ein Buffer für alle Nachrichten, die zu ihm ankommen. Das Senden findet asynchron statt.

Empfangsverhalten

Es findet eine Reaktion auf eine ankommende Nachricht statt. Dabei wird spezielle Behandlungs-Methode ausgeführt. Effekte der Behandlung sind das ändern des privaten Zustandes, senden einer Nachricht oder das Erzeugen eines neuen Actors. Intern sequentiell, also nur eine Nachricht auf einmal bedienbar.

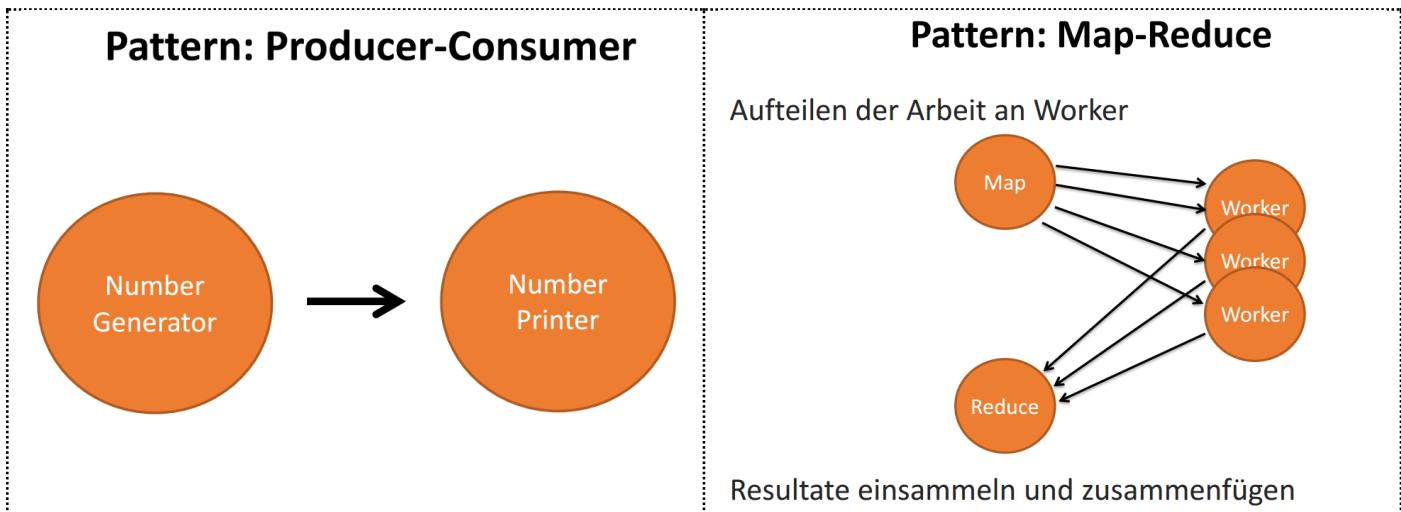
Beispiel**Actor Referenzen**

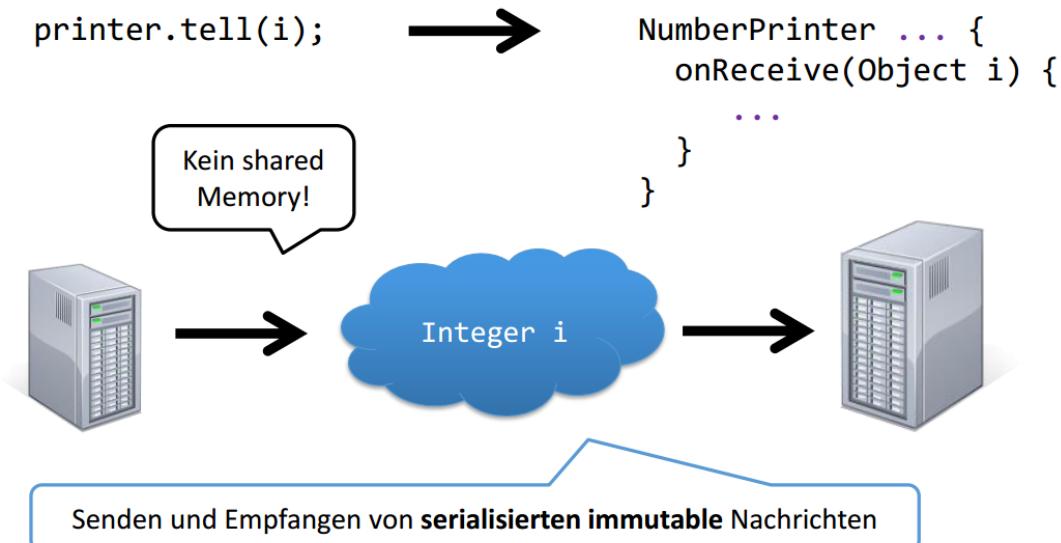
Die ActorRef ist die Adresse eines Actors. Der Actor ist bei Fehlverhalten neu startbar und behält seine Adresse. Es dient zur Entkoppelung von «Interface» und Instanz. Dies ist eine Vorbereitung für die Verteilung. Die ActorRef ist zudem immutable (also in Message verschickbar).

Die Actor Referenz verhindert zudem die Methodenaufrufe und Variablenzugriffe. Daher nur ein reiner Nachrichtenaustausch.

Beispiele für Actor Anwendungen

- Alternative zu Threads (Explizite Kommunikation statt Shared Memory)
- Transaction-Processing (Koordination über verschiedene Systeme)
- Backend für Service (REST, SOAP, Websockets)
- Kommunikations-Hub (Router, Load-Balancer)

Patterns



Verteilte Actors – Server

```
ActorSystem system = ActorSystem.create("System");
ActorRef printer = system.actorOf(Props.create(...), "printer");

application.conf
akka {
    actor {
        provider = "akka.remote.RemoteActorRefProvider"
    }
    remote {
        enabled-transports = ["akka.remote.netty.tcp"]
        netty.tcp {
            hostname = "server"
            port = 2552
        }
    }
}
```

Genau gleich

Verteilte Actors – Client

```
ActorSystem system = ActorSystem.create("producer");

ActorSelection printer = system.actorSelection(
    "akka.tcp://System@server:2552/user/printer");
printer.tell(123, ActorRef.noSender());
```

Actor Adresse

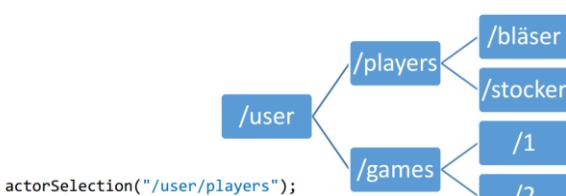
Remote Lookup findet mit system.actorSelection mittels einem URL statt.

ActorSelection ist leichtgewichtiger als ActorRef. Kann 0...n actor umfassen und kann zu einem ActorRef aufgelöst werden.

Die Remote Erzeugen findet mit system.actorOf(...) statt. Application.conf spezifiziert, wo der Actor erstellt wird.

Actor Hierarchies

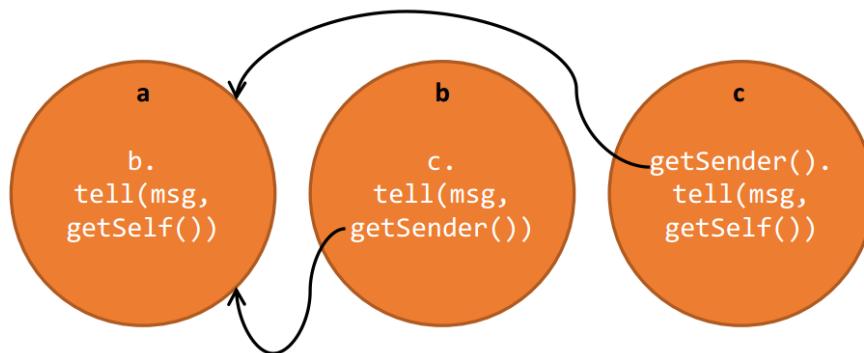
Hierarchien von Actors sind möglich. Es ist passen zum URL Adressierungsschema gelöst und lässt Supervision von Actors zu. Der Erzeuger ist der Parent. ActorSelection selektiert einen Teilbaum. Broadcasts sind möglich.



Senden

Sender

Mit tell(msg, sender) wird der Sender der Message mitgegeben. Dies ist nützlich bei Antwort an Sender-Actor. Typischerweise getSelf(), bzw. getSender() bei Forward (Methoden von Actor).



Synchrones Senden

Das Actor-Modell ist grundsätzlich asynchron. Synchrones Senden-Empfangen ist möglich mit Futures.

```
Future<Object> result = Patterns.ask(actorRef, msg, timeout);


Antwort ist Untyped


```

Messages

Müssen Serializable Classes sein und Immutable (Value Objects) haben (also das Attribut final, Collections in Collections.unmodifiableList wappen und keine Methoden die Seiteneffekte haben).

Typischerweise einfache Wrapper-Klassen mit Fokus auf die Attribute.

Es entsteht viel Schreibaufwand für Message-Klassen. Besser ist dies mit der Scala API möglich. (Links Java, Rechts Scala API).

```

public class Booking {
    final String name;
    public Booking(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    ...
}

class Booking(val name: String)
Resultiert in selbem Bytecode
class Confirmation(val id: Int)
class FullyBooked()
  
```

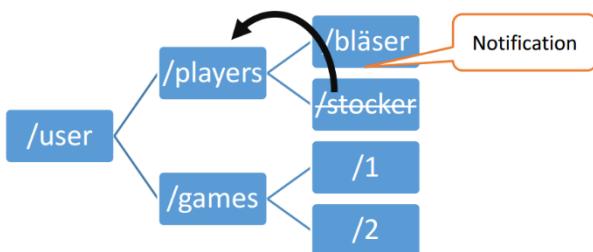
Laufzeitsystem

Akka verwendet Dispatchers zur Ausführung. Typischerweise ein Java Fork-Join Thread Pool. Nicht ein Thread pro Actor. Bei synchronem Send & Receive kommt es zu Warteabhängigkeiten zwischen Actors. Der Thread Pool hat keine Lösung (Deadlock bei fixer Anzahl), es muss Stack pro Actor instanziert werden. Thread pro Actor führt daher zu sehr vielen Threads.

Synchrones Send & Receiver wird daher nicht empfohlen.

Supervision

Actors können andere Actors überwachen. Bei Exception wird der Supervisor benachrichtigt. Parents überwachen per Default ihre Kinder.



Der Supervisor kann verschieden, je nach Fehler, reagieren.

Resume	Child macht weiter (behält interner Zustand)
Restart	Child wird neu gestartet (verliert Zustand)
Stop	Child wird nicht mehr ausgeführt
Escalate	Supervisor gibt auf und meldet selber seinem Supervisor einen Fehler

Nicht für Programmierfehler, sondern nur für externe Ursachen (Netzwerk, Files).

Der Parent von '/user' ist der Root Guardian. Zusätzlicher '/system' Actor für Logging, Shutdown ist vorhanden.

System Shutdown

Wann kann ein Actor System beendet werden?

- Alle Mailboxen leer? Actor könnte noch beschäftigt sein

Applikation muss Actors selber stoppen

```

nmer
skursiv
...
getContext().stop(actorRef);   Stoppt nach Bearbeitung der
                               aktuellen Message
getContext().stop(getSelf()); 
getContext().system().terminate(); Stoppt bei Behandlung
                               der Poison Pill
actor.tell(PoisonPill.getInstance(), sender);
victim.tell(Kill.getInstance(), sender);
...
  
```

Startet Supervision Behandlung!

Schwächen

- Protokoll des Actor nur implizit vorhanden
 - o Formales Protokoll fehlt
 - o Grundsätzliches Problem bei Actors im Gegensatz zu CSP
- Akka hat keine Typsicherheit
 - o Welche Message kann ein Actor behandeln?
- Akka: Diskrepanz JVM und Actor Model
 - o Shared Memory und Mutability
 - o Braucht spezielle Referenzen zwischen Actors
 - o Leicht verletzbare Regeln → Laufzeitfehler

GPU Parallelisierung 1

Die Graphic Processor Units (GPUs) sind Co-Prozessoren des Systems mit mehreren tausenden Cores.

Die CPUs bieten wenige Cores (meist 4 – 20 Cores) und sind allgemein schnelle Prozessoren.

GPU's bieten sehr viele Cores (512 – 5760 Cores) und sind spezifische langsamere Prozessoren.

Das Ziel ist es hier die GPU Many Cores für die Parallelisierung zu nutzen.

Quiz – Letztes Kapitel

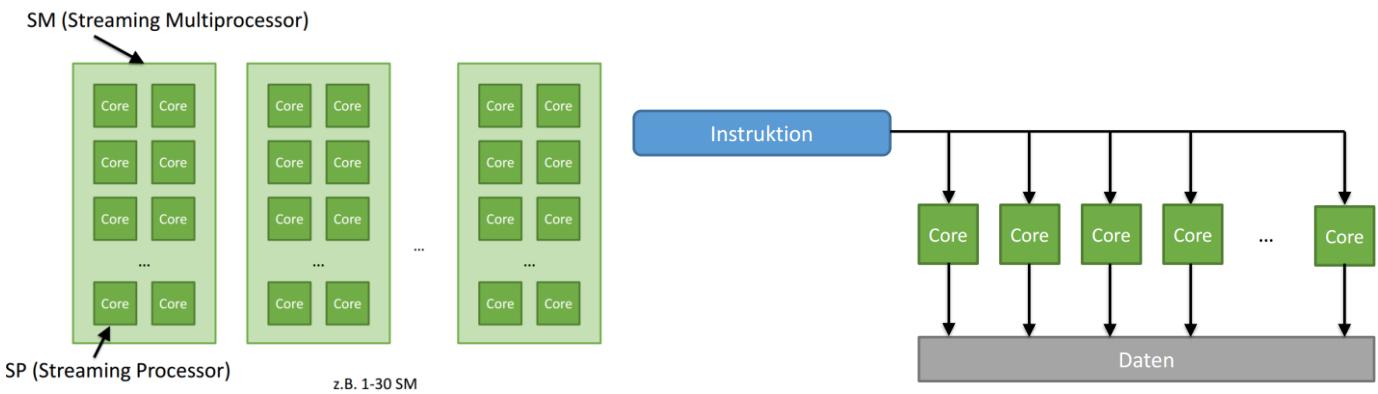
Welche Vorteile bietet das Actor Model für die Parallelisierung? Keine Data Races mehr.

GPU Co-Prozessor Architektur

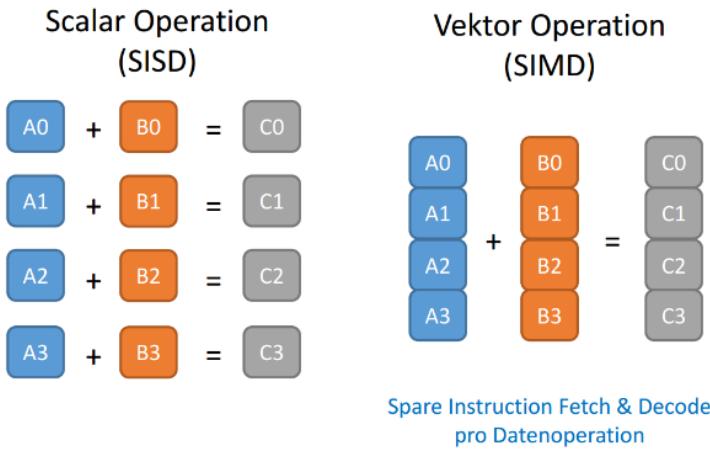
CPU versus GPU

GPUs verwenden mehr Transistoren für Recheneinheiten als die CPUs.

GPU Aufbau



SIMD



Streaming Multiprocessor ist prinzipiell SIMD (Single Instruction Multiple Data). Die Cores führen dieselbe Instruktion auf unterschiedlichen Daten/Speicherstellen aus. SMID bedeutet Vektorparallelität.

GPU Parallelisierungspotential
Cores innerhalb SM nur mit Vektor-Parallelisierung effizient nutzbar. Alle Cores führen dann die gleichen Instruktionen aus. Einzelne Cores können die Instruktionen auch nicht ausführen.

GPUs vs. CPUs

GPU – Video Gaming	CPU – General Purpose
<ul style="list-style-type: none"> - Extrem hohe Datenparallelität - Wenig Verzweigungen - Kein beliebiges Warten bei Parallelität - Einfache, aber viele Cores - Kleine Caches pro Core 	<ul style="list-style-type: none"> - Niedrige Datenparallelität - Viel Verzweigungen - Beliebige Thread-Synchronisation - Wenig, aber mächtige Cores - Größere Caches in Chip
Ziel: Hoher Gesamtdurchsatz	Ziel: Niedrige Latenz pro Thread

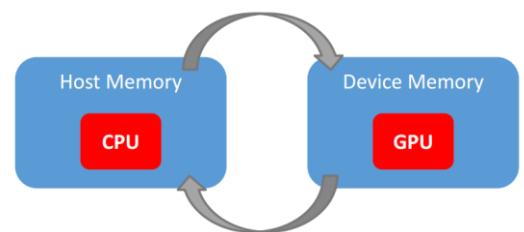
	CPU Intel Xeon E7-8870 Skylake (Q2 2016)	GPU Nvidia Tesla P100 Pascal (Q2 2016)
Cores	20	56 SM mit 64 Cores = 3584 Cores
Taktrate	2.1 GHz	1.3 GHz
Logische Cores / logische Threads	40	56 SM mit je 2048 Resident Threads = 114'688
Memory Bandbreite	85 GB/s	180 GB/s
L1 Cache Grösse	64 KB/Core	64 KB/SM, von L1 + L2
L2 Cache Grösse	256 KB/Core	(shared) geteilt
L3 Cache Grösse	50 MB zusammen	-

NUMA Modell

Non-Uniform Memory Access

Es gibt kein gemeinsamen

Hauptspeicher zwischem GPU und CPU.
Es ist daher ein explizites Übertragen
notwendig. Da ein unterschiedlicher
Instruktionssatz vorhanden ist, ist ein
spezielles komplieren/designen
notwendig.



CUDA Programmiermodell

CUDA steht für Computer Unified Device Architecture und ist proprietär für Nvidia Grafikkarten. Die Aktuelle Version ist mit 8 datiert.

General Purpose Programming Model

Parallelisierung mit sehr vielen (vektorparallelen) Threads. Das Video-Computing und CUDA teilen sich die GPUs. Es gibt ein API und Compiler für die Programmiersprache C/C++.

Nachfolgend wird CUDA anhand des Beispieles der Vektoraddition erklärt.

Sequentiell

```
void VectorAdd(float *A, float *B, float *C, int N) {
    for (int i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
    }
}
```

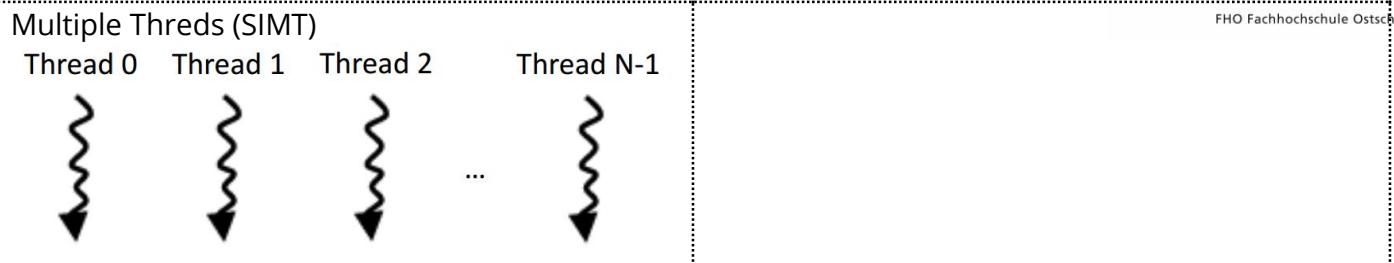
CUDA Kernel

<pre>// kernel definition __global__ void VectorAddKernel(float *A, float *B, float *C) { int i = threadIdx.x; C[i] = A[i] + B[i]; }</pre>	GPU (Device)	<pre>int main() { ... // kernel invocation VectorAddKernel<<<1, N>>>(A, B, C); ... }</pre>	CPU (Host)
--	-------------------------	--	-----------------------

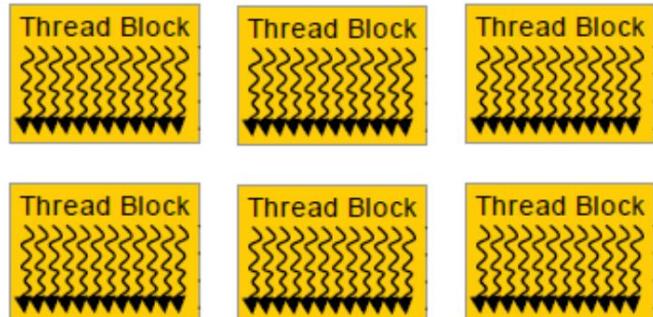
CUDA Threads

Der gleiche Kernel wird von mehreren Threads ausgeführt. Nach dem Prinzip Single Instruction

<pre>__global__ void VectorAddKernel(float *A, float *B, float *C) { int i = threadIdx.x; C[i] = A[i] + B[i]; }</pre>	Thread Id (0 .. N-1)
---	-----------------------------



CUDA Blocks



Die Threads sind in Blöcke gruppiert. Ein Block => gleicher Streaming Multiprozessor. Dies wird im Programmiermodell sichtbar. Threads können innerhalb eines Blocks interagieren.

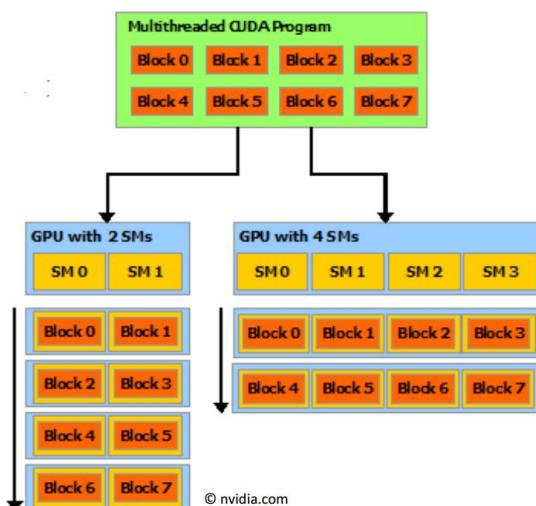
CUDA Ausführungsmodell

Thread = virtueller Skalarprozessor

Block = virtueller Multiprozessor

Die Blöcke müssen unabhängig sein. Also Run to Completion. Denn es gibt eine beliebige Ausführungsreihenfolge.

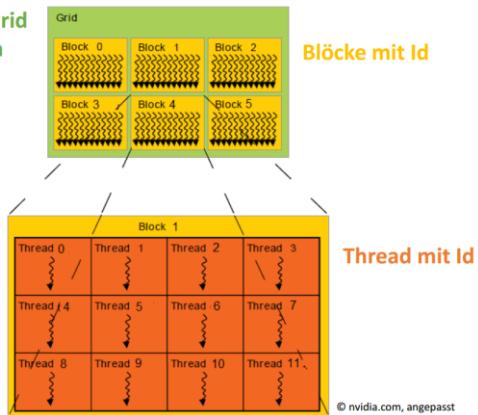
CUDA Thread Pool Abstraktion



Der Grad der Parallelität wird durch die GPU bestimmt. Dabei findet eine automatische Skalierung statt.

Thread Hierarchie

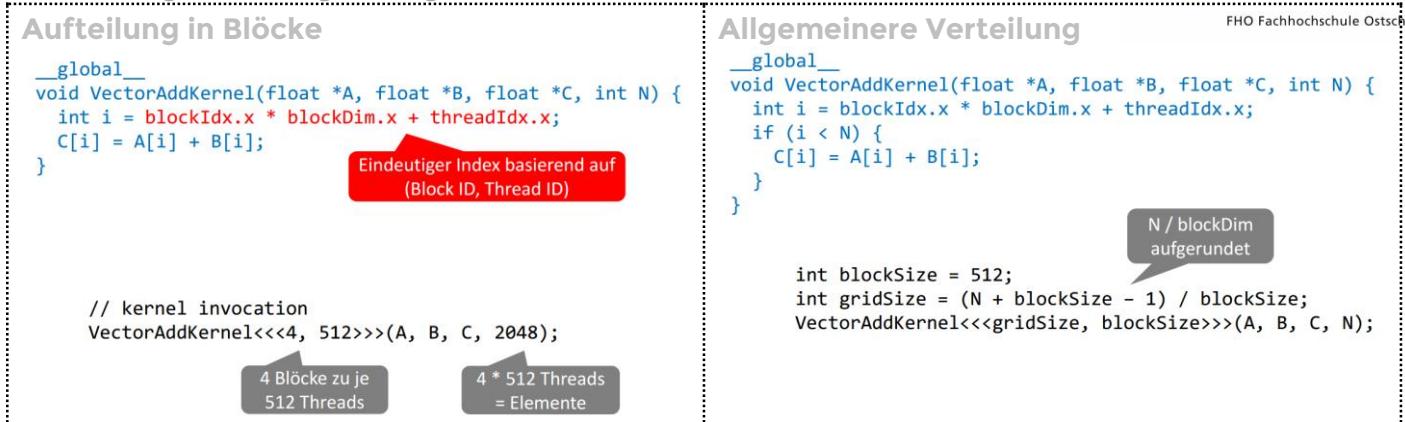
Kernel auf Grid ausführen



Datenaufteilung

Jede Kernel-Ausführung bestimmt seinen Datenteil. `threadIdx.x` – Nummer des Threds innerhalb des Blocks, `blockIdx.x` – Nummer des Blocks und `blockDim.x` – Block-Grösse. Zudem sind weitere Dimensionen `y,z` nutzbar. Der Programmierer modelliert die Datenaufteilung selber.

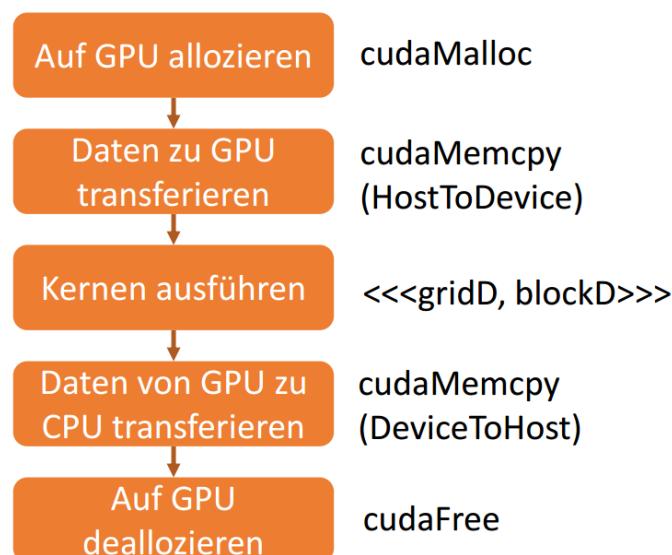
Parallele Programmierung ParProg



Boundary Check

Die if Schleife oben ist da, da es mehr Threads als zu bearbeitende Daten geben kann. Beispielsweise N = 1000. 2 Blöcke mit 512 Threads => 24 Threads sind unnütz. Die Threads mit $i \geq N$ dürfen Array dann nicht zugreifen.

CUDA Ausführung



Programmgerüst

```
void CudaVectorAdd(float* A, float* B, float* C, int N) {
    size_t size = N * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

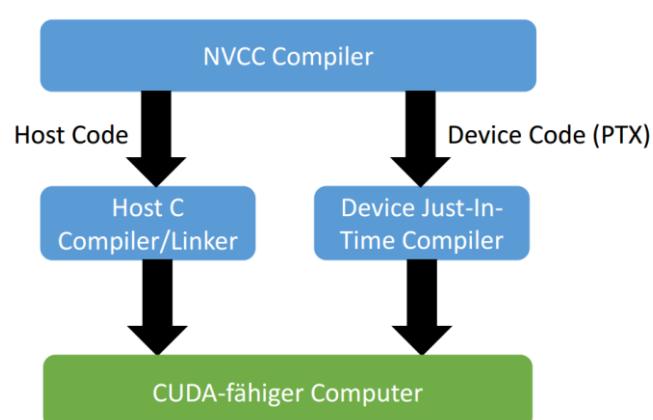
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    int blockSize = 512;
    int gridSize = (N + blockSize - 1) / blockSize;
    VectorAddKernel<<<gridSize, blockSize>>>(d_A, d_B, d_C, N);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

CUDA Compilation



CUDA Speicher-Funktionen

cudaMalloc

- Alloziert Objekt in Device Global Memory
- Parameter: Pointer-Adresse, Size (Bytes)

cudaFree

- Dealloziert Objekt in Device Global Memory
- Parameter: Pointer-Adresse

cudaMemcpy

- Kopiert Speicher zwischen CPU/GPU
- Target Pointer, Source Pointer, Size, Destination (zu oder von GPU)

Fehlerbehandlung

Return Codes von CUDA Funktionen beachten. Fehler, falls Code != cudaSuccess. Dazu am besten Hilfsfunktion für Fehlerbehandlung erstellen.

Parallele Programmierung ParProg

Somit ergibt sich dann folgendes verbessertes CUDA Gerüst.

```
handleCudaError(cudaMalloc(&d_A, size));
handleCudaError(cudaMalloc(&d_B, size));
handleCudaError(cudaMalloc(&d_C, size));

handleCudaError(cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice));
handleCudaError(cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice));

int blockSize = 512, gridSize = (N + blockSize - 1) / blockSize;
VectorAddKernel<<<gridSize, blockSize>>>(d_A, d_B, d_C, N);
handleCudaError(cudaGetLastError());

handleCudaError(cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost));
```

```
handleCudaError(cudaFree(d_A));
handleCudaError(cudaFree(d_B));
handleCudaError(cudaFree(d_C));
```

```
void handleCudaError(cudaError error) {
    if (error != cudaSuccess) {
        fprintf(stderr, "CUDA: %s!\n",
                cudaGetString(error));
        exit(EXIT_FAILURE);
}
```

Die Kernel Funktion hat das `_global_` Keyword. Dies erlaubt Lauch vom Host mit `<<<gridSize, blockSize>>`. Der Kernel kann auch andere `_device_` Funktionen aufrufen.

	Läuft auf	Aufruf von
<code>_global_ void KernelFunc()</code>	Device	Host
<code>_device_ float DeviceFunc()</code>	Device	Device
<code>_host_ float HostFunc()</code>	Host	Host

Quelle: B. Kirk, W. Hwu. Programming Massively Parallel Processors, MK 2013.

Device Query

Die GPU Fähigkeiten und Limiten können abgefragt werden (`cudaGetDeviceProperties()`).

Technical Specifications	Compute Capability										
	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2
Maximum number of resident grids per device (Concurrent Kernel Execution)	16	4		32		16	128	32	16		
Maximum dimensionality of grid of thread blocks					3						
Maximum x-dimension of a grid of thread blocks	65535				2 ³¹ -1						
Maximum y- or z-dimension of a grid of thread blocks					65535						
Maximum dimensionality of thread block						3					
Maximum x- or y-dimension of a block					1024						
Maximum z-dimension of a block					64						
Maximum number of threads per block					1024						

Wahl der Launch Configuration

- Maximale Anzahl Threads per Block
 - o Abhängig von GPU, z.B. 512 oder 1024.
- Blockgrösse als Vielfaches von 32
 - o Sonst sehr ineffizient
- Überflüssige Threads vermeiden
 - o 2 Blöcke à 1024 → 548 unnütze Threads
- Streaming Multiprocesstor ausschöpfen
 - o Limite für Resident Blöcke und Threads, z.B. 8 und 1536
- Grosse Blockgrösse hat Vorteile
 - o Threds können nur in Block interagieren

GPU Parallelisierung 2

Quiz Konfiguration

Vektorlänge 1500

- Maximale Anzahl Threads per Block = 1024
- Maximale Anzahl Resident Blocks = 8
- Maximale Anzahl Resident Threads = 1536.

Resident Threads & Blocks

Anzahl Blöcke/Threads, die in GPU geladen sein können. Davon sind zu einer Zeit nur wenige effektiv in Ausführung. Falls eine Berechnung blockiert, wechselt die GPU zu anderen Resident Block & Resident Thread. Analoges Konzept wie CPU Hyper Threading.

Blocks	Threads per Block	Unnütze Threads	Threads In SM
12	128	36	1024
10	160	100	1280
7	224	68	Alle
6	256	36	Alle
5	320	100	Alle
4	384	36	Alle
3	512	36	Alle
2	1024	476	1024

Favorit

Nur 1 Block in SM wegen Resident Thread Limite

7

Speichermodell

3D Thread Hierarchie

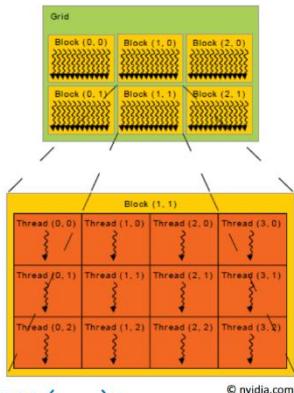
(x, y, z) für threadIdx und blockIdx (1D falls y ,z ungenutzt, 2D falls z ungenutzt) → Modellierhilfe: 2D Matrizen, 3D Würfel

Beispiel

```
dim3 gridSize(3, 2);

dim3 blockSize(4, 3);

Call<<<gridSize, blockSize>>>(...);
```



C Limitation

Mehrdimensionales Array wird nicht direkt unterstützt.

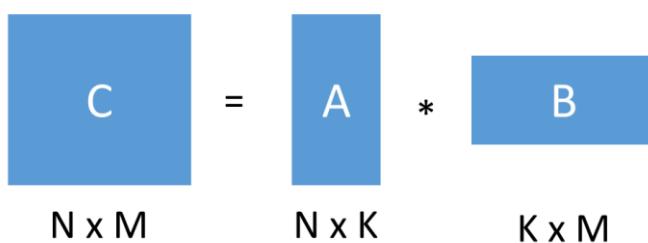
Gewünscht, aber nicht unterstützt

```
float[,] matrix = new float[NofRows, NofCols];
matrix[row, col] = ...
```

In C manuell Low-Level linearisieren

```
float *matrix =
    (float *)malloc(NofRows * NofCols * sizeof(float));
matrix[row * NofCols + col] = ...
```

$$C = A * B$$



Sequentielle Matrix Multiplikation

$$C_{i,j} = \sum_{k=0}^{K-1} A_{i,k} * B_{k,j}$$

```
float sum = 0;
for (int k = 0; k < K; k++) {
    sum += A[i, k] * B[k, j];
}
C[i, j] = sum;
```

Keine multidimensionale Arrays in C

Parallele Programmierung FS2017, Luc Bläser

14

Parallelisierung

```
__global__
void multiply(float *A, float *B, float *C) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < M) {
        float sum = 0;
        for (int k = 0; k < K; k++) {
            sum += A[i * K + k] * B[k * M + j];
        }
        C[i * M + j] = sum;
    }
}
```

Jeder Thread ergibt ein Element in C. Daher $N * M$ Threads. Ein Block ist eine Partition von C. Eine erste Lösung mit Randbehandlung würde wie folgt aussehen.

Diese Lösung ist nicht sehr effizient. Global Memory ist relativ teuer: ca. 600 Zyklen. Die Threads lesen in diesem Fall wiederholt dieselben Elemente von A & B..

Thread 0, 0: A[0, 0] .. A[0, K-1]

B[0, 0] .. B[K-1, 0]

Thread 0, 1: A[0, 0] .. A[0, K-1]

B[0, 1] .. B[K-1, 1]

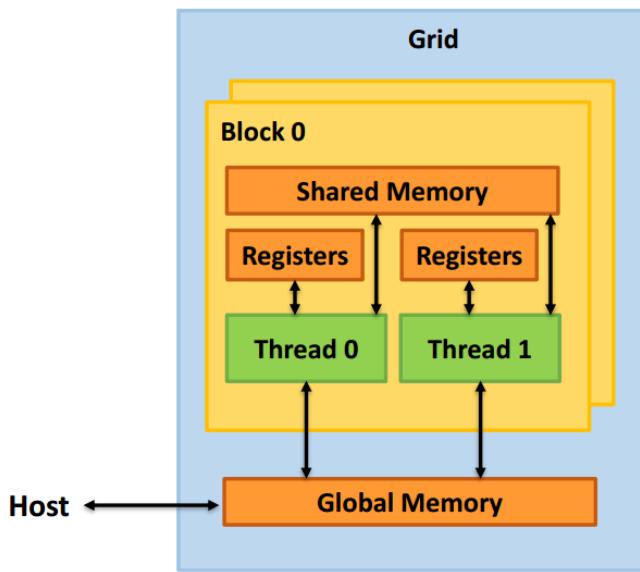
Thread 1, 0: A[1, 0] .. A[1, K-1]

B[0, 0] .. B[K-1, 0]

Thread 1, 1: A[1, 0] .. A[1, K-1]

B[0, 1] .. B[K-1, 1]

⇒ Eine Effizienzsteigerung kann mit einem Cache Speicher erfolgen.



Vereinfachtes Speichermodell

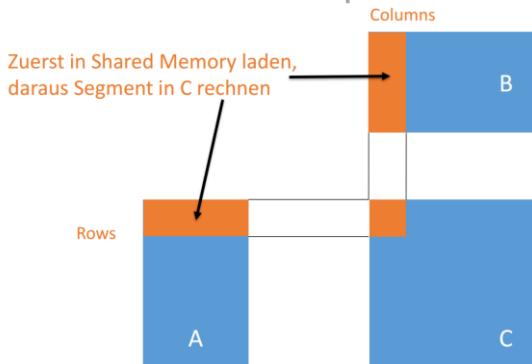
Shared Memory

Per Streaming Multiprozessor, schnell ca. 4 Zyklen, nur zwischen Threads innerhalb Block sichtbar, Paar KB, `__shared__ float x`.

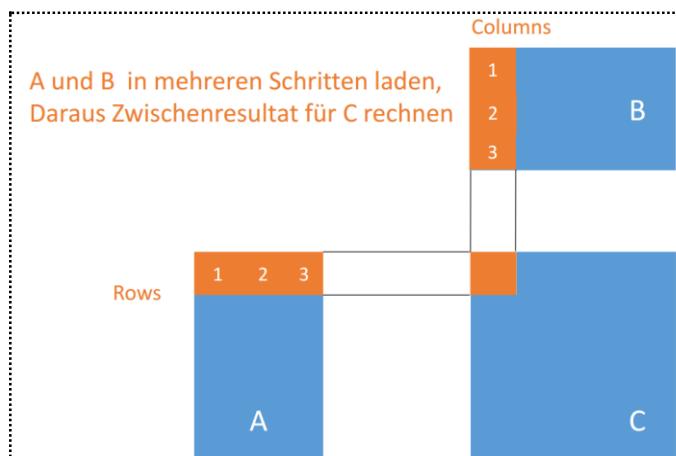
Global Memory

«Main Memory» in GPU Device, langsam ca. 400 – 600 Zyklen, allen Threads sichtbar, mehrere GB, `cudaMalloc()`.

Schnelle Matrix Multiplikation



Bei begrenztem Shared Memory lassen sich A und B in mehreren Schritten laden und daraus dann das Zwischenresultat für C errechnen.



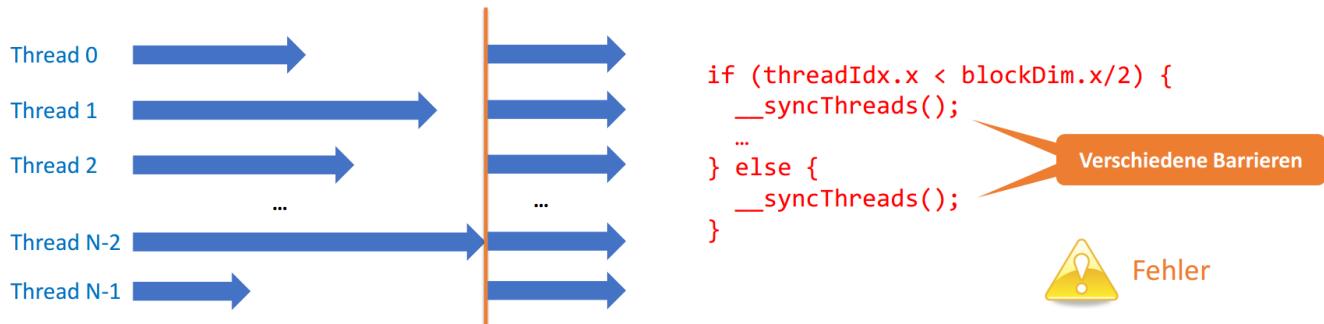
Gerüst des Algorithmus

```
float sum = 0.0;
for (int tile = 0; tile < nofTiles; tile++) {
    // Tile von A und B in Shared Memory lesen
    // Jeder Thread liest ein Element von jedem Tile
    __syncthreads();
    // Multiplizierte Zeile von A-Tile mit
    // Spalte von B-Tile aus dem Shared Memory
    sum += partialProduct;
    __syncthreads();
}
C[row * M + col] = sum;
```

Synchronisation

CUDA Barriere

`_syncthreads()` synchronisiert alle Threads innerhalb eines Blocks. Keine Synchronisation zwischen den Blöcken.



Intra-Block Barriere

Jedes `_syncthreads` Statement ist eine andere Barriere. In if-else nur erlaubt, falls alle Threads eines Blockes entweder nur if- oder nur else-Branch pro Runde wählen. Sonst undefiniert (Blockade/Fehlverhalten).

Erweiterung der Matrix-Multiplikation mit Shared Memory

```

__shared__ float Asub[TILE_SIZE][TILE_SIZE];
__shared__ float Bsub[TILE_SIZE][TILE_SIZE];

int tx = threadIdx.x, ty = threadIdx.y;
int col = blockIdx.x * TILE_SIZE + tx;
int row = blockIdx.y * TILE_SIZE + ty;

for (int tile = 0; tile < nofTiles; tile++) {
    Asub[ty][tx] = A[row * K + tile * TILE_SIZE + tx];
    Bsub[ty][tx] = B[(tile * TILE_SIZE + ty) * M + col];
    __syncthreads();
    // Multipliziere Zeile von A-Tile mit
    // Spalte von B-Tile aus dem Shared Memory
    __syncthreads();
}

```

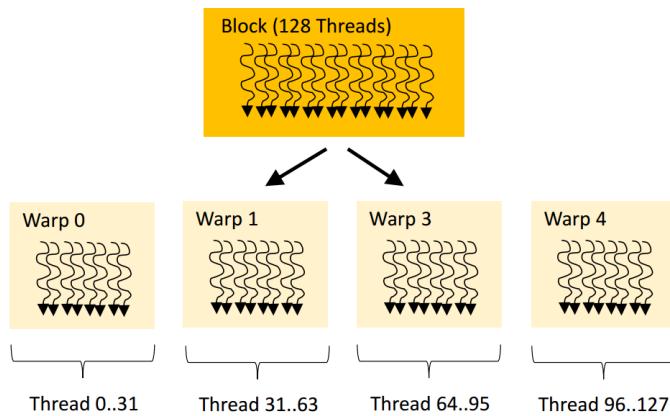
Die Deklaration findet mit dem Keyword `_shared_` statt. Es ist eine statische Array-Grösse notwendig (Begrenzer Speicher z.B. 48 KB). Mehrdimensionalität bei statischer Grösse erlaubt. Sonst muss es in C linearisiert werden.

```

float sum = 0.0;
for (int tile = 0; tile < nofTiles; tile++) {
    Asub[ty][tx] = A[row * K + tile * TILE_SIZE + tx];
    Bsub[ty][tx] = B[(tile * TILE_SIZE + ty) * M + col];
    __syncthreads();
    for (int ksub = 0; ksub < TILE_SIZE; ksub++) {
        sum += Asub[ty][ksub] * Bsub[ksub][tx];
    }
    __syncthreads();
}
C[row * M + col] = sum;

```

Warps



Stream Multiprozessor kann alle Warps eines Blocks beherbergen. Aber nur wenige laufen gleichzeitig echt parallel (1 bis 24).

Abbildung auf Prozessoren

Ein Block läuft immer in einem Streaming Multiprozessor (SM). Ein SM kann eventuell mehrere Blöcke beherbergen. Ein Wrap läuft (falls aktiv) auf den Stream Prozessoren (SPs) eines einzigen SM. Deswegen SIMD. Latenzverringerung: Falls ein Wrap auf Speicher wartet, führt SM nächsten Warp aus, analog zu Hyperthreading Modell.

Divergenz

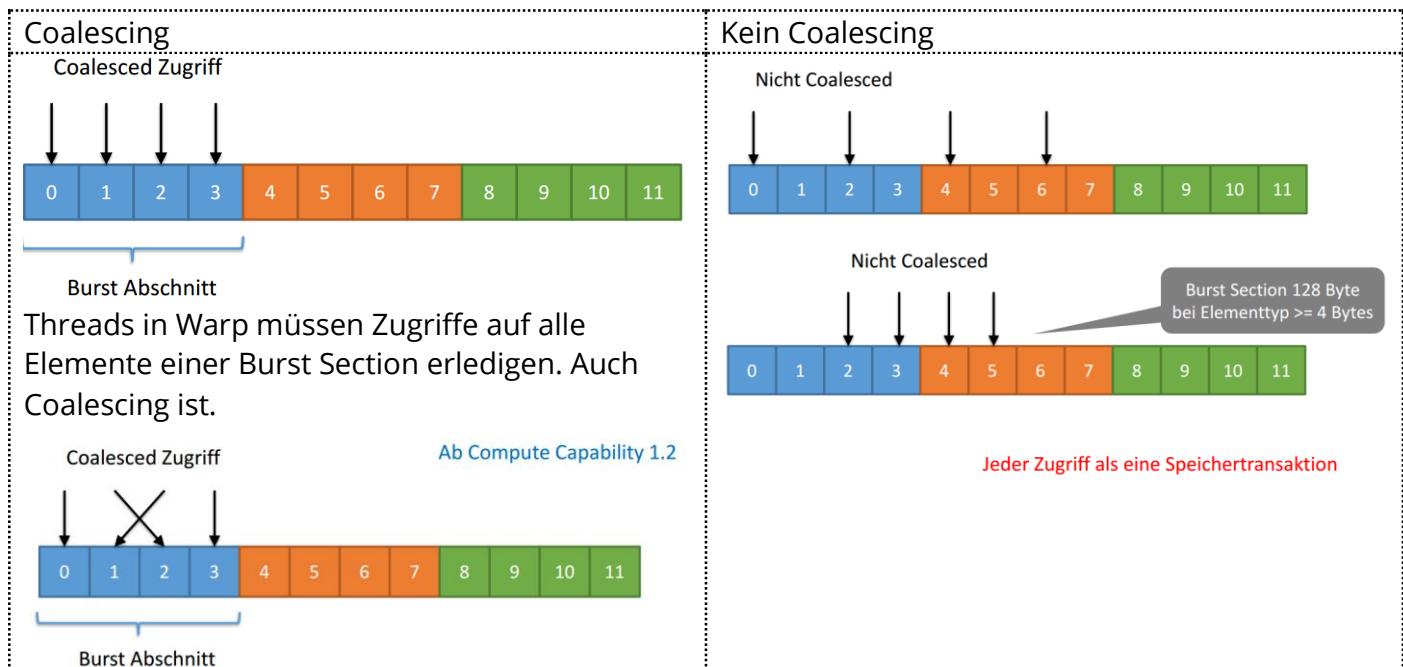
Dies ist ein Performance Problem, wenn unterschiedliche Verzweigungen (if/switch/while/do/for) im selben Warp vorhanden sind. SM führt Instruktion der einen Verzweigung durch, die anderen Threads müssen warten. Dann wieder der anderen Verzweigung. Die einen Threads müssen warten.

Schlechter Fall	Guter Fall
<p>Divergenz innerhalb derselben Wrap</p> <pre>if (threadIdx.x > 1) { ... } else { ... }</pre>	<p>Gleiche Verzweigung innerhalb Wrap</p> <pre>if (threadIdx.x / 32 > 1) { ... } else { ... }</pre>

Wraps werden sequentiell nach threadIdx.x/y/z in 32 Schritten aufgeteilt.

Coalescing

Das Zugriffsmuster der Threads ist entscheidend für die Performance. Falls Threads aufeinanderfolgende Daten zugreifen, kann dies in einer Transaktion (Memory Burst) erledigt werden. Sonst viele einzelne teure Zugriffe (z.B. je 400 Zyklen pro Global Device Memory Access).



Die Zugriffe möglich wie folgt (um)designen.

Data[(Ausdruck ohne threadIdx.x + threadIdx.x)]

Performance Empfehlungen

- Datenaustausch zwischen CPU/GPU minimieren
- Divergenz innerhalb Warp vermeiden
- Global Memory Zugriffe reduzieren (Shared Mem)
- Coalescing ermöglichen
- GPU ausschöpfen (Resident Blocks /Threads)
- Wenig lokale Variablen (=Register) pro Thread

Cluster Parallelisierung

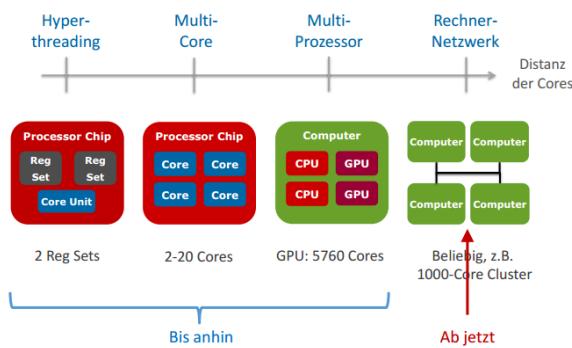
Quiz vom letzten Kapitel

Welche Performance-Aspekte mussten wir bei der GPU-Parallelisierung beachten?

- Shared Memory
- GPU Utilisation
- Divergenz minimieren
- Coalescing

HPC Cluster Architektur

Stufen der Parallelisierung



Motivation

Wir möchten eine möglichst hohe parallele Beschleunigung (Faktor 100 und mehr) erreichen.

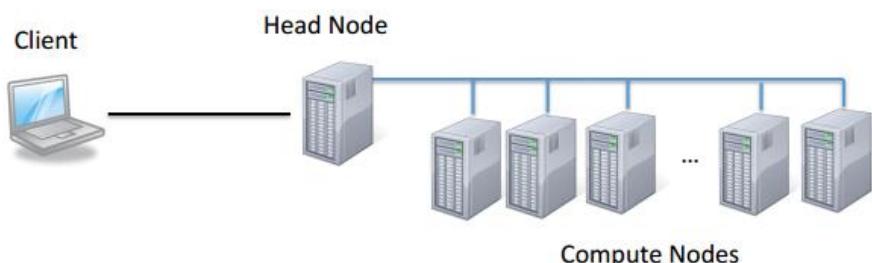
Also viele CPU Cores statt nur viele GPU Cores. GPUs sind wegen SIMD oft zu einschränkend.

Computer Cluster

Ein Computer Cluster ist ein Verbund von leistungsfähigen Rechenknoten. Meist sind es

gleichartige Rechnerknoten, festverbunden an einem Standort und haben ein sehr schnelles Interconnect (z.B. 60Gbit/s Switch). Dies wird vor allem in der Domäne des wissenschaftlichen Rechnens genutzt.

Der ParProg HPC Cluster in Azure besteht aus 25 Compute Nodes mit je 4 CPU Cores zu je 2.4 GHz.



Job Manager

All Jobs	Name	Priority	State	Owner	Progress	Create Time	Submit Time	Requested Nodes	Error Message	Pending Reason
735	Test1	Normal	Queued	HPC\User-01	0	05.05.2017 17:01:07	05.05.2017 17:01:23			Not Enough Resource
734		Normal	Canceled	HPC\User-01	0	05.05.2017 16:58:36	05.05.2017 16:59:23		Canceled by user	Message:None
731		Normal	Canceled	HPC\hpc-admin	0	05.05.2017 16:41:04	05.05.2017 16:41:07		Canceled by user	Message:None
729	Available Software Updates for Node Report Post Step	Finished		HPC\hpc-admin	100	02.05.2017 15:47:06	02.05.2017 15:47:06			None
728	SOA Service Loading Test Post Step	Finished		HPC\hpc-admin	100	02.05.2017 15:43:33	02.05.2017 15:43:33			None

Einfacher HPC Job starten

Job Name:

Project Name:

Priority:

Task name:

Command line:

Working directory:

Standard input:

Standard output:

Standard error:

Number of (nodes / sockets / processors) for the task.

Minimum: Maximum:

HPC Job

Zusammengehörige Ausführung im Cluster. Wird vom Client lanciert und besteht aus einem oder mehreren Tasks,

HPC Task

Ausführung eines Executables (Command Befehls). Er operiert auf Files in File Share des Cluster. Abhängigkeiten zwischen Tasks definierbar.

Verteiltes Programmiermodell

Programme auf mehreren Nodes ausführen können

- Kein Shared Memory (NUMA) zwischen den Nodes
- Shared Memory (SMP) für Cores innerhalb Node

Actor Model/CSP für beide Modelle geeignet

- Egal, ob auf gleichem oder entferntem System
- Nachrichtenaustausch zwischen Prozessen

MPI (Message Passing Interface)

MPI basiert auf Actor/CSP Prinzip und ist die übliche Wahl für heterogene Parallelisierung z.B. Cluster, Multi-Core. Es ist ein Industries-Standard einer Library. Meist für C, Fortan und auch für .NET, Java.

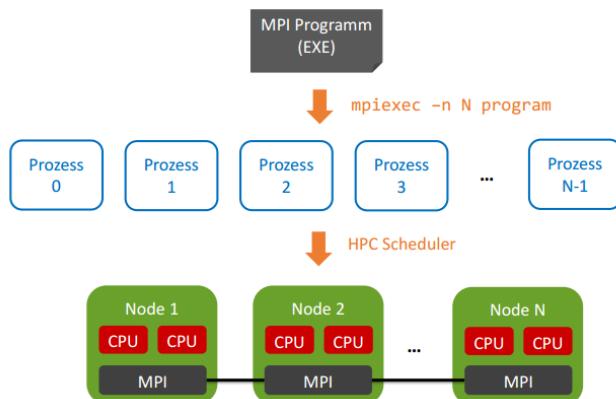
Erstes Programm

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char* argv[]){
    MPI_Init(&argc, &argv); MPI Initialisierung
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); Prozess Identifikation
    printf("MPI process %i", rank);
    MPI_Finalize(); MPI Finalisierung
    return 0;
}
```

```
using MPI;
using System;

class Program {
    public static void Main(string[] args) {
        using (new MPI.Environment(ref args)) {
            int rank = Communicator.world.Rank;
            Console.WriteLine("MPI process {0} ", rank);
        }
    }
}
```



Die Ausführung findet mit mpiexec -n 16 FirstMpiprogram.exe statt. N gibt dabei die Anzahl Cores an. Der HPC Scheduler verteilt die Prozesse dann auf den Nodes.

Single Program Multiple Data

Das MPI Programm wird in mehreren Prozessen gestartet. Jeder Prozess hat seine Identifikation (Rank). Die Prozesse arbeiten unabhängig in ihrem Adressraum. Alle Prozesse starten und terminieren synchron.

Die Prozesse können untereinander kommunizieren. Also Senden und Empfangen von Nachrichten und die Synchronisation mit Barrieren.

Communicator



Ist eine Gruppe von MPI-Prozessen. Dies erlaubt Kommunikation zwischen Prozessen. In «Communicator World» sind alle Prozesse einer MPI-Programmausführung drin. Eigene Gruppen sind auch definierbar.

Prozess Identifikation

Rank = Nummer innerhalb einer Gruppe (Aufsteigende Nummerierung von 0). Die eindeutige Identifikation ist (Rank, Communicator).

Communicator.world.Rank Prozess-Nummer (0..Size-1)

Communicator.world.Size Gesamtanzahl der Prozesse

Direkte Kommunikation

Senden world.Send(value, receiverRank, messageTag)

Receive world.Receive(senderRank, messageTag, out value)

World = Communicator.world, messageTag = Frei wählbare Nummer für Nachrichtenart (>=0).

Parallele Programmierung ParProg

Beispiel – Senden und Empfangen

```

var world = Communicator.world;
int rank = world.Rank;
int size = world.Size;
int tag = 1;
if (rank == 0) {
    int value = new Random().Next();
    for (int to = 1; to < size; to++) {
        world.Send(value, to, tag);
    }
} else {
    int value;
    world.Receive(0, tag, out value);
    Console.WriteLine("{0} received by {1}", value, rank);
}
    
```

 Was bewirkt dieses MPI-Programm?

Verschiedene Verhalten

Prozess 0

```

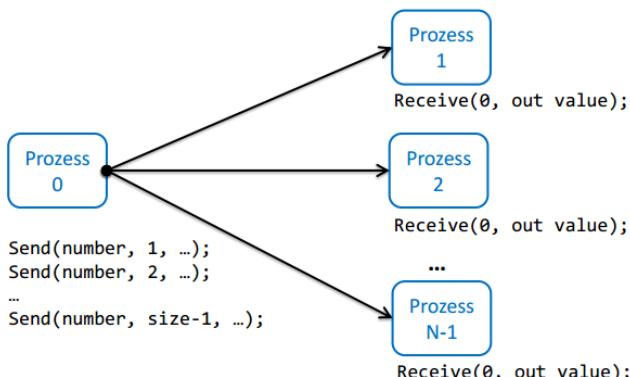
int value = new Random().Next();
for (int to = 1; to < size; to++)
{
    world.Send(value, to, tag);
}
    
```

Prozesse 1 bis N-1

```

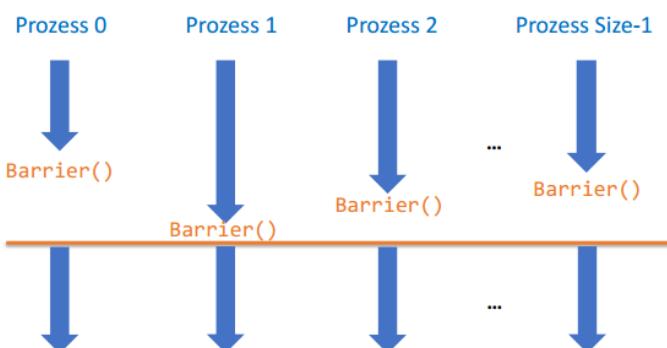
int value;
world.Receive(0, tag, out value);
Console.WriteLine(...);
    
```

Kommunikationsablauf



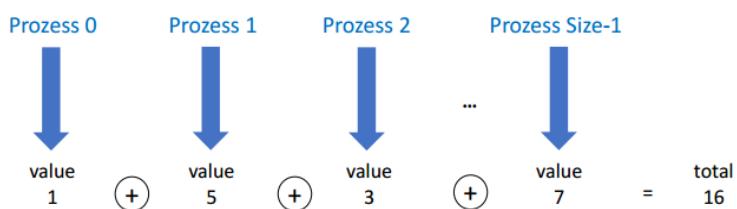
MPI Barriere

Communicator.world.Barrier() wartet, dass alle Prozesse die Barriere erreichen.



Reduktion

Mit world.Allreduce(value, (a,b) => a + b) findet eine Aggregation der Teilresultate zwischen den Prozessen statt.



T Allreduce(T value, Op<T>)

- Beliebige Aggregationsoperation, z.B. als Lambda
- Default-Initialwerte für T, z.B. 0 bei int
- Jeder erhält das Gesamtresultat als Rückgabewert
- Implizite Barriere/Broadcast über alle Prozesse

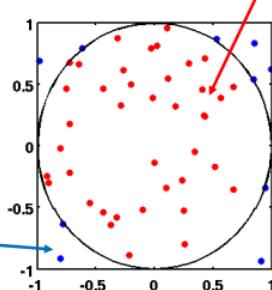
T Reduce(T value, Op<T>, int rank)

- Nur ein Prozess (rank) sieht das Gesamtresultat
- Effizienter als Allreduce, weil kein Broadcast

Ist eine randomisierte Berechnung von Pi. Generiert zufällige Punkte aus der Fläche, schaut dann ob Sie im Einheitskreis liegen und gibt die Trefferrate als eine Annäherung von Pi an.

$$\frac{\pi}{4} = \frac{\#Hit}{\#Points}$$

Miss



Sequentieller Algorithmus

```
long CountHits(long trials) {
    long hits = 0;
    Random random = new Random();
    for (long i = 0; i < trials; i++) {
        double x = random.NextDouble();
        double y = random.NextDouble();
        if (x * x + y * y <= 1) { hits++; }
    }
    return hits;
}
```

```
long hits = CountHits(Trials);
double pi = 4 * ((double)hits / Trials);
```



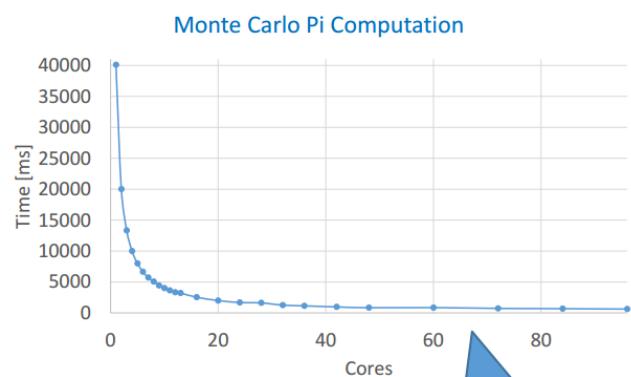
Wie lässt sich das mit MPI parallelisieren?

MPI Parallelisierung

```
int rank = world.Rank;
int size = world.Size;

long hits = CountHits(Trials/size);
long totalHits = world.Reduce(hits, (a, b) => a + b, 0);

if (rank == 0) {
    double pi = 4 * ((double)totalHits / Trials);
    ...
}
```



«Embarrassingly Parallel»

Reactive Programming

Ist ein Modell der impliziten Parallelisierung und ist deskriptiv (bezogen auf den Datenfluss).

Quiz – Vorheriges Kapitel

```
int rank = Communicator.world.Rank;
int size = Communicator.world.Size;

string text = "Hello from " + rank;
int right = (rank + 1) % size;
int left = (rank + size - 1) % size;

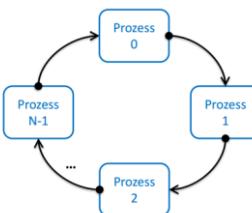
Communicator.world.Send(text, right, 0);
Communicator.world.Receive(left, 0, out text);
```



Was macht dieses MPI-Programm?
Gibt es ein Problem?

Das Programm schickt dem rechten Nachbarn einen Text.

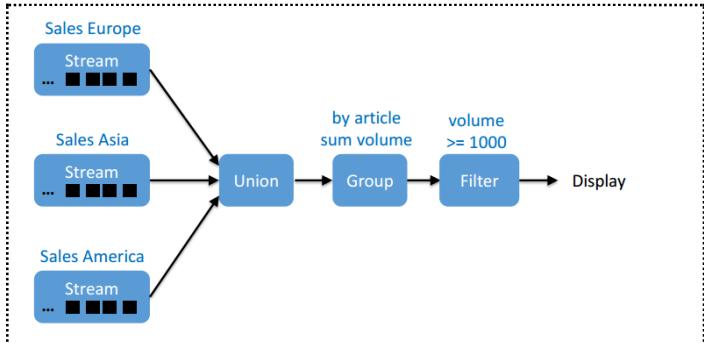
Das Send ist daher allenfalls blockierend (d.H. wartet dass Gegenseite empfängt). Bei einem Kreise kommt es dann zu einem Deadlock (Alle wollen senden und niemand empfängt).



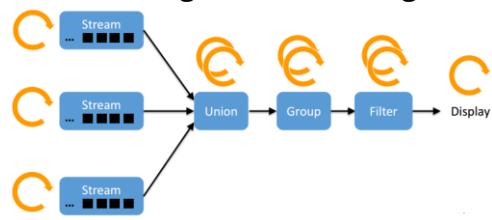
Die Lösung ist ein nicht blockierendes Senden.

```
var request = Communicator.world.ImmediateSend(text, right, 0);
Communicator.world.Receive(left, 0, out text);
request.Wait();
```

Programm-Datenflüsse



Das Beispiel ist auf mehreren Stufen parallelisierbar. Horizontal als parallele Pipeline und Vertikal durch Teilung der Datenmenge.



Mainstream Datenfluss-Modelle

.NET LINQ (Language-Integrated Query)

Eingebettete C# Syntax im Stil von SQL oder explizit über Methoden («Extensions Methods»), Parallelisierung auf .NET Task Parallel Library (TPL).

Java 8 Stream API als Nachahmer

Keine eigene Syntax, via Methoden («Default Methods»), Eingeschränkter (z.B. Gruppierung verlässt Stream API), Parallelisierung auf Fork Join Thread Pool.

LINQ via Extensions

Methods

```
salesEurope.
Union(salesAsia).
Union(salesAmerica).

GroupBy(item => item.Article, item => item.Volume).

Select(category =>
new { Key = category.Key, Value = category.Sum() }).

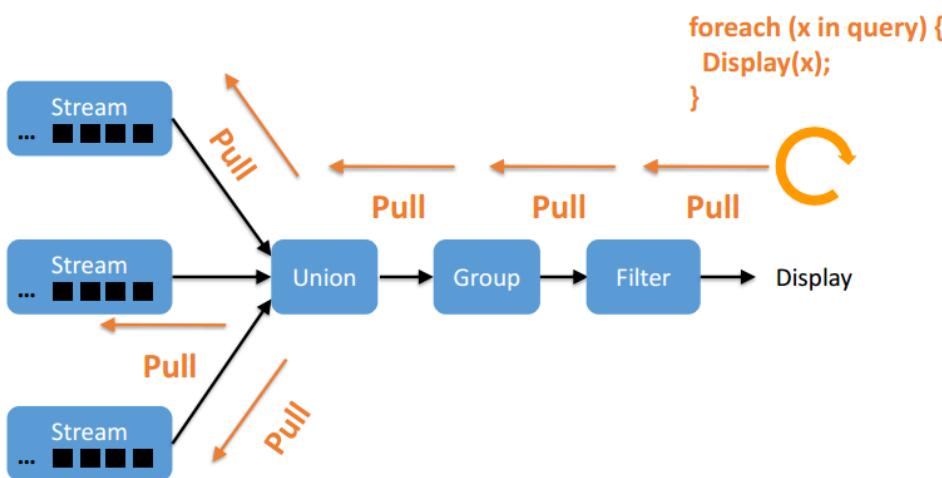
Where(category => category.Value >= 1000);
```

Eingebettete LINQ-Syntax

```
from entry in
salesEurope.
Union(salesAsia).
Union(salesAmerica)
group
entry by entry.Article into category
let sum = category.Sum(e => e.Volume)
where
sum >= 1000
select
new { category.Key, sum };
```

Parallel LINQ (PLINQ)

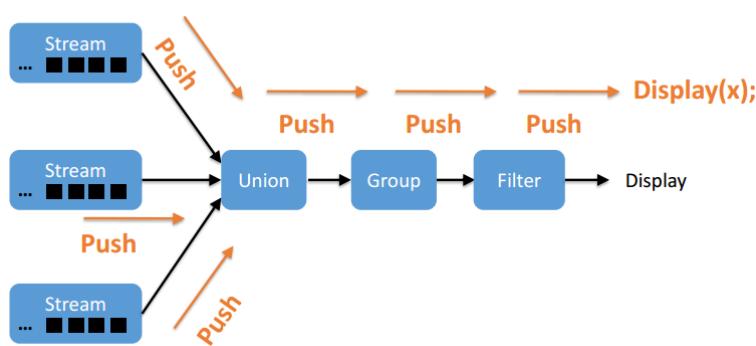
```
from entry in
salesEurope.AsParallel().
Union(salesAsia.AsParallel()).
Union(salesAmerica.AsParallel())
group
entry by entry.Article into category
let sum = category.Sum(e => e.Volume)
where
sum >= 1000
select
new { category.Key, sum };
```



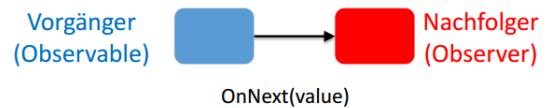
LINQ funktioniert nach dem Pull-Mechanismus. Die Auswertung beginnt durch Iterieren der Abfragen. Dabei werden die Pipeline-Schritte rückwärts ausgelöst. Die Input-Quelle ist passiv. Deswegen muss der Input vollständig vorliegen. Er wird dann durch eine Query ausgelesen und verarbeitet.

Das Pull-Modell funktioniert nicht, falls der Input sukzessive mit Pausen ankommt (User Input, Netzwerk) oder die Länge des Streams unbekannt bzw. unendlich ist. Die Lösung ist das Push-Modell aka Reactive.

Reactive Programming (Push-Mechanismus)

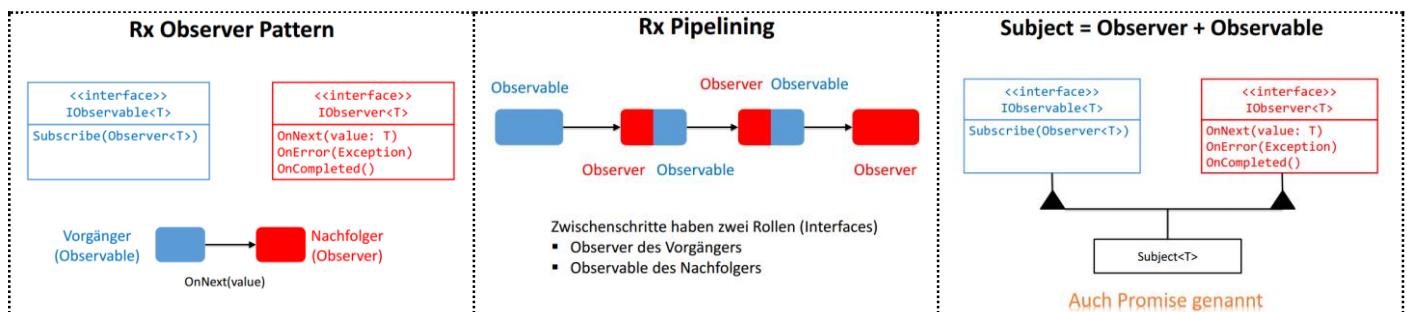


Der Input und die Arbeitsschritte sind hier aktiv (Lösen pro Wert einen Ereignis aus (OnNext)). Dabei findet dann eine Verkettung der Arbeitsschritte statt. Der Nachfolgeschritt abonniert Events des Vorgängers.



Rx.NET (Reactive Extensions)

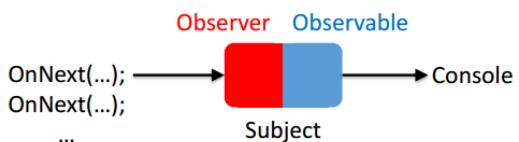
Der Erfinder für das Reactive Programming von .NET ist Erik Meijer (2009). Es basiert auf dem Datenfluss-Paradigma. Hauptpunkte sind Asynchrone Events + LINQ. Die Inputs können asynchron als Events ankommen. Die Verarbeitungsschritte sind als LINQ beschreibbar. Die Outputs sind wiederum asynchrone Events (ideal für GUIs).



```
var subject = new Subject<string>();
subject.Subscribe(Console.WriteLine);

subject.OnNext("A");
subject.OnNext("B");
subject.OnNext("C");

subject.OnCompleted();
```



Der Observer kann beliebig viele Werte erhalten und somit beliebige Verzögerungen zwischen OnNext(). Am Ende der Sequenz bei Erfolg OnCompleted() und bei Fehler OnError(). Nach dem Ende des Subjects werden weitere Aufrufe ignoriert.

Ad-Hoc Observer Erzeugung

```
subject.Subscribe(
    DelegateForOnNext,
    DelegateForOnError,
    DelegateForOnCompleted
);
```

OnError und
OnCompleted
optional

Beispiel:

```
subject.Subscribe(
    value => Console.WriteLine("{0} received", value),
    exception => Console.WriteLine("{0} thrown", exception),
    () => Console.WriteLine("completed", value)
);
```

Buffer-Varianten

Subject	Observer erhält zukünftige Werte	Kein Buffer
ReplaySubject	Observer erhält alle alten Werte	Unbeschränkter Buffer
BehaviorSubject	Schickt letzten Wert und zukünftige	Buffer von einem Element
AsyncSubject	Schickt letzten Wert bei OnCompleted	Buffer von einem Element

Rx und Concurrency

Default ist alles sequentiell jedoch asynchron. Die Concurrency ist einfach einstellbar mit dem Scheduler bei ObserveOn().



Rx Scheduler

Default

TaskPoolScheduler.Default

NewThreadScheduler.Default

DispatcherScheduler.Current

Synchrone Ausführung (gleicher Thread)

Parallele Ausführung mit TPL

Alle Aufrufe dieses Observable in neuen Thread

GUI Thread

So wird eine Aktion im GUI Thread ausgeführt.

```
combinedSales.ObserveOnDispatcher().Subscribe(
    entry =>
        textBlock.Text += entry.Article + " " + entry.Volume
);
```

Ausführung auf
GUI Thread

Mögliche Concurrency Fehler

Race Conditions

Mit Seiteneffekten in Observer möglich. Vermeiden oder wenn nötig synchronisieren.

Deadlocks

Bei Warteabhängigkeiten in Observer. Blockierende Aufrufe wie First(), Last() vermeiden.

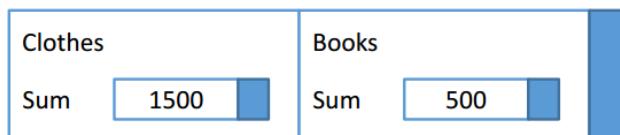
RX mit LINQ kombinieren

```
var sales =
    from entry in
        salesEurope.ToObservable().
    Merge(salesAsia.ToObservable()).
    Merge(salesAmerica.ToObservable())
group
    entry by entry.Article into category
let
    sum = category.Sum(e => e.Volume)
select
    new { category.Key, sum };
```



sum ist kein Skalar,
sondern Observable

Da ein Observable von Einträgen, deren Sum-Field wiederum ein Observable (mit einem Wert) ist.



```
from category in sales
from total in category.sum
where total >= 1000
select new { category.Key, total };
```

Vorgefertigte Observables

Observable.Return("Value")	Liefert den Wert, dann Completed
Observable.Empty()	Liefert sofort Completed
Observable.Never()	Liefert nie etwas (auch nicht Completed)
Observable.Throw(exception)	Liefert sofort Error

Observable.Range(-10, 20) Zahlen von -10 bis 10

Observable.Generate(
0,
value => value < 100,
value => value + 2,
value => value
)

source.Take(10) Nächste 10 Werte von source,
danach Completed

Observable.Interval(TimeSpan.FromMilliseconds(250))	Liefert 0, 1, 2, ... im ZeitIntervall von 250 ms
Observable.Timer(TimeSpan.FromDays(1))	Liefert «0» erst in einem Tag
source.Delay(TimeSpan.FromSeconds(1))	Verzögert alle Werte von source einmal um 1 Sekunde

Events zu Observables

Event Quellen zu Observables umwandeln

```
Observable.FromEventPattern
    <MouseButtonEventHandler, MouseButtonEventArgs>(
    h => window.MouseDown += h,
    h => window.MouseDown -= h
);
```

An- und Abmelden beim
Event spezifizieren

Hot = Aktiv	Cold = Passiv
- Notifizieren spontan - Auch ohne registrierte Observers	- Notifizieren on request - Erst bei Anmeldung von Observers
<code>Observable.Interval(...)</code>	<code>collection.ToObservable()</code>
<code>Observable.Timer(...)</code>	<code>Observable.Range(...)</code>
<code>Observable.FromEventPattern(...)</code>	<code>Observable.Generate(...)</code>

Fazit

Vorteile

- Aktive Datenflüsse statt nur passive LINQ
- Skalierbare Parallelität durch Wahl der Scheduler
- Durchgängig asynchron

Nachteile

- Zerstückelung komplexer Logiken in Handler
- Allfälliger Kontext muss durchgeschleust werden
- Komplizierte Aggregation (Observable statt Skalar)

Software Transactional Memory

Motivation

Es gibt diverse Probleme des Shared Mutable Memory. Bei der expliziten Synchronisation kommt es zu Deadlocks, Starvation und es ist mit hohen Kosten verbunden. Bei ungenügender Synchronisation kommt es zu Race Conditions.

Die Idee des STM stammt aus Datenbanksystem. Eine Umsetzung kann durch SW oder auch HW stattfinden. Das Ziel sind keine Races, keine Deadlocks und (keine Starvation).

Historisches

Maurice Herlihy, J. Eliot und B. Moss haben die Idee in 1992 populär gemacht. In 1995 erfolgt eine Implementation rein auf Software Transaktionen und in 2013 wieder auch in Hardware.

Software Transactions

```
atomic {
    success = balance >= amount;
    if (success) {
        balance -= amount;
    }
}
```

Sind atomare Sequenzen von Operationen mit Read/Write auf Speicher. Konzeptionell wie eine grosse atomare Instruktion. Es sollten keine (inkonsistenten) Zwischenzustände bemerkbar sein.

ACI Transaktionen

Atomicity	Vollständig oder gar nicht sichtbar
Consistency	Programm vor und nach Transaktion gültig
Isolation	Effekte wie eine serielle Ausführung

Im Gegensatz zur Datenbank keine Durability/Persistenz.

Konzept

Es ist eine deskriptive Programmierung. Man gibt an was atomar ist und nicht wie!. Dabei findet eine automatische Isolation statt und man überlässt die korrekte Ausführung dem System. Dabei sind nur die Speicherzugriffe isoliert, die Seiteneffekte aber nicht. Die Implementierung ist meist ein optimistisches Concurrency Control.

STM versus Locking

STM	Locking
<pre>void deposit(int amount) { atomic { balance += amount; } } boolean withdraw(int amount) { atomic { if (balance >= amount) { balance -= amount; return true; } else { return false; } } }</pre> <div style="background-color: #d3d3d3; padding: 5px; width: fit-content;"> Deskriptiv: «Atomar und isoliert» </div>	<pre>void deposit(int amount) { synchronized(this) { balance += amount; } } boolean withdraw(int amount) { synchronized(this) { if (balance >= amount) { balance -= amount; return true; } else { return false; } } }</pre> <div style="background-color: #d3d3d3; padding: 5px; width: fit-content;"> Imperativ: «Monitor Lock & Unlock» </div>

Nested Transactions

Sind beim STM atomar und konsistent (Geldsumme invariant). Beim Locking nicht atomar und Inkonsistenz zeitweise sichtbar.

```
void transfer(Account from, Account to, int amount)

STM
atomic {
    if (from.withdraw(amount)) {
        to.deposit(amount);
    }
}
```

Geschachtelte Transaktionen

```
Locking
if (from.withdraw(amount)) {
    to.deposit(amount)
}
```

Geld fehlt zwischen Withdraw/Deposit

Keine Races, keine Deadlocks

Es gibt auch keine Races (also auch im Fehlerfall atomar) und keine Deadlocks.

Bei Locking besteht eine Deadlock-Gefahr und das Geld kann im Fehlerfall verloren gehen.

```
STM
atomic {
    if (from.withdraw(amount)) {
        to.deposit(amount);
    }
}
```

```
Locking
synchronized(from) {
    if (from.withdraw(amount)) {
        to.deposit(amount);
    }
}
```

Niemand sieht Stand zwischen Überweisung

Transaktionsaufführung

Meist durch optimistisches Concurrency Control (OCC). Das heißt un synchronisiert ausführen und bei Konflikten dann ein Rollback durch das System. Die Transaktionen können unerwartet abbrechen. Dabei gibt es ein automatisches Retry solcher Transaktionen. Dies sollte für die Anwendung unbemerkbar sein.

Typische Probleme

- Seiteneffekte in SW-Transaktion bleichen sichtbar
 - o Sehe Abbruch und Wiederholung
 - o Keine IO: Konsole, Log, Files, Netzwerk, UI etc.
- Starvation-Gefahr bei OCC
 - o Transaktion u.U. wiederholt wegen Konflikt abbrechbar.

Warten auf Bedingungen

```
atomic {
    if (balance < amount) {
        retry;
    }
    balance -= amount;
}
```

Ein Retry innerhalb einer Transaktion. Somit kann ich die Transaktion ausdrücklich abbrechen und es findet eine automatische Wiederholung zu einem späteren Zeitpunkt statt. Durch System Monitoring findet die Wiederholung erst statt, wenn es eine Zustandsänderung gibt. Keine explizite Notifizierung.

Nested Transactions

Transaktion innerhalb Transaktion. Commit erst bei Top-Level Transaktion.

```
atomic {
    from.withdraw(amount);
    to.deposit(amount);
}

void withdraw(int amount) {
    atomic {
        if (balance < amount) {
            retry;
        }
        balance -= amount;
    }
}
```

```
atomic {
    if (y > 0) {
        z = x / y;
    }
}
```

Andere Transaktion schreibt zwischendrin y = 0

Div by Zero, wenn Konflikt nicht direkt erkannt worden ist

Bei Konflikten Transaktionen direkt abbrechen. Sonst

inkonsistente Weiterführung. Unerlaubte Folgefehler/Exceptions.

Praktische STM

- ScalaSTM : Scala und Java
- Intel C++ STM

Hardware Support

Im Intel Transactional Synchronization Extentions (TSX). Dieser unterstützt sogar Nested Transactions. Zudem kann dieser vom STM Framework genutzt werden. Somit ist eine hohe Effizienz möglich.

STM auf Java

Nachfolgend einige Beispiele der Scala STM mit Java 8.. Keine spezielle JVM, dafür Kompromisse.

Wrapping von Variablen

Ist ein Wrapper für transaktionselle Zugriffe. Müssen sonst auch JVM-Ebene detektiert werden. get() und set() sind möglich.

In Transaktionen sind keine normalen Variablenzugriffe möglich. Sind nicht transaktionsell (wie sonstige Seiteneffekte). Ausnahme sind lokale Variablen/Parameter.

```
final Ref.View<Integer> balance = STM.newRef(0);
final Ref.View<LocalDate> lastUpdate = STM.newRef(LocalDate.now());
```

Transaktion

```
import scala.concurrent.stm.japi.STM;

void deposit(int amount) {
    STM.atomic(() -> {
        balance.set(balance.get() + amount);
        lastUpdate.set(LocalDate.now());
    });
}
```

Retry

```
void withdraw(int amount) {
    STM.atomic(() -> {
        if (balance.get() < amount) {
            STM.retry();
        }
        balance.set(balance.get() - amount);
        lastUpdate.set(LocalDate.now());
    });
}
```

Nested Transactions

```
void transfer(BankAccount from,
              BankAccount to, int amount) {
    STM.atomic(() -> {
        from.withdraw(amount);
        to.deposit(amount);
    });
}
```

Exceptions

Rollback und Abbrechen mit Exception (kein Retry)

```
void deposit(int amount) {
    STM.atomic(() -> {
        if (balance.get() >= Limit) {
            throw new RuntimeException("Balance limit");
        }
        balance.set(balance.get() - amount);
        lastUpdate.set(LocalDate.now());
    });
}
```

Nur Zugriff auf Ref.View<T> Wrapper erlaubt. T muss immutable sein (sonst auch nicht atomar).

Die indirekten Strukturen sind nicht transaktionell. Bei Ref.View<Map<K, V>> ist die Map nicht transaktionell. Dafür sollte man dann vordefinierte transaktionselle Collection verwenden (STM.newMap(), STM.newSet() oder STM.newArrayListAsList()).

```
final Map<String, BankAccount> accounts = STM.newMap();  
  
BankAccount openAccount(String name) {  
    return STM.atomic(() -> {  
        BankAccount account = new BankAccount();  
        accounts.put(name, account);  
        return account;  
    });  
}
```

Transaktionstheorie

Jede serielle Ausführung der Transaktion ist korrekt. Nebenläufige Ausführung auch korrekt, sofern die Effekte gleich wie irgendeine serielle Ausführung sind. Der Begriff der Serialisierbarkeit ist in der DBS1 Vorlesung zu finden.

Write Skew Problem

```
atomic {  
    if (b.onDuty) {  
        a.onDuty = false;  
    }  
}  
  
atomic {  
    if (a.onDuty) {  
        b.onDuty = false;  
    }  
}
```

Kein Write-Write Konflikt

Konsistenz: Bereitschaftsdienst, minimal einer muss «on duty» sein.

Isolations-Fehler

In Scala STM nicht möglich. Das heißt korrekte Isolation. In anderen Systemen wie Multiverse, Akka STM ist dies aber möglich. (z.B. Falls einfache Snapshots Isolation benutzt wird).

Starvation Probleme

Wiederholter Abbruch einer Transaktion bei OCC. Kann immer wieder in Konflikte geraten. Besonders für lang-laufende Transaktion. Keine Fortschrittsgarantie Starvation in der Regel möglich. Scala STM macht dies mit beliebigen Wiederholungen.

Diskussion

Deskriptiver Ansatz → Einfaches Programmiermodell, keine Race Conditions und Deadlocks

Komplexe Implementierung → Hohe Laufzeit- und Speicher Kosten, Starvation-Vermeidung

Diverse Schwächen in Frameworks → Seiteneffekte, Starvation-Write Skews, Ref-Wrapper