Datenstrukturen und Grundlagen der nebenläufigen Programmierung in Java

W3L AG
info@W3L.de

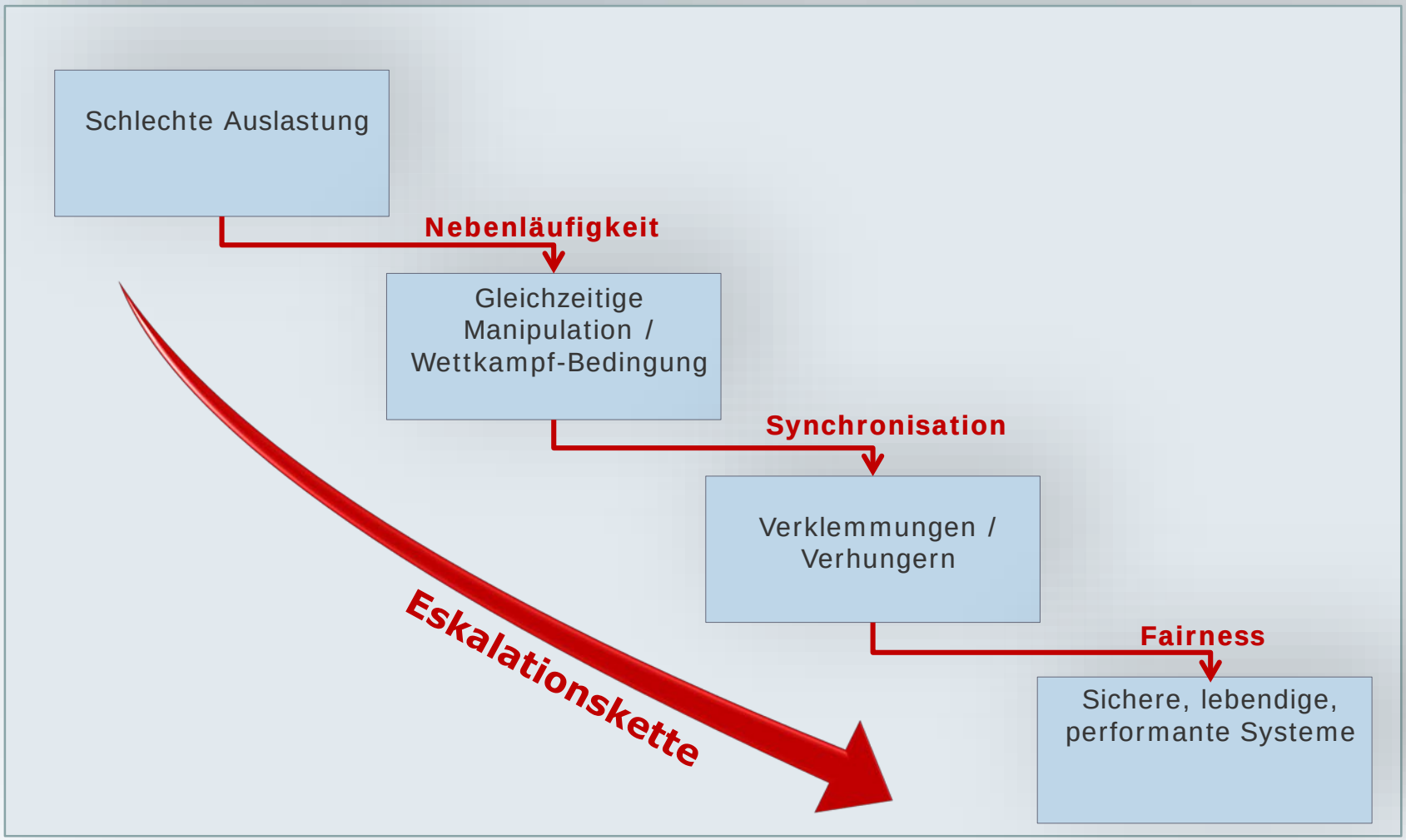
2013

Inhaltsverzeichnis

zurück

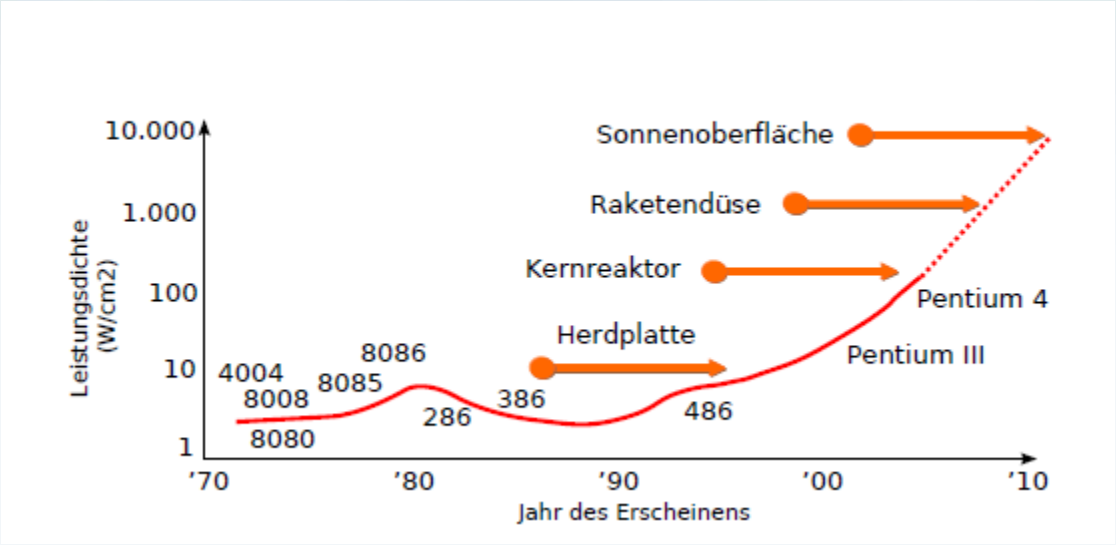
Einführung in die nebenläufige Programmierung

Motivation und roter Faden



Einführung in die nebenläufige Programmierung

Physikalische Randbedingungen



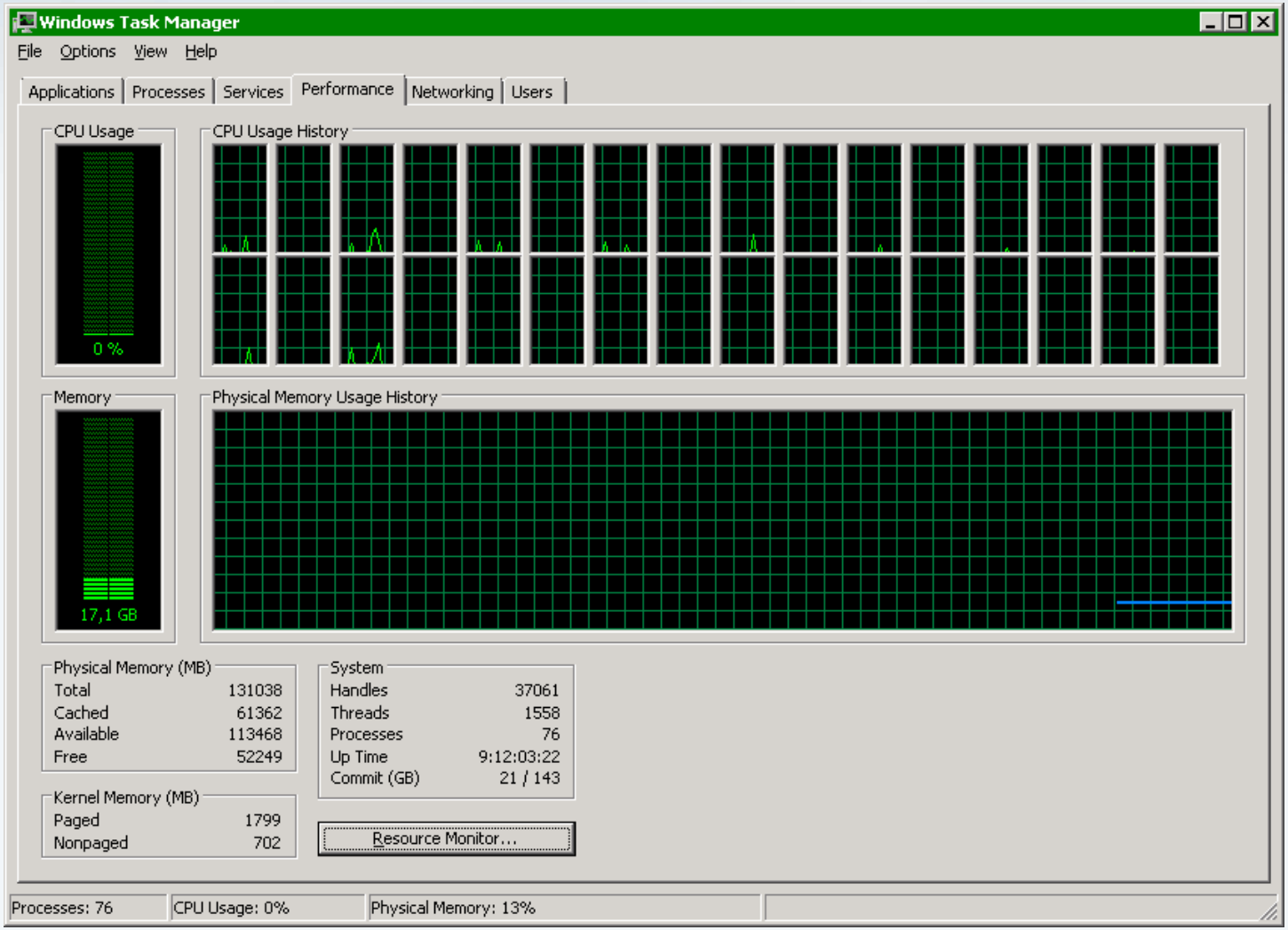
(vgl. Konzeption und Bewertung eines Entwicklungsrahmenwerks zur energieoptimierenden Schaltungssynthese, 2010, Nikolaus Voß)

- Seit Anfang 1990 stieg die Leistungsdichte einer CPU
- Höhere Taktrate bedeuten höhere Leckströme und damit Verluste
- 2010 wäre die Leistungsdichte der Sonne mit 10kW/cm² erreicht gewesen.
  - Umdenken war erforderlich „The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software“

    (vgl. Dr. Dobb‘s Journal, http://www.gotw.ca/publications/concurrency-ddj.htm, 2005)

Einführung in die nebenläufige Programmierung

Heutige Rechnerarchitekturen



Einführung in die nebenläufige Programmierung

- **Was ist Nebenläufigkeit?**
  - **Definition**
    - Zwei Ereignisse sind dann nebenläufig, wenn keines eine Ursache des anderen ist.
    - Man spricht auch von kausal unabhängigen Ereignissen oder Aktionen
    - Sichtweise im **Kausalitätsprinzip** begründet.
    - Nebenläufigkeit ≠ Gleichzeitigkeit
    - Nebenläufigkeit ≠ Parallelität

Einführung in die nebenläufige Programmierung

- **Was ist Nebenläufigkeit?**
  - **Beispiel**
    - Nullstellenberechnung eines Polynoms 2. Grades

$$x^2 + px + q = 0 \qquad x_{1/2} = -\frac{p}{2} \pm \sqrt{(\frac{p}{2})^2 - q}$$

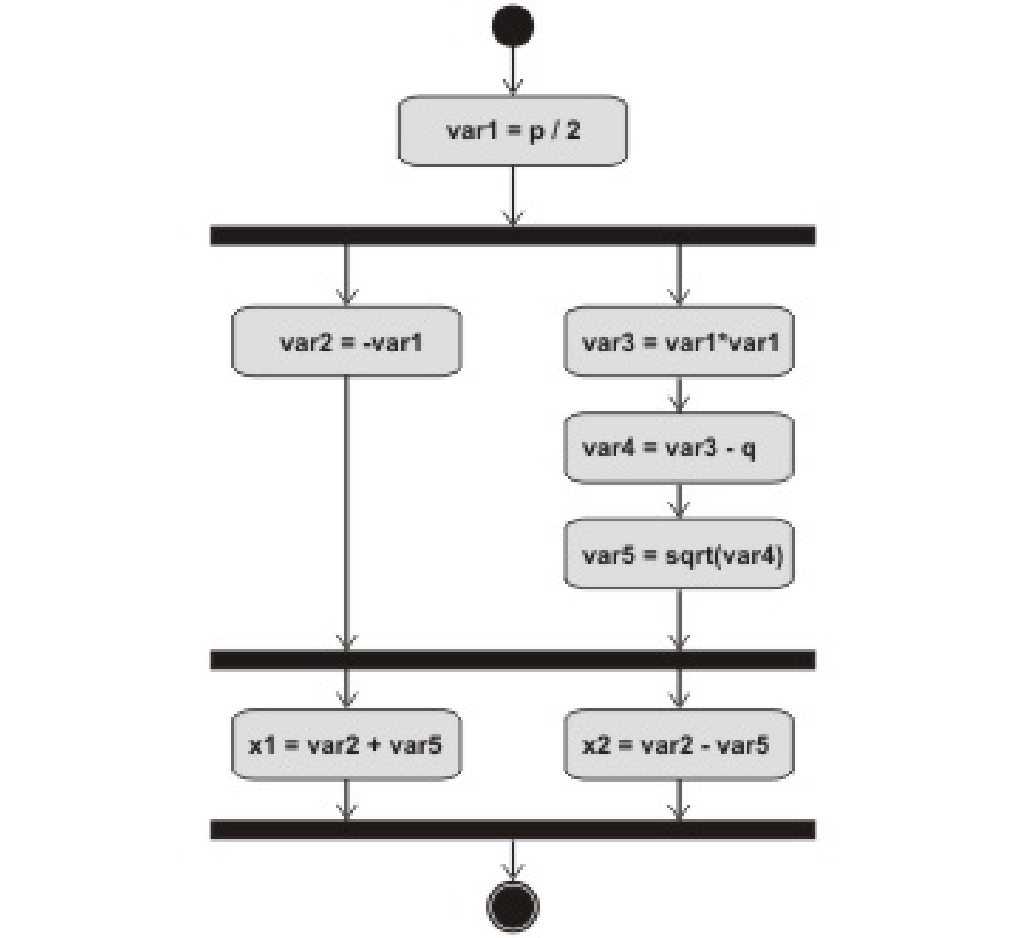
    - Pseudoprogramm

```
Z1 float var1 = p / 2;
Z2 float var2 = -var1;
Z3 float var3 = var1 * var1;
Z4 float var4 = var3 - q;
Z5 float var5 = sqrt(var4);
Z6 float x1 = var2 + var5;
Z7 float x2 = var2 - var5;
```

    - Kausal unabhängige Aktionen ergeben sich dadurch, welche Aktion das Ergebnis einer anderen voraussetzt.

Einführung in die nebenläufige Programmierung

- **Was ist Nebenläufigkeit?**
  - **Beispiel**

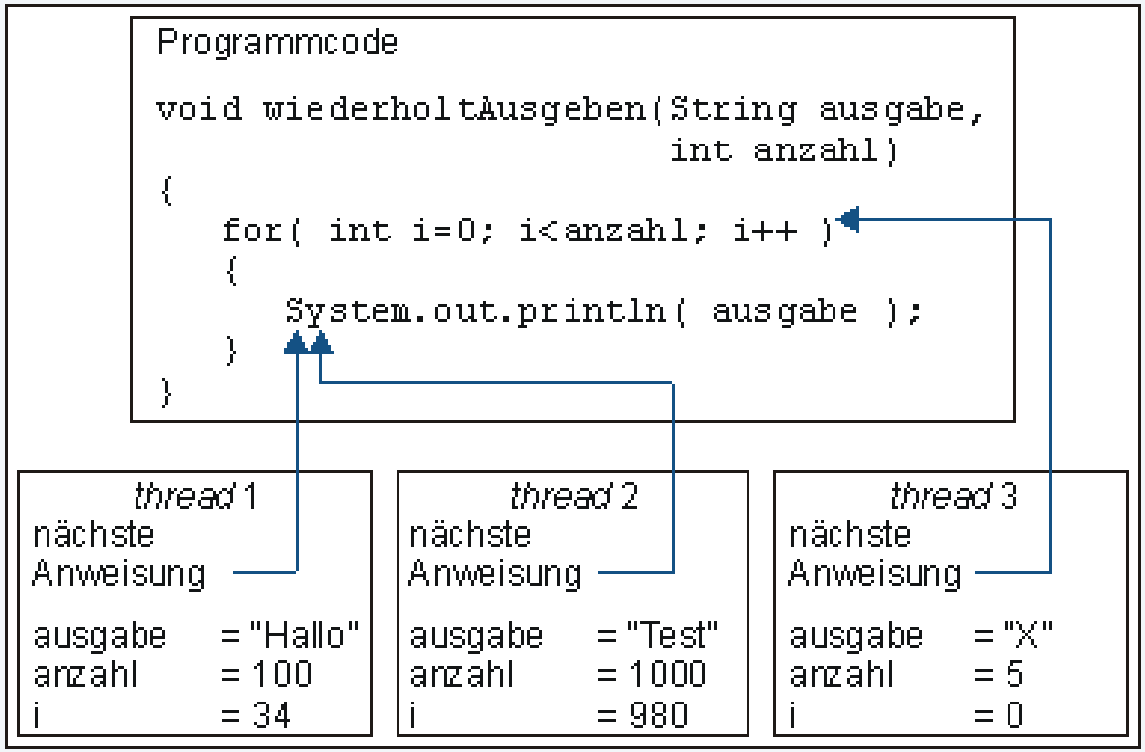


Einführung in die nebenläufige Programmierung

- **Einordnung der parallelen und verteilten Programmierung**
  - Parallele Programmierung
    - Parallel ~ zeitgleich
    - Nebenläufige Aktionen **können** parallel ausgeführt werden!
    - Spezialfall: massiv parallele Programmierung
  - Verteilte Programmierung
    - Es gibt mehrere Kontrollflüsse
    - Manche sind kausal unabhängig voneinander
    - ➔ Die verteilte Programmierung hat viele Bezugspunkte zur nebenläufigen Programmierung

Einführung in die nebenläufige Programmierung

- **Nicht deterministische Ausführung bei nebenläufigen Programmen**
  - Anweisungen werden deterministisch von einem *thread* ausgeführt
  - Da es mehrere Kontrollflüsse in einem Programm geben kann, ...



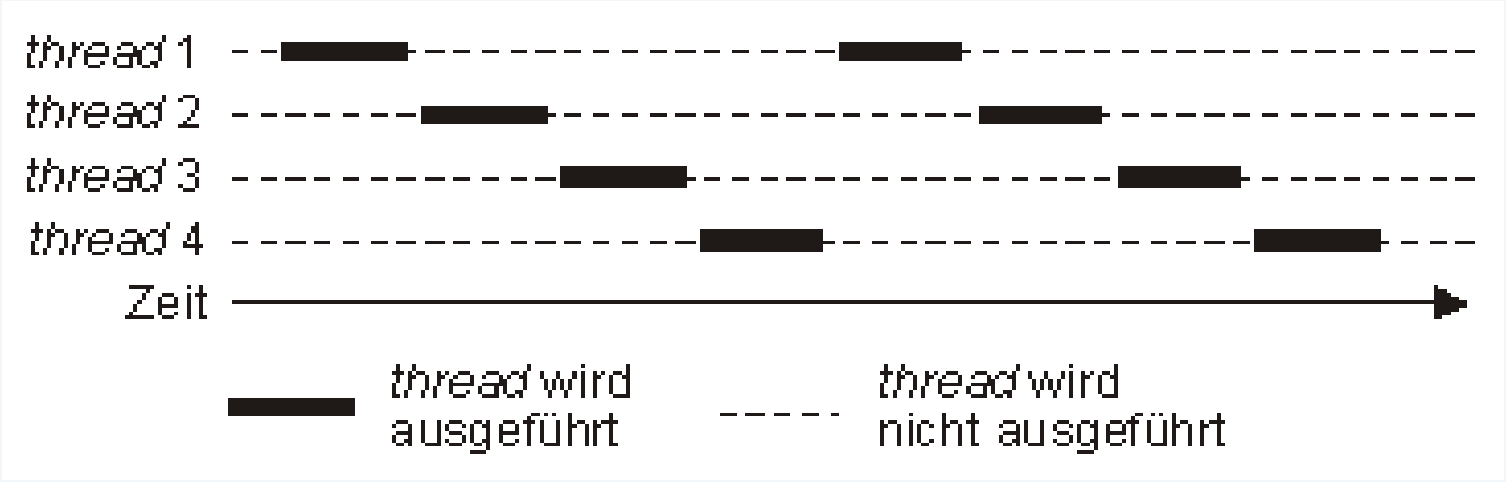
Einführung in die nebenläufige Programmierung

- **Nicht deterministische Ausführung bei nebenläufigen Programmen**
  - ... ist die Reihenfolge der vom Prozessor ausgeführten Anweisungen nicht mehr deterministisch.
  - Warum?

Ausführungs-variante 1	Ausführungs-variante 2	Ausführungs-variante 3	Ausführungs-variante 4
...	...	...	...
Hallo	Hallo	Test	Test
Hallo	Test	Test	X
Hallo	X	Hallo	X
Test	Hallo	Hallo	X
Hallo	Test	Test	X
Hallo	X	Test	Hallo
Hallo	Hallo	Hallo	Hallo
X	Test	Hallo	Hallo
Hallo	X	Test	Test
...	...	...	...

Einführung in die nebenläufige Programmierung

- ***Threads* und *scheduling***
  - Ein Prozessor kann zur selben Zeit immer nur eine Anweisung ausführen
  - Auf Systemen mit einem Prozessor ist daher keine echt parallele Ausführung von Programmen möglich
  - Nebenläufigkeit wird daher von der Hard- und Software (Betriebssystem) simuliert
  - Jeder *thread* wird eine kurze Zeitspanne ausgeführt, um anschließend zu einem anderen *thread* zu wechseln.



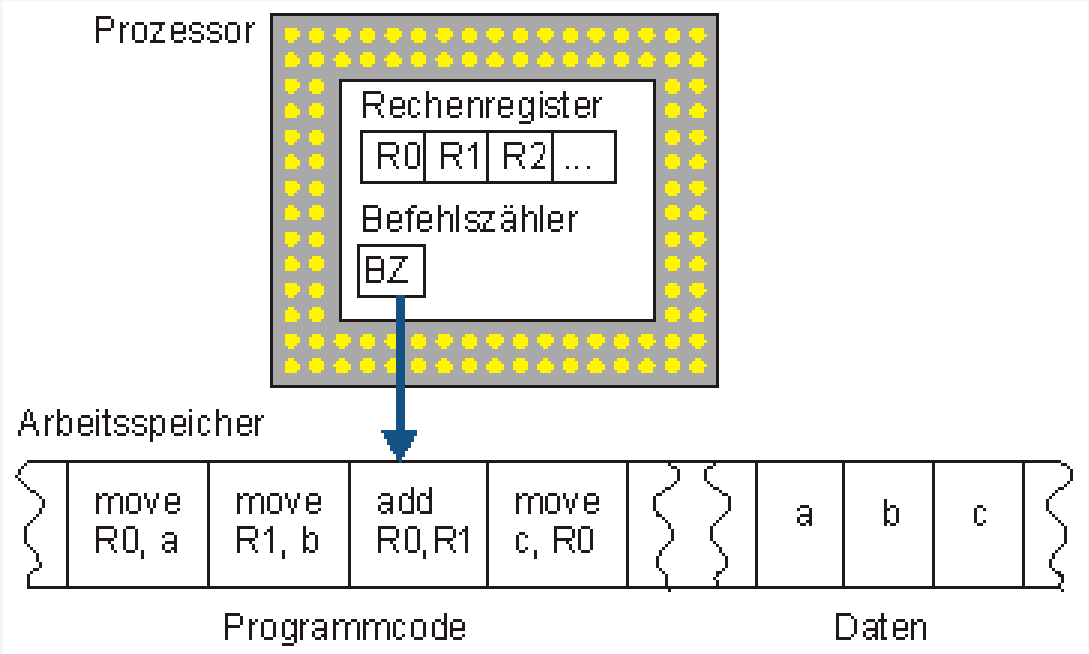
Einführung in die nebenläufige Programmierung

- ***Threads* und *scheduling***
  - Was bedeutet der Wechsel eines Kontrollflusses?
  - Wiederholung: Ein Programm ist eine Aneinanderreihung von Anweisungen!
  - Ein Prozessor liest die Anweisungen aus dem Speicher und führt sie aus. Welche Anweisung aktuell auszuführen ist, wird durch den Befehlszeiger *(program counter* oder *instruction pointer)* markiert.
  - Für die Ausführung der Anweisungen benötigt der Prozessor Rechen- und Adressregister
  - Der aktuelle Zustand eines Prozessors bestimmt sich durch die Werte in allen Rechen- und Adressregister sowie dem Befehlszeiger!
  - Wenn ein thread unterbrochen wird, muss der Zustand des Prozessors gesichert werden. Ansonsten könnte der Prozessor die Anweisungen nicht an derselben Stelle fortsetzen
  - Betriebssysteme verwalten für jeden *thread* daher einen *thread description block*

Einführung in die nebenläufige Programmierung

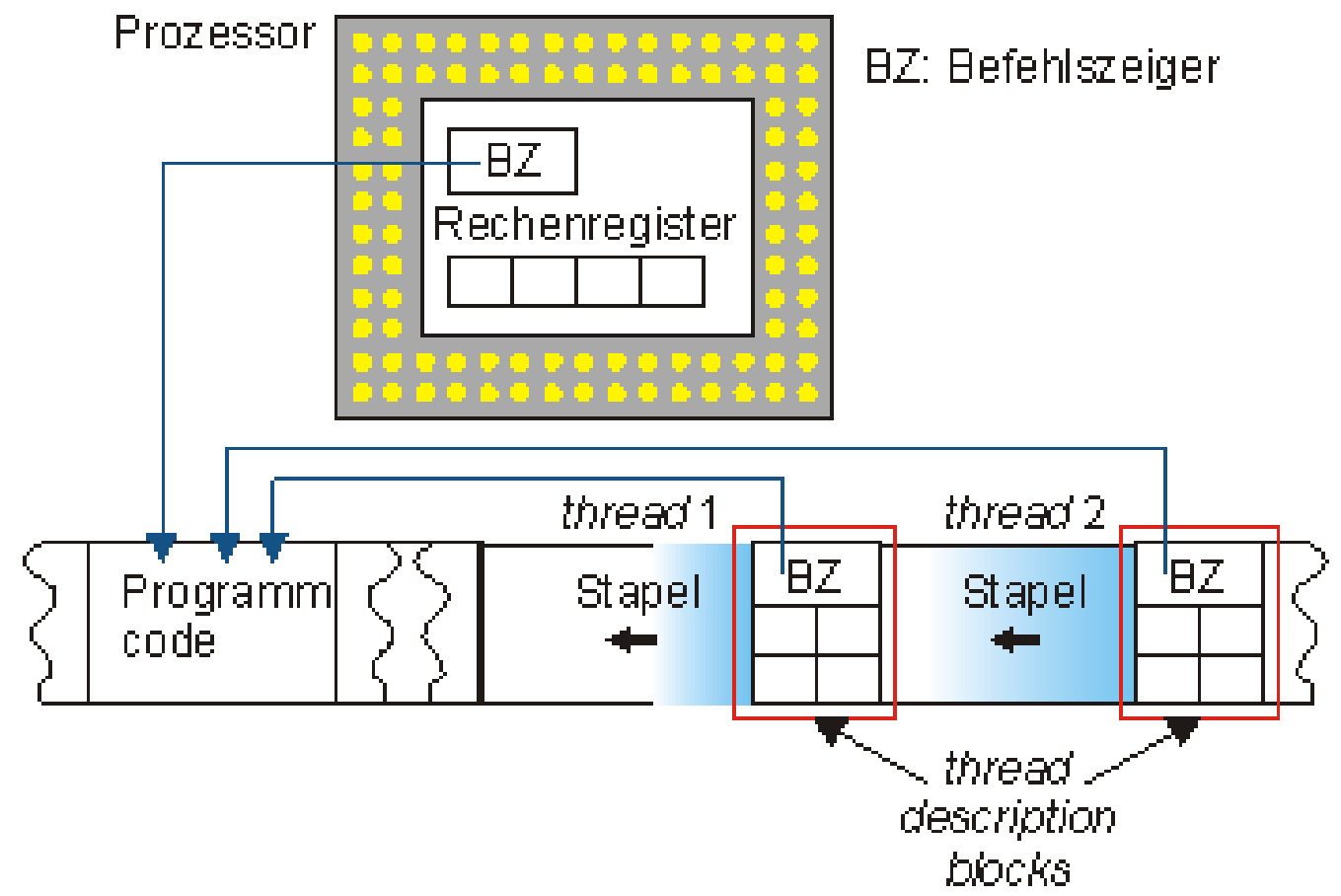
- **Veranschaulichung des Kontrollfluss-Wechsels**
  - Beispiel: Ausführung einer Addition in Pseudo-Assembler (c = a + b)

```
Move R0, a
Move R1, b
Add
Move c, R0
```



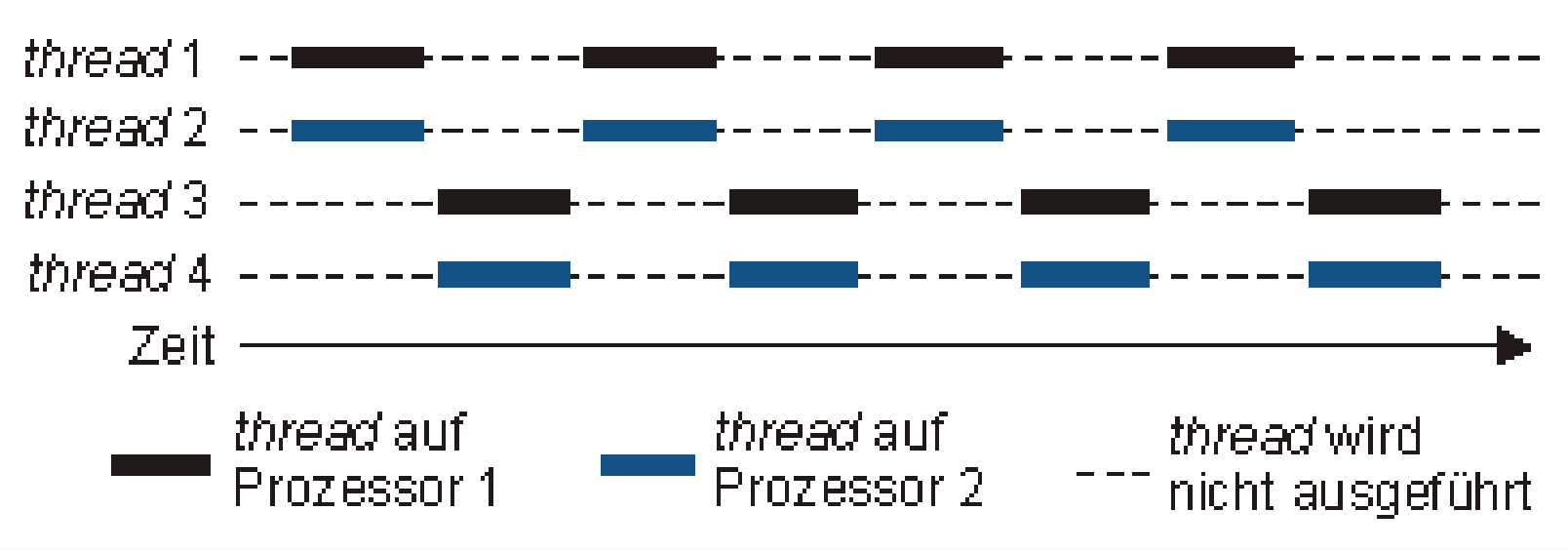
Einführung in die nebenläufige Programmierung

- **Veranschaulichung des Kontrollfluss-Wechsels**
  - Bei mehreren *threads*…



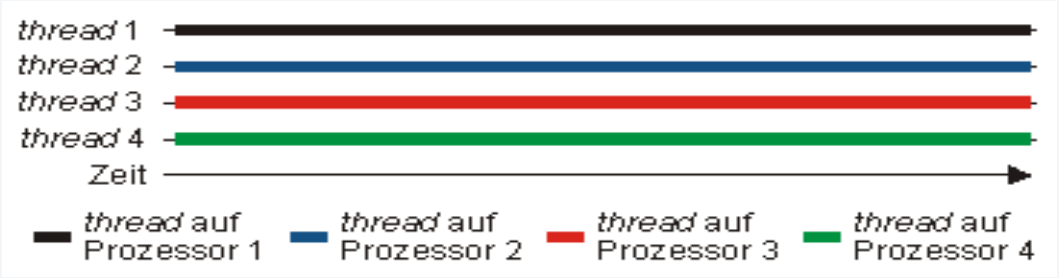
Einführung in die nebenläufige Programmierung

- ***Threads* und *scheduling***
  - Nebenläufigkeit ist nicht nur simuliert.
  - Echt parallele Abarbeitung heutzutage durch Multi-Core-Prozessoren möglich
  - Allerdings: In der Realität immer wesentlich mehr *threads* als Prozessoren
  - Konsequenzen?



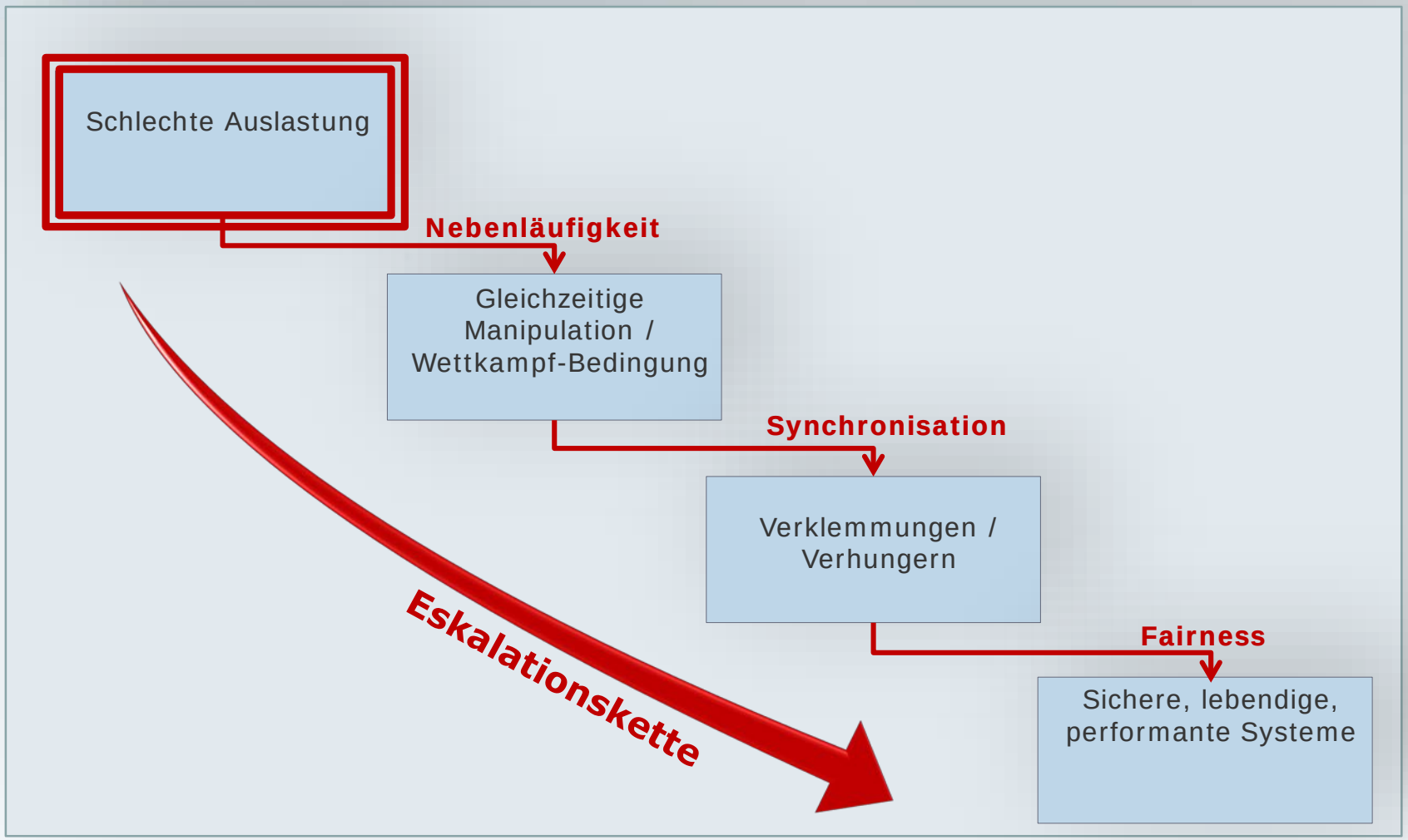
Einführung in die nebenläufige Programmierung

- ***Threads* und *scheduling***
  - *Scheduling* bezeichnet die Umschaltung eines oder mehrerer Prozessoren zwischen threads
  - Der Teil des Betriebssystem, der für die Verteilung und Planung von *threads* zuständig ist, heißt *Scheduler*.
  - Die Ausführungszeit, die ein thread vom Scheduler zugewiesen bekommt, heißt Zeitscheibe *(time slice)*
  - *Scheduling*-Algorithmen bestimmen die Schaltzeitpunkte und Zeitscheiben. Generell gilt: Betriebsgeheimnis von BS-Herstellern!
  - **Wichtig:** Ein Programmier sollte (und kann) keine Annahmen über die Ausführungsreichenfolge oder die Geschwindigkeit von *threads* treffen!
  - **Am besten Annahme:** Ein Prozessor pro *thread*



Einführung in die nebenläufige Programmierung

Motivation und roter Faden

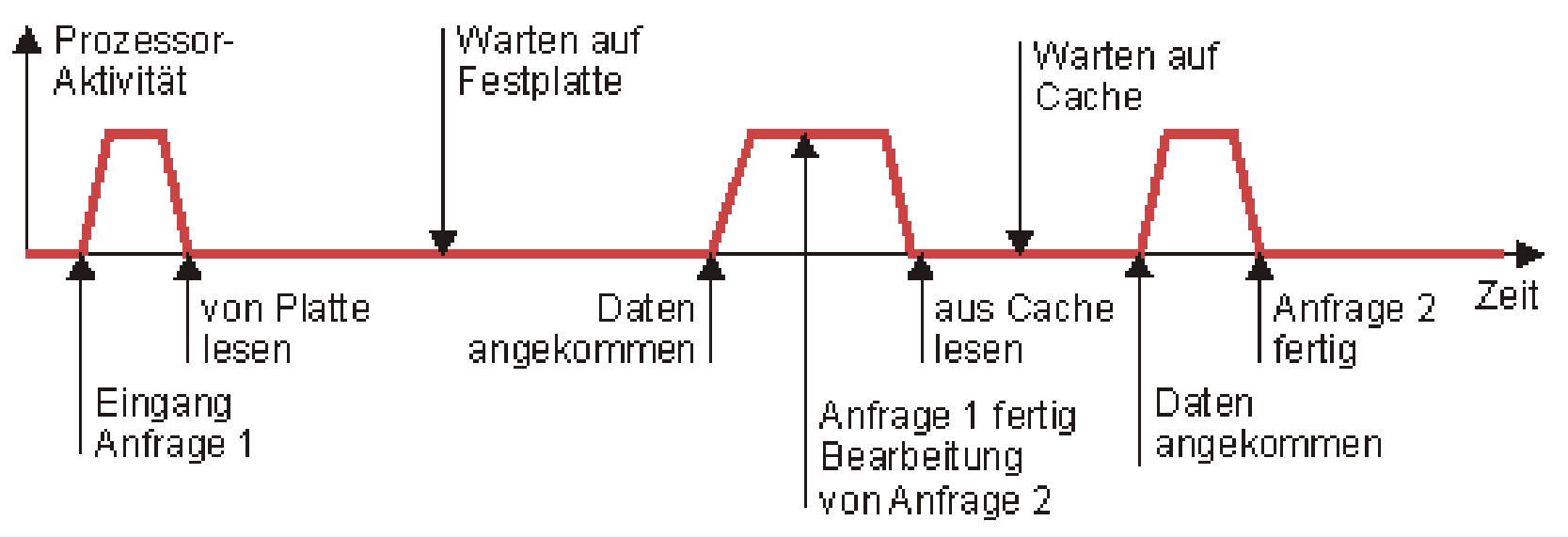


Einführung in die nebenläufige Programmierung

- **Schlechte Auslastung: Typische Problemstellungen**
  - Die Oberfläche friert während einer lange laufenden Operation ein
  - Während einer lange laufenden Operation soll der Benutzer über den Fortschritt informiert werden
  - Server antwortet nicht oder ist ausgelastet
  - Allgemein
    - Es sollen mehrere Dinge gleichzeitig getan werden
    - z.B. Sensoren überwachen und gleichzeitig Ergebnisse ausgeben
    - Annahme dabei: Der Prozessor muss häufig auf andere System-Komponenten wie z.B. die Festplatte oder das Netzwerk warten.

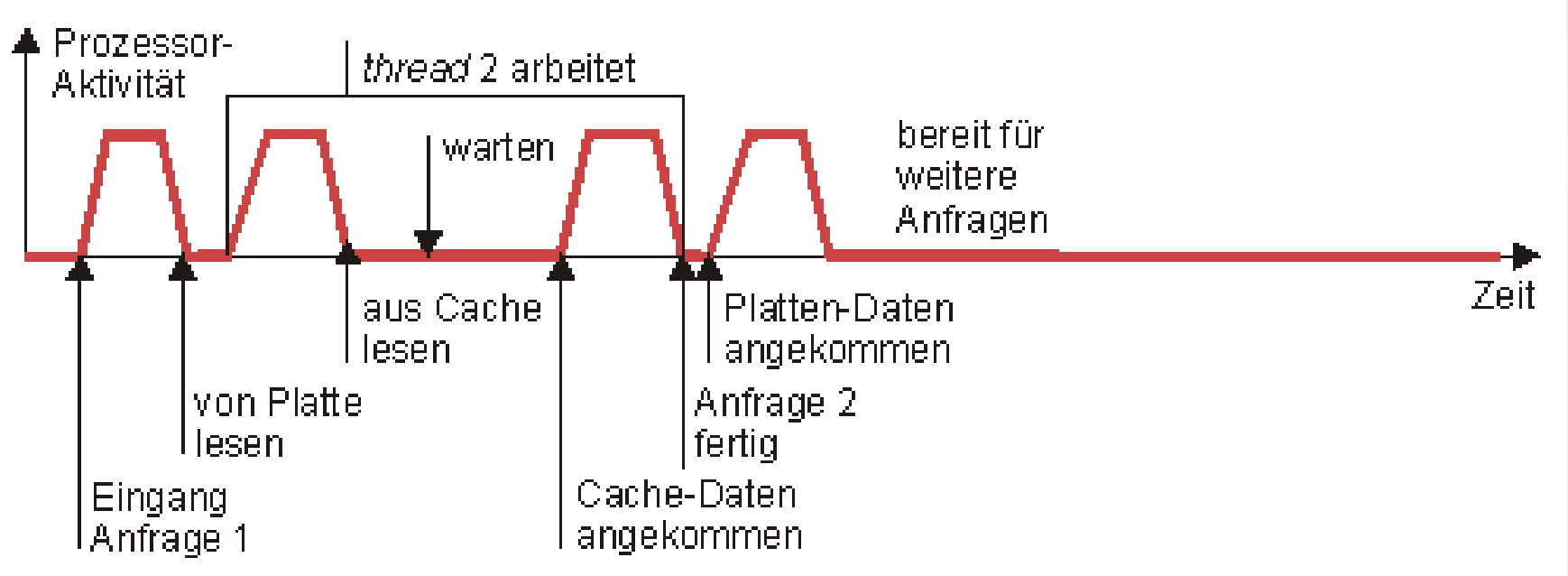
Einführung in die nebenläufige Programmierung

- **Veranschaulichung**
  - Web-Server erhält 2 Anfragen, die von 1 *thread* bearbeitet werden
  - Die Bearbeitung setzt sich aus mehreren Aktionen zusammen:
    - Berechnungen
    - Festplattenzugriffe
    - Zugriffe auf Netzwerkverbindungen (z.B. Zugriff auf DB)



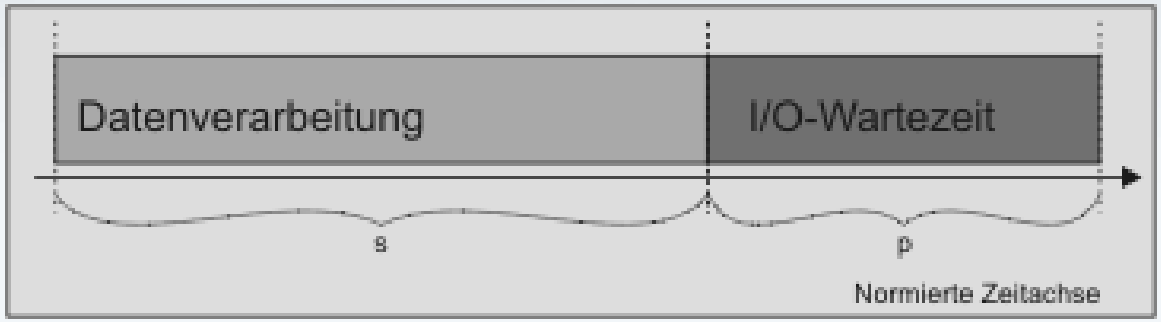
Einführung in die nebenläufige Programmierung

- **Veranschaulichung**
  - Web-Server erhält 2 Anfragen, die von 2 *threads* bearbeitet werden



Einführung in die nebenläufige Programmierung

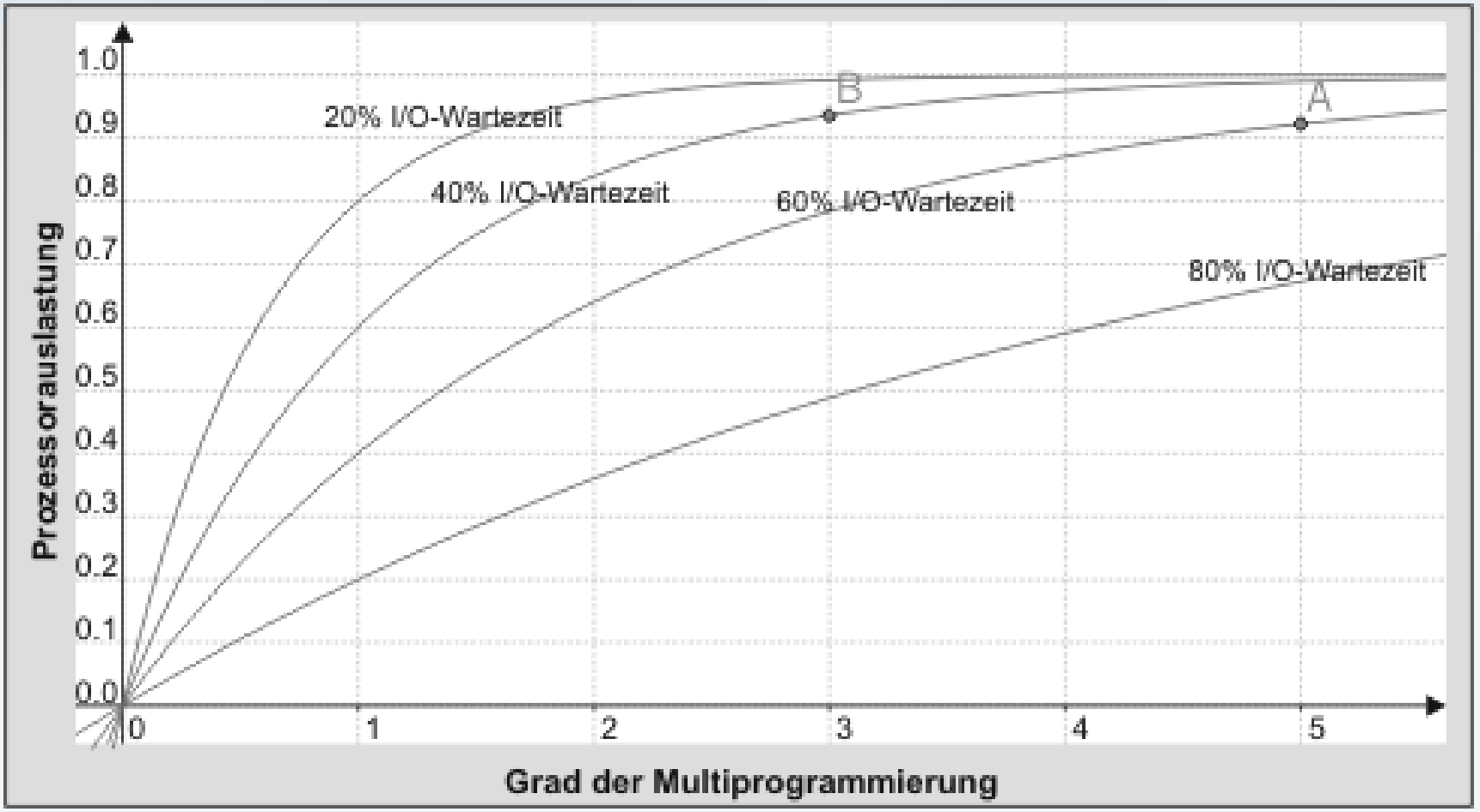
- **Grad der Multiprogrammierung**
  - Bietet ein probabilistisches Modell zur Berechnung der CPU-Auslastung



  - Anteil der Datenverarbeitung = s
  - Anteil der Wartezeit auf I/O-Vorgänge = p mit s + p = 1
  - Bei n unabhängigen threads, die dieselbe I/O-Intensität besitzen, ist die Wahrscheinlichkeit, dass alle n threads warten $\mathbf{p^n}$
  - Prozessorauslastung = $\mathbf{1 - p^n}$
  - Grad der Multiprogrammierung = n

Einführung in die nebenläufige Programmierung

Grad der Multiprogrammierung

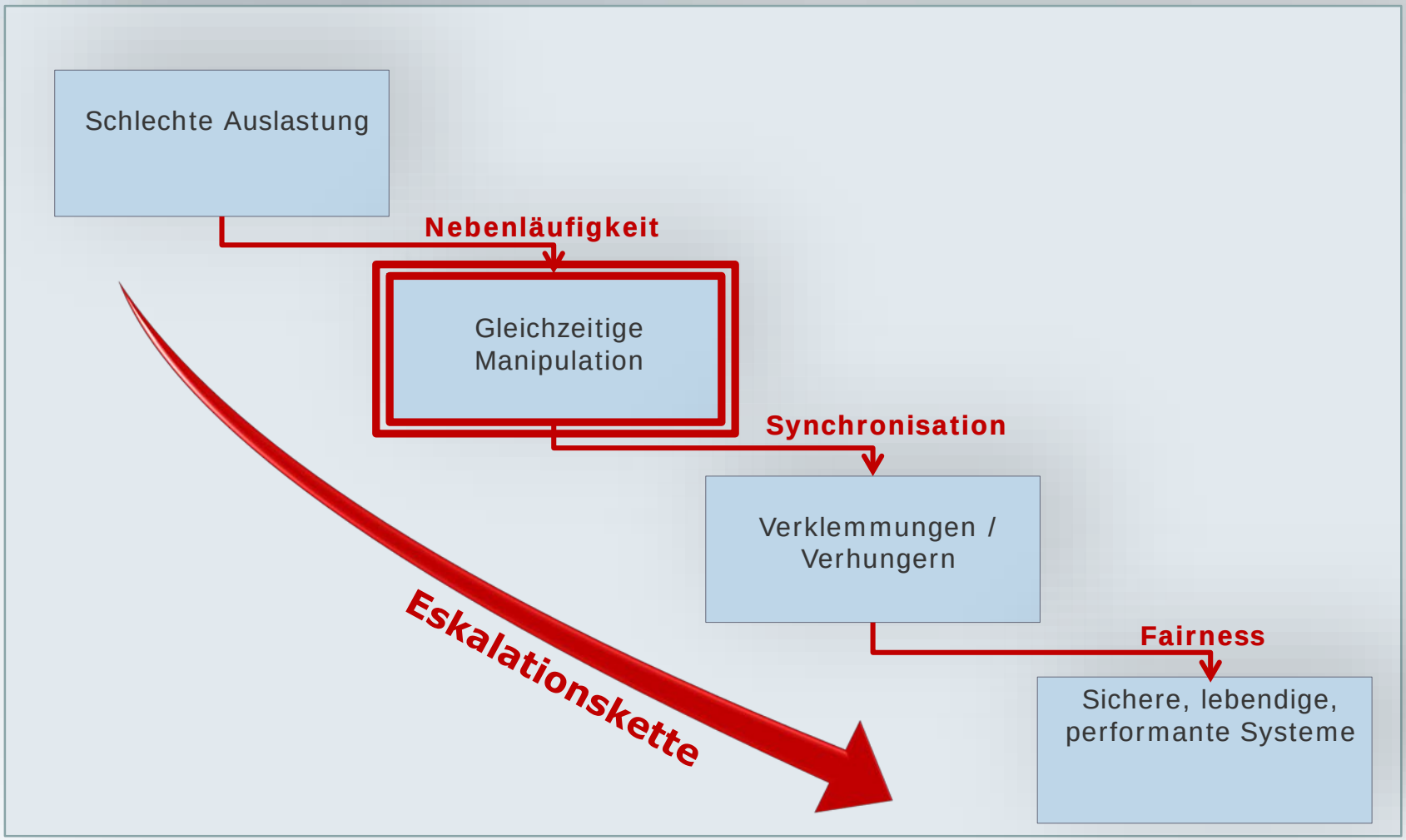


Einführung in die nebenläufige Programmierung

- **Amdahlsches Gesetz**
  - Nach Amdahl ist der Geschwindigkeitszuwachs eines Algorithmus unter Einsatz der nebenläufigen Programmierung lediglich von den kausal abhängigen Aktionen, also den *nicht* nebenläufigen, *nicht* parallelisierbaren und demnach sequenziellen Aktionen beschränkt!
  - Laufzeit setzt sich zusammen aus
    - $\mathbf{Anteil_{sequenziell} + Anteil_{nebenläufig} = T_{gesamt}}$
  - Beim Einsatz der nebenläufigen Programmierung und einem System mit n Prozessoren
    - $\mathbf{Anteil_{sequenziell} + Anteil_{nebenläufig} / n = T_{parallel}}$
  - $\mathbf{T_{parallel} < T_{gesamt}}$ für $n > 1$
  - Geschwindigkeitszuwachs nach dem Modell von Amdahl
  - $\mathbf{S = 1 / Anteil_{sequenziell}}$
  - Beispiel
    - Ein Algorithmus hat einen sequenziellen Anteil von 25 Prozent.
    - Maximaler Geschwindigkeitszuwachs ist daher S = 4!

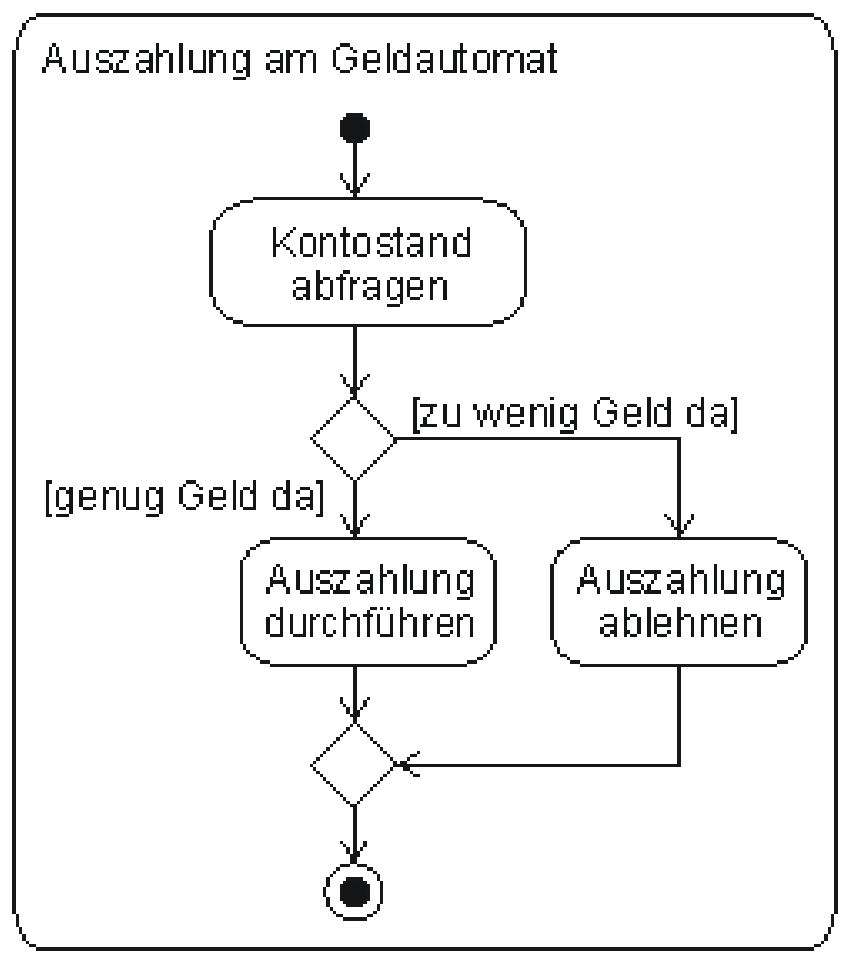
Einführung in die nebenläufige Programmierung

Motivation und roter Faden



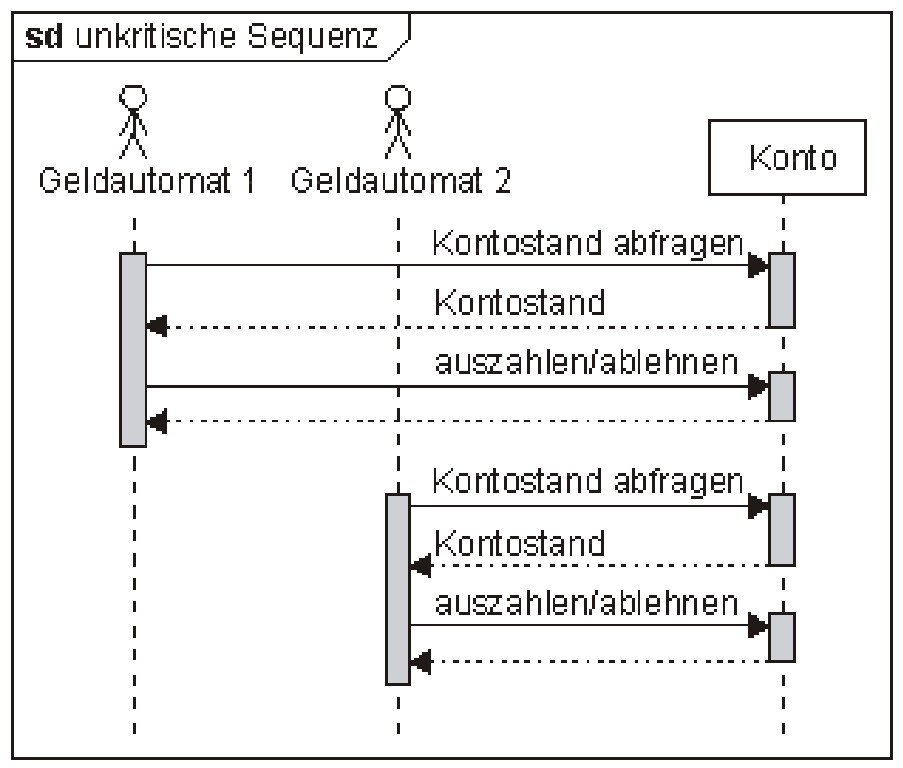
Einführung in die nebenläufige Programmierung

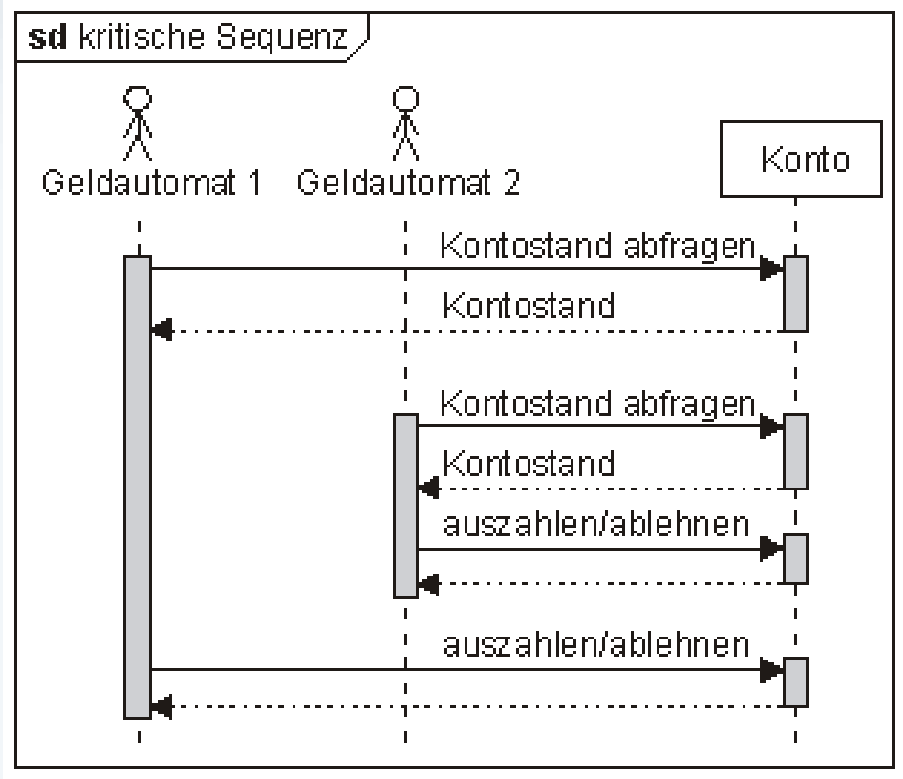
- **Beispiel**
  - Abheben von Geld an einem Geldautomaten



Einführung in die nebenläufige Programmierung

- **Beispiel**
  - Abheben von Geld an einem Geldautomaten



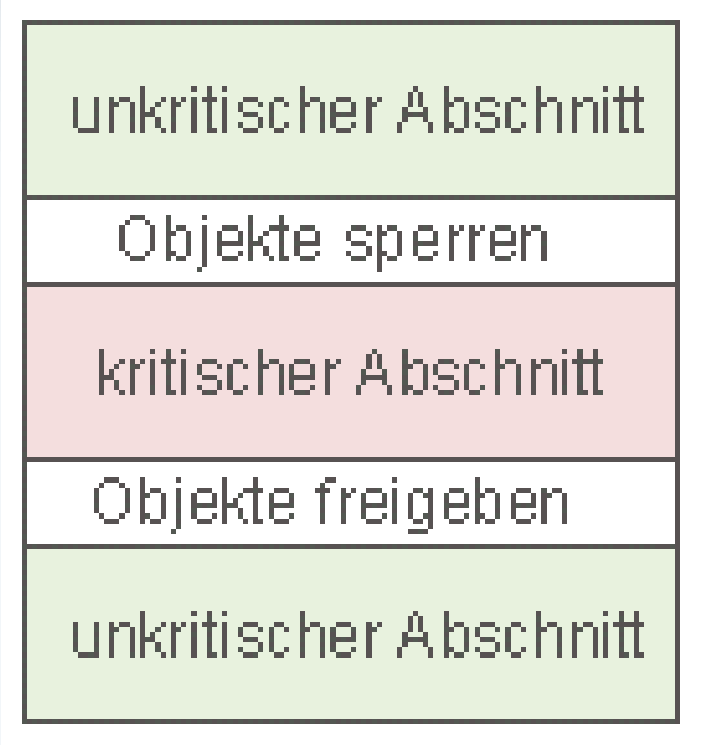


Einführung in die nebenläufige Programmierung

- **Zusammenfassung**
  - Ausführungs-Reihenfolge eines nebenläufigen Programms ist nicht deterministisch (**Wettkampf-Bedingung**)
  - Eine Ausführungs-Reihenfolge, bei der es zu einem fehlerhaften Verhalten kommt, wird **kritische Sequenz** genannt
  - Gibt es eine solche kritische Sequenz, dann hängt die Korrektheit eines Programms vom *scheduling* des Betriebssystems ab
  - Ein Programm sollte daher so gestaltet sein, dass keine kritischen Sequenzen möglich sind!
  - Dazu werden **kritische Abschnitte** definiert, deren Ausführung nicht durch das Umschalten zwischen *threads* gestört werden darf und kann.

Einführung in die nebenläufige Programmierung

- **Kritische Abschnitte**



Einführung in die nebenläufige Programmierung

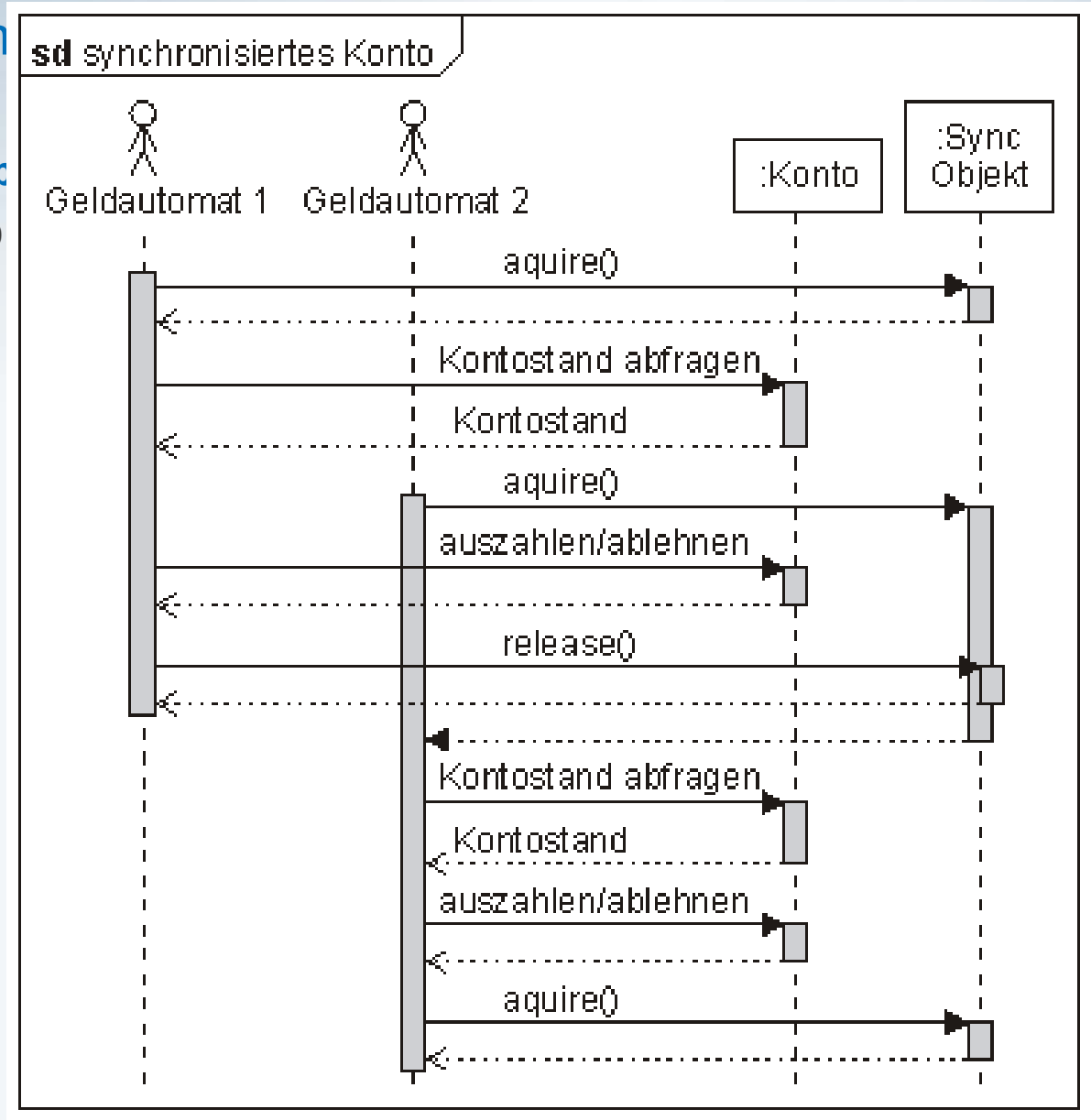
- **Beispiel**
  - Abheben von Geld an einem Geldautomaten

```
syncObjekt.acquire()
if (Deckung ausreichend)
{
        Auszahlung durchführen
}
else
{
        Auszahlung ablehnen
}
syncObjekt.release()
```

Kritischer Abschnitt

Einfüh ng

- **Beisp**
  - Ab



Einführung in die nebenläufige Programmierung

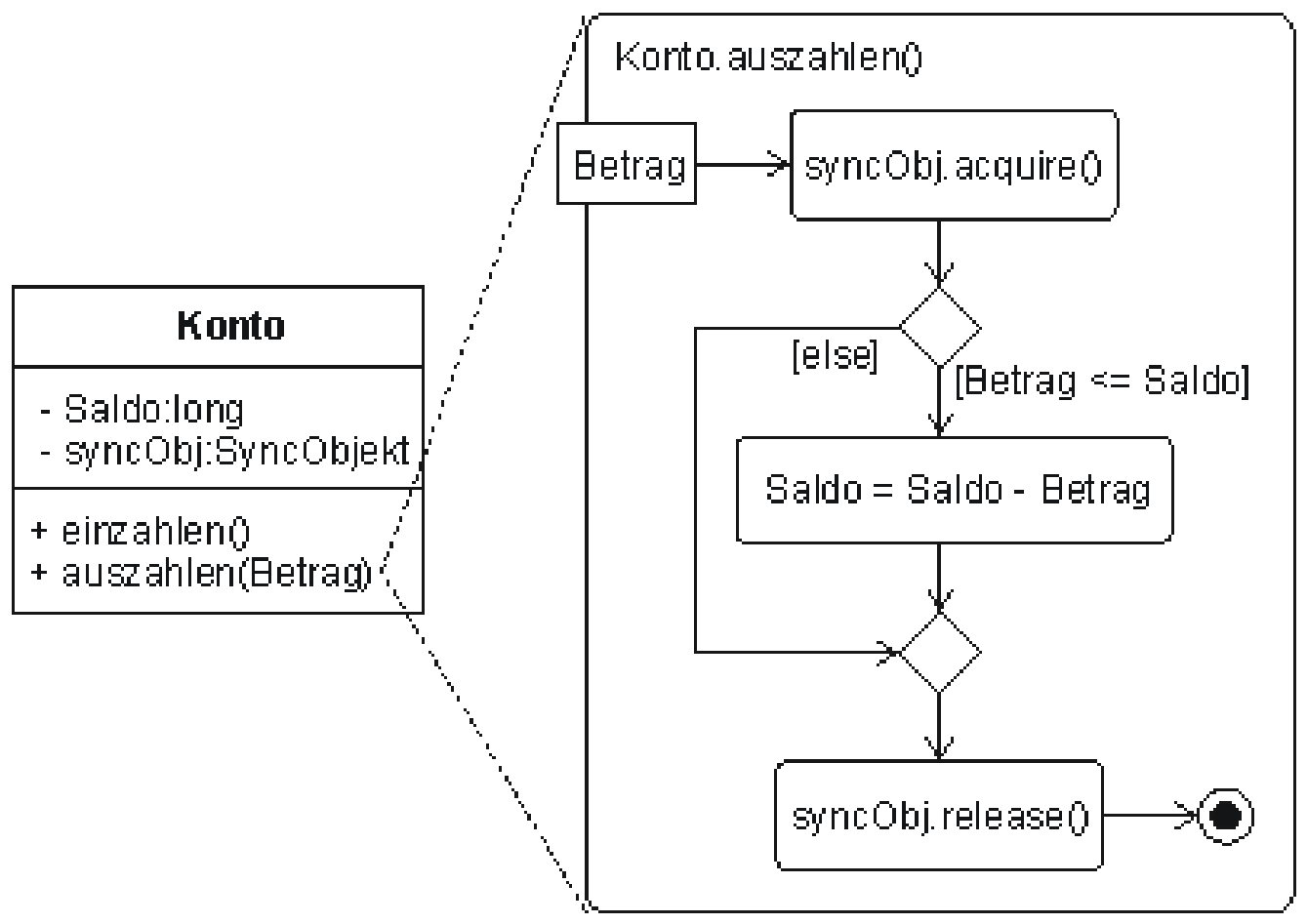
- **Typische Synchronisations-Objekte**
  - **Mutex**
    - Wechselseitiger Ausschluss. Nur ein Besitzer!
  - **Semaphor**
    - Entwickelt in den 1960er Jahren. Es können mehrere *threads* Besitzer eines Semaphors werden. Ein Mutex ist ein Spezialfall eines Semaphors.
  - **Spinlock**
    - Blockiert den aufrufenden *thread* gewöhnlich nicht. Stattdessen fragt der aufrufende *thread* in einer Schleife kontinuierlich den Status des Synchronisations-Objektes ab. I.d.R. sinnvoll bei Mehrprozessor-Systemen. Aber nicht darauf begrenzt.

Einführung in die nebenläufige Programmierung

- **Strukturierung von Synchronisations-Objekten**
  - Gewöhnlich gibt es in einer Anwendung mehrere kritische Abschnitte, die sich bzgl. derselben Daten synchronisieren müssen
  - Ein Synchronisations-Objekt muss dann an verschiedenen Stellen im Programm verwendet werden
  - Beispiel
    - Neben Auszahlungen können auch Überweisungen das Konto überziehen. Die Auszahlung und die Überweisung müssen durch dasselbe Synchronisations-Objekt gesichert werden
  - Kritische Abschnitte und Synchronisations-Objekte sind an vielen Stellen verstreut
    - **Konsequenzen**: Nachvollziehbarkeit sinkt, Fehleranfälligkeit steigt, Wartbarkeit sinkt

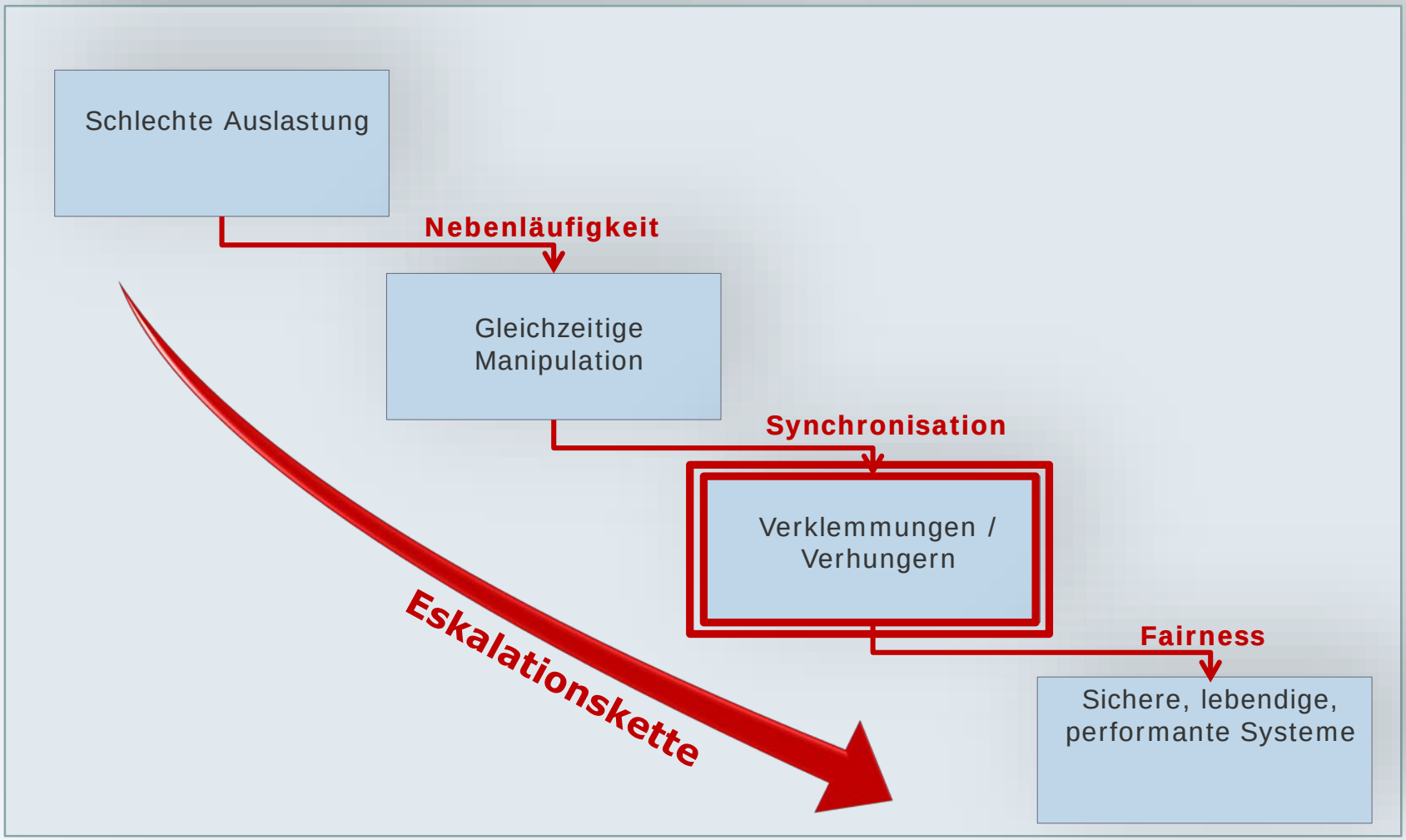
Einführung in die nebenläufige Programmierung

- **Beispiel**
  - Abheben von Geld an einem Geldautomaten



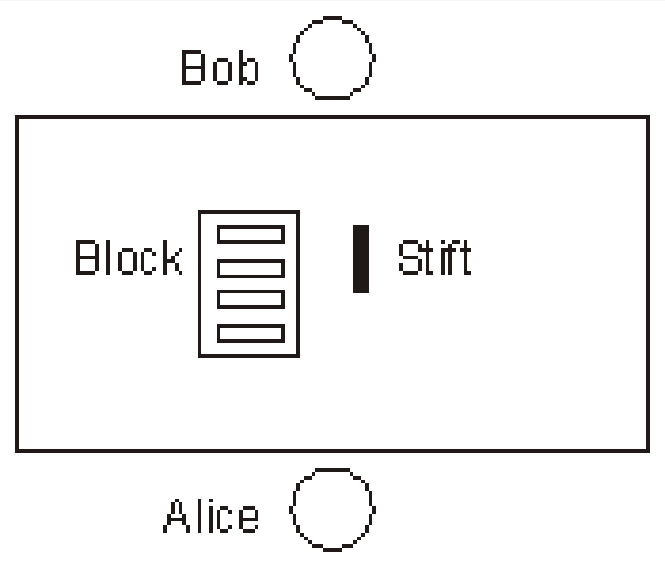
Einführung in die nebenläufige Programmierung

Motivation und roter Faden



Einführung in die nebenläufige Programmierung

- **Verklemmungen**
  - Eine besondere Form des Verhungerns mehrerer *threads* sind Verklemmungen.
    - Eine Verklemmung zwischen zwei *threads* liegt vor, wenn ein *thread* ein Betriebsmittel reserviert hat, das der andere benötigt und umgekehrt.
  - Beispiel: Stift und Papier



Einführung in die nebenläufige Programmierung

- **Definition für Verklemmung**
  - Eine Gruppe von *threads* befindet sich in einer Verklemmung *(deadlock)*, wenn sie (blockierend) auf ein Ereignis warten, das **nur ein thread innerhalb der Gruppe** auslösen kann.
- **Notwendige Bedingungen für Verklemmungen**
  - Halten und Warten / Nachforderung
    - Jeder *thread* hat bereits ein Betriebsmittel für sich reserviert und möchte ein Weiteres reservieren.
  - Keine Unterbrechung
    - Einem *thread* darf ein einmal reserviertes Betriebsmittel nicht wieder entzogen werden können.
  - Zyklisches Warten
    - Es gibt eine Kette von *threads,* in der der Erste auf ein Betriebsmittel des Zweiten wartet und so fort, wobei der letzte wieder auf den ersten thread wartet.
  - Wechselseitiger Ausschluss
    - Ein *thread* muss ein Betriebsmittel exklusiv sperren.

Schnelleinstieg

NEBENLÄUFIGE PROGRAMMIERUNG IN DER PRAXIS

Schnelleinstieg: Threads in Java

- **Einführendes Beispiel: »Hello World«**
  - Java und .NET unterstützen Nebenläufigkeit auf Sprach- und Bibliotheksebene
  - Konsequenz: Nebenläufige Programmierung wird besonders einfach!
  - Schritte zum Starten von Java-threads (es gibt auch eine andere Möglichkeit)
    - Klasse erstellen, die von `java.lang.Thread` erbt.
    - Die geerbte Operation `public void run()` überschreiben.
    - Ein Exemplar dieser Klasse erzeugen…
    - … und die geerbte Operation `start()` aufrufen. Die JVM startet daraufhin unter Zuhilfenahme des Betriebssystems einen neuen *thread*.
    - Der neue thread arbeitet die Operation `publich void run()` ab und endet automatisch beim Verlassen dieser Operation.

Schnelleinstieg: Threads in Java

- **Einführendes Beispiel: »Hello World« in Java**

```
public class HelloWorld extends Thread
{
  public void run()
  {
    for( int i = 0; i < 10000; i++ )
        System.out.print( "Hello " + i + "  " );
  }

  public static void main( String[] args )
  {
    HelloWorld neuerThread = new HelloWorld();
    neuerThread.start();

    for( int i=0; i < 10000; i++ )
      System.out.print( "World " + i + "  " );

  }
}
```

LE2: Schnelleinstieg: Threads in Java

- **Ausführung einzelner Operationen durch einen separaten thread**
  - Runnable-Schnittstelle in Verbindung mit anonymen Klassen.
  - Prinzipielle Vorgehensweise:

```
class OperationAlsNeuerThread
{
  public void operation1()
  {
    Thread t = new Thread( new Runnable(){
      public void run()
      {
        eineOperation();
      }}).start();
  }

  private void eineOperation()
  {
    //Code, der von einem separaten thread
    //ausgeführt wird
  }
}
```

LE2: Schnelleinstieg: Threads in Java

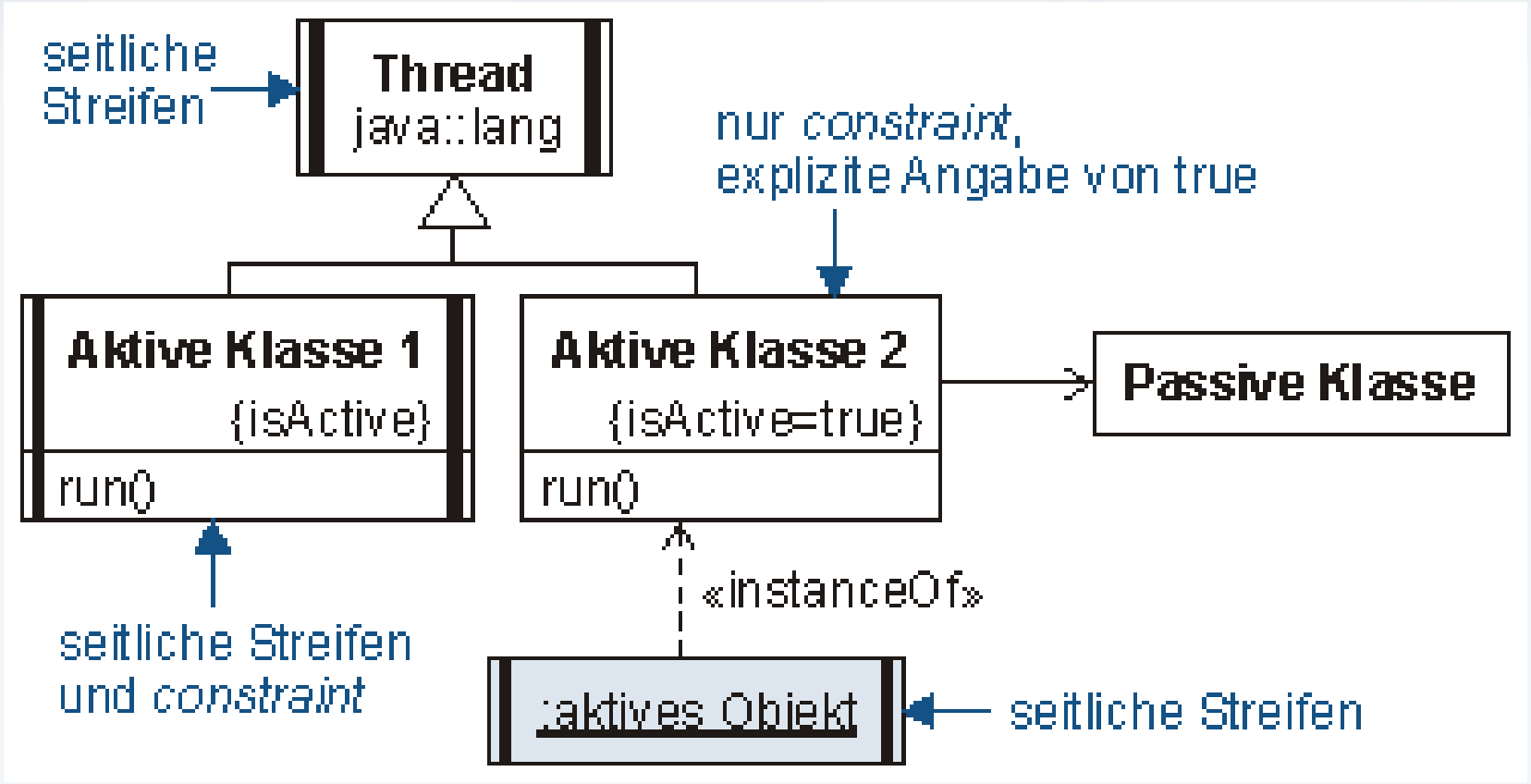
- **Entwurf nebenläufiger Anwendungen**
  - Man muss zwischen aktiven und passiven Klassen unterscheiden
    - Aktive Klassen kapseln und repräsentieren *threads*
    - Passive Klassen enthalten Daten und die Logik der Anwendung
  - Aktive Klassen rufen die Operationen der passiven Klassen auf.
  - Entwicklungsprozess
    - Nebenläufige Anwendungen zunächst wie nicht nebenläufige entwickeln
    - Also beim Entwurf des Fachkonzepts (nur in Grenzen möglich) zunächst auf die Berücksichtigung mehrere *threads* verzichten.
    - Entscheiden, an welchen Stellen Parallelität eingeführt werden kann bzw. muss

LE2: Schnelleinstieg: Threads in Java

- **Nachträgliche Betrachtung der Nebenläufigkeit**
  - Nach Fertigstellung des Fachkonzepts gibt es 3 Möglichkeiten
    - Aktive Klassen werden nachträglich dem Modell hinzugefügt
    - Vorhandene Klassen werden in aktive umgewandelt
    - Die Verwendung von threads findet keinen Niederschlag im Modell
  - Vorteil
    - Etablierte Vorgehensmodelle für die Modellierung des Fachkonzepts können verwendet werden.
    - Das Fachkonzept bleibt zunächst einfacher und leichter verständlich.
    - Nebenläufigkeit führt zu wahrscheinlichen Änderungen an der Anwendung. Diese sollten aber grundsätzlich nicht die Logik des Programms verändern.

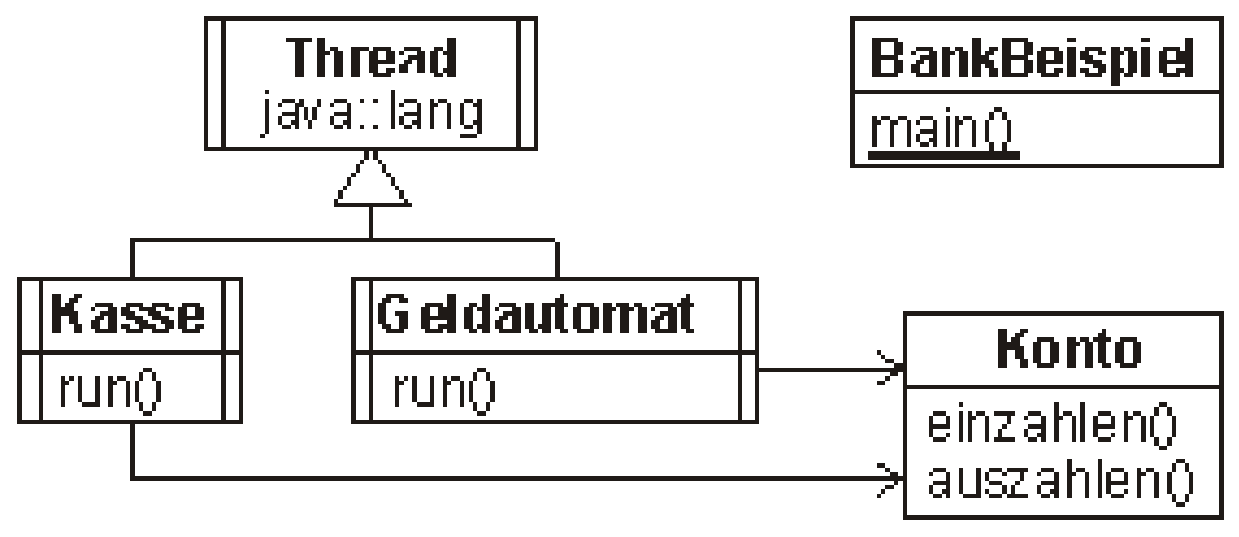
LE2: Schnelleinstieg: Threads in Java

- **Aktive Klassen in der UML**
  - Notation mit senkrechten Linien ist in den meisten Fällen vorzuziehen



Schnelleinstieg: Threads in Java

- **Fallbeispiel: Geldautomat in Java**
  - Eine Klasse `Konto` erlaubt das Ein- und Auszahlen.
  - Beim Auszahlen wird überprüft, ob genug Geld auf dem Konto ist.
  - Zwei Klassen, `Kasse` und `Geldautomat` kapseln threads.
    - An der Kasse wird Geld eingezahlt, welches …
    - … am Geldautomaten abgehoben werden kann.



Schnelleinstieg: Threads in Java

Die Klasse Konto

```
public class Konto
{
  private int Saldo;
  public int getSaldo()
  {
    return this.Saldo;
  }

  public void einzahlen( int Betrag )
  {
    this.Saldo = this.Saldo + Betrag;
  }

  public void auszahlen( int Betrag )
  {
    if( Saldo >= Betrag )
    {
      this.Saldo = this.Saldo - Betrag;
    }
  }
}
```

Schnelleinstieg: Threads in Java

Die Klasse Kasse

```
public class Kasse extends Thread
{
  private Konto einKonto;

  public Kasse( Konto einKonto )
  {
    this.einKonto = einKonto
  }

  public void run()
  {
    einKonto.einzahlen( 15000 ); //150 Euro einzahlen
  }
}
```

Schnelleinstieg: Threads in Java

Die Klasse Geldautomat

```
public class Geldautomat extends Thread
{
  private Konto einKonto;

  public Geldautomat( Konto einKonto )
  {
    this.einKonto = einKonto
  }

  public void run()
  {
    einKonto.auszahlen( 10000 ); //100 Euro auszahlen
  }
}
```

Schnelleinstieg: Threads in Java

Die Klasse Geldautomat

```
public class BankBeispiel
{
  public static void main( ... )
  {
    //allle benötigten Objekte erzeugen
    Konto einKonto  = new Konto();
    Kasse eineKasse = new Kasse( einKonto );
    Geldautomat ga1 = new Geldautomat( einKonto );
    Geldautomat ga2 = new Geldautomat( einKonto );

    //threads starten
    eineKasse.start();
    ga1.start();
    ga2.start();

    try {
      Thread.sleep( 2000 );
    } catch( InterruptedException e ) { }

    System.out.println( "Kontostand: " + einKonto.getSaldo());
  }
}
```

Schnelleinstieg: Threads in Java

- **Ablaufsequenz**
  - Mögliche Ablaufsequenz der Anwendung

Zeit	*thread* ga1	*thread* ga2	*thread* einekasse	einKonto.Saldo
				0
			einzahlen(15000)	15000
	if(Saldo >= 10000)			15000
		if(Saldo >= 10000)		15000
	Saldo = Saldo - 10000			5000
		Saldo = Saldo - 10000		-5000

  - Zusammenfassung
    - Der Algorithmus ist an sich korrekt.
    - Unter nicht nebenläufigen Bedingungen funktioniert alles einwandfrei.
  - Wenn das Ergebnis der nebenläufigen Ausführung eines Programms von der nichtdeterministischen Umschaltung zwischen den *threads* abhängt, spricht man von einer **Wettkampfbedingung** zwischen *threads*.

Schnelleinstieg: Threads in Java

Die Klasse Konto in Java

```
public class Konto
{
  private int Saldo;
  public synchronized int getSaldo()
  {
    return this.Saldo;
  }

  public synchronized void einzahlen( int Betrag )
  {
    this.Saldo = this.Saldo + Betrag;
  }

  public synchronized void auszahlen( int Betrag )
  {
    if( Saldo >= Betrag )
    {
      this.Saldo = this.Saldo - Betrag;
    }
  }
}
```

Datenstrukturen

NEBENLÄUFIGE PROGRAMMIERUNG IN DER PRAXIS

Neues zur Nebenläufigkeit in Java

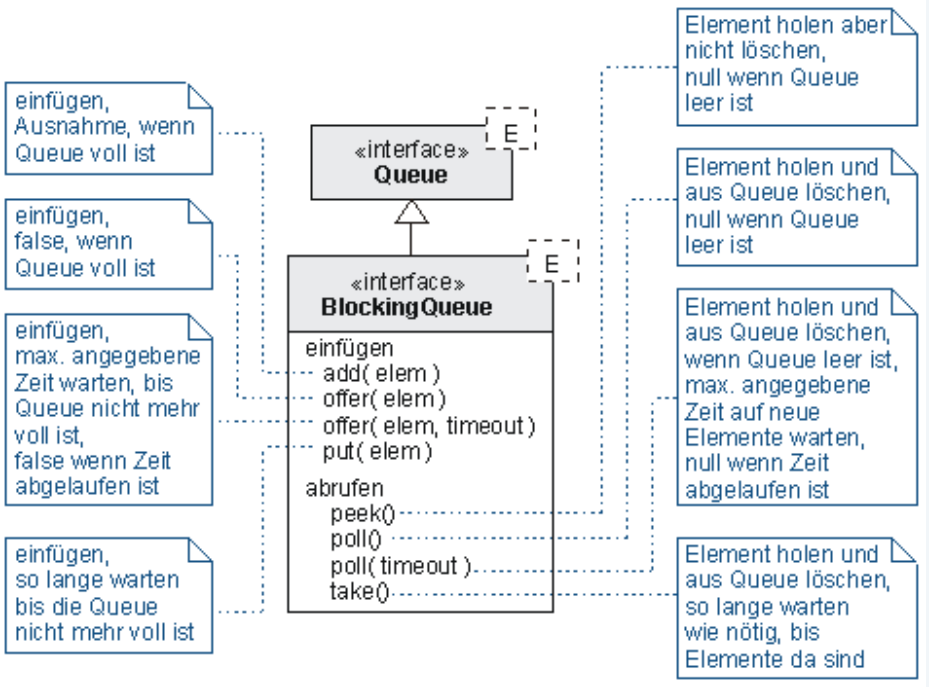
- **Warteschlangen**
  - Die Schnittstelle `BlockingQueue` definiert Warteschlangen, über die mehrere *threads* Daten austauschen können.
  - Die Operationen zum Einfügen und Auslesen von Elementen gibt es jeweils in
    - blockierenden,
    - nicht blockierenden und
    - zeitlich begrenzt blockierenden Varianten.
  - Implementierungen von BlockingQueue arbeiten mit
    - Arrays,
    - verketteten Listen oder
    - ohne Pufferung von Daten.

Neues zur Nebenläufigkeit in Java

- **Warteschlangen**
  - Eine Warteschlange *(queue)* ist für gewöhnlich ein Container, bei dem Elemente nur am Ende eingefügt und nur am Anfang entnommen werden können.
  - Oft realisieren Warteschlagen einen FIFO-Speicher *(first in first out)*.
  - Es gibt aber auch andere Speicherformen, bei denen Elemente z.B. unterschiedliche Prioritäten haben können.
  - **Entkopplung**
    - Warteschlangen dienen meist der Entkopplung, d.h. der losen Kopplung von Komponenten oder Systemen.
    - Die gekoppelten Systeme bleiben unabhängiger voneinander als dies bei direkten Operationsaufrufen zwischen ihnen der Fall wäre.
    - Ein Nachteil der Warteschlangen-basierten Kommunikation ist die geringe Geschwindigkeit eines einzelnen Kommunikations-Schrittes gegenüber einem direkten Operationsaufruf.
    - Warteschlangen-basierte Kommunikation ist nur dort möglich, wo die Kommunikation nur in eine Richtung verläuft, da ansonsten für die Rückgabe von Ergebnissen ein erheblicher Aufwand erforderlich ist.

Neues zur Nebenläufigkeit in Java

- **Nebenläufige Warteschlangen**
  - In `java.util.concurrent` gibt es eine Reihe von Warteschlangen, die für den gleichzeitigen Zugriff durch mehrere *threads* ausgelegt sind.
  - Sie dienen der Kopplung von *threads* in Form von Konsumenten und Produzenten.
  - Als gemeinsame Schnittstelle dient `BlockingQueue`, welche von der bekannten Schnittstelle `java.util.Queue` erbt.



Neues zur Nebenläufigkeit in Java

- **Implementierungen von `BlockingQueue`**
  - `SynchronousQueue`
    - Einfachste Warteschlange. Genau genommen ist es gar keine Warteschlange, denn ein `SynchronousQueue`-Objekt hat keinen internen Container zum Speichern von Elementen.
    - Jeder Aufruf der `put()`-Operation blockiert den aufrufenden *thread* so lange, bis ein Aufruf von `take()` ihn wieder weckt.
  - `ArrayBlockingQueue`
    - Verwaltet intern ein Array fester Größe, in das die Elemente in FIFO-Reihenfolge abgelegt werden.
  - `LinkedBlockingQueue`
    - Eingefügte Element werden als verkettete Liste verwaltet. Auch sie arbeitet nach dem FIFO-Prinzip. Standardmäßig ist die Kapazität nicht eingeschränkt.
  - `PriorityBlockingQueue`

Neues zur Nebenläufigkeit in Java

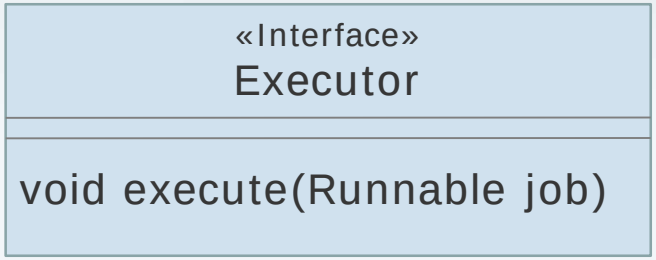
- **Auftragsorientierte Architektur**
  - Ein weiterer Bestandteil von `java.util.concurrent` ist ein Framework zur Erzeugung, Verwaltung und nebenläufigen Ausführung von Aufträgen *(tasks)*.
  - Server-Architektur
    - Häufig arbeiten Server-Anwendungen rein auftragsorientiert.
    - Die Anwendung wartet in einer Endlosschleife darauf, dass von Clients Anfragen eingehen.
    - Diese Anfragen werden als Auftrag aufgefasst und ausgeführt.
    - Das Ergebnis eines Auftrags wird an den anfragenden Client zurückgeschickt.
    - Die Aufträge sind meist unabhängig voneinander.

Neues zur Nebenläufigkeit in Java

- **Auftragsorientierte Architektur**
  - In Java stehen ab der Version 5 Klassen zur einfachen Realisierung einer auftragsorientierten Architektur zur Verfügung. Sie lassen sich in zwei Kategorien aufteilen
    - **Ausführung von Aufträgen ohne Ergebnis**
      Für die Ausführung stehen die Schnittstellen Executor und ExecutorService zur Verfügung. Ihre Hauptaufgabe ist die Entkopplung der Aufträge von den threads, die diese Aufträge ausführen.
    - **Ausführung von Aufträgen mit Ergebnis**
      Für Aufträge, die nach ihrer Ausführung ein Ergebnis liefern, existiert die Schnittstelle Future.

Neues zur Nebenläufigkeit in Java

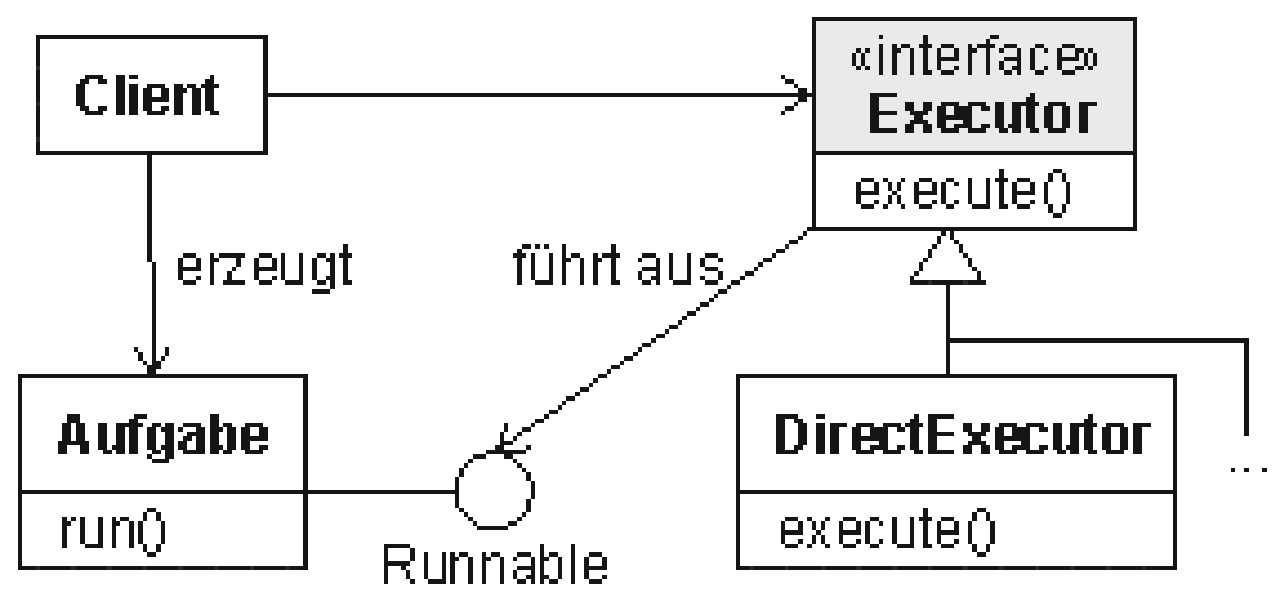
- **Auftragsorientierte Architektur**
  - Die Schnittstelle `java.util.concurrent.Executor` definiert die zentrale Anlaufstelle für Aufträge, die an den Ausführungsdienst gerichtet werden.



  - Durch den Aufruf von `execute()` wird der in Form eines `Runnable`-Objekts übergebene Auftrag ausgeführt.
  - Diese Funktion erinnert stark an die Klasse Thread, die im Konstruktor ebenfalls ein Runnable-Objekt annehmen kann.
  - Was könnte der wesentliche Unterschied sein?

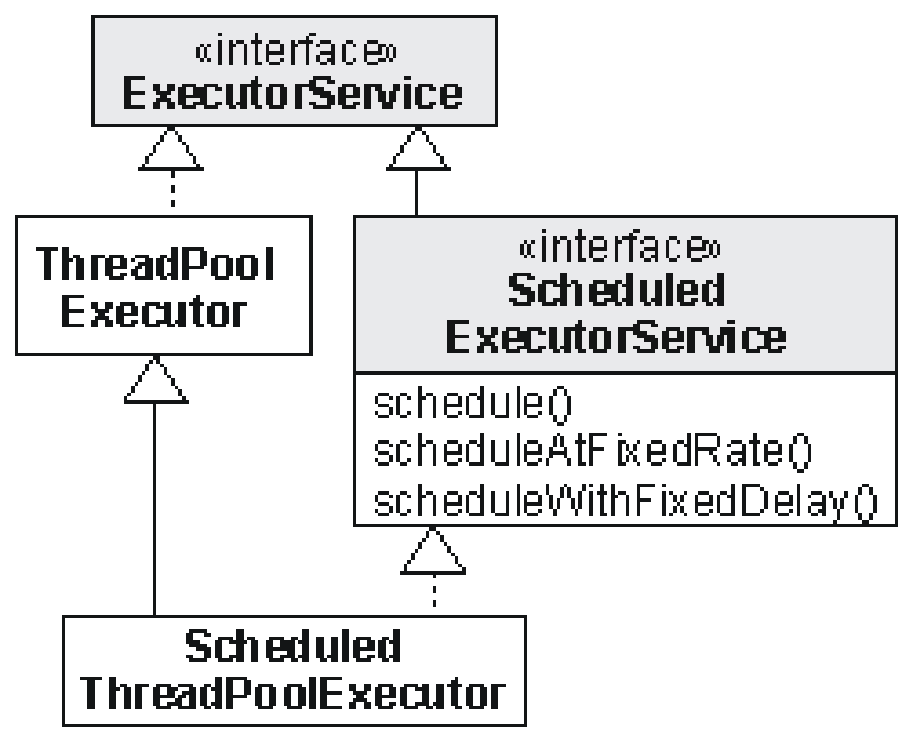
Neues zur Nebenläufigkeit in Java

- **Auftragsorientierte Architektur**
  - Der Unterschied ist, dass die Schnittstelle `Executor` keine Aussage darüber macht, welcher *thread* den Auftrag ausführt, während `Thread` immer einen neuen *thread* zur Ausführung nutzt.
  - Das Erteilen von Aufträgen wird mit `Executor` von der Strategie zur Verteilung der Aufträge auf threads entkoppelt.



Neues zur Nebenläufigkeit in Java

- **Implementierungen von ExecutorService**
  - Die Schnittstelle `ExecutorService` wird von der abstrakten Klasse `AbstractExecutorService` implementiert.
  - Wichtige von ihr abgeleitete Klassen sind `ThreadPoolExecutor` und `ScheduledThreadPoolExecutor`.



Neues zur Nebenläufigkeit in Java

- **Implementierungen von ExecutorService**
  - *thread pools*
    - Die Klasse `ThreadPoolExecutor` verwaltet intern einen *thread pool*.
    - Für die auszuführenden Aufträge wird nur eine bestimmte Anzahl von *threads* erzeugt.
    - Stehen mehr Aufträge zur Bearbeitung an, werden sie nur nacheinander abgearbeitet.
    - Hat ein *thread* einen Auftrag abgearbeitet, holt er sich den nächsten anstehenden Auftrag.
  - Funktionsprinzip eines *thread pools* ähnelt einem Restaurant.
    - Anzahl der Bedienungen, die Essen zu den Gästen bringen, ist fix.
    - Produziert die Küche mehr Essen als die Bedienungen ausliefern können, stauen sich die Teller für einen Moment.
    - Es werden aber kurzfristig keine neuen Bedienungen eingestellt.
    - Ebenfalls werden keine entlassen, falls kurzfristig kein Essen ausgeliefert werden muss.

Neues zur Nebenläufigkeit in Java

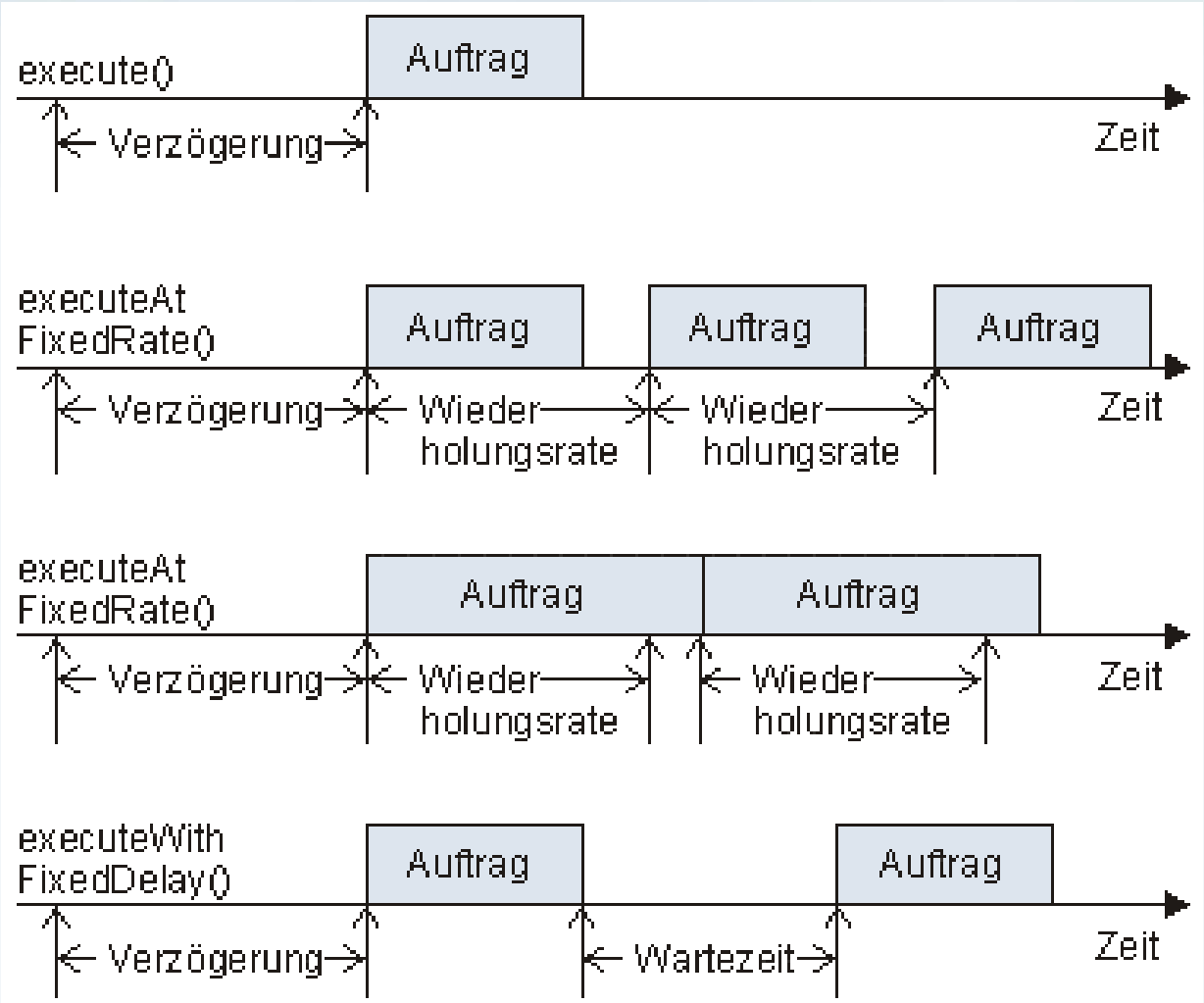
- **Implementierungen von ExecutorService**
  - *thread pools*
    - Motivation ist, dass das Erzeugen und Beenden von *threads* für das Betriebssystem bzw. JVM relativ aufwändig ist.
    - Wenn für jeden Auftrag ein neuer *thread* erzeugt und nach Abarbeitung des Auftrags wieder gelöscht wird, steigt der Zeitbedarf für die *thread*-Verwaltung.

Neues zur Nebenläufigkeit in Java

- **Regelmäßige Ausführung von Aufträgen**
  - Hin und wieder ist es erforderlich, dass Aufträge nicht nur einmalig, sondern wiederholt ausgeführt werden müssen.
  - In der Schnittstelle `ScheduledExecutorService`, die `ExecutorService` erweitert, sind dazu einige Operationen definiert.
    - **`schedule(Runnable auftrag, long verzoegerung, TimeUnit einheit)`**
      Der Auftrag wird übergeben, jedoch erst nach Verstreichen der eingestellten Verzögerung ausgeführt.
    - **`scheduleAtFixedRate(Runnable auftrag, long verzoegerung, long wiederholungsrate, TimeUnit einheit)`**
      Der Auftrag wird erstmalig nach Verstreichen der eingestellten Verzögerung ausgeführt. Anschließend wird er immer in den durch wiederholungsrate angegebenen Abständen ausgeführt.
    - **`scheduleWithFixedDelay(Runnable auftrag, long verzoegerung, long wartezeit, TimeUnit einheit)`**

Neues zur Nebenläufigkeit in Java

Regelmäßige Ausführung von Aufträgen



Neues zur Nebenläufigkeit in Java

- **Aufträge mit Ergebnis**
  - Die einfachste Repräsentation eines Auftrags ist ein Objekt einer Klasse, die die Schnittstelle `Runnable` implementiert.
  - Um Ergebnisse zurückzuliefern, muss ein Auftrag durch `Callable` repräsentiert werden.
    - Die Schnittstelle `Callable` ist in der Version 5 neu in das Java-API aufgenommen worden.
  - `Callable` vs. `Runnable`
    - Die `run()`-Operation von `Runnable` ist vom Typ `void`.
    - Ein Auftrag hat also keine Möglichkeit, einen Wert zurückzuliefern.
    - Die `call()`-Operation von `Callable` definiert im Gegensatz einen Rückgabewert.

Neues zur Nebenläufigkeit in Java

- **Aufträge mit Ergebnis**
  - Beispiel
    - Eine Operation zur Lösung eines linearen Gleichungssystems könnte in einem eigenständigen thread ausgeführt werden und das Ergebnis als Matrix zurückliefern.

```
public class GleichungssystemTest
{
  public static void main( String[] args )
  {
    double[][] matrix = new double[10][10];
    //Matrix initialisieren

    Callable<double[][]> c = new Gleichungssystem( matrix );

    try
    {
      double[][] loesung = c.call();
    }
    catch( Exception e ){}
  }
}
```

Neues zur Nebenläufigkeit in Java

- **Aufträge mit Ergebnis**
  - Beispiel

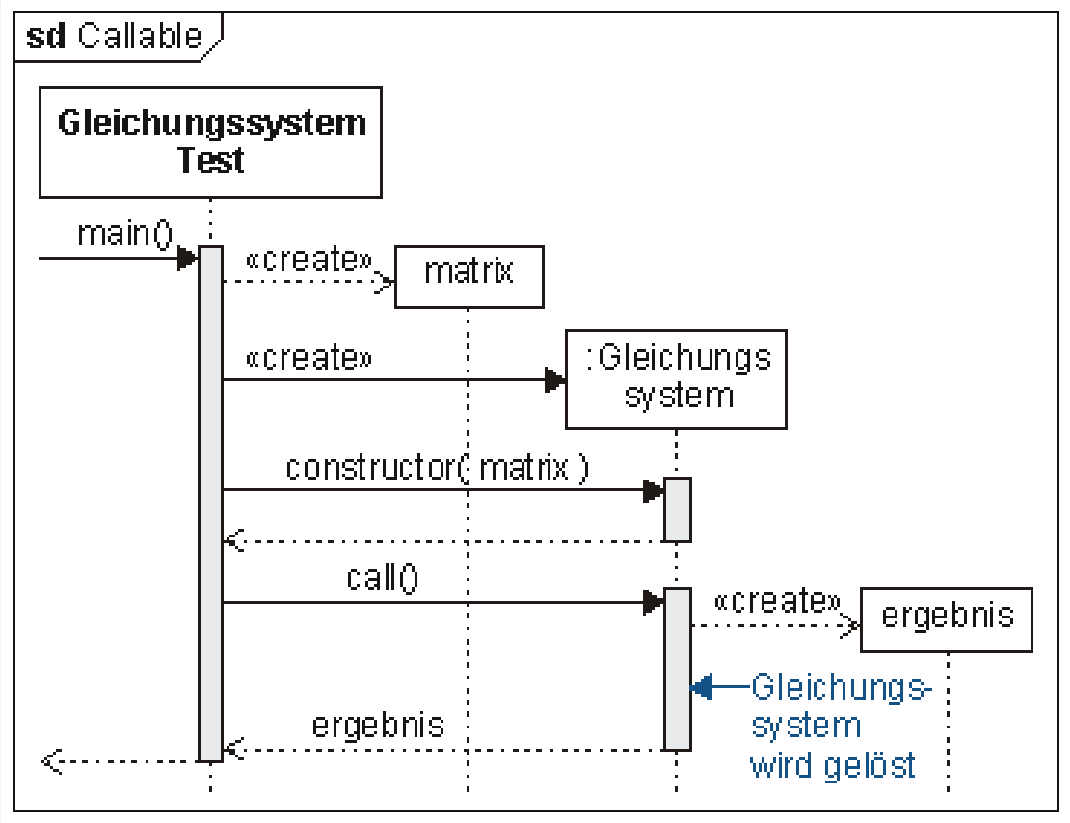
```
public class Gleichungssystem implements Callable<double[][]>
{
  private double[][] gs;

  public Gleichungssystem( double[][] gs )
  {
     this.gs = gs;
  }

  public double[][] call()
  {
    double[][] ergebnis = new double[10][10];
    //löse gs
    return ergebnis;
  }
}
```

Neues zur Nebenläufigkeit in Java

- **Aufträge mit Ergebnis**
  - Beispiel
    - Ausführung des Programms als UML-Sequenzdiagramm
      Der Aufruf von `call()` erfolgt im Beispiel synchron.



Neues zur Nebenläufigkeit in Java

- **Aufträge mit Ergebnis**
  - Die Schnittstelle `Future`
    - `Future` definiert einen in Ausführung befindlichen Auftrag.
    - Der Auftrag liefert ein Ergebnis zurück, welches über die Operationen von `Future` abgefragt werden kann.
    - `Future` ist eine generische Schnittstelle, der Typ-Parameter definiert den Ergebnis-Typ.
  - Wichtige Operationen der Schnittstelle `Future`
    - **`T get()`**
      Die Operation wartet (blockiert den aufrufenden thread) bis der durch das Future-Objekt repräsentierte task abgearbeietet ist und liefert den Rückgabewert.
    - **`T get(long wartezeit, TimeUnit einheit)`**
      Diese Variante wartet nicht unbegrenzt.

Neues zur Nebenläufigkeit in Java

- **Aufträge mit Ergebnis**
  - Funktionsweise
    - Ein Auftragsdienst (`ExecutorService`) verwaltet eine Liste von *threads* und eine Liste von Aufträgen.
    - Die *threads* arbeiten die Aufträge mit einer bestimmten Strategie ab.
    - Aufträge ohne Ergebnisse werden meist durch Aufruf der Operation `execute(Runnable command)` dem Auftragsdienst gemeldet.
    - Bei Aufträgen mit Ergebnis muss die Operation `Future<T> submit(Callable<T> task)` verwendet werden.
    - Das `Future` Objekt repräsentiert einen in Ausführung befindlichen oder in Zukunft auszuführenden Auftrag.
    - Die Synchronisation erfolgt durch den Aufruf der `get()`-Operation auf dem `Future` Objekt.

Neues zur Nebenläufigkeit in Java

Aufträge mit Ergebnis

```
public class GleichungssystemTest
{
  public static void main( String[] args )
  {
    ExecutorService executor = Executors.newFixedThreadPool(3);

    Future<double[][]> last = null;
    for (int i = 0; i < 10; i++)
    {
         //Matrix initialisieren
         double[][] matrix = new double[10][10];
         last = executor.submit<double[][]>(new Gleichungssystem( matrix ));
    }

    try {
           double[][] return = last.get();
    } catch(Exception e) {}

    executor.shutdown();
  }
}
```

- **Frage:** Was verbirgt sich hinter der Operation `get()`?

FALLBEISPIELE

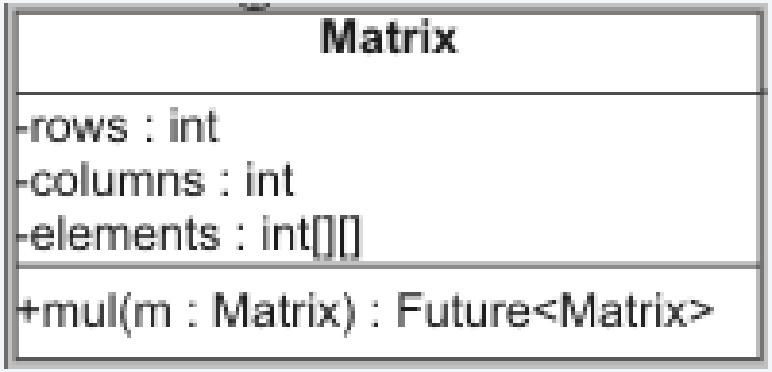
Neues zur Nebenläufigkeit in Java

- **Verwendungsbeispiel: Matrizenmultiplikation**

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \cdots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \cdots & b_{1s} \\ \vdots & \cdots & \vdots \\ b_{n1} & \cdots & b_{ns} \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{n} a_{1i} \cdot b_{i1} & \cdots & \sum_{i=1}^{n} a_{1i} \cdot b_{is} \\ \vdots & \cdots & \vdots \\ \sum_{i=1}^{n} a_{mi} \cdot b_{i1} & \cdots & \sum_{i=1}^{n} a_{mi} \cdot b_{is} \end{pmatrix}$$

  - Berechnung der Elemente der Ergebnismatrix ist kausal unabhängig
  - Skalar-Multiplikation der Spalten- und Zeilenvektoren

- **Wie geht man nebenläufig an dieses Problem heran?**
  - Hinweis zur Schnittstelle



Neues zur Nebenläufigkeit in Java

Verwendungsbeispiel: Matrizenmultiplikation

```
public class Hauptprogramm {
     public static void main(String[] args) {
          Matrix m1 = createRandomMatrix();
          Matrix m2 = createRandomMatrix();

          Future<Matrix> proxy = m1.mul(m2);
          //Tue etwas sinnvolles in der Zwischenzeit
                ...
                ...
                ...

          Matrix ergebnis = proxy.get();
     }
}
```

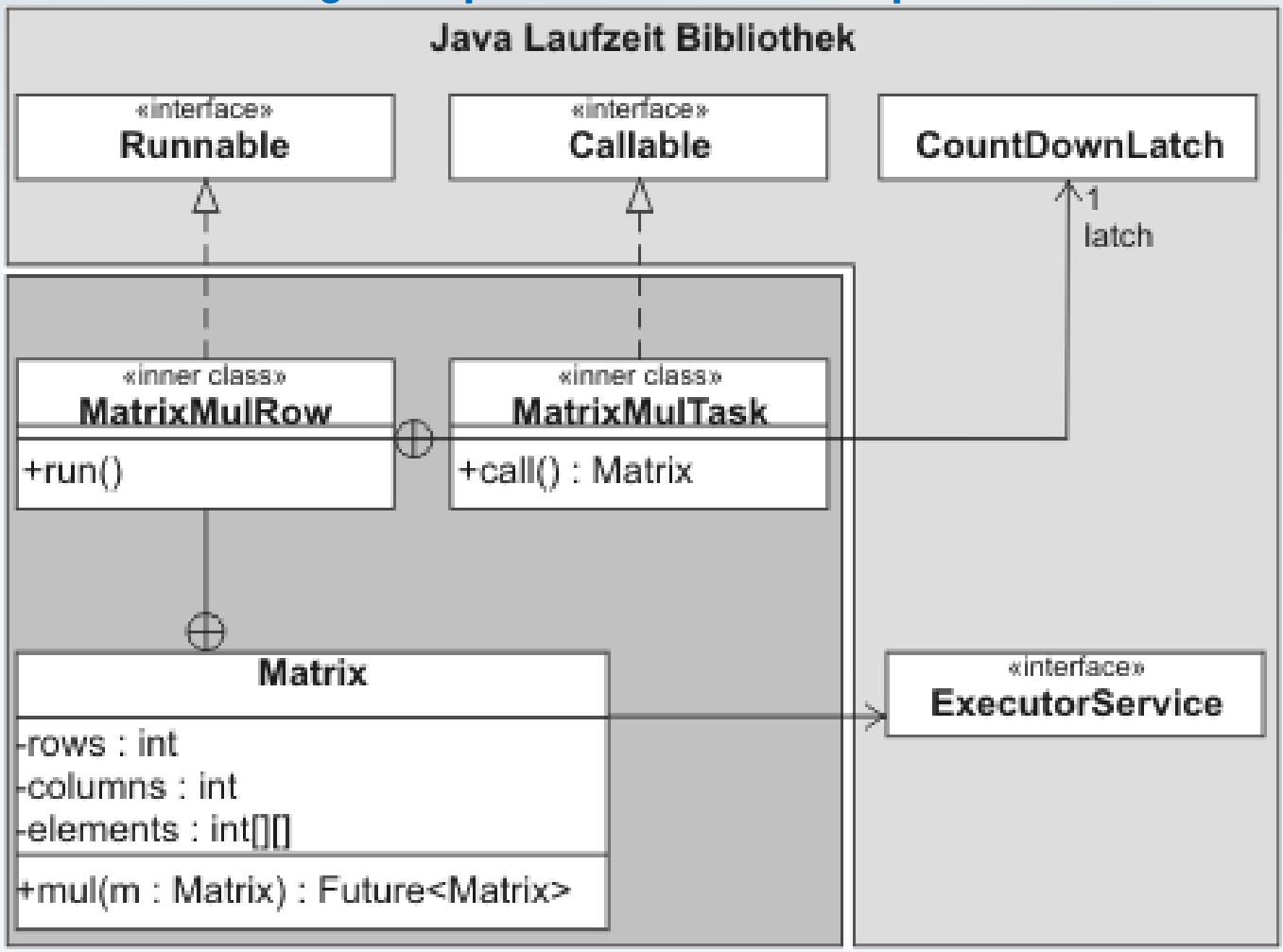
Neues zur Nebenläufigkeit in Java

- **Verwendungsbeispiel: Matrizenmultiplikation**

```
public class Matrix
{
    private int columns;
    private int rows;
    private int[][] elements;
    public Matrix(int w, int h) {
        columns = w;
        rows = h;
        elements = new int[columns][rows];
    }
    public Future<Matrix> mul(Matrix m) throws MatrixIncompatible {
        if (columns != m.rows || rows != m.columns)
            throw new MatrixIncompatible();
        return Matrix.getExecutorService().submit(
            new MatrixMulTask(this, m));
    }
    private static synchronized ExecutorService getExecutorService() {
         //Details sind nicht relevant
    }
}
```

Neues zur Nebenläufigkeit in Java

- **Verwendungsbeispiel: Matrizenmultiplikation**



Neues zur Nebenläufigkeit in Java

Verwendungsbeispiel: Matrizenmultiplikation

```
public class MatrixMulTask implements Callable<Matrix> {
    private Matrix subject, object, ret;
    private CountDownLatch latch;
    public MatrixMulTask(Matrix subject, Matrix object) {
        this.subject = subject; this.object = object;
    }
    public Matrix call() {
        //Ergebnismatrix erzeugen
        this.ret = new Matrix(object.columns, subject.rows);
        //Synchronisations-Objekt
        latch = new CountDownLatch(subject.rows);

        //Zeilenweise werden Autraege erzeugt
        for (int y = 0; y < subject.rows; y++)
            Matrix.getExecutorService().execute(new MatrixMulRow(y));

        //Warten auf den Ausfuehrungsdienst
        try {latch.await();}
        catch(InterruptedException e) {throw new RuntimeException();}
        return ret;

    }
}
```

Neues zur Nebenläufigkeit in Java

- **Verwendungsbeispiel: Matrizenmultiplikation**

```
public class MatrixMulRow implements Runnable {
     private int row;
     public MatrixMulRow(int row) {
         this.row = row;
     }
     public void run() {
         for (int x = 0; x < object.columns; x++) {
             int sum = 0;
             for (int i = 0; i < columns; i++) {
               sum += object.elements[x][i]*
                                       subject.elements[i][row];
             }
             ret.elements[x][row] = sum;
         }
         //Auftrag fertig. Barriere herunterzaehlen
         latch.countDown();
     }
}
```

Neues zur Nebenläufigkeit in Java

- **Verwendungsbeispiel: QuickSort-Verfahren**
  - Von Sir Charles Antony Richard Hoare 1962 publiziert
  - Rekursiver Algorithmus nach dem Teile und Herrsche-Prinzip
  - Grundidee
    - Feld an einem bestimmten Element – Pivotelement **p** – in zwei Hälften teilen
    - Anschließend die Elemente so sortieren, dass **Elemente >= p** in der rechten Hälfte und **Elemente < p** in der linken Hälfte liegen.

Neues zur Nebenläufigkeit in Java

Messung der Laufzeiten

Messwerte auf Intel Core Duo 2.8 GHz

Feldgröße (MB)	Feldgenerierung (ms)	Quicksort Standard (ms)	Quicksort NSP (ms)	Verbesserung (%)
0	0	0	0	0
15	1035	2448	2178	11
35	2327	5690	4533	20
55	3675	9055	6749	25
75	4836	12210	8612	29
95	6102	15419	10850	30
100	6433	16372	11532	30

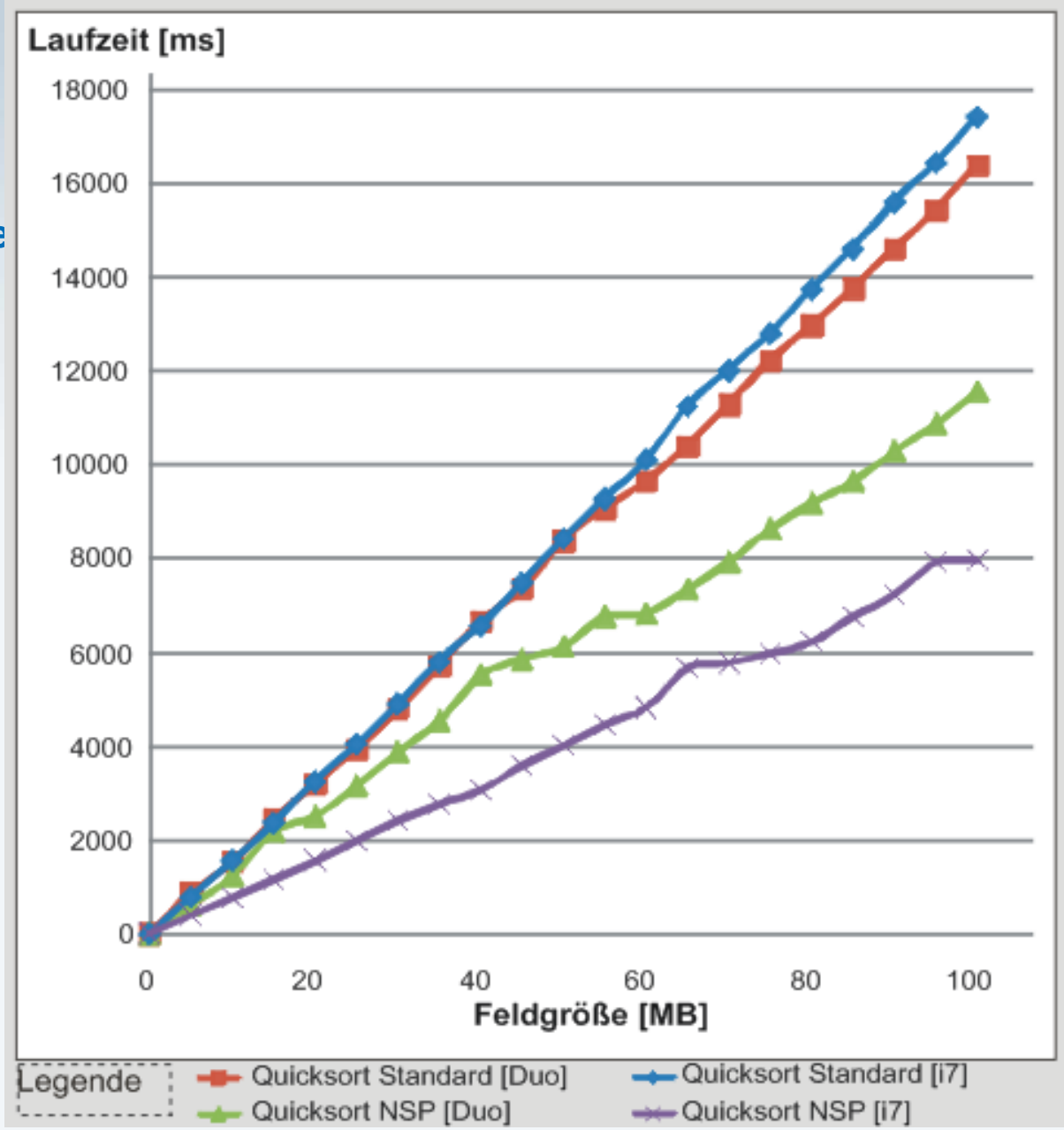
Tab. 1

Messwerte auf Intel Core i7 1.73 GHz

Feldgröße (MB)	Feldgenerierung (ms)	Quicksort Standard (ms)	Quicksort NSP (ms)	Verbesserung (%)
0	0,00	0,00	0,00	0
15	828	2365	1150	51
35	1865	5800	2765	52
55	2940	9267	4448	52
75	3980	12790	5960	53
95	5070	16427	7898	52
100	5348	17403	7950	54

Neu

- Me


Laufzeit [ms]
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
0
20
40
60
80
100
Feldgröße [MB]
Legende
Quicksort Standard [Duo]
Quicksort Standard [i7]
Quicksort NSP [Duo]
Quicksort NSP [i7]

Fazit

- **Nur durch die nebenläufige Programmierung lässt sich die Hardware optimal auslasten**
- **Mehrkernsysteme nicht nur auf Workstations und Server begrenzt**
  - Mobiltelefone
  - Tablets
  - Grafikkarten
- **Softwareingenieure müssen sich mit diesem Thema beschäftigen.**
- **„The Free Lunch Is Over“**

Vielen Dank!

Inhouse-Schulungen

Wir bieten Inhouse-Schulungen und Beratung durch unsere IT-Experten und –Berater.

Schulungsthemen

- Softwarearchitektur (OOD)
- Requirements Engineering (OOA)
- Nebenläufige & verteilte Programmierung

Gerne konzipieren wir auch eine individuelle Schulung zu Ihren Fragestellungen.

Sprechen Sie uns an!
Tel. 0231/61 804-0, info@W3L.de

W3L-Akademie

In Zusammenarbeit mit der Fachhochschule Dortmund bieten wir

zwei Online-Studiengänge

- B.Sc. Web- und Medieninformatik
- B.Sc. Wirtschaftsinformatik

und 7 Weiterbildungen im IT-Bereich an.

Besuchen Sie unsere Akademie!
http://Akademie.W3L.de