

Nebenläufigkeit mit Java

Vorlesung Betriebssysteme

am 21. November 2018



- Wiederholung
 - Prozesse
 - Threads
- Nebenläufigkeit
 - Vorteile
 - Nachteile
 - Race Conditions
 - Deadlocks
- Programmiersprachliche Konzepte zur Synchronisation
 - Semaphore
 - Monitore
- Auffrischung zu Java
 - Kontrollstrukturen
 - Datentypen
 - Objektorientierung
- Threads in Java
- Synchronisation von Threads in Java (Monitor-Prinzip)

- Ein Betriebssystem hat viele Aufgaben:
 - Zuordnung von Ressourcen zu Prozessen
 - **Multiprogramming** zur Verbesserung der Auslastung
 - Bereitstellung von Möglichkeiten zur **Interprozesskommunikation**

- Konzeptuelle Grundlagen:
 1. Ein Computer besitzt eine Menge **von Hardware Ressourcen**
 2. Computerprogramme werden für bestimmte **Aufgaben** geschrieben
 3. Anwendungen sollten **plattformunabhängig** entwickelt werden:
 - Mehrere Programme könnten redundanten Code enthalten
 - Die CPU selbst unterstützt nur begrenzt Multiprogramming... eine Software muss diese Aufgabe übernehmen
 - Daten, E/A-Zugriffe auf Ressourcen usw. müssen bei mehreren aktiven Applikationen vor gegenseitigen Zugriffen geschützt werden
 4. OS ist eine Schicht mit konsistentem **Interface zwischen Hardware und Software** womit Applikationen bequem, sicher und vielfältig zugreifen können
 5. **Abstrakte Ressourcen** und Verwaltung von Zugriffen auf diese

Hintergründe (2)

- Ein OS soll Applikationen verwalten, so dass diese:
 - auf Ressourcen zugreifen können
 - sich einen gemeinsamen Prozessor teilen können
 - den Prozessor und die E/A-Geräte effizient nutzen können

→ Alle modernen Betriebssysteme basieren auf einem Modell, wo die Ausführung einer Applikation der Existenz von mindestens einem Prozess entspricht

Ein Prozess...

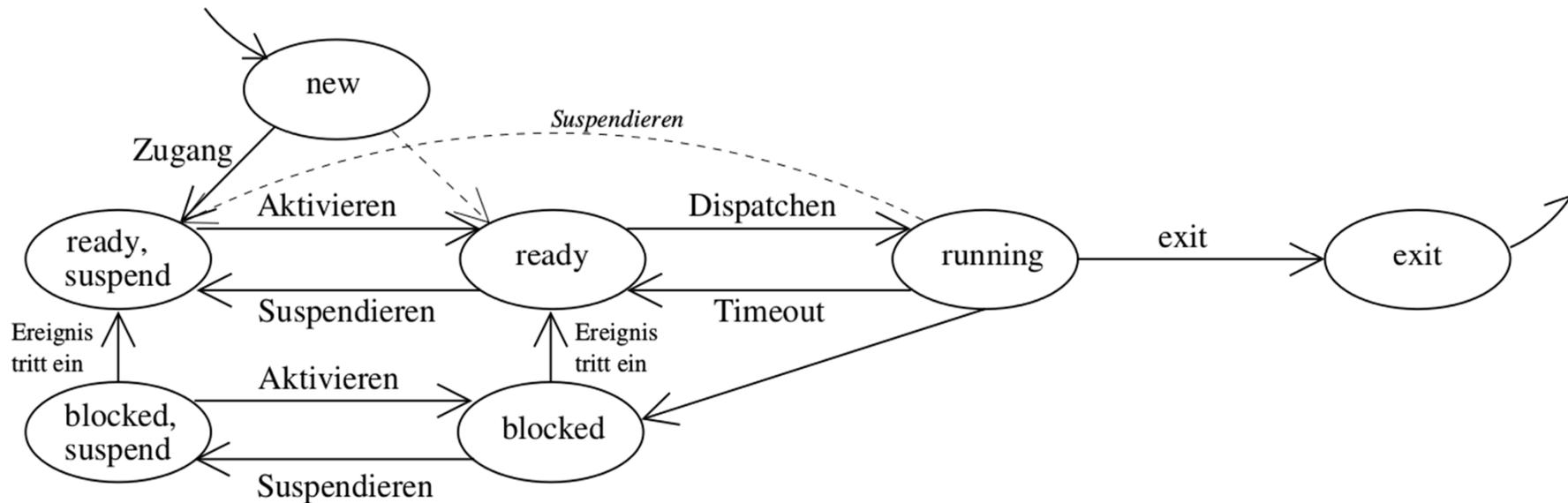
- kann als Eigentümer und Verwalter von Ressourcen betrachtet werden.
- verfügt über einen Bereich im Hauptspeicher.
- kann z.B. über zugeordnete E/A-Geräte, E/A-Kanäle und Dateien verfügen.

- kann als Einheit, die eine gewisse Aufgabe erledigt, betrachtet werden.
- ist ein in Ausführung befindliches (nicht zwingend aktives) Maschinenprogramm.

Beide Aufgaben sind unabhängig voneinander

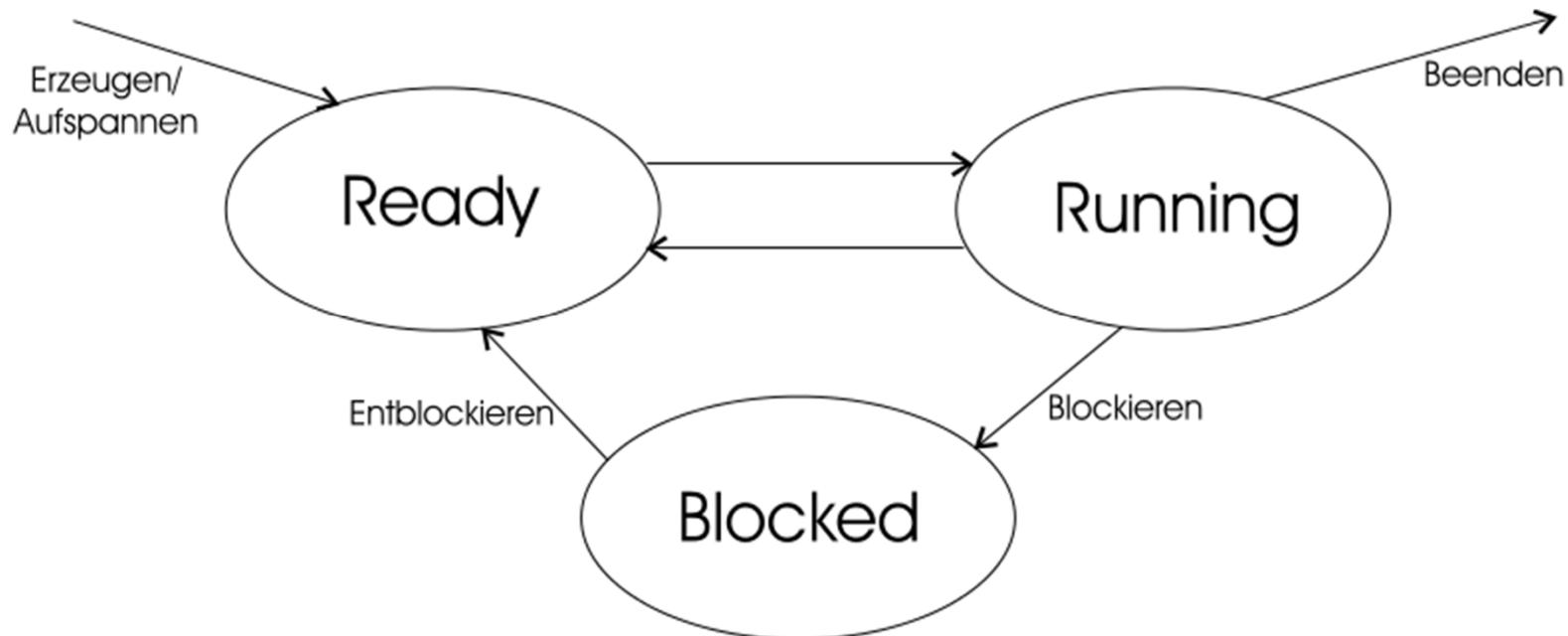
- Einheit, die den **Eigentümer von Ressourcen** bezeichnet, wird weiter **Prozess** genannt
- Einheit, die eine **gewisse Aufgabe** erledigt, wird **Thread (Faden)** genannt

7-Zustands-Prozessmodell:



- Ein blockierter Prozess kann in den Hintergrundspeicher ausgelagert werden
- Entspricht Übergang von **blocked** nach **blocked, suspend**

- Innerhalb eines Prozesses kann es mehrerer Threads geben
- Sind „leichtgewichtige Prozesse“
 - Teilen sich gleichen Adressraum
 - Haben gemeinsamen Zugriff auf Speicher und Ressourcen d. Prozesses
 - Eigener Stack für lokale Variablen
 - Eigener Deskriptor

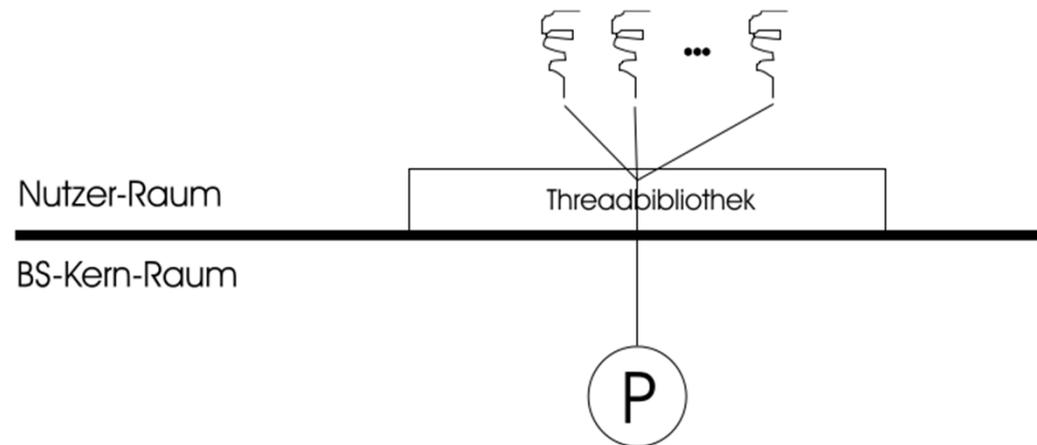


- Grundlegenden Zustände eines Threads: **Running**, **Ready** und **Blocked**
- Zustände zur Suspendierung von Threads nicht sinnvoll, da die Auslagerung ein Konzept von Prozessen ist
 - gemeinsamer Adressraum aller Threads eines Prozesses

Thread-Arten

User-Level-Threads (ULT)

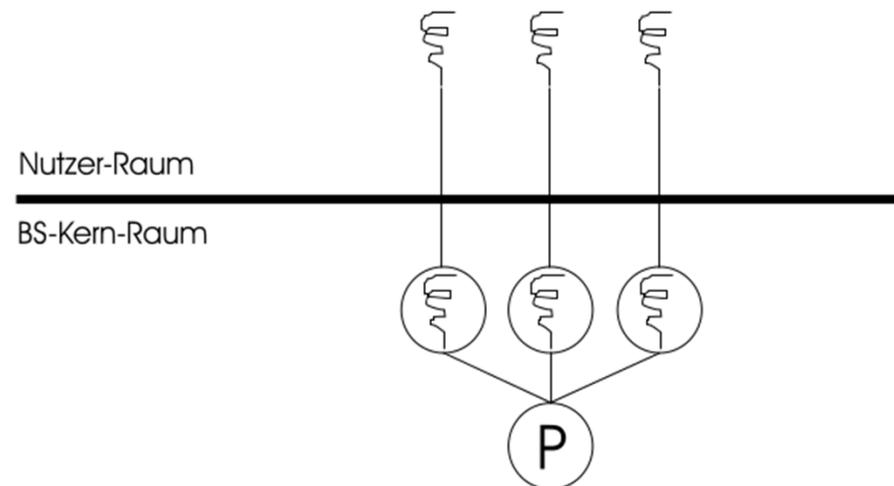
- Threadmanagement Aufgabe der Anwendung
- Threadbibliotheken eine Sammlung von Routinen zum Zwecke des User-level-Thread-Managements
- Anwendung kann jederzeit einen neuen Thread generieren, der innerhalb desselben Prozesses läuft



Thread-Arten

Kernel-Level-Threads (KLT)

- Threadmanagement vom Betriebssystemkern durchgeführt
 - muss vom Betriebssystem unterstützt werden
- KLTs deutlich schneller als Prozesse erzeugt, gewechselt und terminiert werden
- Ist ein Thread blockiert, so kann die Kontrolle einem anderen Thread desselben Prozesses (der sich im Zustand **ready** befindet) übergeben werden
- KLTs können in einer Multiprozessorumgebung echt parallel ausgeführt werden



Mögliche Abläufe:

Sequentielles Programm:

Anweisungen werden Schritt für Schritt hintereinander ausgeführt
("Single Thread of Control")

Paralleles Programm (Nebenläufig):

Anweisungen oder Teile der Anweisungen eines Programms werden
nebeneinander ausgeführt ("Multi Thread of Control")

Echt gleichzeitig parallel

Prozesse/Threads werden auf mehreren
Kernen echt parallel ausgeführt

Zeitlich verzahnt (quasi-parallel)

Prozesse/Threads teilen sich einen Kern
und werden somit durch Scheduling
unterbrochen und verzahnt ausgeführt

Vor- und Nachteile paralleler Programmierung

Vorteile:

- Komplexität in parallele Teilbereiche zerlegen
- Höherer Durchsatz
- Performanz
- Ausnutzung bei eingebetteten und verteilten Systemen

Nachteile:

- Erhöhte Komplexität
- Abläufe sind häufig schwer zu durchschauen
- Sehr fehleranfällig
- Schwer zu Debuggen bei Laufzeitfehlern
- Konzepte zur Synchronisation und Thread-Sicherheit erforderlich, um deterministisches Verhalten zu gewährleisten

Beispiele (I)

- Erfüllung gemeinsamer Aufgabe
 - Aufteilung komplexer mathematischer Berechnungen auf mehrere Prozessoren auf einem oder mehreren Rechnern
 - Beispiel: Parallele Algorithmen für die Matrixmultiplikation

- Sequentielle Multiplikation

$$C = A \cdot B \quad A, B, C \in \mathbb{R}^{2^k \times 2^k} \quad \Rightarrow \quad n^3 = n^{\log_2 8}$$

- Zerlegung in Blöcke

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \quad A_{ij}, B_{ij}, C_{ij} \in \mathbb{R}^{2^{k-1} \times 2^{k-1}}$$

$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$$

$$C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$$

$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$$

$$C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$$

\Rightarrow Auf Multicore Architektur bis zu vierfacher Beschleunigung

- Weitere Optimierungen möglich
 - (Strassen Algorithmus \rightarrow Nutzt Hilfsmatrizen)

Race Condition: Wettlaufsituationen, bei der der zeitliche Ablauf das Ergebnis bestimmt

Beispiel:

- Paralleler Zugriff auf ein Bankkonto (aktueller Kontostand 1000€)
- Prozess A: Verringert den Kontostand um 50€
- Prozess B: Erhöht den Kontostand um 100€

Zeitpunkt	Prozess A	Kontostand	Prozess B
1	Aktuellen Kontostand lesen (1000€)	1000€	Aktuellen Kontostand lesen (1000€)
2	Neuen Kontostand berechnen (950€)	1000€	Neuen Kontostand berechnen (1100€)
3	Neuen Kontostand schreiben	950€	Neuen Kontostand schreiben (Prozess B schreibt kurz nach Prozess A)
		1100€	

→ Zugriff muss synchronisiert werden

Kritische Abschnitte und wechselseitiger Ausschluss

Es gibt Ressourcen, die als ganzes oder bzgl. einzelner Operationen nur exklusiv, d.h. zu einem Zeitpunkt nur durch einen einzigen Prozess nutzbar sind.

kritischen Bereich:

Phase, in der ein Prozess gemeinsam benutzte (globale) Daten oder Betriebsmittel nutzt.

Kritische Bereiche erfordern den wechselseitigen Ausschluss konkurrierender Prozesse.

Anforderungen an eine korrekte Lösung:

1. **Mutual Exclusion:** Zu jedem Zeitpunkt darf sich höchstens ein Prozess im kritischen Bereich befinden.
2. **Progress:** Befindet sich kein Prozess im kritischen Bereich, aber es gibt einen Kandidaten für diesen Bereich, so hängt die Entscheidung, welcher Prozess ihn betreten darf, nur von diesem Kandidaten ab und fällt in endlicher Zeit.
3. **Bounded Waiting:** Zwischen der Anforderung eines Prozesses, in den kritischen Bereich einzutreten und dem tatsächlichen Eintreten in den kritischen Bereich kann eine gewisse Zeitdauer liegen. Es muss jedoch möglich sein, für diese Wartezeit eine endliche obere Schranke anzugeben.

Beispiel mit wechselseitigem Ausschluss

Beispiel:

- Paralleler Zugriff auf ein Bankkonto (aktueller Kontostand 1000€)
- Prozess A: Verringert den Kontostand um 50€
- Prozess B: Erhöht den Kontostand um 100€

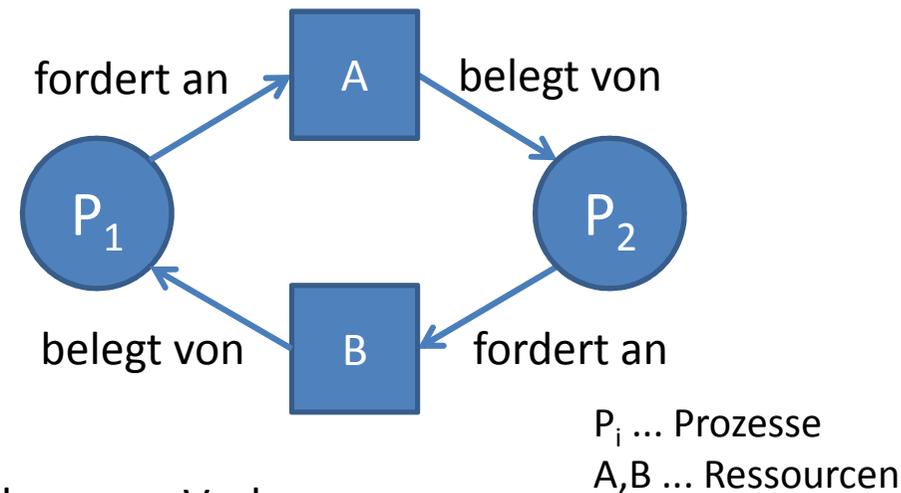
Zeitpunkt	Prozess A	Kontostand	Prozess B
1	Sperren des Kontos gegen fremde Zugriffe	1000€	
2	Aktuellen Kontostand lesen (1000€)	1000€	Versuch: Sperren des Kontos gegen fremde Zugriffe (da nicht möglich: warten)
3	Neuen Kontostand berechnen (950€)	1000€	...
4	Neuen Kontostand schreiben	950€	...
5	Aufhebung der Sperrung	950€	Sperren des Kontos gegen fremde Zugriffe (nun erfolgreich)
6		950€	Aktuellen Kontostand lesen (950€)
7		950€	Neuen Kontostand berechnen (1050€)
8		1050€	Neuen Kontostand schreiben
9			Aufhebung der Sperrung

Zugriff auf unteilbare Ressourcen (kritische Bereiche) birgt die Gefahr von Verklemmungen

Bedingungen:

- Mutual Exclusion: Mindestens 2 unteilbare Ressourcen
- Hold and Wait: Prozess hält Ressource, während er auf eine weitere wartet
- No Preemption: Prozess kann Ressource nicht entzogen werden, während er sie hält
- Circular Wait: Es entsteht eine zyklische Wartesituation

Prozess P ₁	Prozess P ₂
...	...
request(B)	request(A)
...	...
request(A)	request(B)
...	...



→ Mehr zur Deadlock Verhinderung in den kommen Vorlesungen

Software-Konstrukte für den wechselseitigen Ausschluss: Semaphore

Dijkstra, 1965

Ein Semaphor **S** ist eine „Integer-Variable“, die nur durch 3 **atomare Operationen** verändert werden kann. In den meisten Fällen ist einem Semaphor eine Warteschlange zugeordnet

- `init(S, Anfangswert)` setzt `s` auf den Anfangswert

```
S := Anfangswert ;
```

Technische Umsetzung z.B. durch speziellen Maschinenbefehl
Test and Set

- `wait(S)` oder auch `P(S)` versucht `S` zu verringern

```
S := S - 1 ;  
IF ( S < 0 ) THEN  
  <ordne Prozess in Warteschlange an Position -S ein>;
```

muss atomar
geschehen

- `signal(S)` oder auch `V(S)` erhöht `S`

```
S := S + 1 ;  
IF ( S < 0 ) THEN  
  <setze den am längsten wartenden Prozess rechnend>;  
  <erlaube Zugriff auf kritischen Bereich>;
```

muss atomar
geschehen

Software-Konstrukte für den wechselseitigen Ausschluss: Semaphore

Dijkstra, 1965

Verwendung

```
init(sema, 1);  
...  
wait(sema);  
  
<kritischer Bereich>;  
  
signal(sema);  
...  
  
<unkritischer Bereich>;  
...
```

← Initialisierung mit frei wählbarem Bezeichner

} Nutzung

Sämtliche Nutzer dieses kritischen Bereichs müssen den gleichen Semaphor verwenden

Software-Konstrukte für den wechselseitigen Ausschluss: Monitore

C.A.R. Hoare, 1974

Problem: Software-Qualität

In größeren Systemen:

- Synchronisationsoperationen (wait() und signal())
 - umgeben kritische Operationen (z.B. write)
 - müssen explizit gesetzt werden

→ Korrektheitsproblem

Die unabdingbare

- Vollständigkeit
- Symmetrie

der wait()- und signal()-Operationen ist schwierig erreichbar und nachweisbar

Software-Konstrukte für den wechselseitigen Ausschluss: Monitore

C.A.R. Hoare, 1974

Idee:

- Implizite/automatische Synchronisation kritischer Operationen
- Nutzung des Prinzips der Datenabstraktion

Hoare'sche Monitore:

Zusammenfassung von

- Daten
- darauf definierten Operationen
- der Zugriffssynchronisation
 - (einer Warteschlange für ankommende Prozesse/Threads)

zu einem abstrakten Datentyp, dessen Operationen wechselseitigen Ausschluss garantieren

→ Zugriff auf Daten über implizit synchronisierende Operationen

Software-Konstrukte für den wechselseitigen Ausschluss: Monitore

C.A.R. Hoare, 1974

Ziel:

Wechselseitiger Ausschlusses der Monitoroperationen \Leftrightarrow
zu jedem Zeitpunkt ist höchstens ein Thread in einem Monitor aktiv

Threads, die Zugriff auf den Monitor erfordern:

- haben ihn bereits wieder verlassen oder
- warten in der Warteschlange für ankommende Threads auf ihren Eintritt oder
- sind so lange suspendiert, bis der Monitor für sie wieder zur Verfügung steht

Beispiel:

```
Konto k = new Konto(1000);  
k.verringereKontostand(50);  
k.erhoeheKontostand(100);
```

Technische Umsetzung in der Regel durch einen Semaphore/Lock, der mit dem Objekt verbunden ist und am Eingang einer geschützten Methode allokiert und an deren Ende wieder freigegeben wird

Über das Objekt Konto und dessen Methoden wird sichergestellt, dass sich jeweils nur ein Thread in den Methoden (in einer von beiden) befindet (konkrete Beispiele später in Java).

(dem **Javakurs für Anfänger** von Kyrill Schmid/ Lorenz Schauer entnommen)

<http://www.mobile.ifi.lmu.de/lehveranstaltungen/java-fuer-anfaenger-ws1819/>

- Installation
- Der Weg zum ersten Programm:
 - Erstellen des Quellcodes
 - Kompilieren: Übersetzen des Quellcodes (.java) in Bytecode (.class)
 - Starten des Programms durch Übergabe des Bytecodes an den Interpreter JVM
- Kontrollstrukturen
- Datenstrukturen
- Vererbung

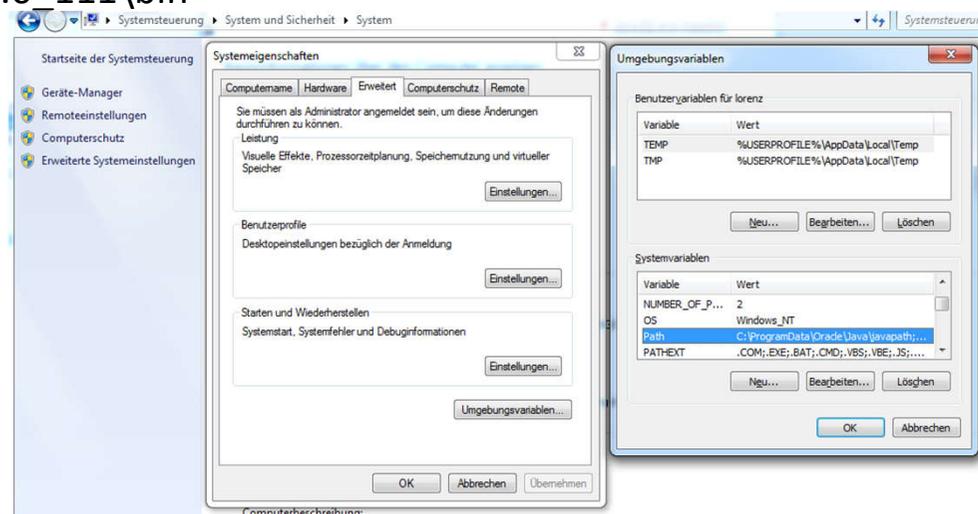
Die 3 wichtigsten Installationsvarianten von Java

- Java Standard Edition SE
 - Java Plattform für Desktop und einfache Serveranwendungen
 - Aktuell (Stand: August 2017) Java 8 Update 144 (Java SE 8u144)
 - Kostenlos auf
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- Java Enterprise Edition EE
 - Java Plattform für komplexere Server und Netzwerkanwendungen
- Java Micro Edition ME
 - Reduzierte Java Plattform für mobile Geräte und eingebettete Systeme
 - Oracle-Lizenz erforderlich

- Schritt 1: JDK Standard Edition SE downloaden
 - Auf <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
 - Richtige Plattform auswählen (Linux, Mac, Solaris, Windows)
 - Gespeicherte **jdk.exe** ausführen und installieren

Schritt 2: Ausführungspfad setzen

- Systemsteuerung -> System -> Erweiterte Systemeinstellungen auswählen -> Unter dem Reiter „Erweitert“ die Schaltfläche *Umgebungsvariablen* anklicken
- Pfad des JDK\bin eintragen. Bsp.:
 - C:\Program Files\Java\jdk1.8.0_111\bin
 - Trennung mit ;



Installationshinweise für OS X unter:

- https://docs.oracle.com/javase/8/docs/technotes/guides/install/mac_jdk.html

• Installationshinweise für Linux (Ubuntu) unter:

- https://wiki.ubuntuusers.de/Java/Installation/Oracle_Java/Java_8

Schritt 3: Installation testen

- In Konsole eingeben: `java -version`
- Ergebnis bspw.:
„`java version 1.8.0_102`“

```
Eingabeaufforderung
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\schauer>java- version
Der Befehl "java-" ist entweder falsch geschrieben oder
konnte nicht gefunden werden.

C:\Users\schauer>java -version
java version "1.8.0_102"
Java(TM) SE Runtime Environment (build 1.8.0_102-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, mixed mode)

C:\Users\schauer>_
```

Der Weg zum ersten Programm

3 Schritte sind zu durchlaufen:

- **Erstellen des Quellcodes**
- Kompilieren: Übersetzen des Quellcodes (.java) in Bytecode (.class)
- Starten des Programms durch Übergabe des Bytecodes an den Interpreter JVM

Der Quellcode kann mit jedem beliebigen Texteditor erzeugt werden.

- Bsp.: Notepad++ (Windows <https://notepad-plus-plus.org/download/v6.8.3.html>)
- Bsp.: Geany (für Linux)
- Java Datei anlegen:
 - Bsp.: NameOhneUmlaute.java
 - Entspricht auch dem Klassennamen! => *public class NameOhneUmlaute*
 - Konvention: Großer Anfangsbuchstabe für Klassen (also auch für Java-Dateien)
 - Bei Windows: Dateiendung einblenden, um .java statt .txt zu erzeugen

```
1 // Package Deklaration
2 package java.fuer.anfaenger;
3
4 // Import externe Klassen
5 import java.util.*;
6
7 // Klassendefinition
8 public class Student{
9
10     // Instanzvariablen
11     private int alter;
12     private String name;
13
14     // Konstruktor
15     public Student(String name){
16         this.name = name;
17     }
18
19     // Methode
20     public void studieren(int jahre){
21         System.out.println(name + " studiert schon "+jahre+" Jahre");
22     }
23 }
```

Der Weg zum ersten Programm

3 Schritte sind zu durchlaufen:

- Erstellen des Quellcodes
- **Kompilieren: Übersetzen des Quellcodes (.java) in Bytecode (.class)**
- Starten des Programms durch Übergabe des Bytecodes an den Interpreter JVM

In der Konsole (cmd bei Windows)

- Wechseln ins Verzeichnis der Java-Datei
- Kompilieren mit dem Befehl: **javac <NameDerDatei>.java**

```
C:\> Eingabeaufforderung
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\schauer>cd Documents

C:\Users\schauer\Documents>javac Test.java

C:\Users\schauer\Documents>
```

Der Weg zum ersten Programm

3 Schritte sind zu durchlaufen:

- Erstellen des Quellcodes
- Kompilieren: Übersetzen des Quellcodes (.java) in Bytecode (.class)
- **Starten des Programms durch Übergabe des Bytecodes an den Interpreter JVM**

Ausführen in der Konsole:

- mit dem Befehl: `java <Name>`
 - **Hinweis:** ohne Endung .class

```
cmd Eingabeaufforderung
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\schauer>cd Documents

C:\Users\schauer\Documents>javac Test.java

C:\Users\schauer\Documents>dir/W
Datenträger in Laufwerk C: ist OS
Volumeseriennummer: 8074-517E

Verzeichnis von C:\Users\schauer\Documents

[.]                [..]
Test.class          Test.java
                   532 Bytes
                   2 Datei(en),
                   3 Verzeichnis(se), 415.401.893.888 Bytes frei

C:\Users\schauer\Documents>java Test
Hello World!

C:\Users\schauer\Documents>
```

Ein Algorithmus lässt sich durch 3 Grundstrukturen beschreiben:

- Anweisungsfolge bzw. Sequenz
 - Wie bisher: Schrittweise Abarbeitung von Befehlen von oben nach unten
- Auswahlstruktur bzw. Selektion
 - Ermöglicht die **bedingte** Ausführung von Anweisungen
 - Bsp.: `if`, `else`, `switch-case`
- Wiederholungsstruktur bzw. Iteration oder Schleife
 - **Mehrmalige** Ausführung der gleichen Anweisungen
 - Beispiele: `while-`, `do-`, `for-`Schleifen

Primitive Typen

- **8** primitive Datentypen in Java vordefiniert
- Erfordern wenig Speicherplatz und geringen Aufwand für Compiler und Interpreter
=> Geschwindigkeitsvorteile

Datentyp	Verwendung	Größe in Bit	Wertebereich
boolean	Wahrheitswert	8	false, true
char	Zeichen	16	0 – 65.535
byte	Ganzzahl	8	-128 bis 127
short	Ganzzahl	16	- 32.768 bis 32.767
int	Ganzzahl	32	-2.147.483.648 bis 2.147.483.647
long	Ganzzahl	64	-2 ⁶³ bis 2 ⁶³ -1
float	Kommazahl	32	Genauigkeit ca. 7 Kommastellen
double	Kommazahl	64	Genauigkeit ca. 15 Kommastellen

Komplexe Typen

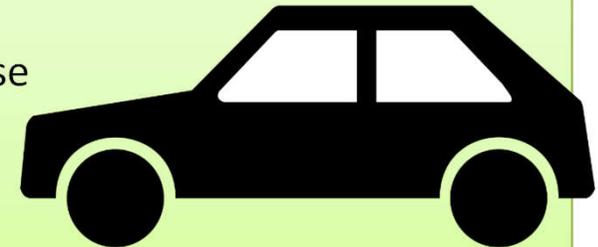
- Jede Klasse (mit mehreren Attributen)
 - Können sich aus verschiedenen Elementen zusammensetzen
 - Bsp.: String, Array, Auto, ...

- Komplexere Programme erfordern Strukturmechanismen
- Daher: **Objektorientierung** als vereinfachte Sichtweise auf komplexe Systeme
 - Klassen bieten eine **vereinfachte Sicht** der realen Welt
 - **Kapselung** von *Eigenschaften* und *Verhalten* gleichartiger Objekte in Klassen
 - **Wartbarkeit** durch Begrenzung von Änderungen auf einzelne Klassen
 - **Wiederverwendung** von bereits vorhandenen Klassen
 - durch Vererbung
- Java Programme bestehen i.d.R. aus **Objekten**, die über **Methodenaufrufe** miteinander **kommunizieren**

Zusammenhang zwischen Klassen und Objekte

Klasse:

- Wird vom Programmierer geschrieben, gespeichert und kompiliert
- Stellt ein Konzept bzw. Plan gleichartiger Objekte dar (Bsp.: Auto)
 - Welche **Eigenschaften** (Attribute) haben die Objekte der Klasse
 - (Bsp.: Name, Preis,...)
 - Welches **Verhalten** (Methoden) bieten die Objekte der Klasse
 - (Bsp.: fahren, bremsen,...)
- Beschreibt dadurch einen Teil der Realität



Objekt (= Instanz einer Klasse):

- Wird beim Ausführen des Programms erzeugt und spätestens beim Beenden verworfen
- Existiert nur im Speicher
- Wird nach dem vorgesehenen Konzept bzw. dem Plan seiner Klasse erstellt:
 - Bekommt Werte für seine Attribute
 - (Bsp.: „Audi Quattro“, 30.000 Euro,...)
 - Verhält sich wie in seiner Klasse definiert
 - (Bsp.: fahren, bremsen, ...)



```

1 // Package Deklaration
2 package java.fuer.anfaenger;
3
4 // Import externe Klassen
5 import java.util.*;
6
7 // Klassendefinition
8 public class Auto{
9
10 // Instanzvariablen
11 private String name;
12 private int preis;
13
14 // Konstruktor
15 public Auto(String name, int preis){
16     this.name = name;
17     this.preis = preis;
18 }
19
20 // Methoden
21 public void fahren(String a, String b){
22     System.out.println("Das Auto faehrt von "+a+" nach "+b);
23 }
24
25 public void bremsen(){
26     System.out.println("Ich bremsse!");
27 }
28
29 }

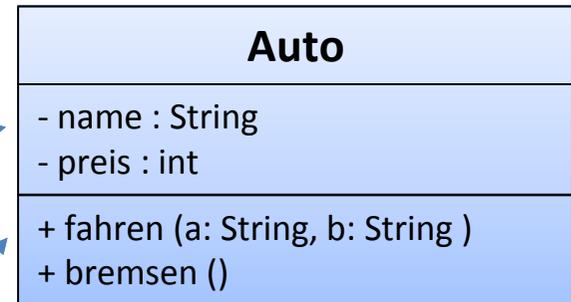
```

Eigenschaften (Attribute)

Konstruktor

Fähigkeiten (Methoden)

Instanz



```

1 public class Main {
2     public static void main(String[] args) {
3         Auto a = new Auto("Audi Quattro", 30000);
4         a.fahren("München", "Wien");
5     }
6 }

```

Bei früheren BS: Nur Simulation der verzahnten Ausführung durch die JVM

- Keine echte Parallelisierung
 - Falls keine direkte Thread-Unterstützung durch das BS möglich

Heute i.d.R.: JVM bildet die Threadverwaltung auf BS ab

- native Threads (bzw. KLT)
- Echte parallele Ausführung auf mehreren Kernen möglich!

Bringt einige Probleme mit sich:

- Nicht deterministisches Verhalten (Race-Conditions)
- Deadlocks
- Lebendigkeit (bsp.: Fairness)
- Sicherheit
- Ressourcenverbrauch

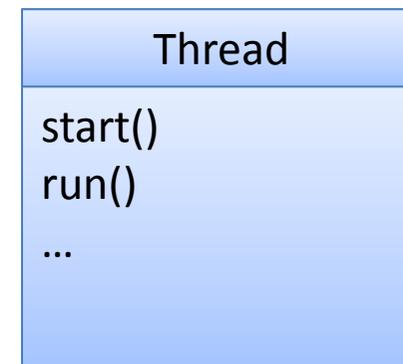
Achten auf Thread-Sicherheit und Synchronisation!

Die Java-Bibliothek besitzt eine Reihe von Klassen, Schnittstellen und Aufzählungen für Nebenläufigkeit:

- Thread
 - Jeder laufende Thread stellt ein Exemplar dieser Klasse dar
- Runnable
 - Programmcode, der parallel ausgeführt werden soll
- Lock
 - Mit Lock können kritische Bereiche markiert werden (nur 1 Thread innerhalb krit. Bereich)
- Condition
 - Threads können auf Benachrichtigungen anderer Threads warten

Threads in Java stellen Objekte der Klasse `java.lang.Thread` dar

- Die Klasse beinhaltet eine `run()` Methode, die den Code beinhaltet, der parallel ausgeführt werden soll
- Die `run`-Methode muss mittels `threadinstanz.start()` ausgeführt werden, damit der Code parallel zum aufrufenden Thread ausgeführt wird!
- **Achtung:** Ruft man `threadinstanz.run()` auf, wird der Code in der `run`-Methode „ganz normal“ also sequentiell ausgeführt



Prinzipiell stehen uns 4 Möglichkeiten zur Verfügung, um Threads in Java programmatisch zu erzeugen:

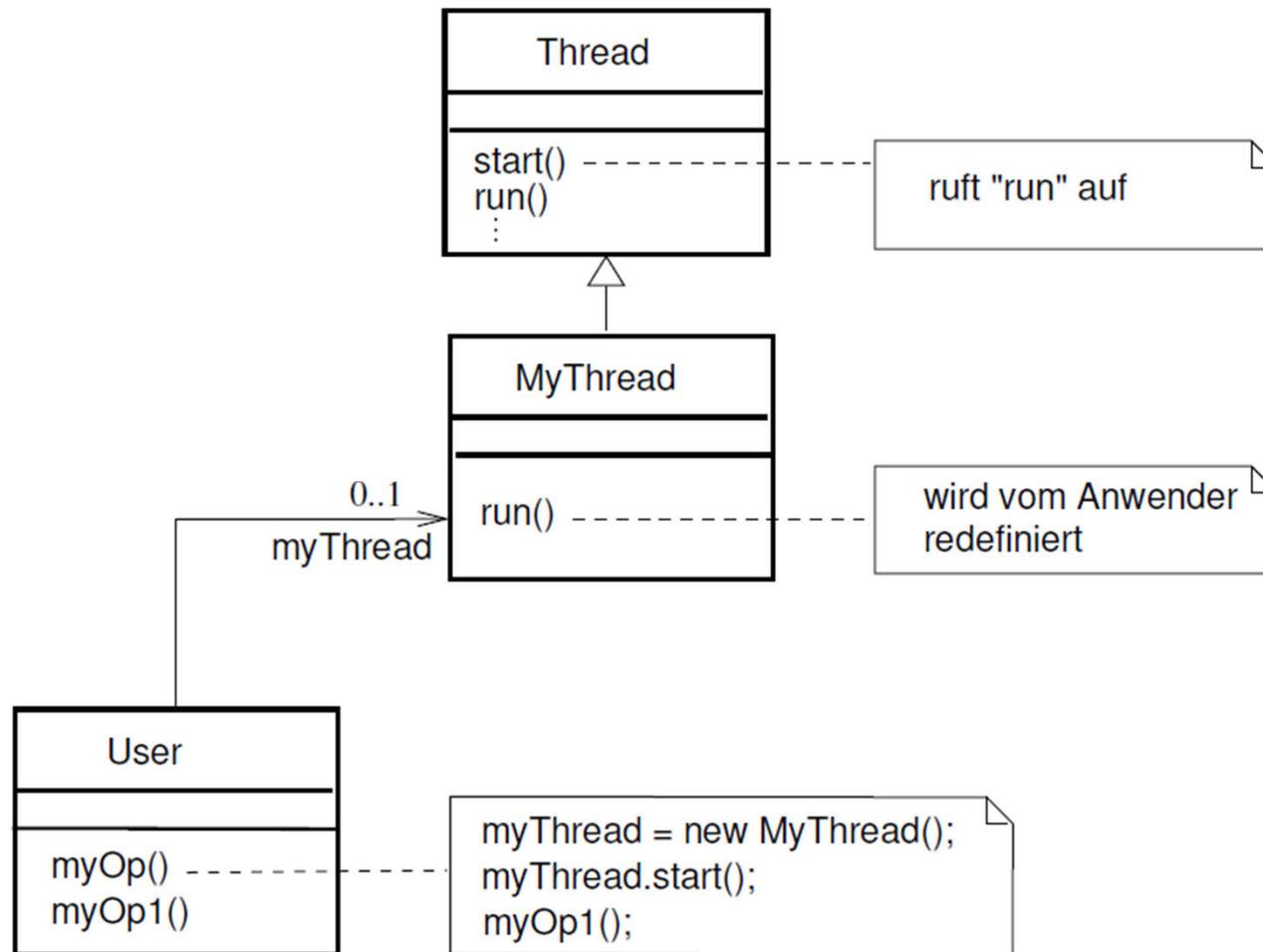
- Thread als eigene Klasse (Datei):
 - Erben von der Klasse Thread
 - Implementieren des Interfaces **Runnable**

- Threads direkt im Quellcode:
 - Anonyme Klasse
 - Entweder **Runnable** oder im Konstruktor von Thread
 - Lambda Ausdrücke (Seit Java 8)

Die 4 Möglichkeiten basieren auf den beiden Konzepten:

- Threads über Vererbung
 - Nachteil: Das Erben einer weiteren Klasse ist nicht möglich!
- Threads über Interface **Runnable**

Threads mittels Vererbung



Quelle: Skript Parallele Programmierung. R. Hennicker 2011

Beispiel-Implementierung mittels Vererbung:

```
// User
public class User {

    public void myOp(){
        MyThread t = new MyThread("Peter");
        t.start();
    }

    public void myOp1(){
        System.out.println("Operation 1.");
    }

    public static void main(String[] args) {
        User u = new User();
        u.myOp();
        u.myOp1();
    }
}
```

```
// MyThread
public class MyThread extends Thread{

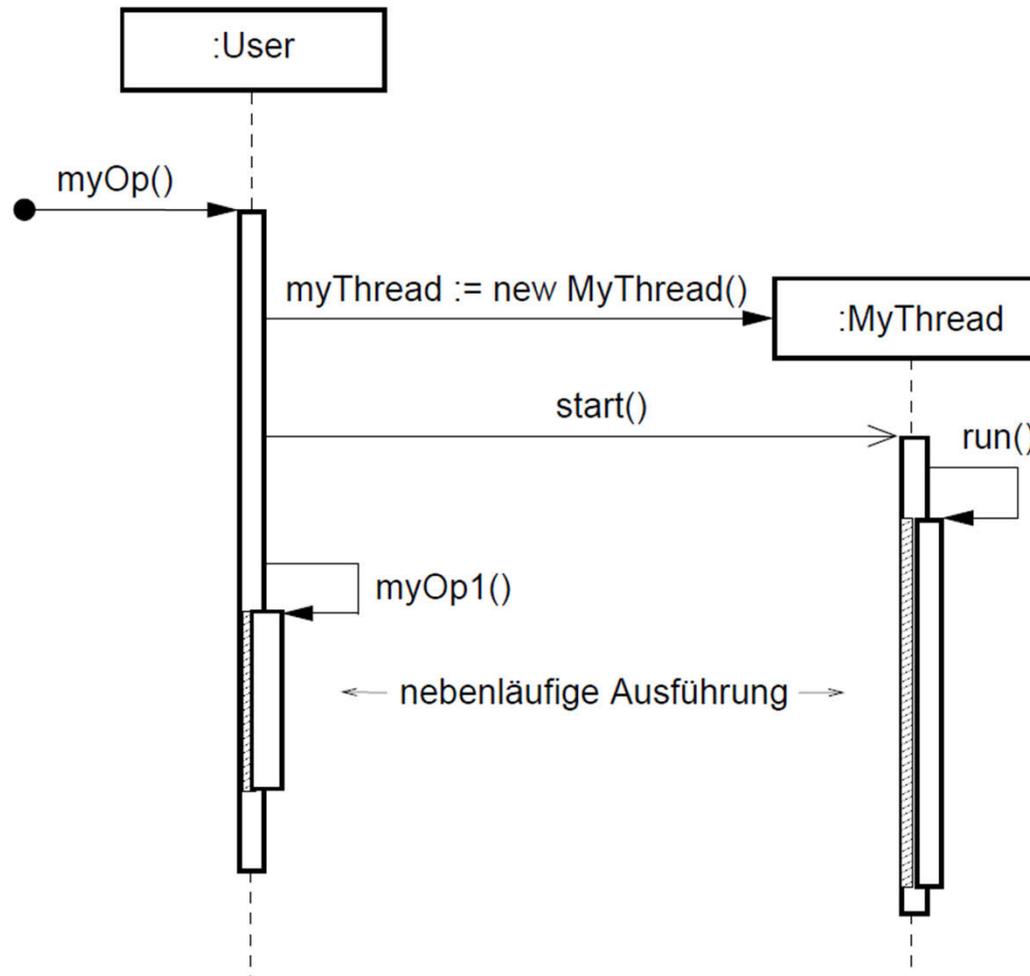
    private String name = "";

    public MyThread(String name){
        this.name = name;
    }

    @Override
    public void run() {
        System.out.println("Hallo ich bin
        "+this.name);
    }
}
```

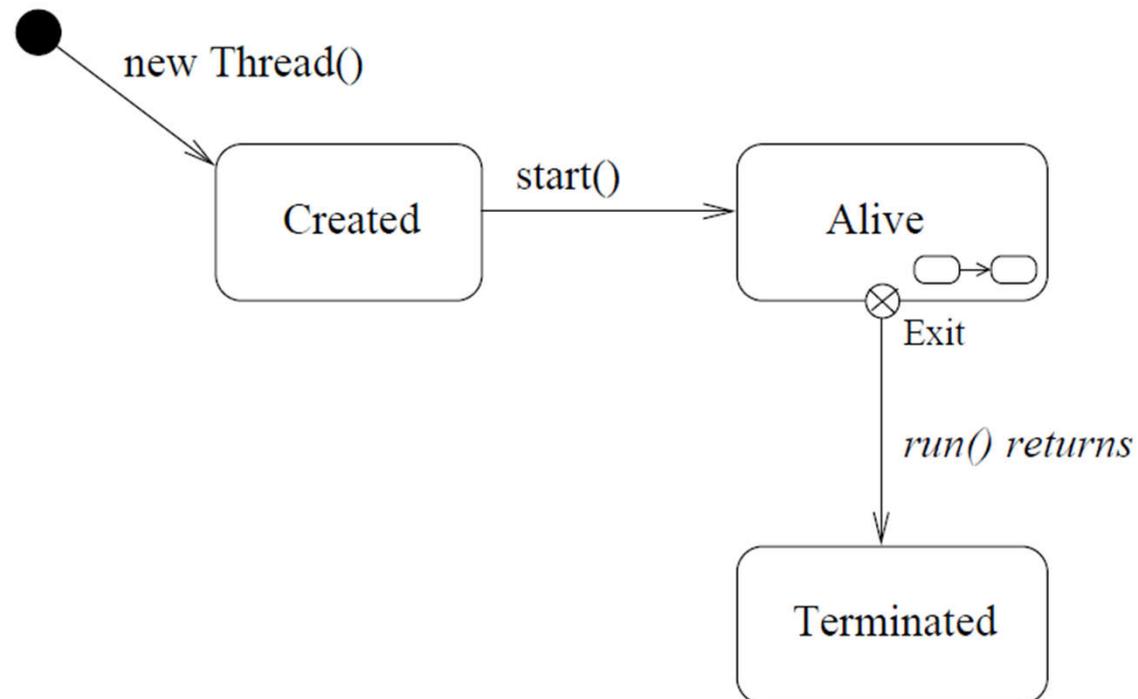
Threads mittels Vererbung

Was passiert: Sequenzdiagramm:



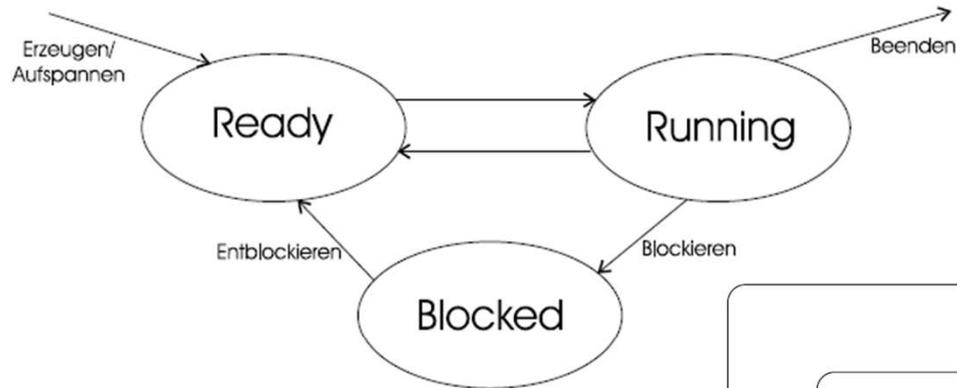
Quelle: Skript Parallele Programmierung. R. Hennicker 2011

Lebenszyklus eines Java-Threads

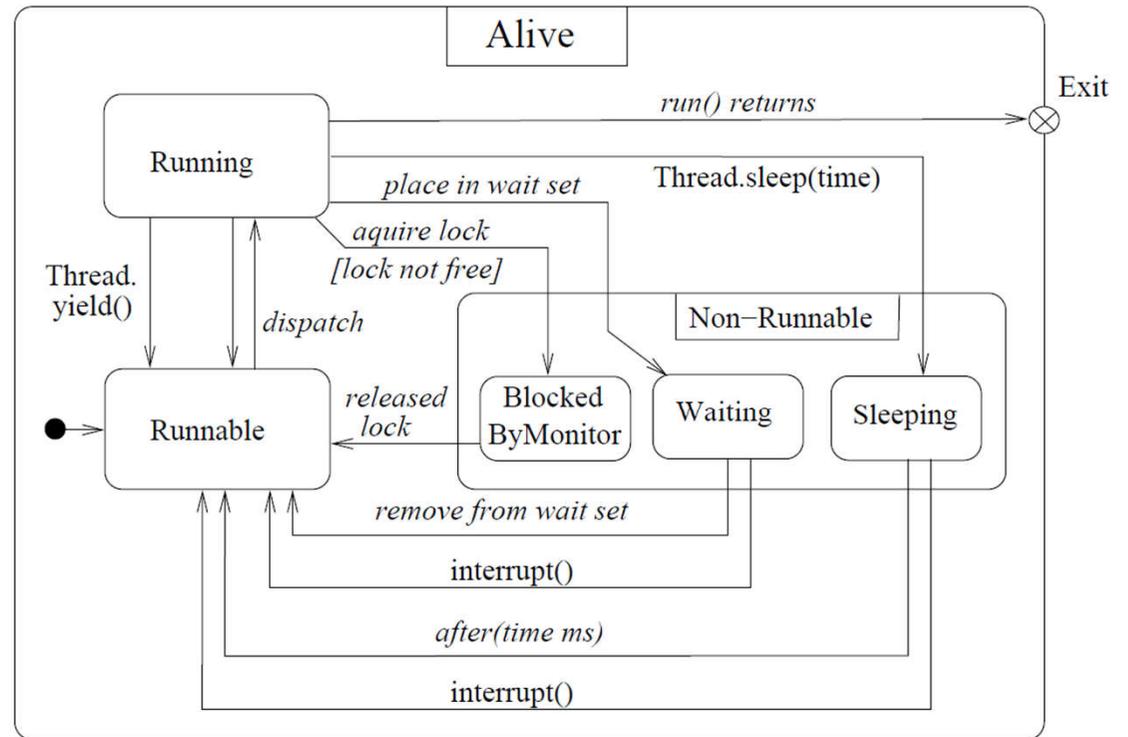


Quelle: Skript Parallele Programmierung. R. Hennicker 2011

- Threadzustände (Alive)



Quelle: Skript Betriebssysteme. C. Linnhoff 2015



Motivation

- Threads verwalten ihre eigenen Daten
 - lokale Variablen
 - und einen Stack
- Sie stören sich also selbst nicht
- Auch Lesen von gemeinsamen Daten ist unbedenklich
- Schreiboperationen sind jedoch kritisch!

- Probleme können durch Scheduling entstehen:
 - Ein Thread arbeitet gerade an Daten, die ein anderer Thread bearbeitet.
 - Hier können gravierende und schwer vorhersehbare Inkonsistenzen entstehen!

- Wir brauchen also Mechanismen, die uns davor schützen!

Kritische Abschnitte im Hinblick auf Threads

- Wenn wir mehrere Threads haben und Programmblöcke, auf die immer nur ein Thread zugreifen sollte, dann müssen diese kritischen Abschnitte geschützt werden!
 - Ist zur gleichen Zeit immer nur ein Thread beim Abarbeiten eines Programmteils, dann liegt ein wechselseitiger Ausschluss bzw. eine atomare Operation vor
 - Arbeitet ein Programm bei nebenläufigen Threads falsch, ist es nicht *thread-sicher* (engl. *thread-safe*).

Prinzipiell sollten kritische Abschnitte und nicht atomare Schreibeoperationen geschützt sein.

- Manche Befehle sehen atomar aus, sind es aber nicht.
 - Bsp.: `i++`

```
// Was passiert bei i++?
```

1. `i` wird gelesen und auf dem Stack abgelegt
2. Danach wird die Konstante 1 auf dem Stack abgelegt
3. Und anschließend werden beide Werte addiert
4. Das Ergebnis wird nun vom Stack geholt und in `i` geschrieben

Aus diesem Grund müsste `i++` geschützt ausgeführt werden.

- Java-Konstrukte zum Schutz der kritischen Abschnitte
 - `Synchronized`
 - Die Schnittstelle `java.util.concurrent.locks.Lock`
 - Wird u.a. von `ReentrantLock` implementiert

Monitore:

- Werden implizit durch die JVM erstellt
- Werden durch eine automatisch verwaltete Sperre realisiert
- Jedes Objekt verfügt über eine Sperre (Lock)
- Eintrittspunkte der Monitore müssen mit dem Schlüsselwort `synchronized` markiert sein
 - Für Klassenmethoden
 - Bsp.: `public synchronized static void doIt(){...}`
 - Für Objektmethoden
 - Bsp.: `public synchronized void makeIt(){...}`
 - Für Blöcke
 - Bsp.: `synchronized (objMitMonitor){...}`
- Ein solcher Eintrittspunkt kann nur betreten werden, wenn das Lock verfügbar ist.
 - **Ansonsten muss der Thread warten**, solange bis das Lock verfügbar ist.

- Ein freies Lock wird beim Betreten einer `synchronized` Methode/Block durch den aufrufenden Thread belegt (oder gehalten)
 - Daraufhin kann kein anderer Thread mehr eine synchronisierte Methode des Objekts betreten (solange bis das Lock wieder freigegeben wird)
- Die statische Methode `Thread.holdsLock()` zeigt an, ob der aktuelle Thread das Lock hält.
- Ein gehaltenes Lock wird freigegeben, wenn:
 - ... die synchronisierte Methode verlassen wird
 - ... eine Ausnahme erfolgt
 - ... ein `wait()` Aufruf getätigt wird.

Durch Ausnahme (Exception)

- Lock wird bei einer unbehandelten `RuntimeException` in einer `synchronized` Methode/Block automatisch durch JVM freigegeben
 - Da bei einer Exception der Block automatisch verlassen wird

Durch Aufruf von `wait()`

- Thread beendet die Abarbeitung
- Geht in den „Blocked“ Zustand
 - Reiht sich in die Warteschlange des Objekts ein
- Das Lock wird freigegeben
- Der Thread wartet nun auf eine Benachrichtigung durch (`notify()` oder `notifyAll()`), dass er wieder weiterarbeiten darf
- Wartende Threads können auch durch einen Interrupt unterbrochen werden
 - Daher: `throws InterruptedException` (Muss abgefangen werden!)
- Aufruf von `wait()`, `notify()` oder `notifyAll()` nur möglich, wenn Lock gehalten wird!
 - Ansonsten Laufzeitfehler: `IllegalMonitorStateException`

Zusammenspiel zwischen `wait()`, `notify()` und `notifyAll()`

Zusammenfassung der Methoden:

- `void wait()` throws `InterruptedException`
 - Thread wartet an dem aufrufenden Objekt darauf, dass er nach einem `notify()` bzw. `notifyAll()` weiterarbeiten kann.
- `void wait(long timeout)` throws `InterruptedException`
 - Wartet auf ein `notify()/notifyAll()` maximal aber eine gegebene Anzahl von Millisekunden. Nach Ablauf dieser Zeit ist er wieder rechenbereit.
- `void wait(long timeout, int nanos)` throws `InterruptedException`
 - Etwas spezifischer als vorher
- `void notify()`
 - Weckt einen beliebigen Thread auf, der an diesem Objekt wartet und sich wieder um das Lock bemühen kann.
 - Erhält er das Lock, kann er die Bearbeitung fortführen.
- `void notifyAll()`
 - Benachrichtigt alle Threads, die auf dieses Objekt warten.

Hinweis: `notify()` bzw. `notifyAll()` i.d.R., wenn aufrufender Thread dann auch das Lock freigibt (also am Ende eines `synchronized` Blocks)

- Sonst werden die Threads zwar aufgeweckt, aber das Lock ist immer noch vom aktuellen Thread belegt („signal and continue“-Prinzip)

Zusammenspiel zwischen `wait()`, `notify()` und `notifyAll()`

Allgemeines Beispiel zum Zusammenspiel zwischen `wait()` und `notify()`

```
public class MeineKlasse{  
  
    private Data state;  
  
    public synchronized void op1() throws InterruptedException {  
        while (!cond1) wait();  
        // modify monitor state  
        notify();  
        // or notifyAll();  
    }  
  
    public synchronized void op2() throws InterruptedException {  
        while (!cond2) wait();  
        // modify monitor state  
        notify();  
        // or notifyAll();  
    }  
}
```

Die `while`-Schleife wird deshalb gebraucht, da bei Fortführung die Synchronisationsbedingung nicht notwendigerweise gelten muss! Ein einfaches `if` kann evtl. nicht ausreichen

Vereinfachtes Beispiel eines Kontos

Beispiel eines Kontos einer Person, um das Arbeitgeber und Vermieter konkurrieren

- hier noch fehlerhaft ohne Synchronisation

```

1 public class Konto {
2     private int Kontostand;
3
4     public Konto(int Anfangswert) {
5         Kontostand = Anfangswert;
6     }
7
8     public void erhoehen(int Wert){
9         Kontostand = Kontostand + Wert;
10        System.out.println("Kontostand: " + Kontostand);
11    }
12    public void verringern(int Wert){
13        Kontostand = Kontostand - Wert;
14        System.out.println("Kontostand: " + Kontostand);
15    }
16 }

```

```

1 public class Main {
2     public static void main(String[] args) {
3         Konto kontoVonPersonX = new Konto(1000);
4
5         Arbeitgeber arbeitgeberY = new Arbeitgeber(kontoVonPersonX);
6         arbeitgeberY.start();
7
8         Vermieter vermmieterZ = new Vermieter(kontoVonPersonX);
9         vermmieterZ.start();
10    }
11 }

```

```

1 public class Arbeitgeber extends Thread {
2     Konto k;
3
4     Arbeitgeber(Konto kontoEinerPerson) {
5         k = kontoEinerPerson;
6     }
7
8     public void run() {
9         while(true){
10            k.erhoehen(10);
11        }
12    }
13 }

```

```

1 public class Vermieter extends Thread {
2     Konto k;
3
4     Vermieter(Konto kontoEinerPerson) {
5         k = kontoEinerPerson;
6     }
7
8     public void run() {
9         while(true){
10            k.verringern(500);
11        }
12    }
13 }

```

Vereinfachtes Beispiel eines Kontos

Beispiel eines Kontos einer Person, um das Arbeitgeber und Vermieter konkurrieren

- hier mit Synchronisation und Sicherstellung, dass der Kontostand nicht unter 0 fällt (auf eine Art, die dem Vermieter nicht gefallen dürfte)
- Klassen: Arbeitgeber, Vermieter und Main unverändert (deshalb nicht aufgeführt)

```

1 public class Konto {
2     private int Kontostand;
3
4     public Konto(int Anfangswert) {
5         Kontostand = Anfangswert;
6     }
7
8     public synchronized void erhoehen(int Wert){
9         Kontostand = Kontostand + Wert;
10        System.out.println("Kontostand: " + Kontostand);
11        notify();
12    }
13    public synchronized void verringern(int Wert) {
14        while (Kontostand - Wert < 0){
15            try {
16                wait();
17            } catch (InterruptedException e) {
18                e.printStackTrace();
19            }
20        }
21
22        Kontostand = Kontostand - Wert;
23        System.out.println("Kontostand: " + Kontostand);
24    }
25 }

```

Warum nicht gleich alles synchronisieren?

Unerwünschte Nebeneffekte können also durch Markieren der kritischen Abschnitte mittels `synchronized` verhindert werden!

Warum dann nicht gleich jede Methode synchronisieren?

Antwort: Führt zu anderen Problemen:

- Synchronisierte Methoden müssen von JVM verwaltet werden, um wechselseitigen Ausschluss zu ermöglichen.
 - Threads müssen auf andere warten können
 - Das erfordert eine Datenstruktur, in der wartende Threads eingetragen und ausgewählt werden
=> Kostet Zeit und Ressourcen!
- Unnötig und falsch synchronisierte Blöcke machen die Vorteile von Mehrprozessormaschinen zunichte.
 - Lange Methodenrümpfe erhöhen die Wartezeit für die anderen!
- Deadlock-Gefahr steigt!



Quelle: http://openbook.rheinwerk-verlag.de/javainsel9/bilder/365_java_09_004.gif

VIELEN DANK