

AI-B.41: Verteilte Systeme

- Lecture Notes [SL] -

IX. Koordination & Synchronisation [I]

C. Schmidt | SG AI | FB 4 | HTW Berlin

Urheberin: Prof. Dr. Christin Schmidt
Verwertungsrechte: keine außerhalb des Moduls

Rückblick letzte Vorlesung

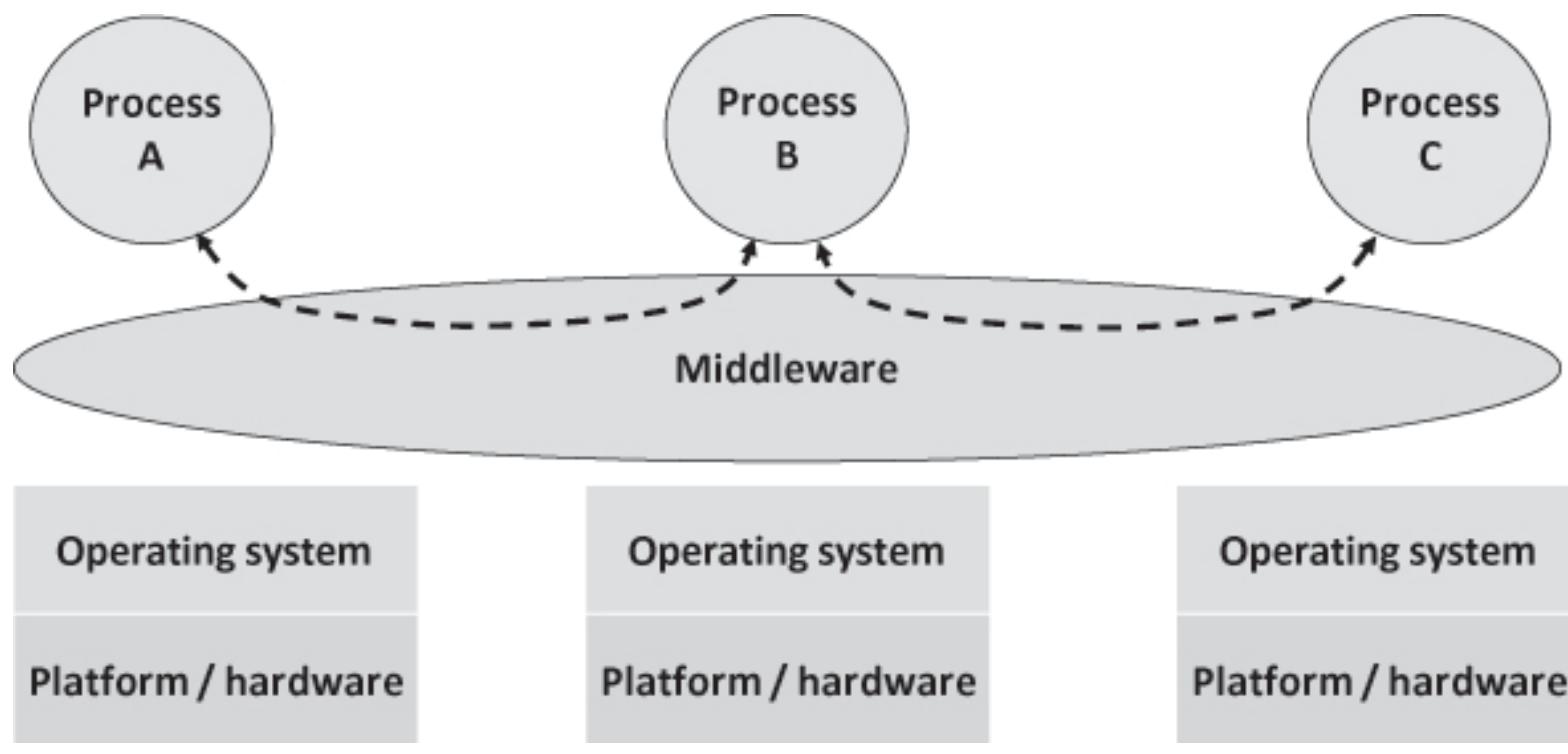
Nach dieser Lehrveranstaltung kennen Studierende idealerweise:

- Erweiterungen elementarer Kommunikationsmechanismen der Interprozesskommunikation auf Basis von Sockets (vgl. Lecture Notes IPC Teil I)
 - Middleware als Softwareschicht zwischen Transport- und Anwendungsschicht zur Erhöhung von Transparenz und Überwindung von Interoperabilitätsproblemen aufgrund von Heterogenität
 - Beispiele für Middleware-Kommunikationsformen sowie Prinzipien verteilter Aufrufe
 - Die Bedeutung von Schnittstellen für Anwendungsentwickler_innen
 - Ausgewählte Beispiele für höhere Konzepte bei der Interprozesskommunikation:
 - Remote Procedure Call (RPC: entfernte Prozeduraufrufe)
 - Remote Method Invocation (RMI: entfernter Methodenaufruf)
 - Entwicklungsschritte verteilter Aufrufe am praktischen Beispiel einer einfachen RMI-Anwendung
 - Serverzustände
- ✓ Studierende kennen grundlegende Aspekte der Interprozesskommunikation auf Middleware-Ebene, Eigenschaften damit verbundener ausgewählter Konzepte und Technologien sowie deren grundlegende Funktionsweise im Zusammenspiel beteiligter Komponenten. Zudem lernen Studierende praktische Beispiele für die Implementierung einfacher verteilter Systeme mittels entfernter Aufrufe und (entfernten) Schnittstellen kennen, welche als Grundlage sowie Bestandteil weiterführender Standards und Middleware-Kommunikationsformen dienen.

Ausgangspunkt [I]

vgl. Schill & Springer [2012:52ff.]

- Bisher sind wir davon ausgegangen, dass miteinander kommunizierende / interagierende Prozesse (z.B. Client, Server) jeweils durch einen Prozess auf Betriebssystemebene implementiert werden (vgl. Lecture Notes IPC I & II).
- Dies ist ausreichend für einfache Anwendungen, allerdings nicht gut skalierbar!

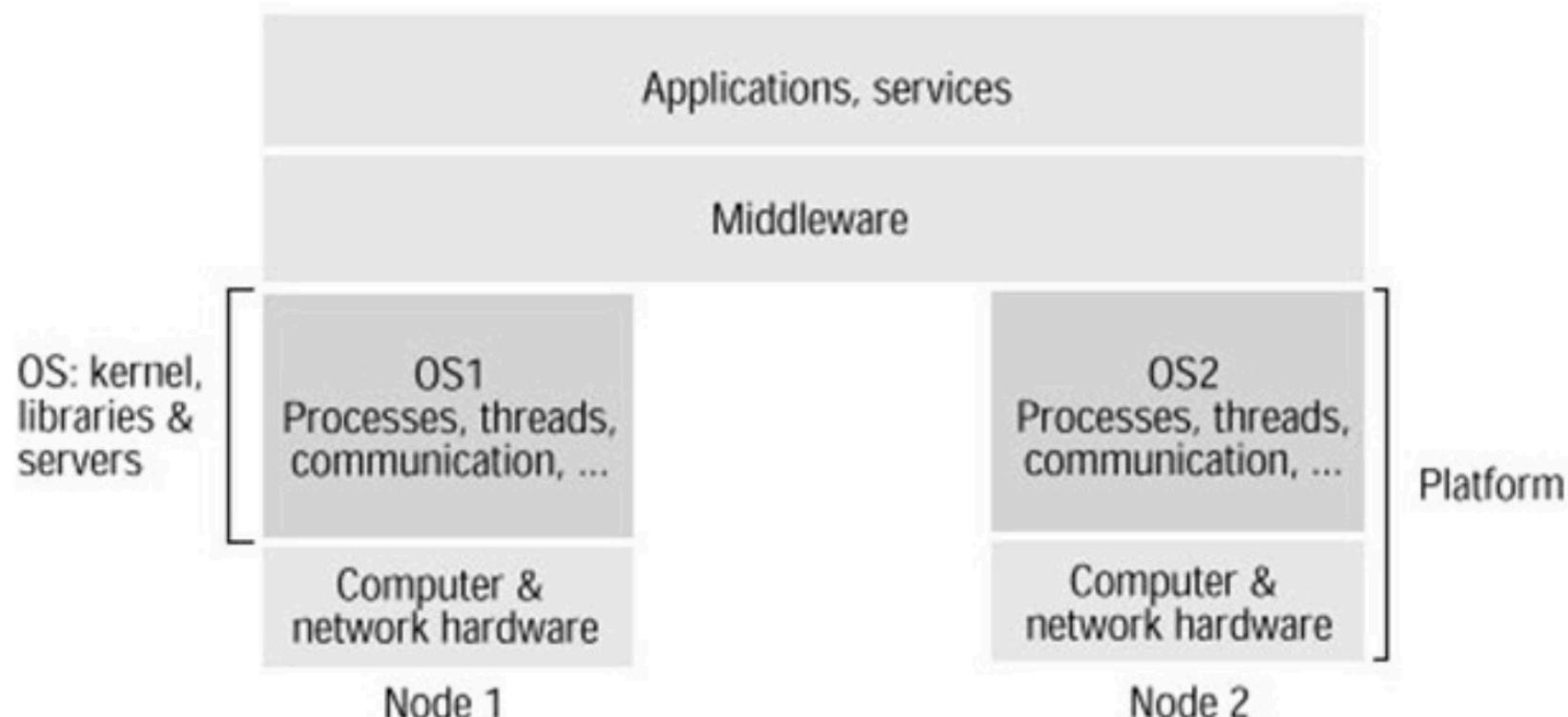


Quelle: Anthony [2016:332]

Ausgangspunkt [II]

vgl. Schill & Springer [2012:52ff.]

- ✓ Bessere Skalierbarkeit durch (quasi-)parallele Prozesse auf Seiten der Knoten
- ✓ Technik: Threads / Leichtgewichtige Prozesse



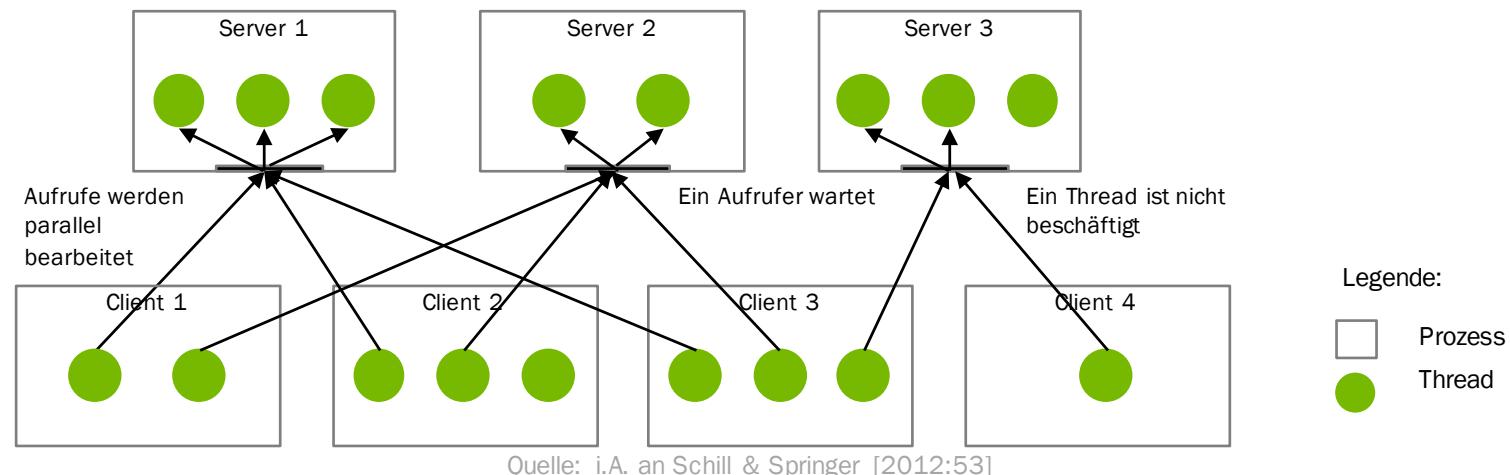
Quelle: Coulouris et al. [2002:252]

Ausgangspunkt [III]

vgl. Weber [1998:101ff.]

Problemstellungen / Randbedingungen:

- Die relevante Information ist über mehrere Maschinen verteilt.
- Entscheidungen werden auf lokalen, eventuell unvollständigen Informationen gefällt.
- Ein Single-Point-of-Failure, d.h. die Fehlfunktion des Gesamtsystems durch Fehlfunktionen einer Einzelkomponente, sollte vermieden werden.
- Das verteilte System arbeitet mit nebenläufigen Prozessen ohne gemeinsamen Adressraum (Fokus der heutigen Lehrveranstaltung, vgl. ff.).
- Es gibt keine gemeinsame Zeitbasis (Fokus der Lehrveranstaltung in der kommenden Woche, vgl. Lecture Notes Synchronisation / Koordination II).



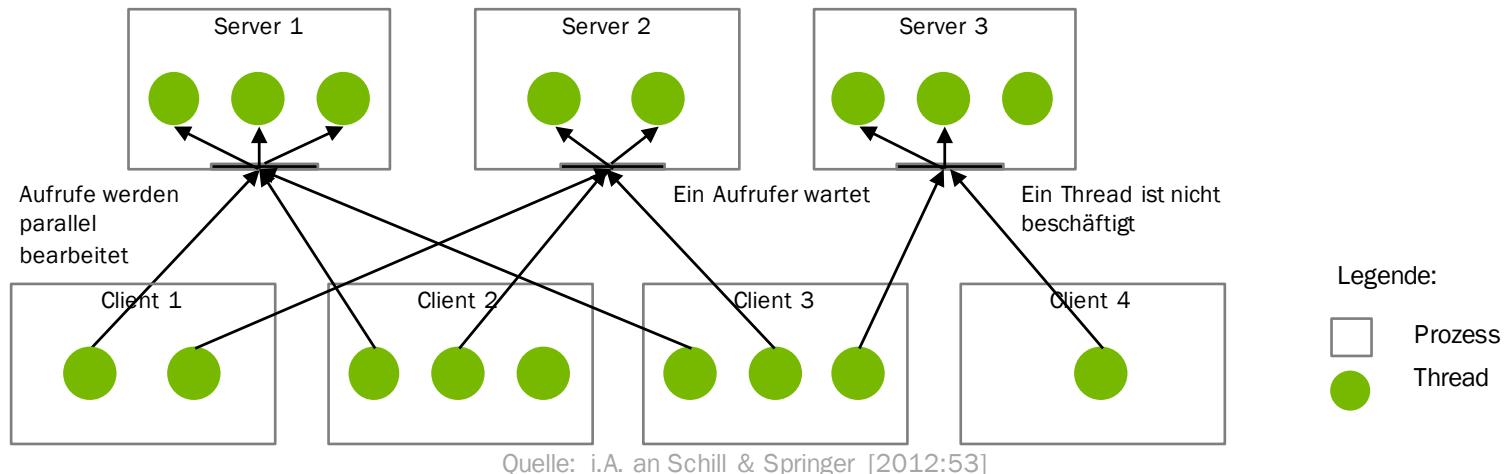
Lernziele heute

Nach dieser Lehrveranstaltung kennen Studierende idealerweise:

- Abgrenzende Eigenschaften der Begriffe Nebenläufigkeit, Parallelität, Konkurrenz und Kooperation im Kontext von Prozessen
- Aspekte der Nebenläufigkeit im Kontext von Betriebsmitteln, welche vom Betriebssystem bereitgestellt werden
- Den Unterschied zwischen Prozessen und Threads sowie den Unterschied bei der Kommunikation zwischen Threads innerhalb eines Prozesses und zwischen Prozessen
- Eigenschaften von Threads als leichtgewichtige Prozesse auf user- und kernel-Level sowie damit zusammenhängende Vor- und Nachteile von Threads
- Praktische Beispiele zur Erzeugung und zum Umgang mit Threads und nebenläufiger Programmierung in JAVA mit Schwerpunkten auf
 - » Grundlegenden Thread & Lock-Primitiven und Mechanismen zur Thread-Manipulation
 - » (A)synchroner Kooperation/Koordination
 - » Synchronisierung
- Perspektiven, Kategorien und Funktionsweisen thread-basierter Systeme mit Fokus auf serverseitige Organisationsformen
- ✓ Studierende kennen grundlegende Aspekte hinsichtlich der Koordination und Synchronisation von nebenläufigen Prozessen und leichtgewichtigen Threads bei der Interprozesskommunikation. Zudem lernen Studierende praktische Beispiele für die Implementierung einfacher Synchronisations- und Koordinationsmechanismen und deren grundsätzliche Einsatzmöglichkeiten im Zusammenspiel verschiedener Knoten eines verteilten Systems.

Nebenläufigkeit (engl.: „concurrency“) [I]

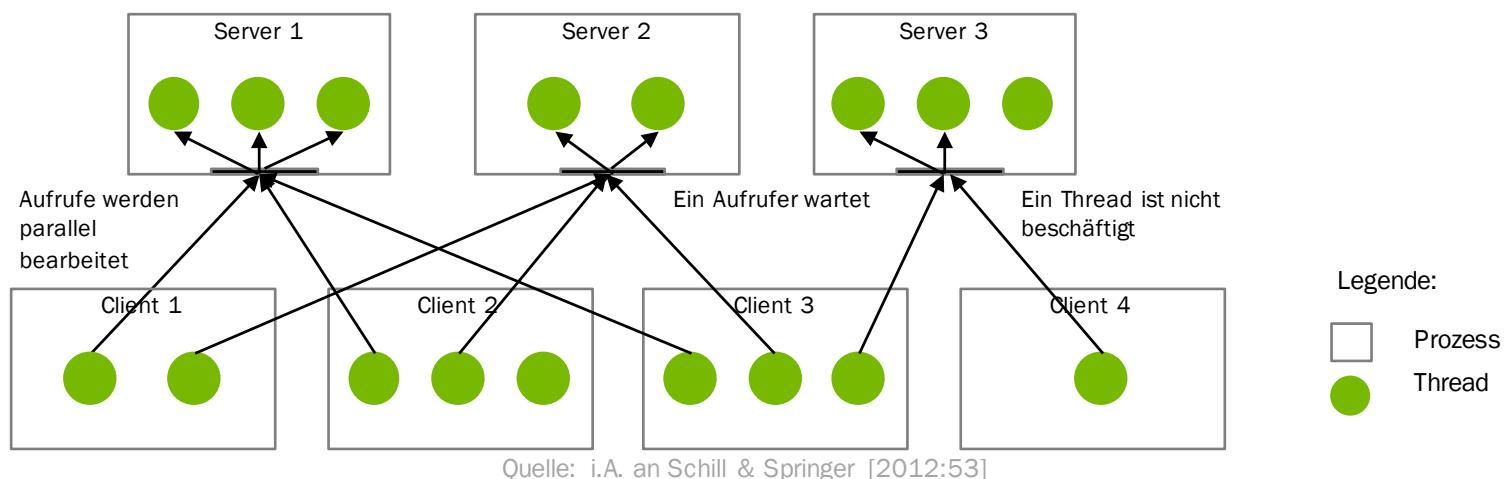
- Ausgangspunkt: Traditionelle Programmieransätze
 - » Programme laufen normalerweise sequentiell ab
 - Aktivitäten werden nacheinander ausgeführt
 - Gleichzeitige Aktivitäten sind nicht möglich
- Beispiel: Stellen Sie sich eine einfache Animation auf einer grafischen Oberfläche vor
 - » Gewünschte Funktionen/Prozesse: Blinken, Laufschrift über Benutzereingabe
 - ✓ Lösungen mit herkömmlichen Schleifen führen nicht zum Ziel!



Nebenläufigkeit (engl.: „concurrency“) [II]

- Nebenläufigkeit (engl.: „concurrency“): Mehrere Vorgänge laufen gleichzeitig auf einem Rechner ab
 - » Parallelität: echte Gleichzeitigkeit (mehrere Prozessoren/mehrere Kerne)
 - » „Pseudo-Parallelität“: Vortäuschung von Gleichzeitigkeit durch schnelles Umschalten zwischen Vorgängen (z.B. bei einem Prozessor/einem Kern)
- Begriffseingrenzung nach Pike [in: Butcher 2014:2]:

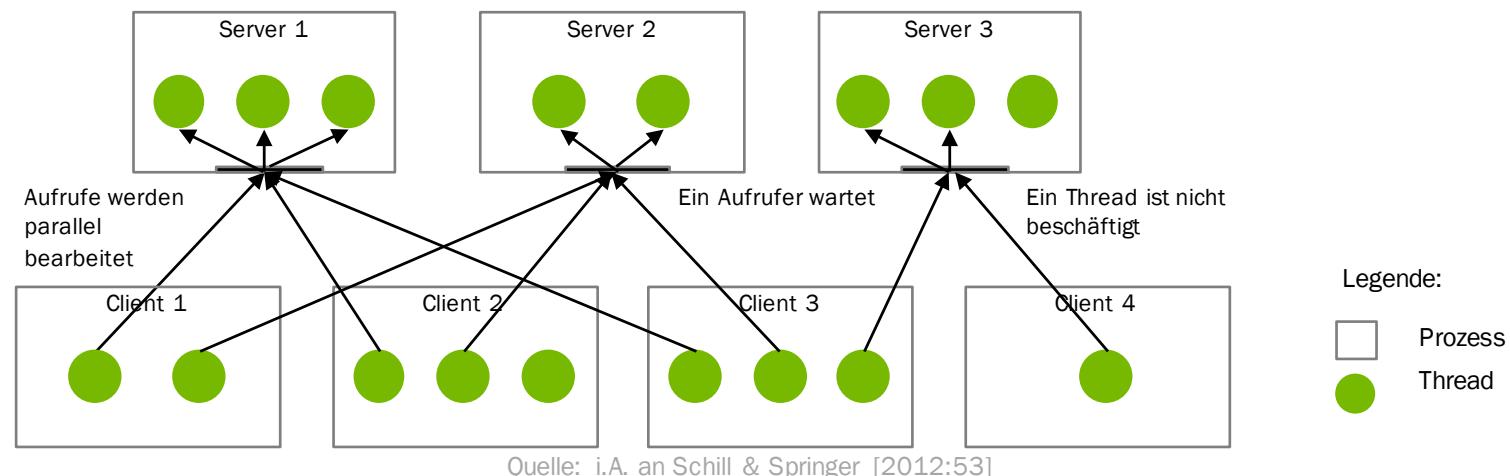
*„Concurrency is about dealing with lots of things at once.
Parallelism is about doing lots of things at once.“*



Nebenläufigkeit (engl.: „concurrency“) [III]

vgl. Bengel et al. [2008:25f.]

- Nebenläufigkeit (concurrent) bedeutet, dass zwei Aktions- oder Aktivitätenstränge oder Prozesse gleichzeitig ablaufen, diese aber nicht notwendigerweise etwas miteinander zu tun haben.
- Die Aktionen sind kausal voneinander unabhängig und können somit unabhängig voneinander ausgeführt werden.
- Bei der Ausführung der nebenläufigen Prozesse stehen sie in Konkurrenz zueinander bei der Belegung der anschließenden Benutzung der Betriebsmittel eines Rechners (vgl. f.)



Prozesse im Kontext der Betriebsmittel / des Betriebssystems

- ✓ Kernfunktionalität eines Betriebssystems: Bereitstellung von Abstraktionen der zu Grunde liegenden physischen Ressourcen

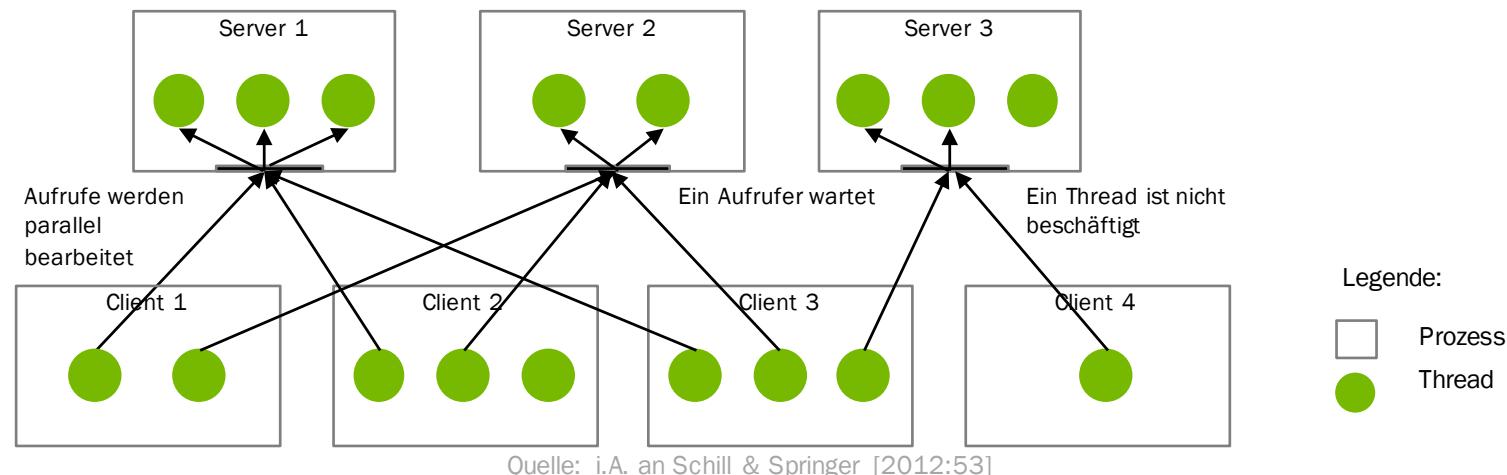


Quelle: i.A. an Coulouris et al. [2002:253]

Ausblick: Kooperierende Prozesse (engl.: „cooperating processes“)

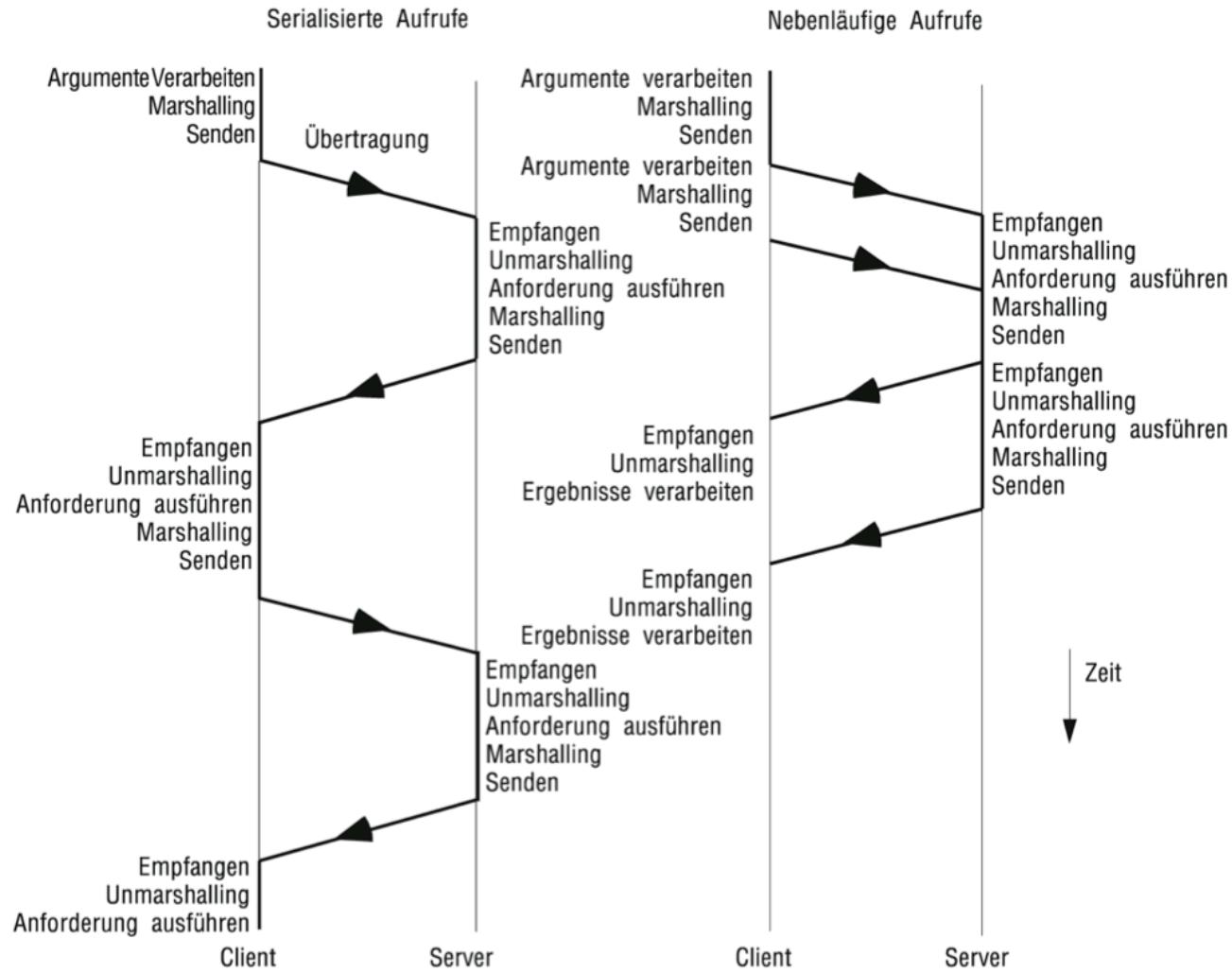
vgl. Bengel et al. [2008:25f.]

- Von kooperierenden Prozessen spricht man:
 - Wenn die nebenläufigen Abläufe (Prozesse) zu einem übergeordnetem Programm gehören oder die verschiedenen Aktivitätenstränge und somit die Prozesse eine gemeinsame Aufgabe lösen und
 - Die Abläufe so logisch miteinander verknüpft sind, dass eine Synchronisation zwischen den Abläufen erfolgen muss.
- ✓ Vertiefung in der kommenden Lehrveranstaltung (vgl. Lecture Notes Synchronisation / Kooperation II)



Nebenläufigkeit (engl. „concurrency“) [IV]

- ✓ Vorteil der Nebenläufigkeit: Zeitersparnis



Quelle: Coulouris et al. [2002:278]

Prozesse und Threads [I]

Ausgangspunkt: 80er Jahre

- Betriebssysteme
 - » Prozessfokussiert (eine einzelne Aktivität mit sequentieller Abarbeitung)
 - » Teure und aufwändige Realisierung von gemeinsamer Ausführung zusammenhängender Aktivitäten
- Steigende Bedeutung von VS
- Lösungsansatz: Erweiterung traditioneller Prozesse im OS um Threads (Nebenläufigkeit)

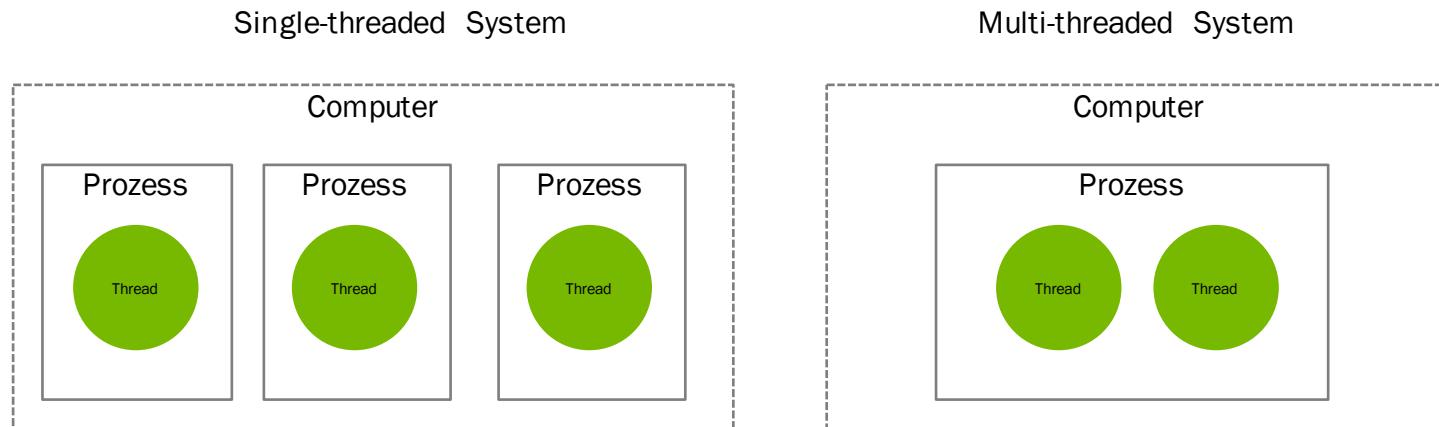
Prozess = Vorgang = Ablauf eines Programms

- Prozess besteht aus
 - » Ausführungsumgebung:
 - Adressraum
 - Kommunikationsschnittstellen (z.B. Netzwerkverbindungen)
 - Lokale Ressourcen (z.B. Speicher)
 - » Thread(s) in der Ausführungsumgebung:
main thread: Hauptkommunikationsstrang (+ n weitere worker-threads)

Prozesse und Threads [II]

Thread

- „Kommunikationsstrang“, leichtgewichtiger Prozess
- Teil eines Prozesses: Aktivität, die innerhalb einer Ausführungsumgebung ausgeführt wird
- Prozesse
 - » Haben mindestens einen Thread („single-threaded“)
 - » Können mehrere Threads haben, welche eine Ausführungsumgebung / einen Adressraum gemeinsam nutzen („Multi-Threading“)
 - ✓ ermöglicht Nebenläufigkeit (Parallelität bei Multiprozessor-Systemen)



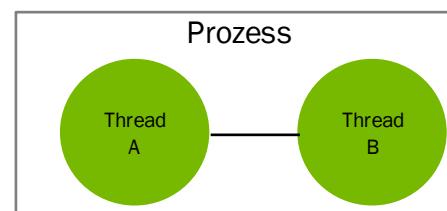
Quelle: i.A. an Weber [1998:159]

Prozessverwaltung: Koordination von Threads [I]

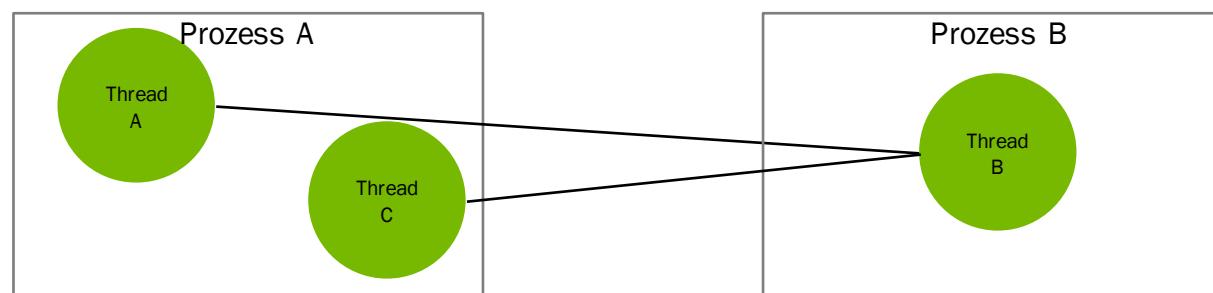
vgl. Weber [1998:159]

Interthread-Kommunikation: innerhalb eines Prozesses

- Threads innerhalb eines Prozesses laufen innerhalb eines Adressraums ab, deshalb haben Sie Zugriff auf gemeinsame Speicherobjekte
- Interthreadkommunikation erfolgt über gemeinsame Variablen und benötigt keinen Nachrichtenaustausch
- Bei nebenläufigem Zugriff auf die gemeinsamen Ressourcen ist eine Synchronisation der beteiligten Threads unter Verwendung klassischer Methoden, z.B. Semaphoren, notwendig.



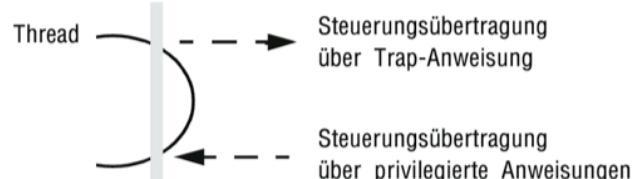
Interthread-Kommunikation: zwischen verschiedenen Prozessen (innerhalb eines Computers, zwischen verschiedenen Computern)



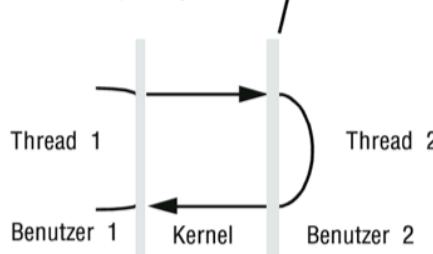
Prozessverwaltung: Koordination von Threads [II]

- ✓ Herausforderung: Domänentransition (= Überschreitung von Adressräumen)

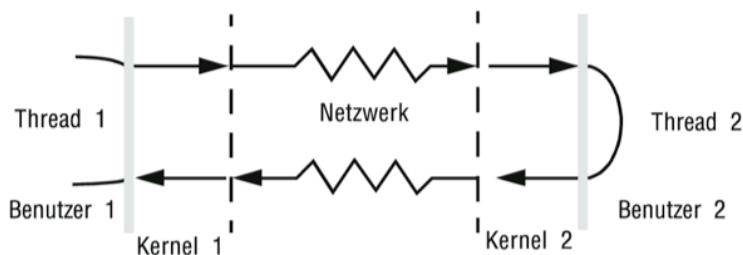
(a) Systemaufruf



(b) RPC/RMI (innerhalb eines Computers)



(c) RPC/RMI (zwischen Computern)



Quelle: Coulouris et al. [2002:278]

Threads: Bewertung

- Vorteile:
 - » Threads sind „billiger“, d.h. der Ressourcenverbrauch für die Initialisierung weiterer Prozesse ist im Vergleich zu Threads höher
 - Zeit (Switching zwischen Threads)
 - Speicher
 - » Kommunikation zwischen Threads durch Nutzung eines Adressraums innerhalb einer Ausführungsumgebung
- Nachteile: Nutzung des gleichen Adressraums!
 - » Sicherheit durch gegenseitigen Zugriff
 - » Datenkonsistenz

Aspekte bei der Implementierung / Programmierung von Threads [IV]

„Das Programmieren von Threads ist die Programmierung nebenläufiger Programme.“

vgl. Weber [1998:162]

Herausforderungen für Entwickler_innen:

- ✓ Programmteile / Prozesse sollen
 - ✓ (Quasi) gleichzeitig ablaufen / bearbeitet werden
 - ✓ Sich gegenseitig nicht blockieren
 - ✓ Nicht endlos in Schleifen laufen und unnötige Ressourcen verbrauchen
- ✓ Verwaltung von Threads
 - ✓ Koordination/Kooperation
 - ✓ Synchronisation

Thread-Programmierung in Java: Ausgewählte Primitiven / Methoden

vgl. Butcher [2014:10ff.]; Rauber & Rünger [2012:337ff.]

Methode	Beschreibung	Anwendung
<code>run()</code>	Enthält die Methoden, die als Thread ausgeführt werden	
<code>start()</code>	Startet die Ausführung eines Threads	Starten / Ausführen von Threads
<code>interrupt()</code>	Unterbricht einen laufenden Thread	
<code>isInterrupted()</code>	Testet, ob ein Thread unterbrochen ist (boolean)	Unterbrechen von Threads
<code>alive()</code>	Liefert <code>true</code> , wenn Thread läuft (boolean)	
<code>join()</code>	Wartet auf das Ende eines Threads	
<code>sleep(long ms)</code>	Unterbricht die Ausführung eines Threads für Zeitraum ms (es gibt auch eine Variation, die um ns ergänzt werden kann, diese wird jedoch von der Mehrzahl der Implementierungen nicht unterstützt)	Scheduling / Koordination von Threads
<code>yield()</code>	Sendet dem Scheduler ein Signal, den nächsten lauffähigen Thread der gleichen Priorität zu bearbeiten	
<code>detach()</code>	Erlaubt (allen) Threads, nebenläufig ausgeführt zu werden (d.h. OS-Scheduler legt Sequenz der Abarbeitung fest)	
<code>wait()</code>	Wartet auf das Eintreffen eines Signals von der Methode <code>notify()</code> oder der Methode <code>notifyAll()</code> ohne Zeitbegrenzung (es kann kein Timeout auftreten) -> entspricht <code>wait(0)</code>	
<code>wait(long ms)</code>	Arbeitet wie <code>wait()</code> mit der Einschränkung, dass maximal ms Millisekunden gewartet wird. Bei einem Wert 0 für ms wird beliebig lange gewartet	
<code>wait(long ms, int ns)</code>	Arbeitet wie <code>wait()</code> mit der Einschränkung, dass maximal ms Millisekunden und ns Nanosekunden gewartet wird	Synchronisierung
<code>notify()</code>	Benachrichtigt genau einen in <code>wait()</code> wartenden Thread. Ausgewählt wird der am längsten wartende Thread (-> es muss auf das zeitliche Verhalten der Threads geachtet werden)	
<code>notifyAll()</code>	Sendet allen wartenden Threads ein Signal	

- ✓ Methoden zum Stoppen (`stop()`), Anhalten (`suspend()`) und Fortfahren (`resume()`) von Threads sind *deprecated*, da sie Deadlocks/Dateninkonsistenzen verursachen

Thread-Programmierung in Java: Erzeugung von n Threads

- ✓ Der Programmcode n zusätzlich definierter Threads muss sich immer innerhalb der Methode *run()* befinden
- ✓ Es gibt zwei Möglichkeiten, in welcher Art von Klasse die Methode *run()* als Grundlage zur Erzeugung von n Threads definiert werden kann
 1. Ableiten aus der Klasse *Thread*
 2. Implementieren der Schnittstelle *Runnable*

Thread-Programmierung in Java: Erzeugung von n Threads

Möglichkeit 1: Ableiten aus der Klasse *Thread* [I]

- Klasse *Thread*
 - » Standardklasse des Package `java.lang` (kann ohne Import-Anweisung verwendet werden)
 - » Nach der Definition einer derartigen Klasse ist es erforderlich
 - Ein Objekt dieser Klasse zu erzeugen (x)
 - Dieses Objekt mit der Methode `start()` zu starten (`x.start();`)
- ✓ Definition einer neuen Klasse *MyThread* als Ableitung aus der Klasse *Thread*

```
1 public class MyThread extends Thread {  
2     public void run() {  
3         System.out.println("Hello World");  
4     }  
5  
6     public static void main(String[] args) {  
7         MyThread t = new MyThread();  
8         t.start();  
9     }  
10 }
```

Thread-Programmierung in Java: Erzeugung von n Threads

Möglichkeit 1: Ableiten aus der Klasse *Thread* [II]

- ✓ Programmcode des zuvor definierten Threads *MyThread*
 - ✓ Wird innerhalb der Methode *run()* eingefügt
 - ✓ Enthält das, was im Rahmen des Threads ausgeführt werden soll

```
1 public class MyThread extends Thread {  
2     public void run() {  
3         System.out.println("Hello World");  
4     }  
5  
6     public static void main(String[] args) {  
7         MyThread t = new MyThread();  
8         t.start();  
9     }  
10 }
```

Thread-Programmierung in Java: Erzeugung von n Threads

Möglichkeit 1: Ableiten aus der Klasse *Thread* [IV]

- ✓ Erzeugung eines Objekts t der Klasse *MyThread* mit der Methode *new()*

```
1 public class MyThread extends Thread {  
2     public void run() {  
3         System.out.println("Hello World");  
4     }  
5  
6     public static void main(String[] args) {  
7         MyThread t = new MyThread();  
8         t.start();  
9     }  
10 }
```

Thread-Programmierung in Java: Erzeugung von n Threads

Möglichkeit 1: Ableiten aus der Klasse *Thread* [V]

- ✓ Starten des Objekts *t* mittels der Methode *start()*

```
1 public class MyThread extends Thread {  
2     public void run() {  
3         System.out.println("Hello World");  
4     }  
5  
6     public static void main(String[] args) {  
7         MyThread t = new MyThread();  
8         t.start();  
9     }  
10 }
```

Thread-Programmierung in Java: Erzeugung von n Threads

Möglichkeit 1: Ableiten aus der Klasse *Thread* [VI]

- ✓ Starten des Objekts *t* mittels der Methode *run()*

```
1 public class MyThread2 extends Thread {  
2     public void run() {  
3         System.out.println("Hello World");  
4     }  
5  
6     public static void main(String[] args) {  
7         MyThread2 t = new MyThread2();  
8         t.run();  
9     }  
10 }
```

Thread-Programmierung in Java: Erzeugung von n Threads

start() vs. run()

- Unterschiede bei der Verwendung der Methoden *start()* und *run()* zum Starten des erzeugten Objekts:
 - » *start()*: Neuer Thread führt Programmcode aus

```
1 public class MyThread extends Thread {  
2     public void run() {  
3         System.out.println("Hello World");  
4     }  
5  
6     public static void main(String[] args) {  
7         MyThread t = new MyThread();  
8         t.start();  
9     }  
10 }
```

- » *run()*: Standardthread führt Programmcode aus:

```
1 public class MyThread2 extends Thread {  
2     public void run() {  
3         System.out.println("Hello World");  
4     }  
5  
6     public static void main(String[] args) {  
7         MyThread2 t = new MyThread2();  
8         t.run();  
9     }  
10 }
```

Thread-Programmierung in Java: Erzeugung von n Threads

Möglichkeit 2: Implementieren der Schnittstelle *Runnable* [I]

- Anwendungsszenarien
 - » In JAVA gibt es keine Mehrfachvererbung
 - » Soll sich bei größeren Programmen die Methode *run()* in einer Klasse befinden, die bereits aus einer anderen Klasse abgeleitet wurde, so kann diese nicht noch zusätzlich aus der Klasse Thread abgeleitet werden
 - » Lösung: Die aus einer anderen Klasse (nicht aus *Thread* abgeleiteten Klasse) abgeleitete Klasse, welche die *run*-Methode enthält, kann die Schnittstelle *Runnable* implementieren
- Beispiel: Erzeugen eines Objekts *runner* der Klasse *HelloWorldExample*, welche das Interface *Runnable* implementiert

```
1 public class HelloWorldExample implements Runnable {  
2     public void run() {  
3         System.out.println("Hello World");  
4     }  
5  
6     public static void main(String[] args) {  
7         HelloWorldExample runner = new HelloWorldExample();  
8         Thread t = new Thread(runner);  
9         t.start();  
10    }  
11 }
```

Thread-Programmierung in Java: Erzeugung von n Threads

Möglichkeit 2: Implementieren der Schnittstelle *Runnable* [II]

- ✓ Erzeugen eines neuen Thread-Objekts *t* der Klasse *Thread*, dem als Argument das Objekt *runner* übergeben wird

```
1 public class HelloWorldExample implements Runnable {
2     public void run() {
3         System.out.println("Hello World");
4     }
5
6     public static void main(String[] args) {
7         HelloWorldExample runner = new HelloWorldExample();
8         Thread t = new Thread(runner);
9         t.start();
10    }
11 }
```

Thread-Programmierung in Java: Ausgewählte Primitiven / Methoden

vgl. Butcher [2014:10ff.]; Rauber & Rünger [2012:337ff.]

Methode	Beschreibung	Anwendung
<code>run()</code>	Enthält die Methoden, die als Thread ausgeführt werden	
<code>start()</code>	Startet die Ausführung eines Threads	Starten / Ausführen von Threads
<code>interrupt()</code>	Unterbricht einen laufenden Thread	
<code>isInterrupted()</code>	Testet, ob ein Thread unterbrochen ist (boolean)	Unterbrechen von Threads
<code>alive()</code>	Liefert <code>true</code> , wenn Thread läuft (boolean)	
<code>join()</code>	Wartet auf das Ende eines Threads	
<code>sleep(long ms)</code>	Unterbricht die Ausführung eines Threads für Zeitraum ms (es gibt auch eine Variation, die um ns ergänzt werden kann, diese wird jedoch von der Mehrzahl der Implementierungen nicht unterstützt)	Scheduling / Koordination von Threads
<code>yield()</code>	Sendet dem Scheduler ein Signal, den nächsten lauffähigen Thread der gleichen Priorität zu bearbeiten	
<code>detach()</code>	Erlaubt (allen) Threads, nebenläufig ausgeführt zu werden (d.h. OS-Scheduler legt Sequenz der Abarbeitung fest)	
<code>wait()</code>	Wartet auf das Eintreffen eines Signals von der Methode <code>notify()</code> oder der Methode <code>notifyAll()</code> ohne Zeitbegrenzung (es kann kein Timeout auftreten) -> entspricht <code>wait(0)</code>	
<code>wait(long ms)</code>	Arbeitet wie <code>wait()</code> mit der Einschränkung, dass maximal ms Millisekunden gewartet wird. Bei einem Wert 0 für ms wird beliebig lange gewartet	
<code>wait(long ms, int ns)</code>	Arbeitet wie <code>wait()</code> mit der Einschränkung, dass maximal ms Millisekunden und ns Nanosekunden gewartet wird	Synchronisierung
<code>notify()</code>	Benachrichtigt genau einen in <code>wait()</code> wartenden Thread. Ausgewählt wird der am längsten wartende Thread (-> es muss auf das zeitliche Verhalten der Threads geachtet werden)	
<code>notifyAll()</code>	Sendet allen wartenden Threads ein Signal	

- ✓ Methoden zum Stoppen (`stop()`), Anhalten (`suspend()`) und Fortfahren (`resume()`) von Threads sind *deprecated*, da sie Deadlocks/Dateninkonsistenzen verursachen

Thread-Programmierung in Java: Ausgewählte Primitiven / Methoden

vgl. Butcher [2014:10ff.]; Rauber & Rünger [2012:337ff.]

Methode	Beschreibung	Anwendung
<code>run()</code>	Enthält die Methoden, die als Thread ausgeführt werden	Starten / Ausführen von Threads
<code>start()</code>	Startet die Ausführung eines Threads	
<code>interrupt()</code>	Unterbricht einen laufenden Thread	
<code>isInterrupted()</code>	Testet, ob ein Thread unterbrochen ist (boolean)	
<code>alive()</code>	Liefert <code>true</code> , wenn Thread läuft (boolean)	Unterbrechen von Threads
<code>join()</code>	Wartet auf das Ende eines Threads	
<code>sleep(long ms)</code>	Unterbricht die Ausführung eines Threads für Zeitraum ms (es gibt auch eine Variation, die um ns ergänzt werden kann, diese wird jedoch von der Mehrzahl der Implementierungen nicht unterstützt)	
<code>yield()</code>	Sendet dem Scheduler ein Signal, den nächsten lauffähigen Thread der gleichen Priorität zu bearbeiten	
<code>detach()</code>	Erlaubt (allen) Threads, nebenläufig ausgeführt zu werden (d.h. OS-Scheduler legt Sequenz der Abarbeitung fest)	Scheduling / Koordination von Threads
<code>wait()</code>	Wartet auf das Eintreffen eines Signals von der Methode <code>notify()</code> oder der Methode <code>notifyAll()</code> ohne Zeitbegrenzung (es kann kein Timeout auftreten) -> entspricht <code>wait(0)</code>	
<code>wait(long ms)</code>	Arbeitet wie <code>wait()</code> mit der Einschränkung, dass maximal ms Millisekunden gewartet wird. Bei einem Wert 0 für ms wird beliebig lange gewartet	
<code>wait(long ms, int ns)</code>	Arbeitet wie <code>wait()</code> mit der Einschränkung, dass maximal ms Millisekunden und ns Nanosekunden gewartet wird	
<code>notify()</code>	Benachrichtigt genau einen in <code>wait()</code> wartenden Thread. Ausgewählt wird der am längsten wartende Thread (-> es muss auf das zeitliche Verhalten der Threads geachtet werden)	Synchronisierung
<code>notifyAll()</code>	Sendet allen wartenden Threads ein Signal	

- ✓ Methoden zum Stoppen (`stop()`), Anhalten (`suspend()`) und Fortfahren (`resume()`) von Threads sind *deprecated*, da sie Deadlocks/Dateninkonsistenzen verursachen

Koordination von Threads

Synchrone Ausführung mittels Primitive *join()* [I]

vgl. Anthony [2016:78f.]

- Thread A (Aufrufer) wartet auf Thread B durch Aufruf von Primitive *join()*, d.h. A blockiert bis B fertig ist.

```
Thread A
{
    ...
    B.join()
    Perform action Y
    ...
}
```

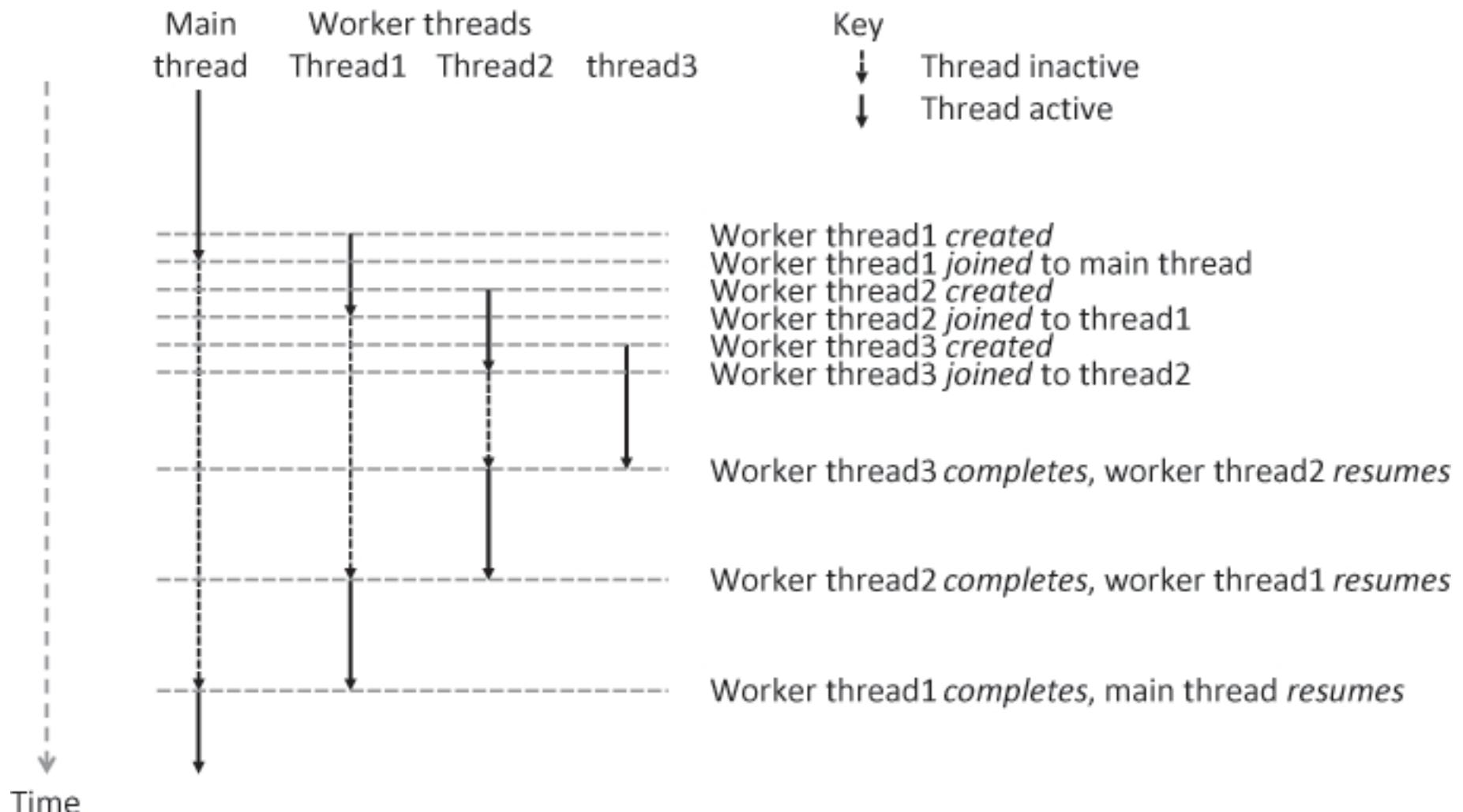
```
Thread B
{
    Perform action X
}
```

Quelle: Anthony [2016:78]

Koordination von Threads

Synchrone Ausführung mittels Primitive *join()* [II]

vgl. Anthony [2016:78f.]



Quelle: Anthony [2016:80]

Koordination von Threads

Asynchrone Ausführung mittels Primitive *detach()* [!]

vgl. Anthony [2016:78f.]

- Wenn synchrone Ausführung nicht nötig ist, kann es (allen) Threads gestattet werden, nebenläufig ausgeführt zu werden (d.h. OS-Scheduler legt Sequenz der Abarbeitung fest)
- Thread A (Aufrufer) wartet auf nicht auf Thread B durch Aufruf der Primitive *detach()*, d.h. er fordert das Gegenteil von *join()* und möchte nebenläufig mit ihm ablaufen.

```
Thread A
{
    ...
    B.detach()
    Perform action Y
    ...
}
```

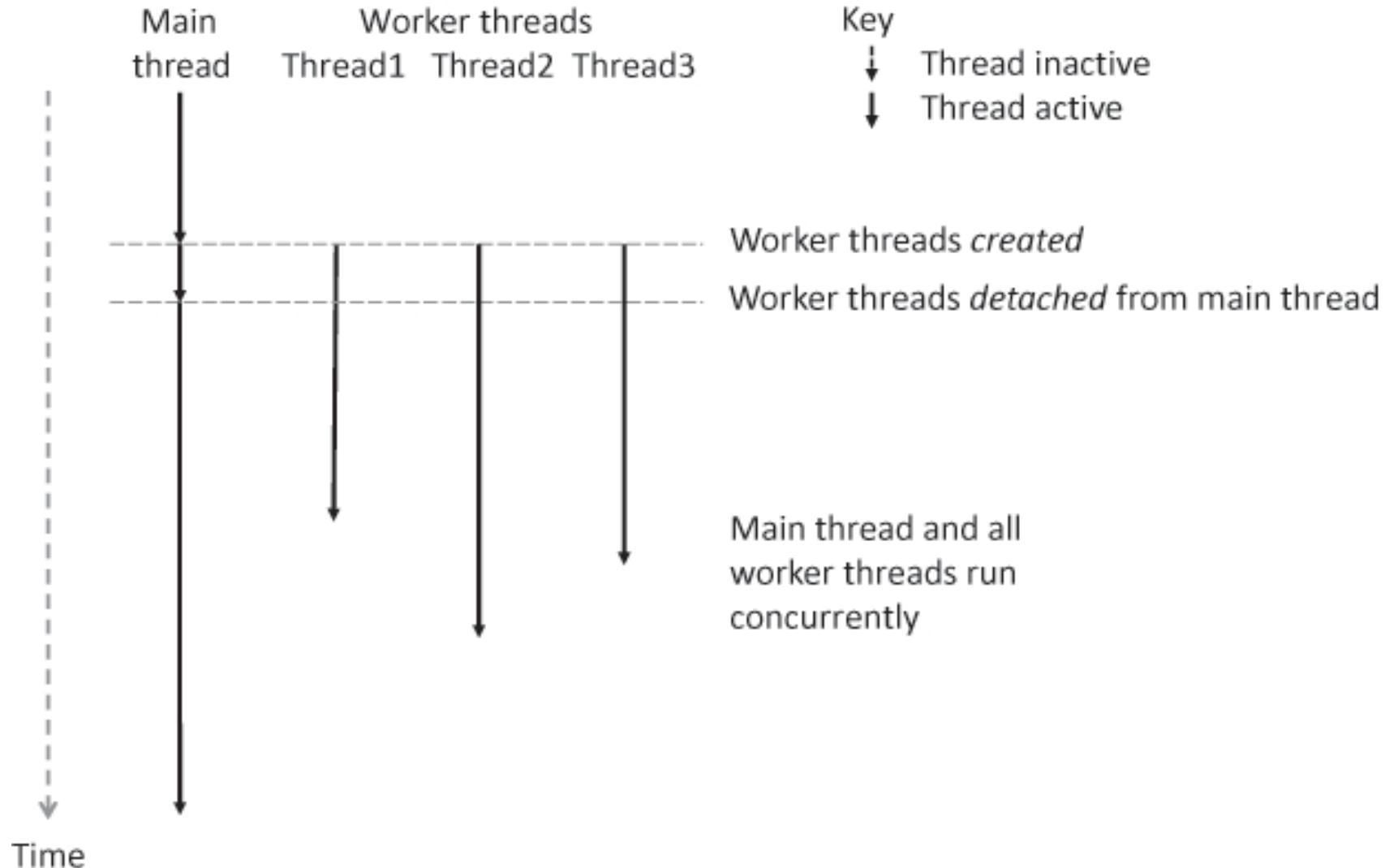
```
Thread B
{
    Perform action X
}
```

Quelle: Anthony [2016:80]

Koordination von Threads

Asynchrone Ausführung mittels Primitive *detach()* [II]

vgl. Anthony [2016:78f.]



Quelle: Anthony [2016:81]

Koordination von Threads

Beispiel 1: Unterbrechung der Ausführung eines Threads mittels Primitive sleep() [I]

sleep()

- ✓ Static-Methode der Klasse *Thread*
- Bewirkt, dass aufrufender Thread die als Argument angegebene Zahl von Millisekunden schläft
- Kann Ausnahmen werfen (*InterruptedException*)
- Zwingend erforderlich: Methode *sleep()* muss in *try-catch*-Block aufgerufen werden
- Problem: Threads werden manchmal gemeinsam geweckt (-> Struktur wird durchbrochen)

```
1 public class PingPong extends Thread {  
2     String Zeichenkette; // auszugebende Zeichenkette  
3     int Verzoegerung; // Pausenlaenge  
4     // Initialisierung von der Zeichenkette und der Pausenlaenge / Verzoegerung  
5  
6     PingPong(String wasSagen, int wieLange) {  
7         Zeichenkette = wasSagen;  
8         Verzoegerung = wieLange;  
9     }  
10  
11    // Aktionen, welche von den erzeugten Threads ausgefuehrt werden  
12    public void run() {  
13        try {  
14            for (;;) {  
15                // Ausgabe der Zeichenkette  
16                System.out.print(Zeichenkette + " ");  
17                // Diesen Thread schlafen legen  
18                sleep(Verzoegerung);  
19            }  
20        } catch (InterruptedException e) {  
21            return;  
22        }  
23    }  
24  
25    // methode main zum Start des Programms  
26    public static void main(String[] args) {  
27        // Erzeugen und Start eines Threads mit 33ms Pause  
28        new PingPong("ping", 33).start();  
29        // Erzeugen und Start eines Threads mit 100ms Pause  
30        new PingPong("PONG", 100).start();  
31    }  
32}
```

Koordination von Threads

Beispiel 1: Unterbrechung der Ausführung eines Threads mittels Primitive sleep() [!!]

`sleep()`

- ✓ Static-Methode der Klasse `Thread`
 - Bewirkt, dass aufrufender Thread die als Argument angegebene Zahl von Millisekunden schlft
 - Kann Ausnahmen werfen (`InterruptedException`)
 - Zwingend erforderlich: Methode `sleep()` muss in `try-catch-Block` aufgerufen werden
 - Problem: Threads werden manchmal gemeinsam geweckt (-> Struktur wird durchbrochen)

```
1 public class PingPong extends Thread {
2     String Zeichenkette; // auszugebende Zeichenkette
3     int Verzoegerung; // Pausenlaenge
4     // Initialisierung von der Zeichenkette und der Pausenlaenge / Verzoegerung
5
6     PingPong(String wasSagen, int wieLange) {
7         Zeichenkette = wasSagen;
8         Verzoegerung = wieLange;
9     }
10
11    // Aktionen, welche von den erzeugten Threads ausgefuehrt werden
12    public void run() {
13        try {
14            for (;;) {
15                // Ausgabe der Zeichenkette
16                System.out.print(Zeichenkette + " ");
17                // Diesen Thread schlafen legen
18                sleep(Verzoegerung);
19            }
20        } catch (InterruptedException e) {
21            return;
22        }
23    }
24
25    // methode main zum Start des Programms
26    public static void main(String[] args) {
27        // Erzeugen und Start eines Threads mit 33ms Pause
28        new PingPong("ping", 33).start();
29        // Erzeugen und Start eines Threads mit 100ms Pause
30        new PingPong("PONG", 100).start();
31    }
32 }
```

Ausführung

Koordination von Threads

Beispiel 2: Unterbrechung der Ausführung eines Threads mittels Primitive sleep() [I]

... zunächst ohne Unterbrechung durch sleep()

```
1 public class Looptest extends Thread {           Ausführung → Thread 2 (1)
2     private String myName;                      Thread 1 (1)
3
4     public Looptest(String name) {               Thread 1 (2)
5         myName = name;                         Thread 3 (1)
6     }                                         Thread 1 (3)
7
8     public void run() {                         Thread 1 (4)
9         for (int i = 1; i <= 100; i++) {        Thread 1 (5)
10            System.out.println(myName + " (" + i + ")"); Thread 2 (2)
11        }                                       Thread 1 (6)
12    }                                         Thread 1 (7)
13
14    public static void main(String[] args) {   Thread 3 (2)
15        Looptest t1 = new Looptest("Thread 1"); Thread 1 (8)
16        Looptest t2 = new Looptest("Thread 2"); Thread 1 (9)
17        Looptest t3 = new Looptest("Thread 3"); Thread 2 (3)
18        t1.start();                           Thread 1 (10)
19        t2.start();                           Thread 3 (3)
20        t3.start();                           Thread 3 (4)
21    }                                         Thread 1 (11)
22 }
```

The code shows a Java program where three threads (t1, t2, t3) are created and started. Each thread prints its name followed by a sequence of numbers from 1 to 100. The output is as follows:

- Thread 2 (1)
- Thread 1 (1)
- Thread 1 (2)
- Thread 3 (1)
- Thread 1 (3)
- Thread 1 (4)
- Thread 1 (5)
- Thread 2 (2)
- Thread 1 (6)
- Thread 1 (7)
- Thread 3 (2)
- Thread 1 (8)
- Thread 1 (9)
- Thread 2 (3)
- Thread 1 (10)
- Thread 3 (3)
- Thread 3 (4)
- Thread 1 (11)
- Thread 2 (4)
- Thread 1 (12)
- Thread 1 (13)
- Thread 3 (5)

Koordination von Threads

Beispiel 2: Unterbrechung der Ausführung eines Threads mittels Primitive sleep() [II]

... und im direkten Vergleich dazu mit Unterbrechung durch sleep() in try-catch-Block

```
1 public class Looptest2 extends Thread {  
2     private String myName;  
3  
4     public Looptest2(String name) {  
5         myName = name;  
6     }  
7  
8     public void run() {  
9         for (int i = 1; i <= 100; i++) {  
10             System.out.println(myName + " (" + i + ")");  
11             try {  
12                 sleep(100);  
13             } catch (InterruptedException e) {  
14             }  
15         }  
16     }  
17  
18     public static void main(String[] args) {  
19         Looptest2 t1 = new Looptest2("Thread 1");  
20         Looptest2 t2 = new Looptest2("Thread 2");  
21         Looptest2 t3 = new Looptest2("Thread 3");  
22         t1.start();  
23         t2.start();  
24         t3.start();  
25     }  
26 }
```



Thread 1 (1)
Thread 3 (1)
Thread 2 (1)
Thread 2 (2)
Thread 1 (2)
Thread 3 (2)
Thread 2 (3)
Thread 1 (3)
Thread 3 (3)
Thread 2 (4)
Thread 1 (4)
Thread 3 (4)
Thread 2 (5)
Thread 1 (5)
Thread 3 (5)
Thread 3 (6)
Thread 1 (6)
Thread 2 (6)
Thread 3 (7)
Thread 1 (7)
Thread 2 (7)
...

Koordination von Threads

Threads beenden

- Thread endet automatisch, wenn die *run*-Methode, die er ausführt beendet ist
 - » Die Anweisungen der *run*-Methode wurden alle ausgeführt
 - » Innerhalb der *run*-Methode wurde eine *return*-Anweisung ausgeführt
 - » Bei der Ausführung wurde eine *Unchecked-Exception* geworfen und nicht gefangen
- ✓ Wir erinnern uns (vgl. Thread Primitives): Methoden zum Stoppen (*stop()*), Anhalten (*suspend()*) und Fortfahren (*resume()*) von Threads sind *deprecated*, da sie Deadlocks/Dateninkonsistenzen verursachen

Thread-Programmierung in Java: Ausgewählte Primitiven / Methoden

vgl. Butcher [2014:10ff.]; Rauber & Rünger [2012:337ff.]

Methode	Beschreibung	Anwendung
<code>run()</code>	Enthält die Methoden, die als Thread ausgeführt werden	
<code>start()</code>	Startet die Ausführung eines Threads	Starten / Ausführen von Threads
<code>interrupt()</code>	Unterbricht einen laufenden Thread	
<code>isInterrupted()</code>	Testet, ob ein Thread unterbrochen ist (boolean)	Unterbrechen von Threads
<code>alive()</code>	Liefert <code>true</code> , wenn Thread läuft (boolean)	
<code>join()</code>	Wartet auf das Ende eines Threads	
<code>sleep(long ms)</code>	Unterbricht die Ausführung eines Threads für Zeitraum ms (es gibt auch eine Variation, die um ns ergänzt werden kann, diese wird jedoch von der Mehrzahl der Implementierungen nicht unterstützt)	Scheduling / Koordination von Threads
<code>yield()</code>	Sendet dem Scheduler ein Signal, den nächsten lauffähigen Thread der gleichen Priorität zu bearbeiten	
<code>detach()</code>	Erlaubt (allen) Threads, nebenläufig ausgeführt zu werden (d.h. OS-Scheduler legt Sequenz der Abarbeitung fest)	
<code>wait()</code>	Wartet auf das Eintreffen eines Signals von der Methode <code>notify()</code> oder der Methode <code>notifyAll()</code> ohne Zeitbegrenzung (es kann kein Timeout auftreten) -> entspricht <code>wait(0)</code>	
<code>wait(long ms)</code>	Arbeitet wie <code>wait()</code> mit der Einschränkung, dass maximal ms Millisekunden gewartet wird. Bei einem Wert 0 für ms wird beliebig lange gewartet	
<code>wait(long ms, int ns)</code>	Arbeitet wie <code>wait()</code> mit der Einschränkung, dass maximal ms Millisekunden und ns Nanosekunden gewartet wird	Synchronisierung
<code>notify()</code>	Benachrichtigt genau einen in <code>wait()</code> wartenden Thread. Ausgewählt wird der am längsten wartende Thread (-> es muss auf das zeitliche Verhalten der Threads geachtet werden)	
<code>notifyAll()</code>	Sendet allen wartenden Threads ein Signal	

- ✓ Methoden zum Stoppen (`stop()`), Anhalten (`suspend()`) und Fortfahren (`resume()`) von Threads sind *deprecated*, da sie Deadlocks/Dateninkonsistenzen verursachen

Thread-Programmierung in Java: Ausgewählte Primitiven / Methoden

vgl. Butcher [2014:10ff.]; Rauber & Rünger [2012:337ff.]

Methode	Beschreibung	Anwendung
<code>run()</code>	Enthält die Methoden, die als Thread ausgeführt werden	
<code>start()</code>	Startet die Ausführung eines Threads	Starten / Ausführen von Threads
<code>interrupt()</code>	Unterbricht einen laufenden Thread	
<code>isInterrupted()</code>	Testet, ob ein Thread unterbrochen ist (boolean)	Unterbrechen von Threads
<code>alive()</code>	Liefert <code>true</code> , wenn Thread läuft (boolean)	
<code>join()</code>	Wartet auf das Ende eines Threads	
<code>sleep(long ms)</code>	Unterbricht die Ausführung eines Threads für Zeitraum ms (es gibt auch eine Variation, die um ns ergänzt werden kann, diese wird jedoch von der Mehrzahl der Implementierungen nicht unterstützt)	
<code>yield()</code>	Sendet dem Scheduler ein Signal, den nächsten lauffähigen Thread der gleichen Priorität zu bearbeiten	Scheduling / Koordination von Threads
<code>detach()</code>	Erlaubt (allen) Threads, nebenläufig ausgeführt zu werden (d.h. OS-Scheduler legt Sequenz der Abarbeitung fest)	
<code>wait()</code>	Wartet auf das Eintreffen eines Signals von der Methode <code>notify()</code> oder der Methode <code>notifyAll()</code> ohne Zeitbegrenzung (es kann kein Timeout auftreten) -> entspricht <code>wait(0)</code>	
<code>wait(long ms)</code>	Arbeitet wie <code>wait()</code> mit der Einschränkung, dass maximal ms Millisekunden gewartet wird. Bei einem Wert 0 für ms wird beliebig lange gewartet	
<code>wait(long ms, int ns)</code>	Arbeitet wie <code>wait()</code> mit der Einschränkung, dass maximal ms Millisekunden und ns Nanosekunden gewartet wird	Synchronisierung
<code>notify()</code>	Benachrichtigt genau einen in <code>wait()</code> wartenden Thread. Ausgewählt wird der am längsten wartende Thread (-> es muss auf das zeitliche Verhalten der Threads geachtet werden)	
<code>notifyAll()</code>	Sendet allen wartenden Threads ein Signal	

- ✓ Methoden zum Stoppen (`stop()`), Anhalten (`suspend()`) und Fortfahren (`resume()`) von Threads sind *deprecated*, da sie Deadlocks/Dateninkonsistenzen verursachen

Synchronisation von Threads [I]

- Ausgangspunkt: wie wir bisher gesehen haben,
 - » Arbeiten Threads weitgehend autark
 - » Laufen Threads quasi-parallel ab
 - Zwischen den Threads wird umgeschaltet
 - Die Umschaltung ist nicht geordnet
- Problem: In der Praxis wird meist ein kooperatives Verhalten gefordert, welches den geordneten Zugriff von Threads auf gemeinsame Daten/gekapselte Objekte ermöglicht
- Basiskonzepte erfordern über einfache Kontroll- und Steuerungsmechanismen hinaus Möglichkeiten zur geordneten gemeinsamen Nutzung von Objekten, sprich zur Synchronisierung/Beeinflussung der Ablaufsequenz von Threads)
- Lösungsansätze: Synchronisierungs-Methoden (vgl. ff.)

Synchronisation von Threads [II]

Methoden

- Verwendung von Prioritäten mittels
 - » Konstanten (1-10):
 - MIN_PRIORITY: 1
 - NORM_PRIORITY: 5
 - MAX_PRIORITY: 10
 - » Methoden:
 - setPriority(): Setzen einer Priorität
 - getPriority(): Abfragen einer Priorität
 - ✓ Scheduler erkennt durch Wert von Priority, welchem Thread er Vorrang erteilt, falls mehrere Threads auf Rechenzeit warten
- Verwendung weiterer Mechanismen:
 - » Einfache Locks (mit dem Ziel eines gegenseitigen Ausschlusses (engl. mutual exclusion))
 - *volatile*: Schlüsselwort für gemeinsam genutzte Variablen
 - *synchronized*: Schlüsselwort für Codeabschnitte, welche nur jeweils von einem Thread ausgeführt werden sollen
 - » Primitive: *wait()*/*notify()*
 - » Semaphore
- Weitere: race conditions / condition variables, message queues, pipes, ...

Threads: Mutual Exclusion & Locks

vgl. Butcher [2014:9ff.]

- ✓ (Viele) Threads können sich gegenseitig auf die Füße treten
- ✓ Dies kann durch gegenseitigen Ausschluss (mutual exclusion) über die Definition kritischer Abschnitte erfolgen, die jeweils nur ein spezifischer Thread betreten darf
- ✓ Diese kritischen Abschnitte werden mit unterschiedlichen Begriffen assoziiert: locks, mutex, monitor

Beispiel: mehrere Threads [I]

vgl. Butcher [2014:9ff.]

- ✓ Eine Klasse Counting und zwei Threads, die jeweils die Methode increment() 10.000x aufrufen.

```
01:  
02: public class Counting {  
03:  
04:     public static void main(String[] args) throws InterruptedException {  
05:  
06:         class Counter {  
07:             private int count = 0;  
08:             public void increment() { ++count; }  
09:             public int getCount() { return count; }  
10:         }  
11:         final Counter counter = new Counter();  
12:  
13:         class CountingThread extends Thread {  
14:             public void run() {  
15:                 for(int x = 0; x < 10000; ++x)  
16:                     counter.increment();  
17:             }  
18:         }  
19:  
20:         CountingThread t1 = new CountingThread();  
21:         CountingThread t2 = new CountingThread();  
22:         t1.start(); t2.start();  
23:         t1.join(); t2.join();  
24:  
25:         System.out.println(counter.getCount());  
26:     }  
27: }
```

Output (mehrmaliger Aufruf):

10626, 11777

Warum nicht 20.000?

Quellcode: Butcher [2014:12]

Beispiel: mehrere Threads [II]

vgl. Butcher [2014:9ff.]

- ✓ Eine Klasse Counting und zwei Threads, die jeweils die Methode increment() 10.000x aufrufen.

```
01:  
02: public class Counting {  
03:  
04:     public static void main(String[] args) throws InterruptedException {  
05:  
06:         class Counter {  
07:             private int count = 0;  
08:             public void increment() { ++count; }  
09:             public int getCount() { return count; }  
10:         }  
11:         final Counter counter = new Counter();  
12:  
13:         class CountingThread extends Thread {  
14:             public void run() {  
15:                 for(int x = 0; x < 10000; ++x)  
16:                     counter.increment();  
17:             }  
18:         }  
19:  
20:         CountingThread t1 = new CountingThread();  
21:         CountingThread t2 = new CountingThread();  
22:  
23:         t1.start(); t2.start();  
24:         t1.join(); t2.join();  
25:  
26:         System.out.println(counter.getCount());  
27:     }  
28: }
```

Quellcode: Butcher [2014:12]

Output (mehrmaliger Aufruf):

10626, 11777

Warum nicht „20.000“?

Beispiel: mehrere Threads [III]

vgl. Butcher [2014:9ff.]

Warum nicht „20.000“?

Weil es sein kann, dass die beiden Threads die Methode increment()
simultan aufrufen, somit auch die gleichen Zählerwerte aufrufen, diesen
Zählerwert beide inkrementieren und ihn als count mit gleichem Wert
speichern. Obwohl count zwei mal inkrementiert wurde, sieht es aus, als
wäre nur einmal inkrementiert worden...

Beispiel: mehrere Threads [III]

vgl. Butcher [2014:9ff.]

Warum nicht „20.000“?

Antwort:

„Race condition“ („[...] occurs when a second thread modifies the state of one (or more objects), making any assumptions, checks, made by the first threads invalid.“ [<http://www.javacreed.com/what-is-race-condition-and-how-to-prevent-it/>])

Anders ausgedrückt: weil es sein kann, dass die beiden Threads die Methode increment() simultan aufrufen, somit auch die gleichen Zählerwerte aufrufen, diesen Zählerwert beide inkrementieren und ihn als count mit gleichem Wert speichern. Obwohl count zwei mal inkrementiert wurde, sieht es aus, als wäre nur einmal inkrementiert worden...

Frage: wie kann sichergestellt werden, dass nicht zwei Threads gleichzeitig die Methode increment() aufrufen?

Beispiel: Mutual Exclusion mit „synchronized“ (JAVA)

vgl. Butcher [2014:9ff.]

- ✓ Eine Möglichkeit ist das Verwenden des Schlüsselwortes „synchronized“ vor der jeweiligen Methode:

```
01:  
02: public class Counting {  
03:  
04:     public static void main(String[] args) throws InterruptedException {  
05:  
06:         class Counter {  
07:             private int count = 0;  
08:             public synchronized void increment() { ++count; }  
09:             public int getCount() { return count; }  
10:         }  
11:         final Counter counter = new Counter();  
12:  
13:         class CountingThread extends Thread {  
14:             public void run() {  
15:                 for(int x = 0; x < 10000; ++x)  
16:                     counter.increment();  
17:             }  
18:         }  
19:  
20:         CountingThread t1 = new CountingThread();  
21:         CountingThread t2 = new CountingThread();  
22:  
23:         t1.start(); t2.start();  
24:         t1.join(); t2.join();  
25:  
26:         System.out.println(counter.getCount());  
27:     }  
28: }
```

Quellcode: Butcher [2014:13]

synchronized

- Programmberäume werden durch das Schlüsselwort *synchronized* als sogenannter **Monitor** gekennzeichnet
- Verwendung für Methoden und Codeabschnitte mit dem Ziel, dass sich zu einem Zeitpunkt höchstens ein Thread in einem mit *synchronized* gekennzeichneten Abschnitt befindet
- Eine mit *synchronized* gekennzeichnete Methode wird immer vollständig ausgeführt, bevor
 - » die Ausführung dieses Programmteils an einen anderen Thread abgegeben wird oder
 - » ein anderer Thread in diesen Programmteil eingreifen kann

Beispiel: Mutual Exclusion mit „synchronized“ (JAVA)

vgl. Butcher [2014:9ff.]

- ✓ Aber: es ist noch nicht alles rosig! Unser Code enthält noch einen Bug...

```
01:  
02: public class Counting {  
03:  
04:     public static void main(String[] args) throws InterruptedException {  
05:  
06:         class Counter {  
07:             private int count = 0;  
08:             public synchronized void increment() { ++count; }  
09:             public int getCount() { return count; }  
10:         }  
11:         final Counter counter = new Counter();  
12:  
13:         class CountingThread extends Thread {  
14:             public void run() {  
15:                 for(int x = 0; x < 10000; ++x)  
16:                     counter.increment();  
17:             }  
18:         }  
19:  
20:         CountingThread t1 = new CountingThread();  
21:         CountingThread t2 = new CountingThread();  
22:  
23:         t1.start(); t2.start();  
24:         t1.join(); t2.join();  
25:  
26:         System.out.println(counter.getCount());  
27:     }  
28: }
```

Quellcode: Butcher [2014:13]

Beispiel: Mutual Exclusion mit „synchronized“ (JAVA)

vgl. Butcher [2014:9ff.]

- ✓ Aber: es ist noch nicht alles rosig! Unser Code enthält noch einen Bug...
- ✓ Welchen?

```
01:  
02: public class Counting {  
03:  
04:     public static void main(String[] args) throws InterruptedException {  
05:  
06:         class Counter {  
07:             private int count = 0;  
08:             public synchronized void increment() { ++count; }  
09:             public int getCount() { return count; }  
10:         }  
11:         final Counter counter = new Counter();  
12:  
13:         class CountingThread extends Thread {  
14:             public void run() {  
15:                 for(int x = 0; x < 10000; ++x)  
16:                     counter.increment();  
17:             }  
18:         }  
19:  
20:         CountingThread t1 = new CountingThread();  
21:         CountingThread t2 = new CountingThread();  
22:  
23:         t1.start(); t2.start();  
24:         t1.join(); t2.join();  
25:  
26:         System.out.println(counter.getCount());  
27:     }  
28: }
```

Quellcode: Butcher [2014:13]

Beispiel: Mutual Exclusion mit „synchronized“ (JAVA)

vgl. Butcher [2014:13ff.]

- ✓ „Memory visibility“: Das Java Speichermodell definiert, wann Änderungen eines Threads für einen anderen sichtbar werden.
- ✓ „Thread-Safe“: Ohne Synchronisation des Lesens und Schreibens durch Threads gibt es keine Garantien.

```
01:  
02: public class Counting {  
03:  
04:     public static void main(String[] args) throws InterruptedException {  
05:  
06:         class Counter {  
07:             private int count = 0;  
08:             public synchronized void increment() { ++count; }  
09:             public synchronized int getCount() { return count; }  
10:         }  
11:         final Counter counter = new Counter();  
12:  
13:         class CountingThread extends Thread {  
14:             public void run() {  
15:                 for(int x = 0; x < 10000; ++x)  
16:                     counter.increment();  
17:             }  
18:         }  
19:  
20:         CountingThread t1 = new CountingThread();  
21:         CountingThread t2 = new CountingThread();  
22:  
23:         t1.start(); t2.start();  
24:         t1.join(); t2.join();  
25:  
26:         System.out.println(counter.getCount());  
27:     }  
28: }
```

Quellcode: Butcher [2014:13]

Synchronisation von Threads

Lock mit `synchronized` [I]

... zunächst ohne Sperre (engl.: „lock“) mittels synchronized

Synchronisation von Threads

Lock mit `synchronized` [II]

... und im direkten Vergleich dazu mit Sperre (engl.: „lock“) mittels `synchronized`

```
1 public class ThreadDemoSynchronized extends Thread {
2     char c;
3
4     public static void main(String args[]) {
5         ThreadDemoSynchronized t1 = new ThreadDemoSynchronized('x');
6         ThreadDemoSynchronized t2 = new ThreadDemoSynchronized('u');
7         t1.start();
8         t2.start();
9     }
10
11    public ThreadDemoSynchronized(char c) {
12        this.c = c;
13    }
14
15    public void run() {
16        for (int n = 0; n < 10; n++) {
17            Ausgabe.schreibeInZeile(c);
18        }
19    }
20 }
21
22 class Ausgabe {
23     synchronized static void schreibeInZeile(char c) {
24         for (int i = 0; i < 10; i++) {
25             System.out.print(c + " ");
26             int n = 0;
27             double w;
28             while (n < 10000)
29                 w = Math.sqrt(n++);
30         }
31         System.out.println();
32     }
33 }
```



```
^ ^ ^ ^ ^ ^ ^ ^ ^ ^  
x x x x x x x x x x  
x x x x x x x x x x  
x x x x x x x x x x  
u u u u u u u u u u  
u u u u u u u u u u  
u u u u u u u u u u  
.. .. .. .. .. .. .. ..
```



```
x x x x x x x x x x  
x x x x x x x x x x  
x x x x x x x x x x  
x x x x x x x x x x  
x x x x x x x x x x  
x x x x x x x x x x  
x x x x x x x x x x  
x x x x x x x x x x  
u u u u u u u u u u  
u u u u u u u u u u  
u u u u u u u u u u  
u u u u u u u u u u  
u u u u u u u u u u  
u u u u u u u u u u  
u u u u u u u u u u  
x x x x x x x x x x  
x x x x x x x x x x  
x x x x x x x x x x
```

Synchronisation von Threads

Lock mit *synchronized* [III]

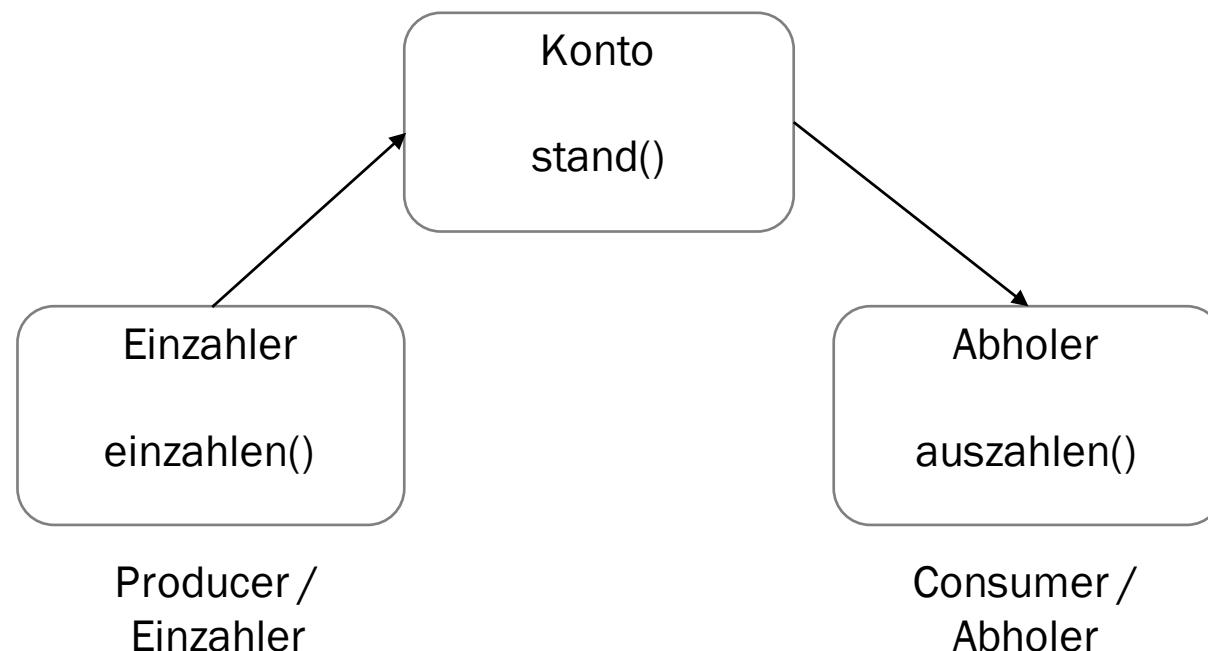
- Ausgangspunkt: Zwar ist nun festgelegt, dass jeweils nur ein Thread auf die Methode `schreibeInZeile` zugreifen kann (entweder `t1` („x“) oder `t2` („u“)), aber eine geordnete Sequenz der Threads ist nicht erreicht (z.B. kann keine Ordnung dergestalt erreicht werden, dass `t2` immer erst dann ausgeführt wird, wenn `t1` (10x) komplett ausgeführt worden ist)
 - ✓ Lösungsansatz: Verwendung der Primitiven `wait()`/`notify()` (vgl. ff.)

Synchronisation von Threads

Verwendung der Primitiven `wait()`/`notify()` am Beispiel einer Kontoverwaltung [I]

Beispiel: Kontoverwaltung einer Bank (Trivialfall)

- Der Abholer soll erst dann einen Betrag abholen, wenn ein Einzahler diesen vorher auch eingezahlt hat
- Der Abholer soll genau den Betrag abholen, der zuvor auch eingezahlt wurde



Synchronisation von Threads

Verwendung der Primitiven `wait()`/`notify()` am Beispiel einer Kontoverwaltung [II]

Szenario 1: unsynchronisiert, Programm: Konto

```
1  /**
2   * Das Konto hat drei Attribute:
3   * stand: wird bei jeder Einzahlung erhöht und bei jeder Auszahlung verringert
4   * einz: der letzte eingezahlte Betrag
5   * ausz: der letzte ausgezahlte Betrag
6   * Im Beispiel ist Höhe von ausz = einz
7  */
8
9  class KontoUnsyn {
10    private int einz, ausz, stand;
11
12    // Die Auszahlung in Höhe der Einzahlung
13    public int auszahlung() {
14        ausz = einz;
15        stand -= ausz;
16        return ausz;
17    }
18
19    // Die Einzahlung in Höhe eines Integer-Wertes
20    public void einzahlung(int wert) {
21        einz = wert;
22        stand += wert;
23    }
24
25    // Der Kontostand
26    public int stand() {
27        return stand;
28    }
29 }
```

Synchronisation von Threads

Verwendung der Primitiven `wait()`/`notify()` am Beispiel einer Kontoverwaltung [III]

Szenario 1: unsynchronisiert, Programm: Einzahler (producer)

- Thread, der während der Laufzeit 10 Einzahlungen (`einzahlung(betrag)`) auf das Konto `konto` (Instanz der zuvor definierten Klasse `KontoUnsyn`) vornimmt
- Der eingezahlte Betrag ist ein Zufalls Wert zwischen 0 und 1000

```
1 class EinzahlerUnsyn extends Thread {  
2     private KontoUnsyn konto;  
3  
4     public EinzahlerUnsyn(KontoUnsyn k) {  
5         this.konto = k;  
6     }  
7  
8     public void run() {  
9         int betrag;  
10        for (int i = 0; i < 10; i++) {  
11            betrag = (int) (Math.random() * 1000);  
12            konto.einzahlung(betrag);  
13            System.out.println("Einzahler " + betrag + " Stand: " + konto.stand());  
14            try {  
15                sleep((int) (Math.random() * 1000));  
16            } catch (InterruptedException e) {  
17            }  
18        }  
19    }  
20}
```

Synchronisation von Threads

Verwendung der Primitiven `wait()`/`notify()` am Beispiel einer Kontoverwaltung [IV]

Szenario 1: unsynchronisiert, Programm: Abholer (consumer)

- Thread, der zehnmal einen Betrag vom Konto abhebt

```
1 class AbholerUnsyn extends Thread {  
2     private KontoUnsyn konto;  
3  
4     public AbholerUnsyn(KontoUnsyn k) {  
5         this.konto = k;  
6     }  
7  
8     public void run() {  
9         int wert = 0;  
10        for (int i = 0; i < 10; i++) {  
11            wert = konto.auszahlung();  
12            System.out.println("Abholer    " + wert + "  Stand: " + konto.stand());  
13            try {  
14                sleep((int) (Math.random() * 1000));  
15            } catch (InterruptedException e) {  
16            }  
17        }  
18    }  
19 }
```

Synchronisation von Threads

Verwendung der Primitiven `wait()`/`notify()` am Beispiel einer Kontoverwaltung [V]

Szenario 1: unsynchronisiert, Programm: Kontoverwaltung

- Erzeugung/Instanziierung von
 - » Einzahler
 - » Abholer
- Start der Threads

```
1 public class KontoVerwUnsyn {
2     public static void main(String[] s){
3         KontoUnsyn konto = new KontoUnsyn();
4         EinzahlerUnsyn eu = new EinzahlerUnsyn(konto);
5         AbholerUnsyn au = new AbholerUnsyn(konto);
6         eu.start();
7         au.start();
8     }
9 }
```

Synchronisation von Threads

Verwendung der Primitiven `wait()`/`notify()` am Beispiel einer Kontoverwaltung [VI]

Szenario 1: unsynchronisiert, Ergebnis: Fehler durch fehlende Synchronisierung

- ✓ Bei fehlender Synchronisation:
 - ✓ Kontostand kann negativ werden
 - ✓ Einzahlung / Abholung erfolgen ggf. mehrmals hintereinander und nicht geordnet (erst Einzahlung, dann Abholung)
- ✓ Ursache: Einzahler und Abholer greifen beliebig auf das Konto zu
- ✓ Lösungsansatz: Szenario 2: synchronisierte Kontoverwaltung (vgl. ff.)

Ausführung →

Abholer 0 Stand: 0
Einzahler 431 Stand: 431
Abholer 431 Stand: 0
Abholer 431 Stand: -431
Einzahler 643 Stand: 212
Abholer 643 Stand: -431
Einzahler 226 Stand: -205
Abholer 226 Stand: -431
Einzahler 658 Stand: 227
Abholer 658 Stand: -431
Abholer 658 Stand: -1089
Einzahler 814 Stand: -275
Abholer 814 Stand: -1089
Einzahler 760 Stand: -329
Einzahler 608 Stand: 279
Abholer 608 Stand: -329
Einzahler 762 Stand: 433
Abholer 762 Stand: -329
Einzahler 573 Stand: 244
Einzahler 736 Stand: 980

Synchronisation von Threads

Verwendung der Primitiven `wait()`/`notify()` am Beispiel einer Kontoverwaltung [VII]

Szenario 2: synchronisiert, Programm (unverändert): Einzahler (producer)

- Thread, der während der Laufzeit 10 Einzahlungen (`einzahlung(betrag)`) auf das Konto `konto` (Instanz der zuvor definierten Klasse `KontoUnsyn`) vornimmt
- Der eingezahlte Betrag ist ein Zufalls Wert zwischen 0 und 1000
 - » Der eingezahlte Betrag ist ein Zufalls Wert zwischen 0 und 1000

```
1 class Einzahler extends Thread {  
2     private Konto konto;  
3  
4     public Einzahler(Konto k) {  
5         this.konto = k;  
6     }  
7  
8     public void run() {  
9         int betrag;  
10        for (int i = 0; i < 10; i++) {  
11            betrag = (int) (Math.random() * 1000);  
12            konto.einzahlung(betrag);  
13            System.out.println("Einzahler " + betrag + " Stand: " + konto.stand());  
14            try {  
15                sleep((int) (Math.random() * 1000));  
16            } catch (InterruptedException e) {  
17            }  
18        }  
19    }  
20}
```

Synchronisation von Threads

Verwendung der Primitiven `wait()`/`notify()` am Beispiel einer Kontoverwaltung [VIII]

Szenario 2: synchronisiert, Programm (unverändert): Abholer (consumer)

- Thread, der zehnmal einen Betrag vom Konto abhebt

```
1 class Abholer extends Thread {  
2     private Konto konto;  
3  
4     public Abholer(Konto k) {  
5         this.konto = k;  
6     }  
7  
8     public void run() {  
9         int wert = 0;  
10        for (int i = 0; i < 10; i++) {  
11            wert = konto.auszahlung();  
12            System.out.println("Abholer    " + wert + "  Stand: " + konto.stand());  
13            try {  
14                sleep((int) (Math.random() * 1000));  
15            } catch (InterruptedException e) {  
16            }  
17        }  
18    }  
19 }
```

Synchronisation von Threads

Verwendung der Primitiven `wait()`/`notify()` am Beispiel einer Kontoverwaltung [IX]

Szenario 2: synchronisiert, Programm (unverändert): Kontoverwaltung

- Erzeugung/Instanziierung von
 - » Einzahler
 - » Abholer
- Start der Threads

```
1 public class KontoVerw {  
2     public static void main(String[] s) {  
3         Konto konto = new Konto();  
4         Einzahler e = new Einzahler(konto);  
5         Abholer a = new Abholer(konto);  
6         e.start();  
7         a.start();  
8     }  
9 }
```

Synchronisation von Threads

Verwendung der Primitiven `wait()`/`notify()` am Beispiel einer Kontoverwaltung [X]

Szenario 2: synchronisiert, Programm: Konto

```
1 class Konto {
2     private int einz, ausz, stand;
3     private boolean zugriff = false;
4
5     public synchronized int auszahlung() {
6         while (!zugriff) {
7             try {
8                 wait();
9             } catch (InterruptedException e) {
10                }
11            }
12            ausz = einz;
13            stand -= ausz;
14            zugriff = false;
15            notifyAll();
16            return ausz;
17        }
18
19        public synchronized void einzahlung(int wert) {
20            while (zugriff) {
21                try {
22                    wait();
23                } catch (InterruptedException e) {
24                }
25            }
26            einz = wert;
27            stand += wert;
28            zugriff = true;
29            notifyAll();
30        }
31
32        public int stand() {
33            return stand;
34        }
35    }
```

- ✓ Zugriff auf Methoden `auszahlung()` und `einzahlung()` wird über die Variable „`zugriff`“ gesteuert
 - » Typ Boolean
 - » Erst eine Einzahlung ändert Wert der Variable `zugriff` auf „`true`“
- // Folge:
 - ✓ Wird die methode `auszahlung()` aufgerufen, wenn der Wert der Variablen noch auf `false` steht, verbleibt der Thread im Wartezustand bzw. die Auszahlung hat keine Änderung des Kontostands zur Folge
 - ✓ Wird die Methode `einzahlung()` aufgerufen, während die Variable `zugriff` den Wert `true` hat, wird der Einzahlungs-Thread in den Wartezustand gesetzt bzw. hat keine Änderung des Kontostands zur Folge
 - ✓ Erst eine Einzahlung bei `zugriff false`

Synchronisation von Threads

Verwendung der Primitiven `wait()`/`notify()` am Beispiel einer Kontoverwaltung [XI]

Szenario 2: synchronisiert, Programm: Konto

```
1 class Konto {  
2     private int einz, ausz, stand;  
3     private boolean zugriff = false;  
4  
5     public synchronized int auszahlung() {  
6         while (!zugriff) {  
7             try {  
8                 wait();  
9             } catch (InterruptedException e) {  
10                }  
11            }  
12            ausz = einz;  
13            stand -= ausz;  
14            zugriff = false;  
15            notifyAll();  
16            return ausz;  
17        }  
18  
19        public synchronized void einzahlung(int wert) {  
20            while (zugriff) {  
21                try {  
22                    wait();  
23                } catch (InterruptedException e) {  
24                    }  
25                }  
26                einz = wert;  
27                stand += wert;  
28                zugriff = true;  
29                notifyAll();  
30            }  
31  
32            public int stand() {  
33                return stand;  
34            }  
35        }
```

Ausführung

Einzahler	83	Stand:	83
Abholer	83	Stand:	0
Einzahler	572	Stand:	572
Abholer	572	Stand:	0
Einzahler	532	Stand:	532
Abholer	532	Stand:	0
Einzahler	731	Stand:	731
Abholer	731	Stand:	0
Einzahler	467	Stand:	467
Abholer	467	Stand:	0
Einzahler	209	Stand:	209
Abholer	209	Stand:	0

Probieren Sie es selbst aus:

- ✓ Verändern Sie die Werte für `sleep` beim Einzahler und Abholer (z.B. auf 3000ms)

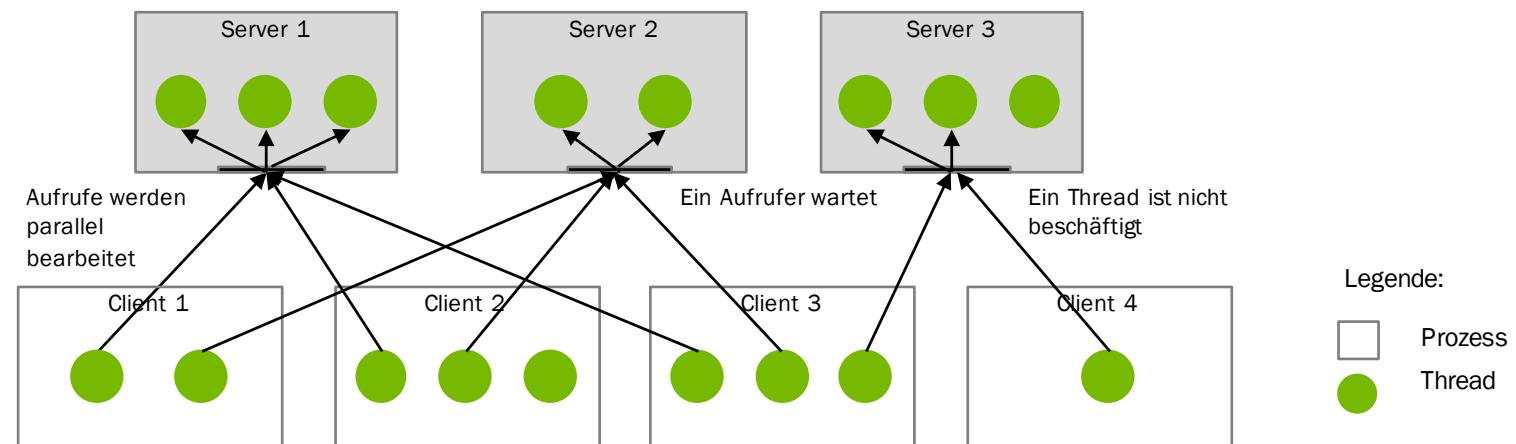
Synchronisation von Threads

Semaphoren

- Ausgangspunkt:
 - » *synchronized()* ermöglicht den gegenseitigen Ausschluss von gleichzeitig ausgeführten Programmteilen durch Threads
 - » *synchronized()* ist in Java implementiert
 - Herausforderung: Andere Programmiersprachen
 - Erzeuger-Verbraucher-Problem
-
- ✓ Lösungsansatz: Semaphoren (vgl. Appendix, Module Programmierung)

Threadbasierte Systeme: serverseitige Organisationsformen [I]

- ✓ Die Aufgabenverteilung und -zuordnung zu verschiedenen Threads eines **Servers** kann unterschiedlich organisiert werden.
- ✓ Unterscheidung nach Weber [1998:159ff.]:
 - » Dispatcher/Worker-Modell
 - » Team-Modell
 - » Pipeline-Modell
- ✓ Unterscheidung von Multi-threaded Server-Architekturen nach Coulouris et al. [2002:262ff.]
 - » Worker-Pool-Architektur
 - » Alternativen:
 - Thread-pro-Anforderung
 - Thread-pro-Verbindung
 - Thread-pro-Objekt

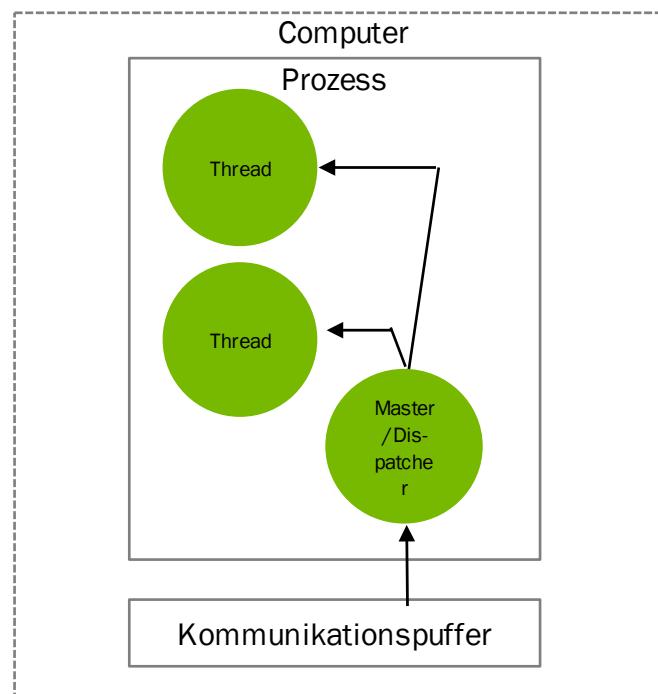


Threadbasierte Systeme: serverseitige Organisationsformen [II]

vgl. Weber [1998:159ff.]

Dispatcher/Worker-Modell:

- ein Thread empfängt Aufträge und teilt sie anderen Bearbeitungsthreads zu, die für diesen Zweck bereit stehen oder neu erzeugt werden (vgl. Worker-Pool, ff.)
- Master-/Slave-Modell: Empfangs- und Zuteilungsthread ist der Master-Thread



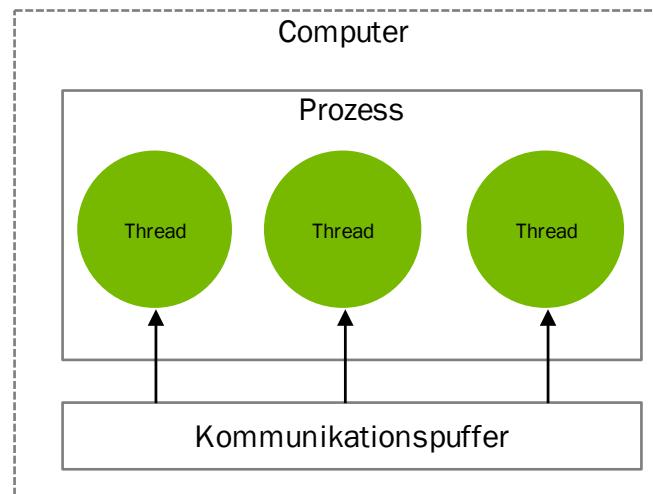
Quelle: i.a. an Weber [1998:160]

Threadbasierte Systeme: serverseitige Organisationsformen [III]

vgl. Weber [1998:159ff.]

Team-Modell (vgl. Worker-Pool):

- Alle Threads sind gleichberechtigt
- Hat ein Thread seine bisherige Aufgabe abgearbeitet, so bedient er sich selbst mit einem neuen Auftrag aus dem Eingangspuffer. Dieser Puffer wird somit zu einer gemeinsam benutzten Ressource und muss entsprechend gegen konkurrierenden Zugriff geschützt werden



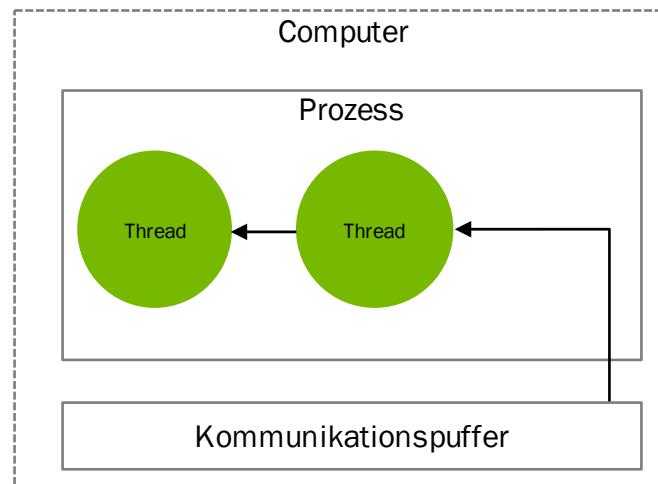
Quelle: i.a. an Weber [1998:160]

Threadbasierte Systeme: serverseitige Organisationsformen [IV]

vgl. Weber [1998:159ff.]

Pipeline-Modell:

- Threads sind logisch in einer Kette hintereinander angeordnet.
- Jeder Thread bearbeitet nur einen Teil des Auftrags und gibt ihn dann zur Weiterbearbeitung an den nachfolgenden Thread.
- ✓ Dazu muss natürlich der Auftrag eine entsprechende Struktur haben, um diese fließbandartige Bearbeitung zuzulassen.



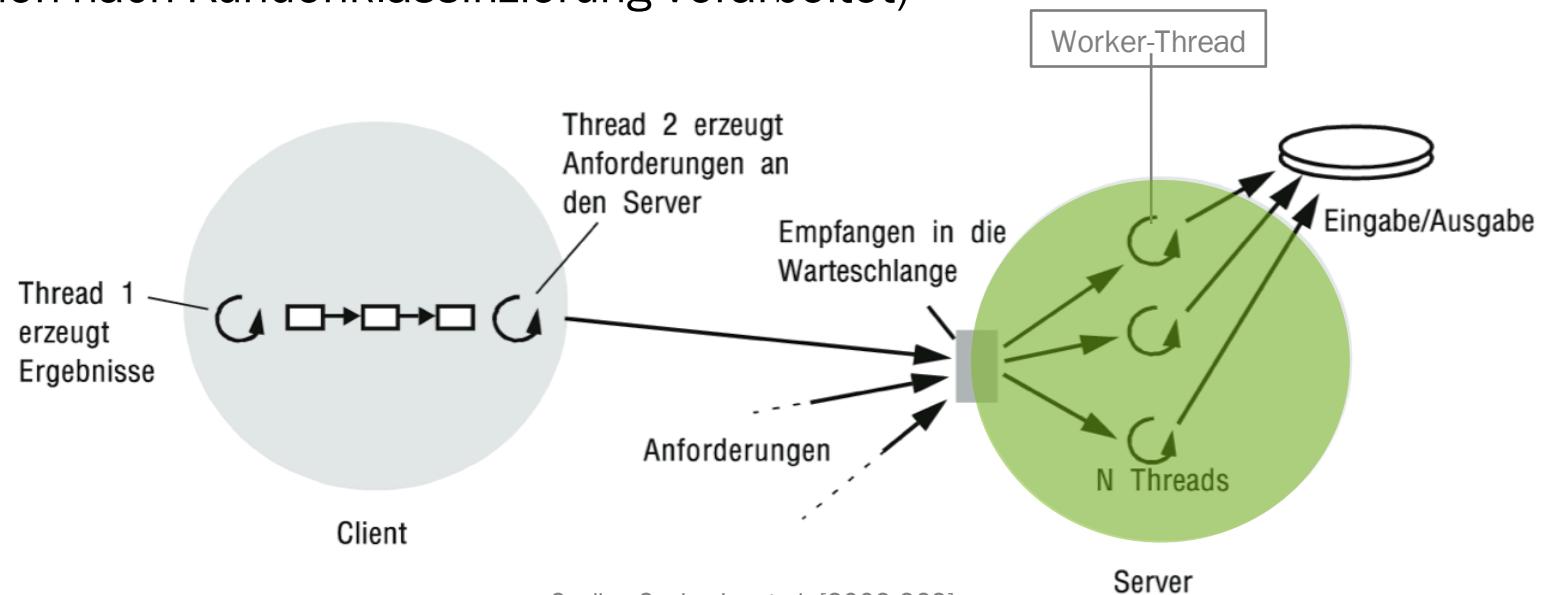
Quelle: i.a. an Weber [1998:160]

Threadbasierte Systeme: serverseitige Organisationsformen [V]

vgl. Coulouris et al. [2002:262ff.]

Worker-Pool-Architektur (vgl. Team-Modell)

- Server:
 - » Erzeugt festen Pool an „Arbeits“-Threads, welche die in der Warteschlange befindlichen Anforderungen verarbeiten
 - » Empfang/Warteschlange wird realisiert durch I/O-Thread
- Variationen durch multiple I/O-Threads mit festen Pools von Worker-Threads
- Beispiel: Ticket-System mit unterschiedlichen Prioritäten (Kundenanfragen werden nach Kundenklassifizierung verarbeitet)

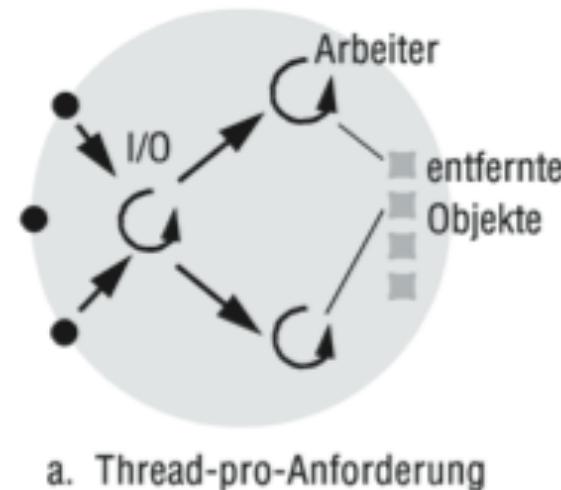


Threadbasierte Systeme: serverseitige Organisationsformen [VI]

vgl. Coulouris et al. [2002:262ff.]

Alternativen zur Worker-Pool-Architektur: Thread-pro-Anforderung

- I/O-Thread erzeugt für jede einzelne Anforderung einen einzelnen Thread
- Thread zerstört sich nach Verarbeitung der Anfrage
- Vorteil: Wegfall der Warteschlange (Maximierung des Durchsatzes)
- Nachteil: Overhead durch Erzeugung/Zerstörung vieler Threads



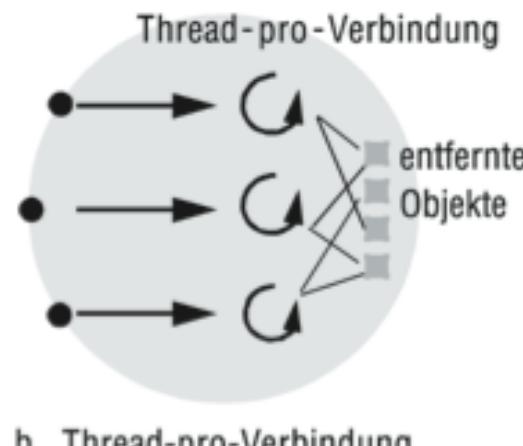
Quelle: Coulouris et al. [2002:265]

Threadbasierte Systeme: serverseitige Organisationsformen [VII]

vgl. Coulouris et al. [2002:262ff.]

Alternativen zur Worker-Pool-Architektur: Thread-pro-Verbindung

- Server erzeugt für jede einzelne Verbindung vom Client einen einzelnen Thread
- Beliebig viele Anforderungen können von diesem einen Thread bei bestehender Verbindung zu einem Client verarbeitet werden
- Thread zerstört sich nach Abbruch der Verbindung
- Vorteil: Reduktion des Overheads
- Nachteil: Verzögerungen durch unterschiedlich ausgelastete Arbeits-Threads



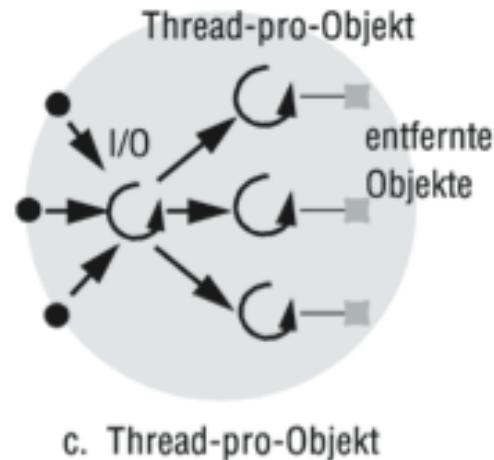
Quelle: Coulouris et al. [2002:265]

Threadbasierte Systeme: serverseitige Organisationsformen [VII]

vgl. Coulouris et al. [2002:262ff.]

Alternativen zur Worker-Pool-Architektur: Thread-pro-Objekt

- Jedem angeforderten Objekt ist ein Thread zugeordnet
- I/O-Thread verwaltet und ordnet Anforderungen in Abhängigkeit des angeforderten Objekts zu
- Vorteil: Reduktion des Overheads
- Nachteil: Verzögerungen durch unterschiedlich ausgelastete Arbeits-Threads



Quelle: Coulouris et al. [2002:265]

Threads: Herausforderungen

- Race Conditions
 - » Tipp: Locks (z.B. intrinsisch in JAVA via synchronized)
- Memory Visibility
 - » Locks auf schreibende und lesende Methoden
- Multiple Locks
 - » Risiken:
 - Deadlocks („Deadlock is a danger whenever a thread tries to hold more than one lock.“ [Butcher 2014: 17])
 - Tipps: Beachten einer globalen, fixierten Ordnung der Locks, Locks möglichst kurz halten
 - Alien methods [Butcher 2014:19]
 - Tipp: Während ein Thread einen Lock hält, sollten keine Methoden aufgerufen werden, die unbekannt sind und unbekannte Ergebnisse erzeugen (z.B. weitere Locks)

Threads: Bewertung

vgl. Butcher [2014:45ff.]

- Vorteile:
 - » Breites Anwendungsfeld zur Lösung verschiedener Probleme
 - » Integration in viele Programmiersprachen
- Nachteile:
 - » Pseudoparallelität, keine echte Parallelität
 - » Shared Memory (kein Distributed Memory): Threads können nur diejenigen Probleme lösen, die auf Ihren Knoten „passen“
 - » Schwierig zu testen (schwierig: Maintenance / Refactoring)

Ausblick:

- Advanced
(<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/package-summary.html>)
 - » ReentrantLock
 - Hand-over-Hand Locking
 - Condition variables
 - Atomic variables

Scala (auf JAVA-Basis):

- https://twitter.github.io/scala_school/concurrency.html

Python

- <https://www.python-kurs.eu/threads.php>

C++

- <https://www.geeksforgeeks.org/multithreading-in-cpp/>

Danke. Lernziele erreicht?

Nach dieser Lehrveranstaltung kennen Studierende idealerweise:

- Abgrenzende Eigenschaften der Begriffe Nebenläufigkeit, Parallelität, Konkurrenz und Kooperation im Kontext von Prozessen
 - Aspekte der Nebenläufigkeit im Kontext von Betriebsmitteln, welche vom Betriebssystem bereitgestellt werden
 - Den Unterschied zwischen Prozessen und Threads sowie den Unterschied bei der Kommunikation zwischen Threads innerhalb eines Prozesses und zwischen Prozessen
 - Praktische Beispiele zur Erzeugung und zum Umgang mit Threads und nebenläufiger Programmierung in JAVA mit Schwerpunkten auf
 - » Grundlegenden Thread & Lock-Primitiven und Mechanismen zur Thread-Manipulation
 - » (A)synchroner Kooperation/Koordination
 - » Synchronisierung
 - Perspektiven, Kategorien und Funktionsweisen thread-basierter Systeme mit Fokus auf serverseitige Organisationsformen
-
- ✓ Studierende kennen grundlegende Aspekte hinsichtlich der Koordination und Synchronisation von nebenläufigen Prozessen und leichtgewichtigen Threads bei der Interprozesskommunikation. Zudem lernen Studierende praktische Beispiele für die Implementierung einfacher Synchronisations- und Koordinationsmechanismen und deren grundsätzliche Einsatzmöglichkeiten im Zusammenspiel verschiedener Knoten eines verteilten Systems.

Quellen

Anthony, R. (2016) *Systems Programming - Designing and Developing Distributed Applications*; Amsterdam et al.: Morgan-Kaufman / Elsevier.

Bengel, G. (2004) *Grundkurs Verteilte Systeme*; Wiesbaden: Vieweg.

Bengel, G.; Baun, C.; Kunze, M.; Stucky, K.-U. (2008) *Masterkurs Parallele und Verteilte Systeme*; Wiesbaden: Vieweg & Teubner.

Butcher, P. (2014) *Seven Concurrency Models in Seven Weeks – When Threads Unravel*. In: Tate, B. (Ed.) *The Pragmatic Programmers Series*. Dallas, Raleigh: The Pragmatic Programmers / The Pragmatic Bookshelf.

Coulouris, G.; Dollimore, J.; Kindberg, T. (2002) *Verteilte Systeme - Konzepte und Design*; 3., überarbeitete Auflage; München: Pearson Studium.

Coulouris, G.; Dollimore, J.; Kindberg, T., Blair, G. (2012) *Distributed Systems - Concepts and Design*; 5., überarbeitete Auflage; Boston et al.: Addison-Wesley, .

Dunkel, J.; Eberhart, A.; Fischer, S.; Kleiner, C.; Koschel, A. (2008) *Systemarchitekturen für Verteilte Anwendungen*; München: Hanser.

Oechsle, R. (2011) *Parallele und verteilte Anwendungen in JAVA*; 3., erweiterte Auflage; München: Carl Hanser.

Oracle (n.n.) *Java Thread Primitive Deprecation*; online:
<https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html> [last accessed: 15 II 18]

Rauber, T.; Rünger, G. (2012) *Parallele Programmierung*; Berlin, Heidelberg: Springer/eXamen.press.

Schill, A.; Springer, T. (2012) *Verteilte Systeme*; 2. Auflage; Berlin, Heidelberg: Springer Vieweg.

Tanenbaum, A.; van Steen, M. (2002) *Distributed Systems – Principles and Paradigms*, 2nd edition. Upper Saddle River: Pearson/ Prentice Hall,
online: <http://barbie.uta.edu/~jli/Resources/MapReduce&Hadoop/Distributed%20Systems%20Principles%20and%20Paradigms.pdf>
[letzter Zugriff: 9 II 18]]

Tanenbaum, A.; van Steen, M. (2008) *Verteilte Systeme – Prinzipien und Paradigmen*; 2., überarbeitete Auflage; München: Pearson Studium.

Weber, M. (1998) *Verteilte Systeme*; Heidelberg, Berlin: Spektrum.

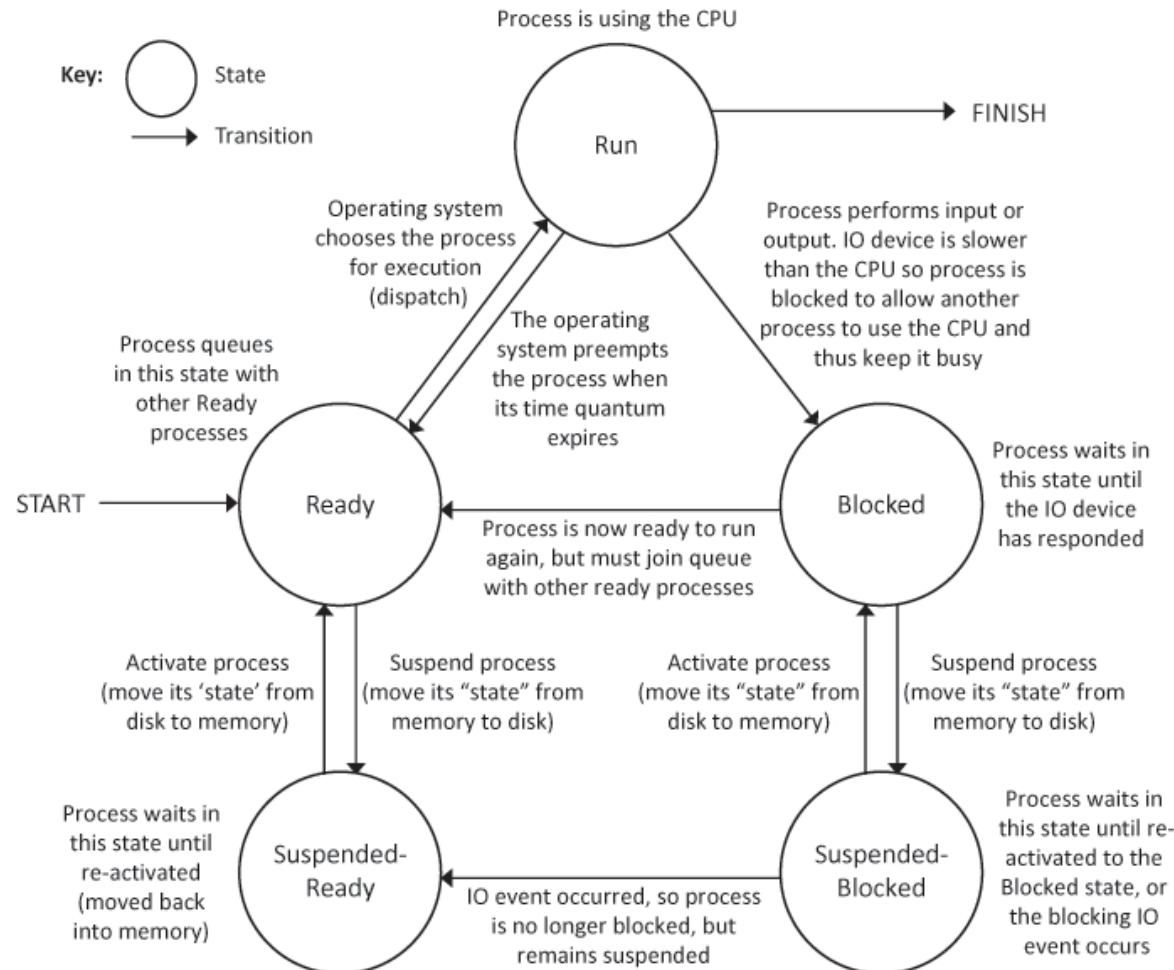
- Appendix -

Ergänzende Materialien zur
Vertiefung im Eigenstudium

- » Prozessverhalten (Zustandsmodell)
- » Eigenschaften von Threads als leichtgewichtige Prozesse auf user- und kernel-Level sowie damit zusammenhängende Vor- und Nachteile von Threads
- » Synchronisation von Threads mittels Semaphore (JAVA)

Prozesse und Threads

Prozessverhalten (Zustandsmodell):



Quelle: Anthony [2016:49]

Aspekte bei der Implementierung / Programmierung von Threads [I]

User-Level-Threads: Verankerung (nur) im Arbeitsbereich eines Users

User Space					Thread
Laufzeitsystem					
Kernel Space	Kern				

Quelle: i.A. an Weber [1998:161]

Vorteile:

- Das zu Grunde liegende OS braucht nicht verändert zu werden. Alle Systemaufrufe werden an das OS über das Thread-Laufzeitsystem abgewickelt.
- Die threadbasierte Anwendung ist weitgehend unabhängig vom OS und damit leichter portabel.
- Umschaltung zwischen Threads ist schneller, da das Laufzeitsystem lediglich Programmzähler, Registersatz und Kellerzeiger austauscht (es werden keine Kern-Traps ausgelöst, um dann dort zwischen den Threads zu wechseln. Ein Wechsel in den Kern würde natürlich einen Prozesswechsel bewirken, so dass die Idee der Threads verloren geht).
- User-Level Threads skalieren gut, da wenig Informationen pro Thread geführt werden müssen.

Nachteile:

- Blockierende Systemaufrufe: Ein Thread blockiert und mit ihm das Laufzeitsystem, welches vom OS, zusammen mit all seinen Threads, als ein einzelner Prozess gesehen wird.
 - » Die Kontrolle kann somit nicht an einen anderen Thread übergeben werden.
 - » Das Laufzeitsystem hat keine Möglichkeit, laufende Threads zu unterbrechen, wenn diese nicht spezielle Routinen des Laufzeitsystems aufrufen, um ein Scheduling zu erlauben.

Aspekte bei der Implementierung / Programmierung von Threads [II]

Kernel-Level-Threads: Verankerung im Betriebssystem

User Space					Thread
Kernel Space	Kern				

Quelle: i.A. an Weber [1998:161]

Vorteile:

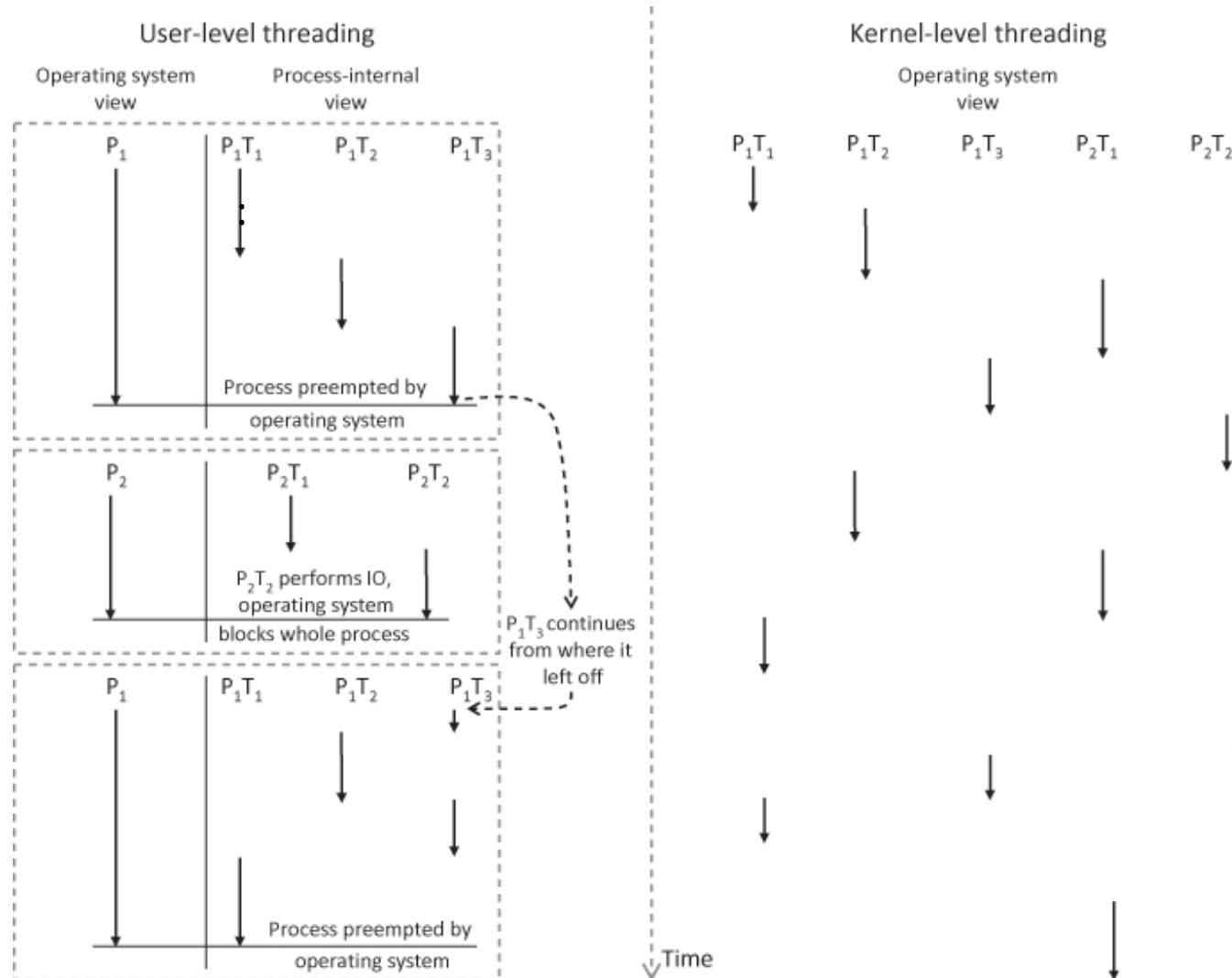
- Jeder Thread wird individuell beim Scheduling behandelt, d.h. ein Prozess blockiert nicht komplett, wenn einer seiner in der Ausführungsumgebung vorhandenen Threads I/O durchführt.
- ✓ Vermeidung der Probleme blockierender Aufrufe von User-Level-Threads

Nachteile:

- Aufgaben des Laufzeitsystems sind hier in den Kern verlagert, deshalb Erhöhung des zeitlichen Aufwands: jeder Aufruf einer Threadverwaltungsoperation wird zu einem Systemaufruf.

Aspekte bei der Implementierung / Programmierung von Threads [III]

User-Level Threads vs. Kernel-Level-Threads



Quelle: Anthony [2016:77]

Synchronisation von Threads

Semaphoren [I]

vgl. Rauber & Rünger [2012:325ff.]

- (Zählender) Semaphor ist ein Datentyp, der einen Zähler realisiert
- Zähler kann nichtnegative Integerwerte annehmen
- Zähler kann mit zwei Operationen manipuliert werden:
 - » Signaloperation: `v()` // freigeben
 - Inkrementiert Zähler,
 - Weckt einen bezüglich des Semaphors blockierten Thread auf, wenn es einen solchen gibt.
 - ✓ Wenn es wartende Threads gibt, sollen diese geweckt werden, ansonsten erfolgt eine Inkrementierung/Erhöhung des Wertes der Integer-Variable
 - » Wait-Operation: `p()` // passieren
 - Blockiert, bis der Wert des Zählers > 0 ist
 - Dekrementiert den Zähler
 - ✓ Falls der Wert eines Semaphors für einen aufrufenden Thread negativ werden würde, so wird dieser blockiert, ansonsten erfolgt die Dekrementierung des Zählers

Synchronisation von Threads

Semaphoren [II]

- ✓ Zähler

```
1  public class Semaphore {
2      private int value;
3
4      public Semaphore(int init) {
5          if (init < 0)
6              init = 0;
7          value = init;
8      }
9
10     public synchronized void p() {
11         while (value == 0) {
12             try {
13                 wait();
14             } catch (InterruptedException e) {
15                 }
16             }
17         value--;
18     }
19
20     public synchronized void v() {
21         value++;
22         notify();
23     }
24 }
```

Synchronisation von Threads

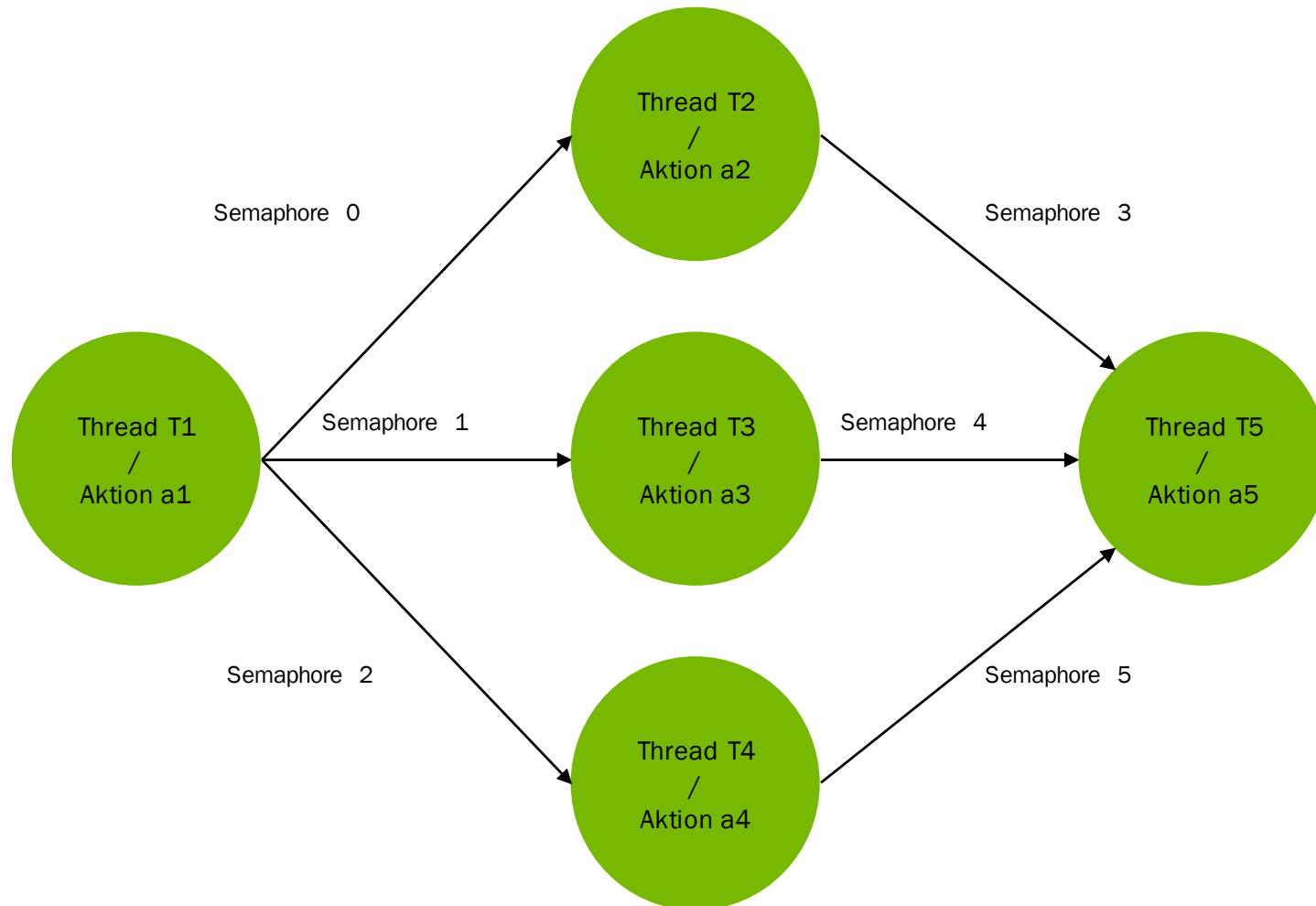
Semaphoren [III]

Gegenseitiger Ausschluss (mutex) im kritischen Abschnitt in der *run*-Methode

- Referenz auf den von allen Threads benutzten Semaphor wird im Konstruktur mit übergeben
 - Konstruktor enthält Aufruf der `start`-Methode
 - Main-Methode: Erzeugung einer Semaphore mit Initialisierungswert 1 (Initialisierungswert gibt an, wie viele Threads den kritischen Abschnitt gleichzeitig durchlaufen können)

Synchronisation von Threads Semaphoren [IV]

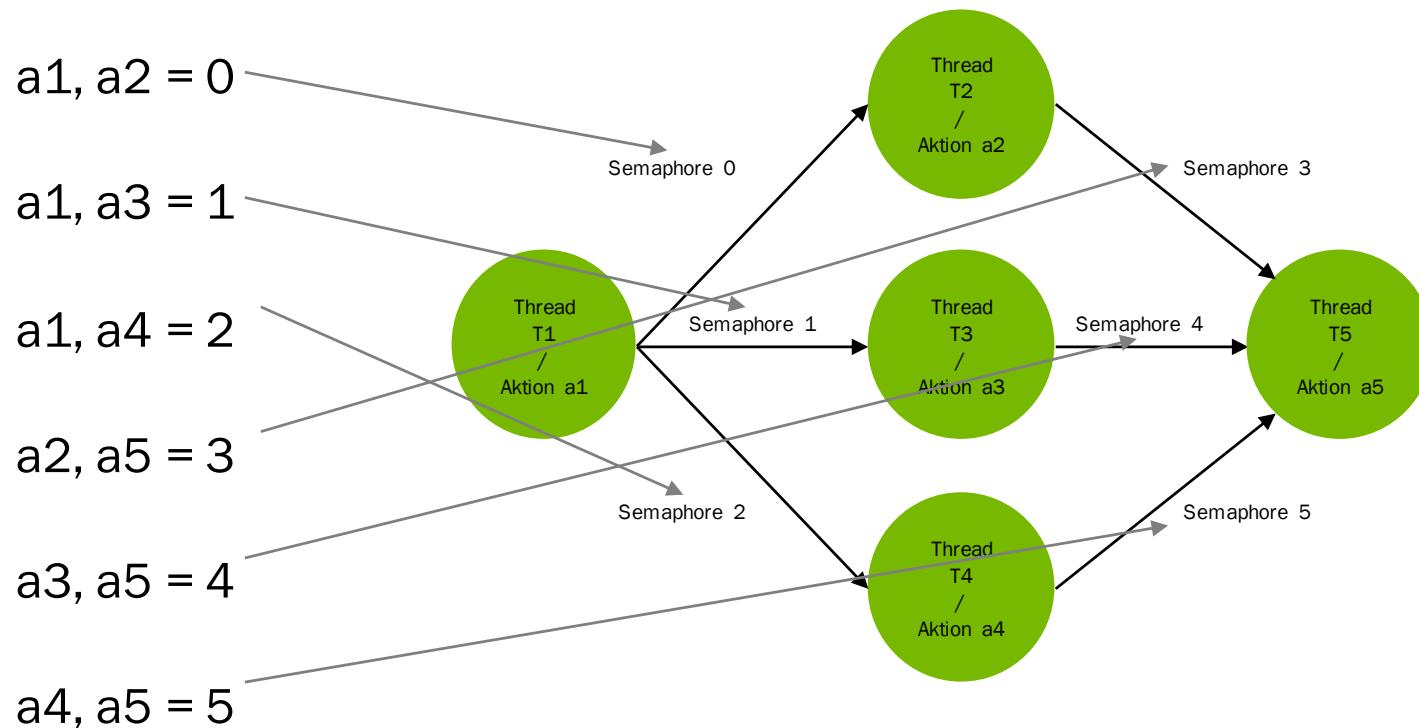
- Beispiel für zeitliche Abfolge von Aktionen in Threads



Synchronisation von Threads

Semaphoren [V]

- Semaphoren verbinden die Kettenglieder miteinander
- Semaphoren-Array für sechs zeitliche Beziehungen (Semaphore[] sems = new Semaphore[6])
- Abgeleitete Elemente im Array: die sechs zeitlichen Beziehungen (in der Ausprogrammierung werden diese bei den jeweiligen Threads genutzt)

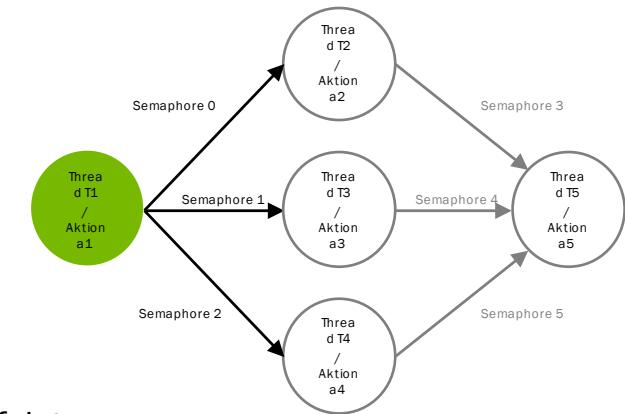


Synchronisation von Threads

Semaphoren [VI]

Was passiert?

- Initialisierung eines Threads T1
- Erzeugen der Methode a1, die „a1“ ausgibt
- Methode *run()* enthält die Ausführungsmethoden des Threads T1 wie folgt:
 - » a1()
 - » „Freigeben“/Wecken der Semaphoren 0,1,2 durch jeweiligen Aufruf der Methode v()



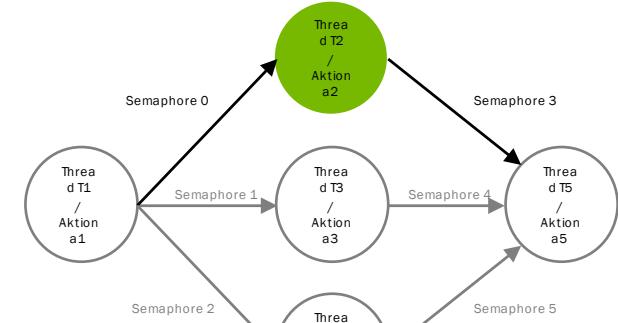
```
1  class T1 extends Thread {  
2      private Semaphore[] sems;  
3  
4      public T1(Semaphore[] sems, String name) {  
5          super(name);  
6          this.sems = sems;  
7          start();  
8      }  
9  
10     private void a1() {  
11         System.out.println("a1");  
12     }  
13  
14     public void run() {  
15         a1();  
16         sems[0].v();  
17         sems[1].v();  
18         sems[2].v();  
19     }  
20 }
```

Synchronisation von Threads

Semaphoren [VII]

Was passiert?

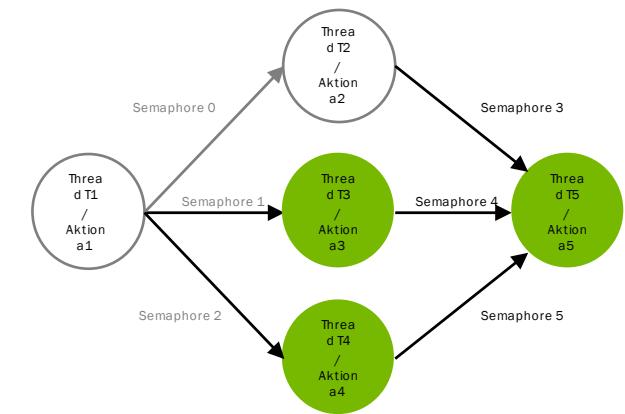
- Initialisierung eines Threads T2
- Erzeugen der Methode a2, die „a2“ ausgibt
- Bevor a2 ausgeführt werden kann, muss die Methode *p()* auf die entsprechende Semaphore (ankommender Pfeil) ausgeführt werden, also enthält die Methode *run()* die Ausführungsmethoden des Threads T2 wie folgt:
 - » Passieren/Blockieren der Semaphore 0 durch Aufruf der Methode *p()*
 - » a2()
 - » „Freigeben“/Wecken der Semaphore 3 durch Aufruf der Methode *v()*



```
1 class T2 extends Thread
2 {
3     private Semaphore[] sems;
4     public T2(Semaphore[] sems, String name)
5     {
6         super(name);
7         this.sems = sems;
8         start();
9     }
10    private void a2()
11    {
12        System.out.println("a2");
13    }
14    public void run()
15    {
16        sems[0].p();
17        a2();
18        sems[3].v();
19    }
20 }
```

Synchronisation von Threads Semaphoren [VIII]

Wie soll nun komplettierend die Anordnung der restlichen Aktivitäten mittels Threads realisiert werden?



Welche Ausführungsmethoden sollen die *run()*-Methoden der jeweiligen Threads enthalten:

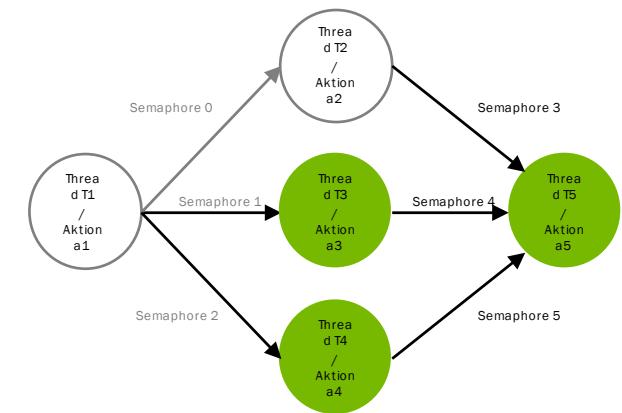
- Thread T3?

- Thread T4?

- Thread T5?

Synchronisation von Threads Semaphoren [IX]

Wie soll nun komplettierend die Anordnung der restlichen Aktivitäten mittels Threads realisiert werden?



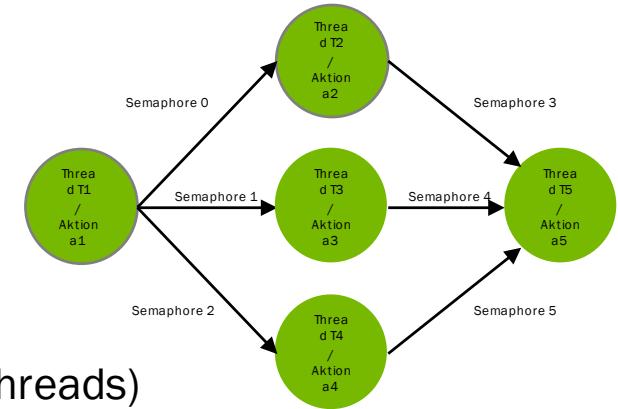
Welche Ausführungsmethoden sollen die *run()*-Methoden der jeweiligen Threads enthalten:

- Thread T3:
 - » Passieren/Blockieren der Semaphore 1 durch Aufruf der Methode *p()*
 - » Ausführung von *a3()*
 - » „Freigeben“/Wecken der Semaphore 4 durch Aufruf der Methode *v()*
- Thread T4:
 - » Passieren/Blockieren der Semaphore 2 durch Aufruf der Methode *p()*
 - » Ausführung von *a4()*
 - » „Freigeben“/Wecken der Semaphore 5 durch Aufruf der Methode *v()*
- Thread T5:
 - » Passieren/Blockieren der Semaphoren 3,4 und 5 durch jeweiligen Aufruf der Methode *p()*
 - » Ausführung von *a5()*

Synchronisation von Threads

Semaphoren [X]

- Methode main()
 - » Neue Semaphore als Array der Größe 6 für zeitliche Relationen zwischen den einzelnen Aktionen (Threads)
 - » Initialisierungswert 0, da
 - Kein gegenseitiger Ausschluss gefordert
 - Alle Threads zunächst schlafen sollen



```
1 public class TimingRelation {
2     public static void main(String[] args) {
3         Semaphore[] sems = new Semaphore[6];
4         for (int i = 0; i < sems.length; i++) {
5             sems[i] = new Semaphore(0);
6         }
7         new T1(sems, "T1");
8         new T2(sems, "T2");
9         new T3(sems, "T3");
10        new T4(sems, "T4");
11        new T5(sems, "T5");
12    }
13 }
```