Nebenläufigkeit

- Motivation und Parallelität
- Definitionen und Übersicht
- Continuations und Koroutinen
- Threads in Java
- Erste Beispiele

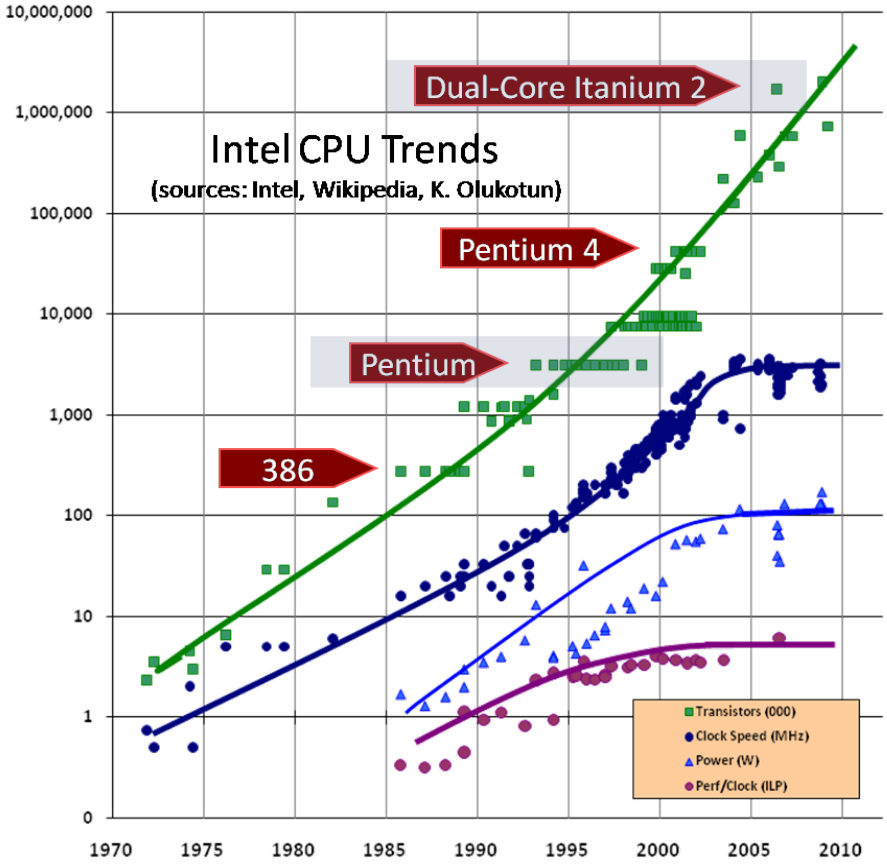
Motivation für Nebenläufigkeit

Nebenläufigkeit =

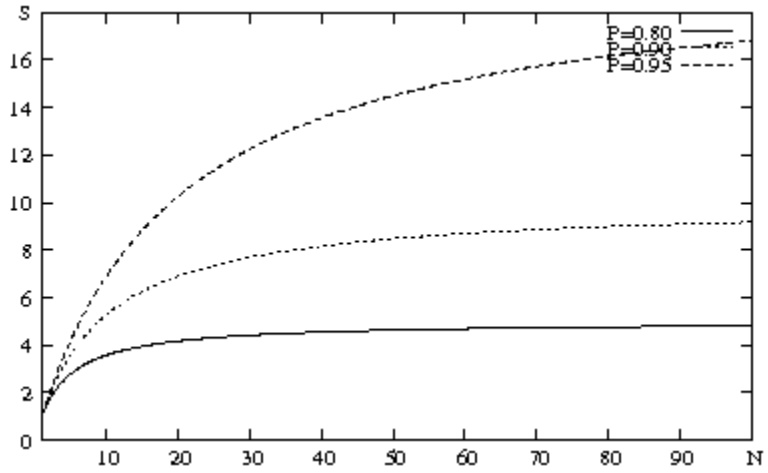
mehrere Kontrollflüsse werden in nicht festgelegter Weise abgearbeitet. Dabei verfügt jeder Kontrollfluss über einen eigenen lokalen Zustand.

- Graphische Oberflächen benötigen einen unabhängigen Kontrollfluss
- Die Simulation realer Objekte wird erleichtert
- Mehrere Eingabequellen müssen effizient beantwortet werden
- Blockierende (oder langsame) Kommunikation soll nicht das gesamte Programm blockieren
- …
- Steigerung der Rechenleistung durch Ausnutzung der Multicore-Architektur
- Effiziente und parallel arbeitende Webserver
- Verteilte Systeme
- Höchstleistungscomputing mit Parallelrechner

The End of the Free Lunch ...



Aber es gibt auch Grenzen der Parallelverarbeitung (Amdahl's Law)

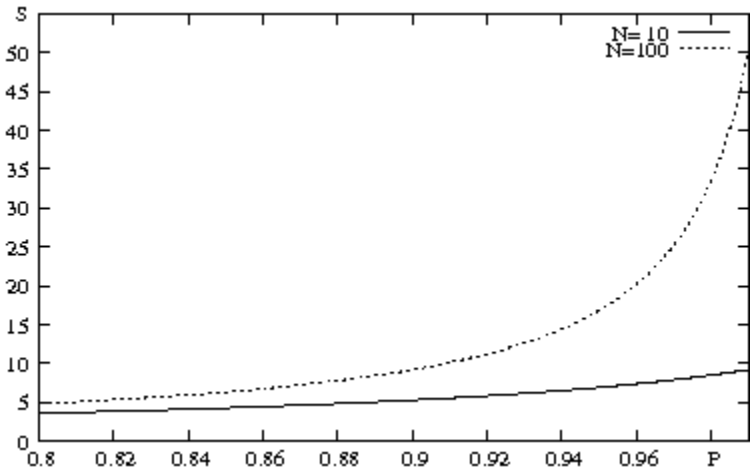


N = Anzahl Prozessoren
P = Parallelisierbarer Anteil des Programms
S = Geschwindigkeitssteigerung (speed up)
$= T_0 / T_N$

Amdahl wollte seinerzeit zeigen, dass sich Parallelverarbeitung kaum lohnt.

Das Gesetz sagt aus, dass die Geschwindigkeitssteigerung durch sequentielle Programmteile entscheidend begrenzt wird.

Sehr rechenintensive Anwendungen = sehr parallele Programme



Wenn wir einen Rechner mit N Prozessoren ausnutzen wollen,
brauchen wir eine hohe Parallelisierbarkeit.

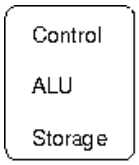
Bei vielen Problemen (Simulation physikalischer Prozesse, LHC)
ist die Parallelisierbarkeit beliebig groß!

Aber „normale“ Anwendungen sind kaum parallelisierbar !

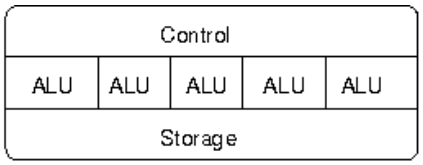
Geschichte der Verwendung von Nebenläufigkeit

- Die effiziente Ausnutzung von langsamer Peripherie führte zum *Multitasking*.
- Die ersten interaktiven *Mehrbenutzersysteme* (*multics*, *tops10*, *unix*) basierten auf Multiprocessing.
- *Echtzeitsysteme* (embedded systems) benötigen zumindest eine einfache Form von Nebenläufigkeit (Interrupt-Technik).
- Die ersten *OO-Sprachen* (Simula, Smalltalk 80) enthielten Nebenläufigkeit.
- In den 70ern und 80ern wurden unterschiedliche Modelle von OO und nicht OO Nebenläufigkeit untersucht.
- Seit den 70ern gibt es intensive Entwicklungen und Anwendungen im Bereich der Parallelverarbeitung (Vektorrechner, Parallelrechner, Clustersysteme).
- Fast jeder Processor verfügt heute über mehrere Rechenerkerne (multi core).
- **Heute ist Multiprocessing und Multithreading (praktisch) auf jedem System und in jeder Programmiersprache möglich (und nötig).**

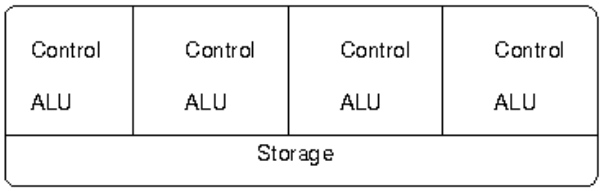
Flynn'sche Taxonomie von Rechnerarchitekturen



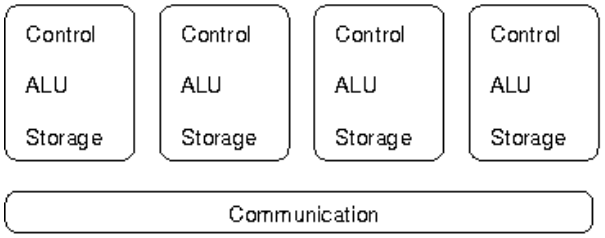
SISD



SIMD



MIMD shared Memory



MIMD distributed Memory = Cluster

SISD = **S**ingle **I**nstruction **S**ingle **D**ata
SIMD = **S**ingle **I**nstruction **M**ultiple **D**ata
MIMD = **M**ultiple **I**nstruction **M**ultiple **D**ata

shared Memory = (teilweise) gemeinsamer Speicher
distributed Memory = verteilter (lokaler) Speicher

Programmiermethoden:

SIMD ähnlich:
- parallele Datenstrukturen für Funktionen höherer Ordnung

gemeinsamer Speicher mit
- Monitor / Sperre
- atomaren Operationen
- STM

lokaler Speicher mit
- Botschaftenaustausch

Hardwarearchitektur kann anders als Softwarearchitektur sein

Parallelrechner

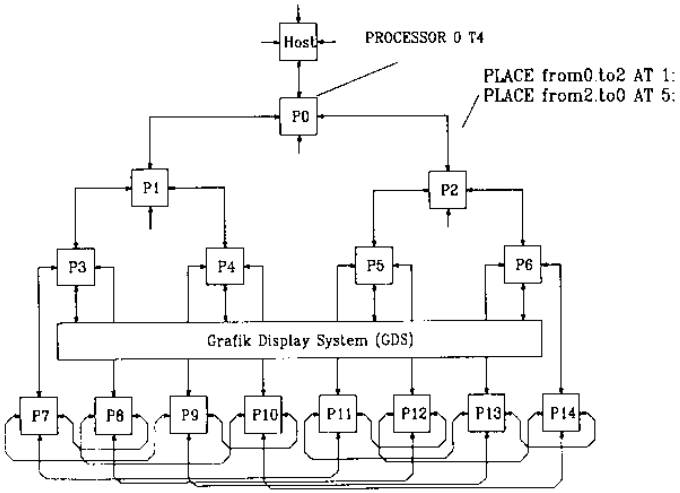
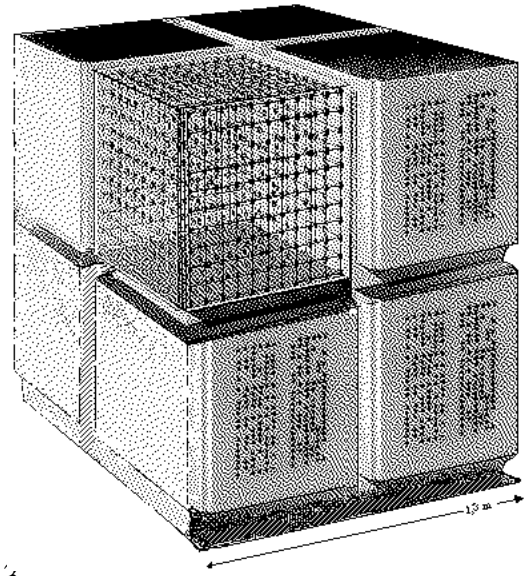


Bild 3.3: *Verschaltung des zu Bild 3.4 gehörenden Transputernetzes*

```
{{{F prog.tsr
...  change this if you want an other configuration
...  SC processl
...  SC graphics
...  declarations

PLACED PAR  -- total of 17 processors as PROG + 1 processor as EXE
  ...  tree level 0 (1 processor )
  ...  tree level 1 (2 processors)
  ...  tree level 2 (4 processors)
  ...  tree level 3 (8 processors)
  ...  grafik-display-system
}}}
```

Bild 3.4: *Plazierung separat compilierter Prozesse auf einem Transputernetz (Folds geschlossen)*



Parallele Programmierung

- **Vektoroperationen** auf Datenstrukturen (z.B. Graphikkarte)

- **Unabhängige Ausführung** (Scatter-Gather, Fork-Join)

- **Implizite Datenparallelität**: Kontrollstrukturen (z.B. openMP) oder Datenstrukturen (z.B. Scala) werden als parallel deklariert. Ein internes Framework steuert die Ausführung.

- Fertige **parallele Algorithmen** (z.B. Hadoop / MapReduce)

- Multi-Threadprogrammierung mit **gemeinsamem Speicher**. Die Herausforderung besteht in der sicheren Vermeidung von Wettlaufbedingungen und Verklemmungen.

- Multi-Threadprogrammierung (oder Programmierung verteilter Prozesse) mit Kommunikation mittels **Botschaftenaustausch** (z.B. Actor-Modell).

Threadausführung und die Elemente eines Java-Programms

Die Ausführung eines Java-Programms umfasst:

- ***Klassen*** zur Beschreibung von Objekten
- ***Objekte*** zum Speichern von Information
- ***Methoden***, die sagen, wie etwas gemacht wird
- Einen ***Thread*** der sich durch die Methoden durchschlängelt und jeweils genau eine Methode ausführt. Der Main-Thread startet mit der Funktion `main()`
- Eventuell weitere Threads (automatisch oder programmiert) (***Multithreading***)
- Verschiedene ***Speicherbereiche*** (Stack, Heap, Register, Cache)
- Den ***Prozess*** der virtuellen Maschine in dem sich das alles abspielt

Java ist so entworfen, dass man architekturunabhängig korrekte Multithreading-Programme schreiben kann – aber das ist nicht einfach!!

Hinweis: Thread = „Faden“ (ausgesprochen [thred] – nicht ~~thri:d~~)

Sequentielle Programmausführung

Ein **sequentielles Java-Programm entspricht dem prozeduralen Paradigma**. Soweit man die Ergebnisse *beobachten* kann, ist die Ausführungsreihenfolge exakt (evtl. zusammen mit externen Daten) durch das Programm vorgegeben.

Der **Compiler**, die Java-Laufzeitumgebung und der Prozessor **können** innerhalb dieser Grenzen **Modifikationen vornehmen** um die Effizienz zu steigern. Und das tun sie auch!!!

- Soweit es keinen Einfluss auf die Ergebnisse hat, kann die Reihenfolge von Befehlen verändert werden (***reordering***).

- Daten können zeitweise in Registern und Cache-Speichern gehalten werden. In dieser Zeit ist der Inhalt des Hauptspeichers nicht korrekt. Es ist denkbar, dass ein Objekte niemals im Hauptspeicher erscheint (***visibility***) .

Im Ergebnis kann erst durch diese Maßnahmen die Leistungsfähigkeit moderner Computer erreicht werden!

Diese beiden Punkte sind vielen Java-Entwicklern nicht bekannt! Folge: Programmfehler !!!

Nebenläufigkeit

Unter Nebenläufigkeit versteht man die gleichzeitige oder **quasigleichzeitige Ausführung** von Programmen oder Programmteilen.

In einem nebenläufigen Programm ist die Reihenfolge der Befehlsausführung nicht vollständig festgelegt. Das Programm kann durch einen oder mehrere Prozessoren ausgeführt werden.

Nebenläufige Abläufe *können* über **gemeinsamen Speicher** verfügen.

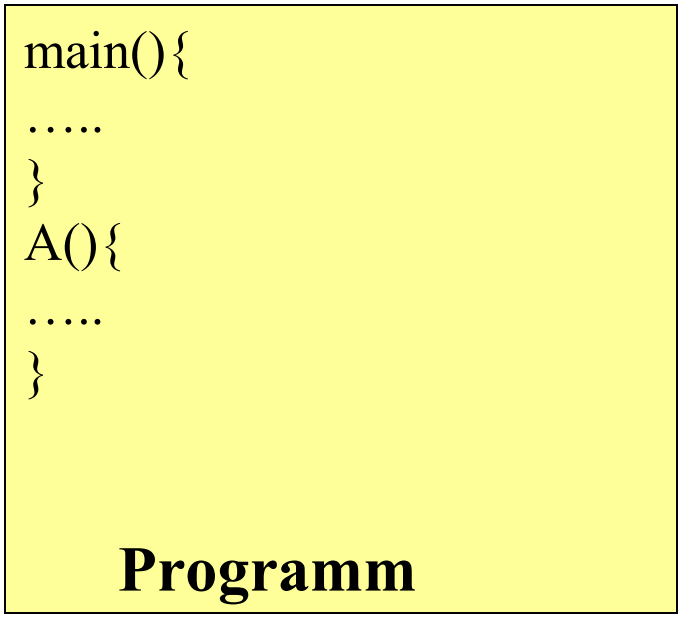
Gemeinsamer Speicher ist effizient, stellt aber ein großes Problem dar. Gründe sind die **sequentielle Optimierung** der Ausführung (s.o.) und die nötige Koordination in der **Änderung von Variablen**, die durch die Programmlogik bestimmt ist (Auschluss von Wettlaufbedingungen – **race conditions**).

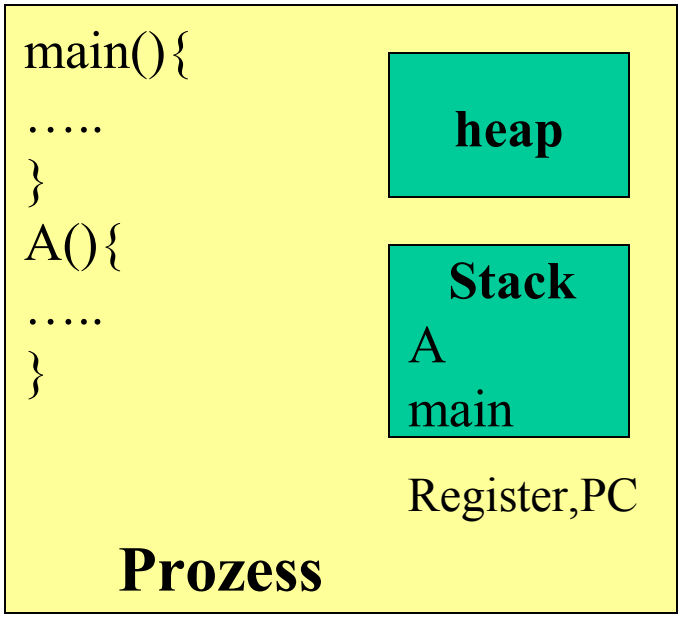
Die Regelung der Veränderung und der Sichtbarkeit der Werte von gemeinsamen Variablen wird durch **Synchronisation** erreicht.

Programme die unabhängig von den Ausführungsmöglichkeiten eine nebenläufigen Programms immer das gleiche Verhalten zeigen, sind **threadsicher**.

Ein Prozess ist ein in der Ausführung begriffenes Programm

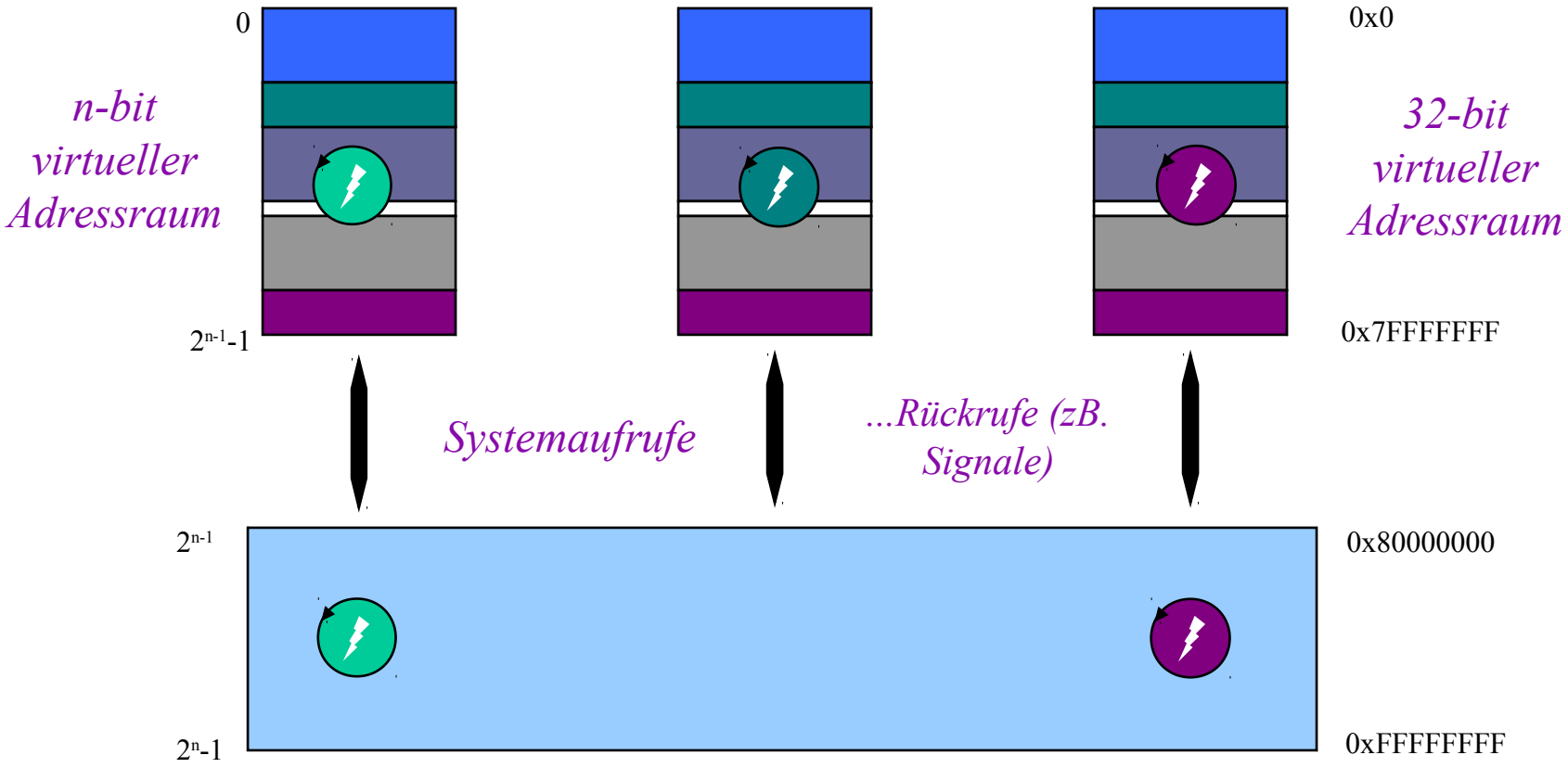
- Ein Programm beschreibt *statisch* Struktur und Ablauf.
- Ein Prozess ist die *dynamische* Ausführung des Programm.





Prozesse werden von dem Betriebssystem verwaltet

Interprozesskommunikation über das Betriebssystem ist langsam!



Der relative Aufwand für das Starten von Threads und Prozessen

Eine Messung ergibt (nach Abwarten der internen Java-Optimierung) in etwa die folgenden Verhältnisse:

run	1	-	-
pool	1,5	1	-
thread	100	65	1
process	2000	1300	20

- run: Objekterzeugung/Methodenaufruf/Zeitmessung
- pool: Ausführung der Methode durch einen zuvor gestarteten Thread
- thread: Ausführung der Methode durch einen neuen Thread
- process: Ausführen eines externen Prozesses (C-Programm)

Ausführung mit JDK 1.7 und Linux 3.2.0. Es lief immer nur 1 Thread (aber sehr häufig wiederholt).

Fazit: naive Nebenläufigkeit kann sehr schlecht sein!

Threads

Ein *Thread* ist eine Ausführungsreihenfolge innerhalb eines Prozesses. Ein Prozess enthält mindestens 1 Thread (sequentieller Prozess). Ein Prozess kann mehrere Threads enthalten.

Threads verfügen in der Regel über eigenen (exklusiven) Speicherraum aber auch über gemeinsam genutzten Speicher.

- Jeder Thread verfügt über einen separaten Stack (Rücksprung, lokale Variable).

- Jeder Thread verfügt über einen eigenen Programmzähler (PC), d.h. eine eigene Kontrolle des Ablaufs.

Threads werden realisiert durch:

system level threads: Betriebssystem (native threads)

user level threads: Laufzeitsystem (z.B. green threads in früherer jvm)
oder selbst programmiert (z.B. low-level C)

Threads und Objektorientierung:

- In **sequentiellen OO-Programmiersprachen** ist zu jedem Zeitpunkt **genau eine Methode aktiv**.
- In dem entgegengesetzten **Actor-Konzept** sind **alle Objekte gleichzeitig aktiv**. Kommunikation erfolgt durch den **Austausch von Nachrichten**.
- Thread-Modelle der **prozeduralen Welt** verlegen die **Nebenläufigkeit auf die Ebene von Prozeduren**.
- In dem **Java** Modell können Methoden `run()`, (Schnittstelle `Runnable`), als nebenläufiger Thread gestartet werden.

Ein Thread ist ein Ablauf. *In Java können Methoden verschiedener Objekte in einem einzigen Thread ausgeführt werden. Umgekehrt kann eine Methode in verschiedenen Threads aufgerufen und ausgeführt werden.*

Funktionsobjekte / Callbacks / Threads

Vergleich: Ein (Java-) Thread hat mit einer Closure gemein, dass er (zunächst) ein Funktionsobjekt ausführt. Dieses hat wie Closures einen lokalen Kontext und den Zugriff auf eine äußere Umgebung.

Im Unterschied zu Closures haben Threads einen eigenen Ablauf.

Callback Funktion.

Definition:Eine Callback-Funktion ist ein Funktionsobjekt, das einem anderen Objekt zwecks Aufruf bei einem Ereignis übergeben wird. Die Ausführung der Callback-Funktion erfolgt im Thread des Aufrufers. Die Callback-Funktion kann aber nur auf die Variablen seine eigenen Umgebung (closure) (und auf Aufrufparameter) zugreifen.

Callback muss kein Mutlithreading bedeuten. Wenn jedoch ja, dann muss man auf die Threadsicherheit achten!

Continuations und Koroutine:

Eine ***Continuation*** speichert den momentanen lokalen Ausführungszustand und ermöglich dessen Fortsetzung zu einem späteren Zeitpunkt.

Herkunft: funktionale Sprachen, formale Definition, Exception-Handling

Das ***Koroutinenkonzept*** stellt rufende und gerufene Prozedure auf eine Stufe. Es ermöglicht es, mehrfach den Kontrollfluss zwischen den unterschiedlichen Kontexten wechseln zu lassen.

Herkunft: Simulation diskreter Systeme, Simula 67, Modula 2

Continuations und Koroutinen enthalten grundsätzlich nur einen Kontrollfluss aber mehrere lokalen Ausführungsumgebungen. Man kann Koroutinen mit Continuations realisieren. Multithreading lässt sich (weitgehend) mit Koroutinen realisieren. Umgekehrt lassen sich Koroutinen durch Multithreading implementieren.

Continuations ermöglichen einfachen Kontextbezug in Webanwendungen.

Beispiel für Continuation (Python)

```
def fib():
    i0, i1 = 0, 1
    while 1:
        yield i0
        i0, i1 = i1, i0 + i1

def main():
    f = fib()
    for i in range(0, 20):
        print i, f.next()
```

Als Ergebnis der Ausführung erscheint eine Liste von Fibonacci-Zahlen.

`yield` speichert den aktuellen Zustand und gibt das Resultat zurück. `next` veranlasst die Fortsetzung der Programmausführung ab `yield`.

Man kann das auch als Beispiel für eine Koroutine auffassen, da durch das Paar `yield` / `next` zwischen zwei Funktionen (main / fib) hin und her gewechselt wird.

Das Beispiel lässt sich in Java (nur) über Threads realisieren.*

*außer man ändert die virtuelle Maschine

Threads in Java:

Threads stellen einen **Ablauf** dar, keinen besonderen Sichtbarkeits- oder Speicherbereich.

Ein Methode wird durch einen Thread ausgeführt. Die lokalen Variablen dieser Methode sind nur lokal in der Methode und damit automatisch nur während ihrer Ausführung in **einem** Thread bekannt.

Objekte (auf dem Heap) können grundstätzlich von allen Threads angesprochen werden. Statische und nichtstatische Variable von **Klassen** können von unterschiedlichen Threads gelesen oder geschrieben werden.

Der Zugriff auf **gemeinsame Variable** bedarf besonderer Vorkehrungen!

Manchmal muss ein Thread **warten** bis ein bestimmter Zustand eingetreten ist.

Die Thread-Ausführung wird durch das ***Scheduling*** gesteuert. Java legt nicht fest, wie dies geschieht.

Java Primitiv-Konstrukte für Threads

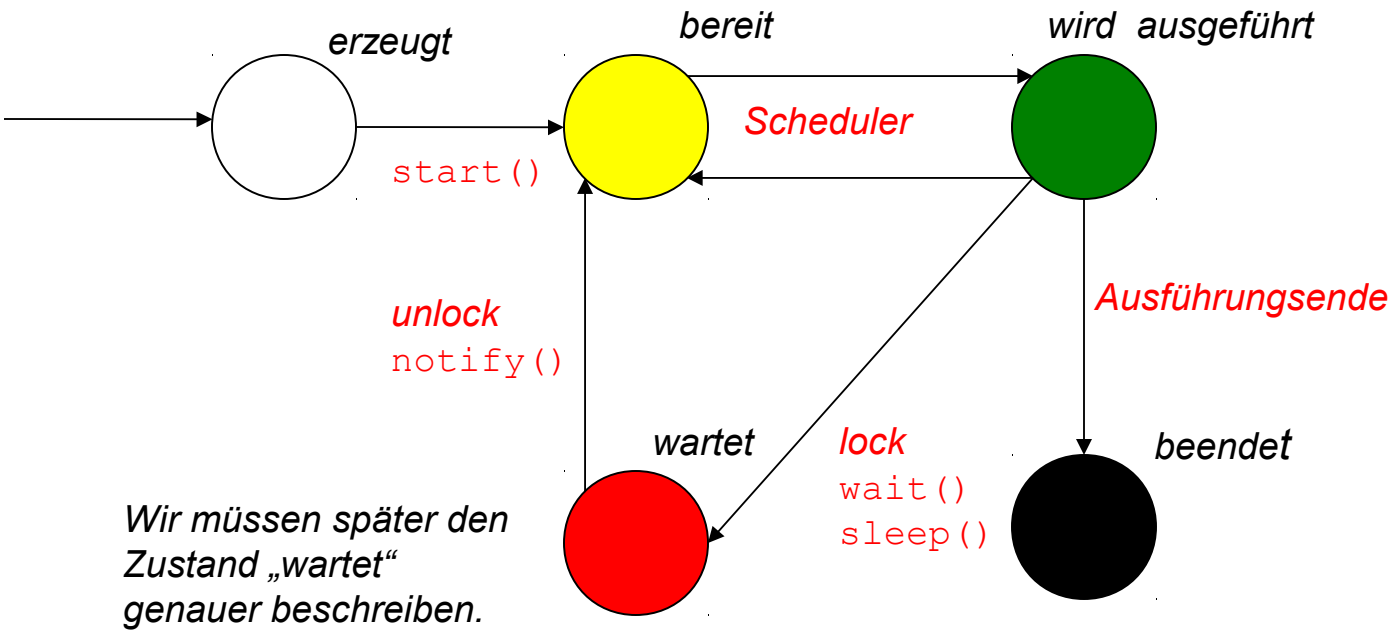
- **`interface Runnable`** als allgemeine Schnittstelle
- **`class Thread`** zur Erzeugung und Steuerung von Threads
- **`synchronized`** Blöcke für gegenseitigen Ausschluss
- **`wait(), notify(), notifyAll()`** zum Warten auf Bedingungen
- **`volatile`** garantiert, dass der aktuelle Wert einer Variablen vorliegt
- **`interrupt() / InterruptedException`** zur sicheren Terminierung

Java Primitive verwenden das Monitorkonzept:

1. **Objekte** verwalten die **Sperre** (*lock*) zu gesicherten (synchronized) Methoden und Programmabschnitten.
2. **Objekte** verwalten **Wartelisten** (*wait set*) von Threads die von diesem Objekt eine Nachricht (`notify()`, `notifyAll()`) erwarten.

Zustandsdiagramm der Threadausführung (vereinfacht)

Die Wirkungsweise der Thread-Konstrukte und des automatischen Schedulings lässt sich durch Zustandsübergänge beschreiben.



Zustände und Aktionen

***erzeugt* (NEW):** das Thread-Objekt ist erzeugt worden.

`start()`: die Methode `run()` des Runnable-Objekts wird aufgerufen

***bereit* / *wird ausgeführt* (RUNNABLE):** grundsätzlich befindet sich der Thread in der Ausführung, kann aber vom Scheduler zeitweise zurückgestellt werden.

Scheduler: diese Komponente der jvm oder des Betriebssystems entscheidet darüber, welcher der bereiten Thread wie lange die CPU zugeteilt bekommt.

***wartet* (BLOCKED, WAITING, TIMED_WAITING):** auf Grund von gegenseitigem Ausschluss (`synchronized`) oder explizitem Warten (`wait()`, `sleep()`) kann ein Thread momentan nicht ausgeführt werden.

unlock / `notify()`: nach Ende des gegenseitigen Ausschluss oder Ende des Wartezustands (Zeitablauf, Unterbrechung, `notify()`) wird der Thread wieder ausführungsbereit.

***beendet* (TERMINATED):** der Thread-Ablauf ist beendet (in der Regel: Ende der Ausführung von `run()`. Das Thread-Objekt und das evtl. separate Runnable-Objekt und ist noch vorhanden und seine Methoden können noch aufgerufen werden.

Scheduling von Threads

Die einfachste Form ist das *scheduling on demand*. Nur bei wenigen genau definierten Sprachkonstrukten wird die Ausführung eines Thread unterbrochen.

Preemptives Scheduling unterbricht laufende Threads wenn bestimmte äußere Bedingungen vorliegen (abgelaufenes Zeitfenster)

Die Sprachbeschreibung von Java legt die Art des Scheduling nicht fest. Dies geschieht in der virtuellen Maschine und im Betriebssystem.

In Java können Threads in der Zuteilung von Rechenzeit bevorzugt werden (*Prioritäten*). Die Behandlung von Prioritätshinweisen ist durch die Sprache aber nicht festgelegt.

Konsequenz: portable Java-Programme können nur wenige Annahmen über den Ablauf von Threads machen.

Automatische Threads in Java

Unabhängig von besonderen Anweisungen des Programms enthalten Java-Programme mehrere Threads:

- **main-Thread**, der das eigene Programm ausführt.
- Threads für die automatische **Speicherbereinigung**.
- **Event-Thread** für die Behandlung der Benutzerinteraktion.
- und weitere ...

Diese automatischen Threads stellen (in einfachen Programmen) keine Probleme dar.

Von welchen Objekten ist die Rede, wenn man über Threads spricht ?

Objekte gehören vom Grundsatz her nicht zu einem bestimmten Thread!!!

Ein Thread befindet sich zu einem bestimmten Zeitpunkt in der Methode eines Objekts (**aktuelles this-Objekt**).

Jeder Thread verfügt über ein Objekt, dass wichtige Informationen über den Thread speichert und wichtige Aktionen verfügtbar macht. Dieses Objekt ist eine **Instanz der Klasse `Thread`** (oder einer davon abgeleiteten Klasse)

Es gibt ein paar **Klassenmethoden der Klasse `Thread`** (z.B. `Thread.currentThread()`).

Man kann mit einem Thread **Thread-lokale Klassenvariable** verknüpfen (`ThreadLocal`).

Der Threadablauf startet immer in der **Methode `run()`** eines Objekts, das die **Schnittstelle `Runnable`** implementiert. Der Thread wird spätestens am Ende von `run()` beendet.

Explizites Erzeugen von Threads

Java-Threads müssen direkt oder indirekt die Schnittstelle `java.lang.Runnable` implementieren und damit über eine Methode `run()` verfügen. Die Methode `run()` wird beim Start des Thread aufgerufen. Sie stellt praktisch die jeweilige main-Funktion dar.

```
package java.lang;
public interface Runnable {
    public void run();
}
```

Um einen Thread zu erzeugen, benötigt man die Hilfe der Klasse `Thread`, die ebenfalls bereits die Schnittstelle `Runnable` implementiert.

Es ergeben sich unterschiedliche Möglichkeiten Threads zu erzeugen:

- Statische Erweiterung der Klasse `Thread` durch **Vererbung**
- Dynamische Erweiterung der Klasse `Thread` durch **Delegation (besser)**
- Erzeugen von Threads durch anonyme Klassen (**Funktionsobjekt**)

Threadsobjekte durch anonyme Klassen

```
class ThreadApplication {

  ...
  Thread t = new Thread() {
    public void run() {
        // hier steht was getan wird.
    }
    // weitere Methoden ...
  };


  // Starten von Threads:
  t1.start(); // jetzt wird t1.run() ausgeführt

  // jetzt laufen main und t1.run()
  ...
```

Beenden von Threads

Ein Thread endet, wenn die run-Methode beendet wird (main-Thread endet am Ende von main()).

Ein Java-Prozess endet, wenn alle Threads beendet sind.

Ausnahme: wenn man vor seinem Start einen Thread in den “Dämon-Status” versetzt (Aufruf setDemon(true)), spielt der Thread für die Lebensdauer des Prozesses keine Rolle.

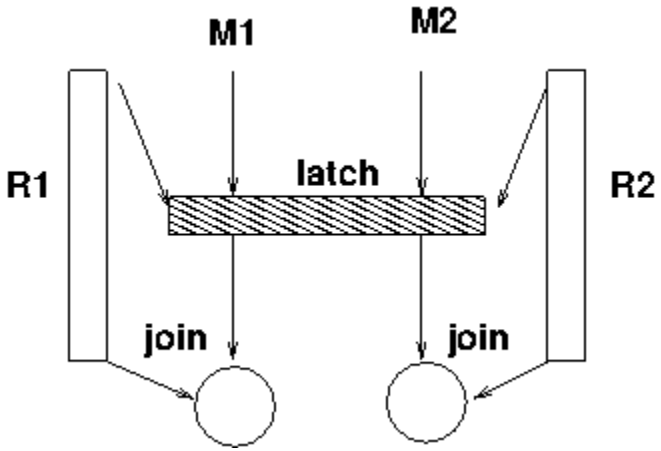
Durch den Aufruf **`System.exit()`** wird immer der Prozess sofort beendet (“abgewürgt”).

Ansonsten kann man den Thread nur durch entsprechende Mitteilung (evtl. mithilfe von `Thread.interrupt()`) dazu bewegen sich selbst zu beenden.

Mittels der Methode `join()` kann man darauf warten, dass ein bereits ausgeführter Thread beendet ist.

(das folgende Beispiel benutzt Bibliotheksklassen um sicherzustellen, dass alles funktioniert – das ist IMMER besser als die Primitivoperationen zu nutzen!)

Beispiel für Threadfunktionen



M1, M2 sind zwei Überwachungsthreads, R1, R2 machen irgendwas.

M1 und M2 starten erst, wenn R1 und R2 ausgeführt werden.

Sie warten dann auf das Ende „ihres“ Threads und melden dass der überwachte Thread beendet ist.

(latch ist nötig, weil join nicht wartet, wenn der betreffende Thread noch nicht läuft)

Beispiel: Klasse für nebenläufige Ausführung

Die Annotation `@Immutable` beschreibt, dass sich der Objektzustand nie ändert, so dass das Objekt völlig problemlos gleichzeitig in verschiedenen Threads genutzt werden kann.

(`println` ist threadsicher)

```
@Immutable
public final class Runner implements Runnable {
    private final int maxCount;
    private final CountDownLatch latch;

    public Runner(CountDownLatch latch, int maxCount) {
        this.maxCount = maxCount;
        this.latch = latch;
    }

    public void run() {
        latch.countDown(); // signal that this thread is running
        String name = Thread.currentThread().getName();
        for (int i = 0; i < maxCount; i++)
            System.out.println(name + ": " + i);
    }
}
```

Beispiel: Klasse die andere Threads überwacht

```
@Immutable
public class TerminationTrace implements Runnable {
    private final Thread toMonitor;
    private final CountDownLatch latch;

    public TerminationTrace(CountDownLatch latch, Thread toMonitor) {
        this.toMonitor = toMonitor;
        this.latch = latch;
    }

    public void run() {
        try {
            latch.await();      // wait until all threads are running
            toMonitor.join();   // wait until toMonitor has terminated
            System.out.println(toMonitor.getName() + " ist beendet");
        }
        catch (InterruptedException shouldNeverHappen) {
            throw new RuntimeException(shouldNeverHappen);
        }
    }
}
```

Beispiel: Steuerung durch main

```
import java.util.concurrent.CountDownLatch;

public class Main {
    public static void main(String[] a) {
        // we got 2 „worker" threads:
        final CountDownLatch latch = new CountDownLatch(2);

        Runner r = new Runner(latch, 1000);

        Thread t1 = new Thread(r);
        Thread t2 = new Thread(r);

        new Thread(new TerminationTrace(latch, t1)).start();
        new Thread(new TerminationTrace(latch, t2)).start();

        t1.start();
        t2.start();

        System.out.println("main ist (schon) fertig");
    }
}
```