



TECHNISCHE HOCHSCHULE MITTELHESSEN

THM

**CAMPUS
GIESSEN**

MNI

Mathematik, Naturwissenschaften
und Informatik

Nebenläufigkeit und Parallelität

CS1016 Programmierung interaktiver Systeme

von Prof. Dr. Weigel



Nebenläufigkeit und Parallelität

1. Nebenläufigkeit und Parallelität
2. Parallelität in interaktiven Systemen
3. Java Threads
4. Mutex und Semaphore

Motivation

Sequentielle Prozesse erzeugen Bottlenecks, weil...

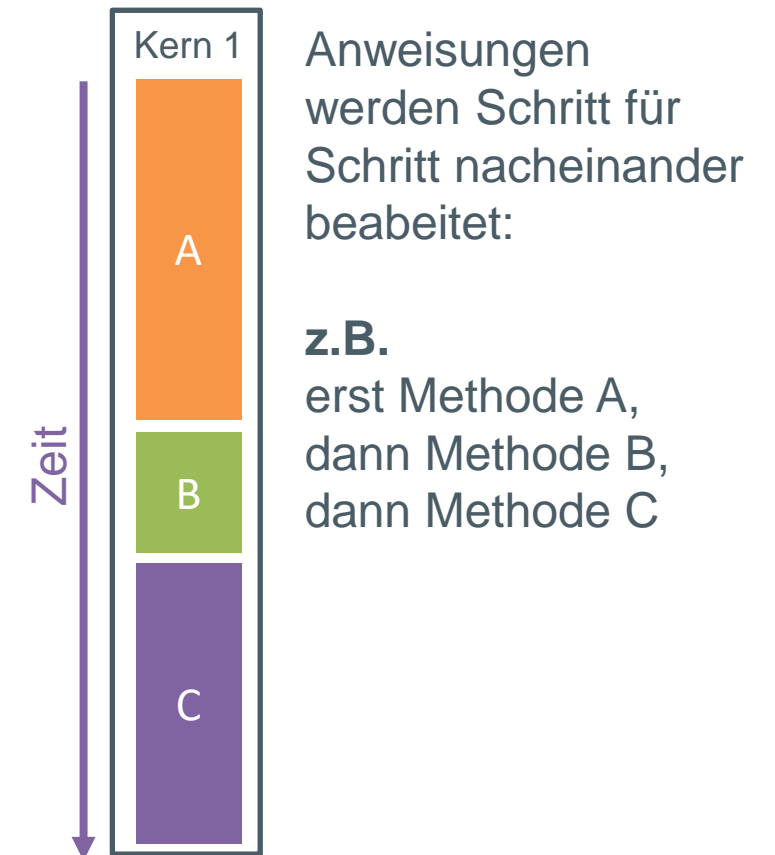
1. Manche Aktivitäten sehr lange dauern, z.B.:
 - komplexen Berechnungen
 - Up-/Download von Daten aus dem Internet
 - Laden und Speichern von Daten (z.B. Festplatte, Netzwerkspeicher)
2. Es entstehen Wartezeiten bei neuen Events, z.B.
 - Benutzereingaben
 - Netzwerk- und Internetkommunikation

Heutige Computersysteme sind für Multi-Tasking ausgelegt!

Parallele Programme können mehrere CPU Kerne verwenden

- beschleunigt Berechnungen
- reduziert Wartezeiten

Sequentielles Prozess



Nebenläufigkeit und Parallelität

Nebenläufigkeit

Aktivitäten haben keine kausale Abhängigkeit voneinander
→ Resultate von A werden nicht für B benötigt (und umgekehrt)

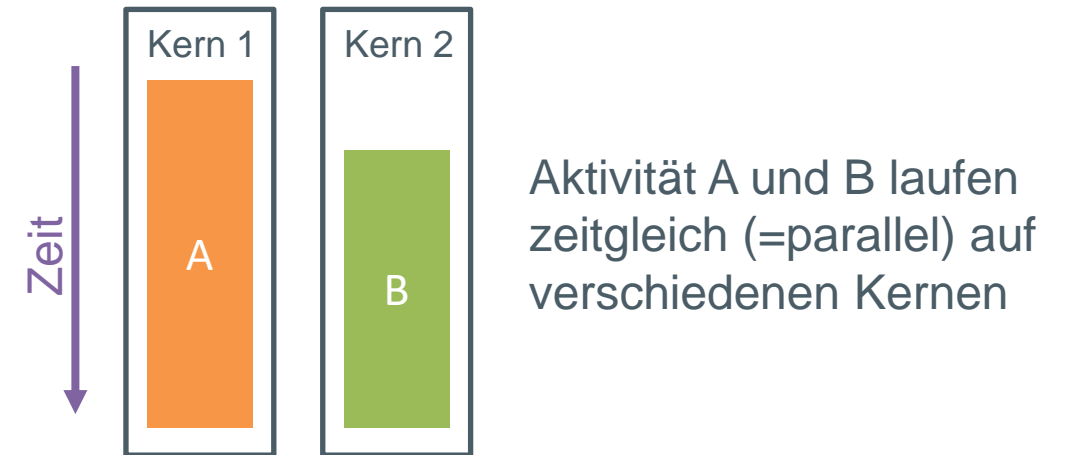
Zeitliche Verzahnung

Wechsel zwischen Aktivitäten
auf dem gleichen CPU Kern



Parallelität

Aktivitäten laufen gleichzeitig
auf unterschiedlichen CPU Kernen



→ Nebenläufigkeit ist Voraussetzung für Parallelität



Nebenläufigkeit und Parallelität

1. Nebenläufigkeit und Parallelität
2. Parallelität in interaktiven Systemen
3. Java Threads
4. Mutex und Semaphore

Das Problem mit dem *sequentiellen* Processing Main-Loop

```
sketch.settings();  
sketch.setup();
```

```
// Vereinfachtes Beispiel eines Main-Loops  
while(true) {  
    // Handle User Input  
    mouseX = ...  
    mouseY = ...  
    if(mouseX!=oldMouseX || mouseY!=oldMouseY)  
        sketch.mouseMoved();  
  
    // Draw at 60 FPS (default)  
    if(<time to draw next frame>)  
        sketch.draw();  
}
```

60FPS bedeutet:

draw() wird alle $16, \bar{6}$ ms aufgerufen

Was passiert wenn **draw() länger braucht?**

1. Die Framerate bricht ein (< 60FPS)
2. Benutzereingaben werden blockiert

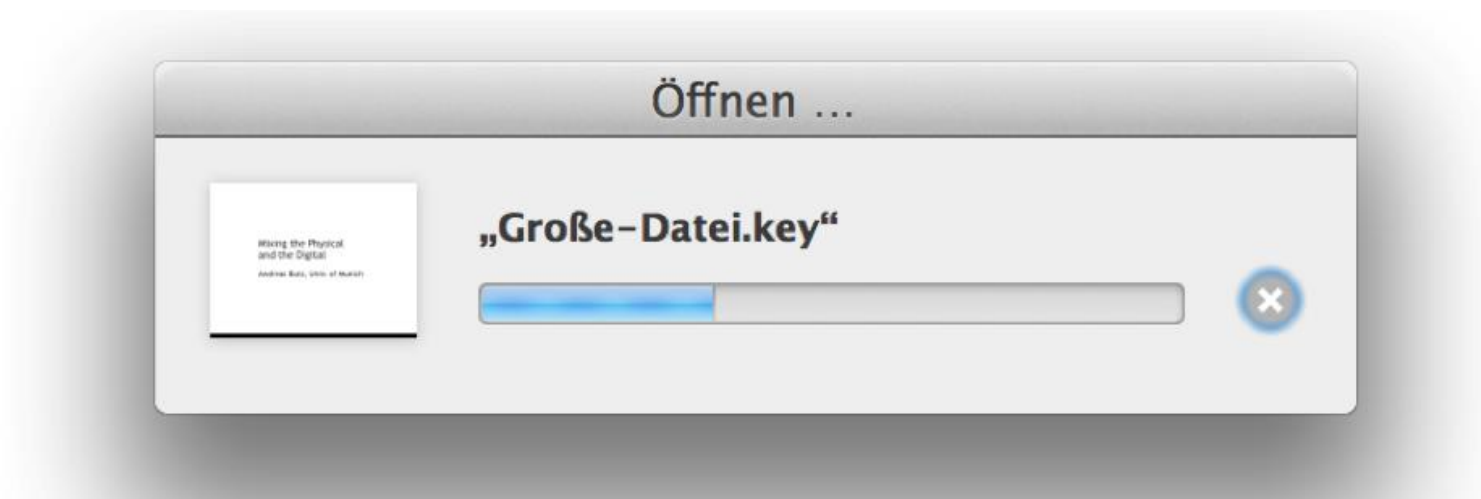
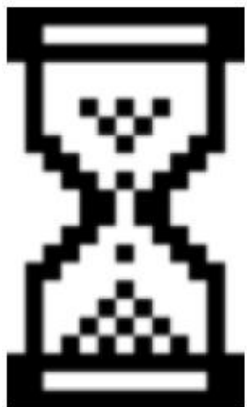
→ Bei sehr großen Verzögerungen:
Die Anwendung reagiert nicht mehr ("friert ein")

Beispiele für Verzögerungen:

- Scene mit vielen Gegnern in einem Videospiel
- Komplexe Berechnungen
- Explorer wartet auf Infos zu Netzwerk-Ordnern

Grundregeln für die UI Gestaltung: Feedback

- Rückmeldung, dass eine Funktion ausgeführt wird / wurde
- Visuelles, auditives oder haptisches Signal



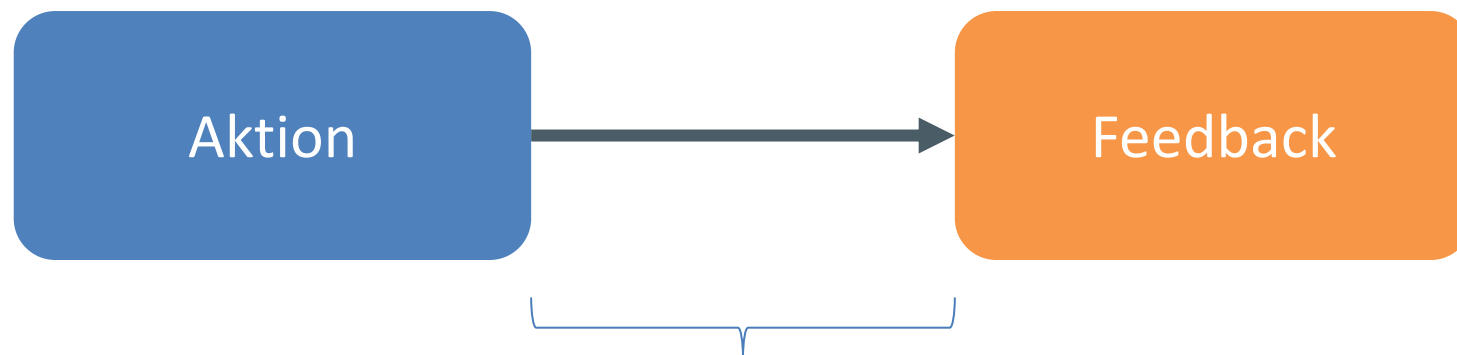
Fehlermeldungen:

- Möglichst verständlich (nicht: "Fehler 1 ist aufgetreten")
- Mit Lösungshinweis

[Bildquelle: <http://www.mmibuch.de>]

Verknüpfung mit einer Aktion

Feedback sollte möglichst zeitnah erfolgen



0-100ms: Direkter kausaler Zusammenhang

< 1s: Feedback wird der Aktion zugeordnet

> 1s: Kein direkter Zusammenhang

Lösung: Nebenläufigkeit

Die GUI soll **nicht** von...

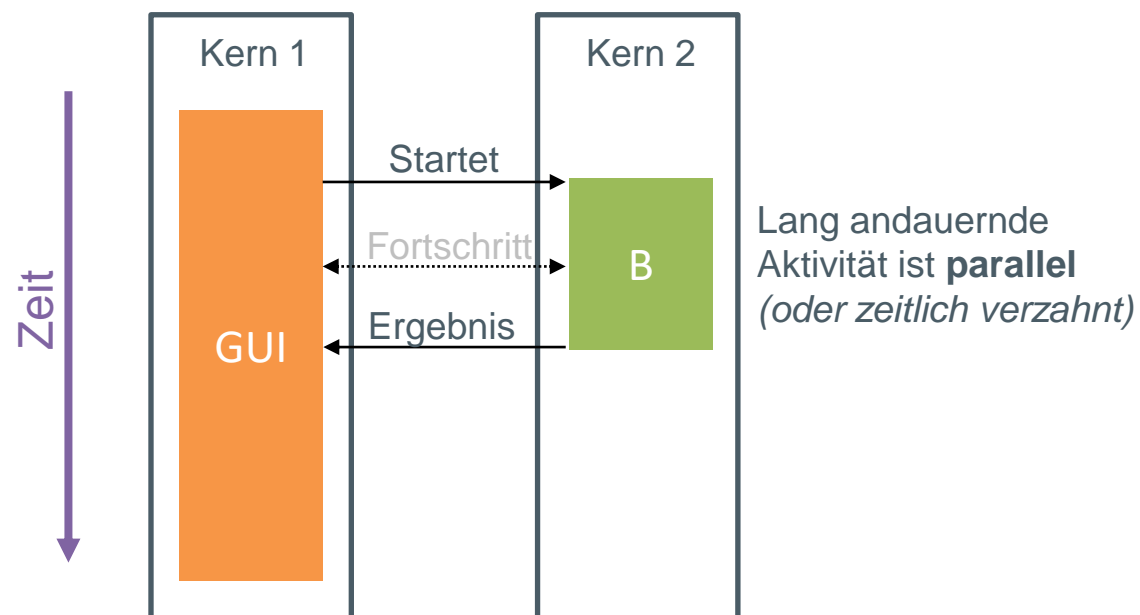
- komplexe Berechnungen
- Laden von Dateien
- Laden von Netzwerkressourcen

blockiert werden!

Die GUI...

- gibt den Fortschritt aus (=Feedback)
- erlaubt es Berechnungen abubrechen (=Fehlertoleranz)

Nutzung von Nebenläufigkeit:





Nebenläufigkeit und Parallelität

1. Nebenläufigkeit und Parallelität
2. Parallelität in interaktiven Systemen
3. **Java Threads**
4. Mutex und Semaphore

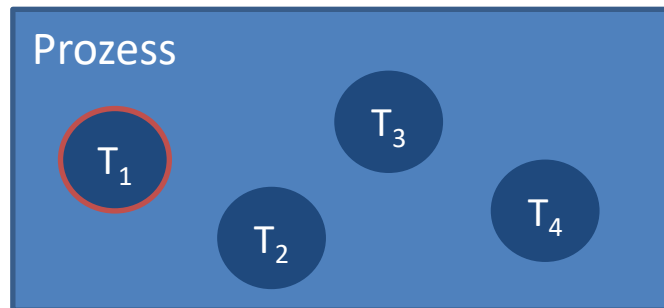
Prozesse vs. Threads

Prozess

Ein Programm während der Ausführung

Prozesse beinhalten:

- Adressraum
- Datenspeicher
- Programmcode
- ...



Thread

Ein Ausführungsfaden in einem Prozess

- Es gibt **einen Haupt-Thread** pro Prozess
→ Bei Java der Aufruf von main()
- Ein Prozess kann **mehrere Threads** besitzen

Threads haben den gleichen Adressraum

Zugriff auf den gleichen Speicher und Ressourcen

Aber eigener Stack + Deskriptor

Java nutzt Threads vom Betriebssystem

→ Erlaubt parallele Ausführung auf mehreren Kernen

Threads in Java

```

class SimpleThread extends Thread {
    private String name;

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

    @Override
    public void run() {
        while(true) {
            try {
                System.out.println(name + " is running");
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
  
```

start() ruft intern run() auf

Startbar mit:

```

var t1 = new SimpleThread("T1");
t1.start();
var t2 = new SimpleThread("T2");
t2.start();
System.out.println("End of Main Thread");
  
```

Ausgabe:

```

End of Main Thread
T1 is running
T2 is running
T2 is running
T1 is running
T2 is running
...
  
```

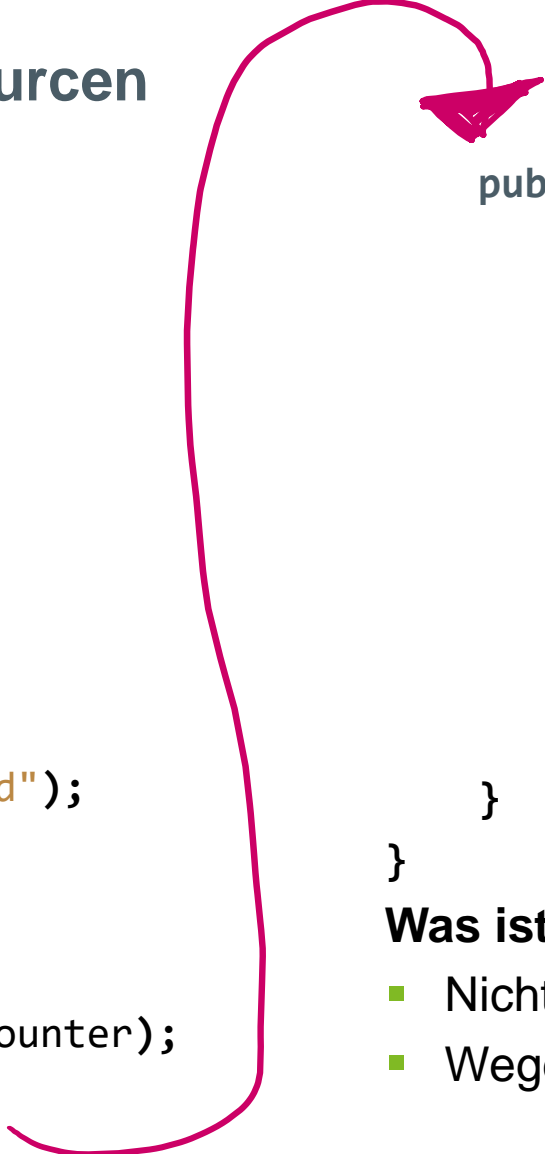
Reihenfolge
nicht fest

Vorsicht bei Zugriff auf Daten & Ressourcen

```
class CounterThread extends Thread {
    static int counter;

    public static void main(String[] args)
        throws InterruptedException {
        var t = new Thread[5];
        for(int i=0; i<t.length; i++) {
            t[i] = new CounterThread();
            t[i].start();
        }
        System.out.println("All threads started");

        for(int i=0; i<t.length; i++)
            t[i].join();
        System.out.println("End result is "+ counter);
    }
}
```



```
public void run() {
    for(int i=0; i<100; i++) {
        var t =(int)(Math.random()*10);
        try {
            int value = counter;
            Thread.sleep(t);
            counter = value + 1;
            Thread.sleep(t);
        } catch [...] {}
    }
}
```

Was ist das Ergebnis? 500?

- Nichtdeterministisch!
- Wegen Race Condition



Nebenläufigkeit und Parallelität

1. Nebenläufigkeit und Parallelität
2. Parallelität in interaktiven Systemen
3. Java Threads
4. **Mutex und Semaphore**

Problem mit Ressourcen



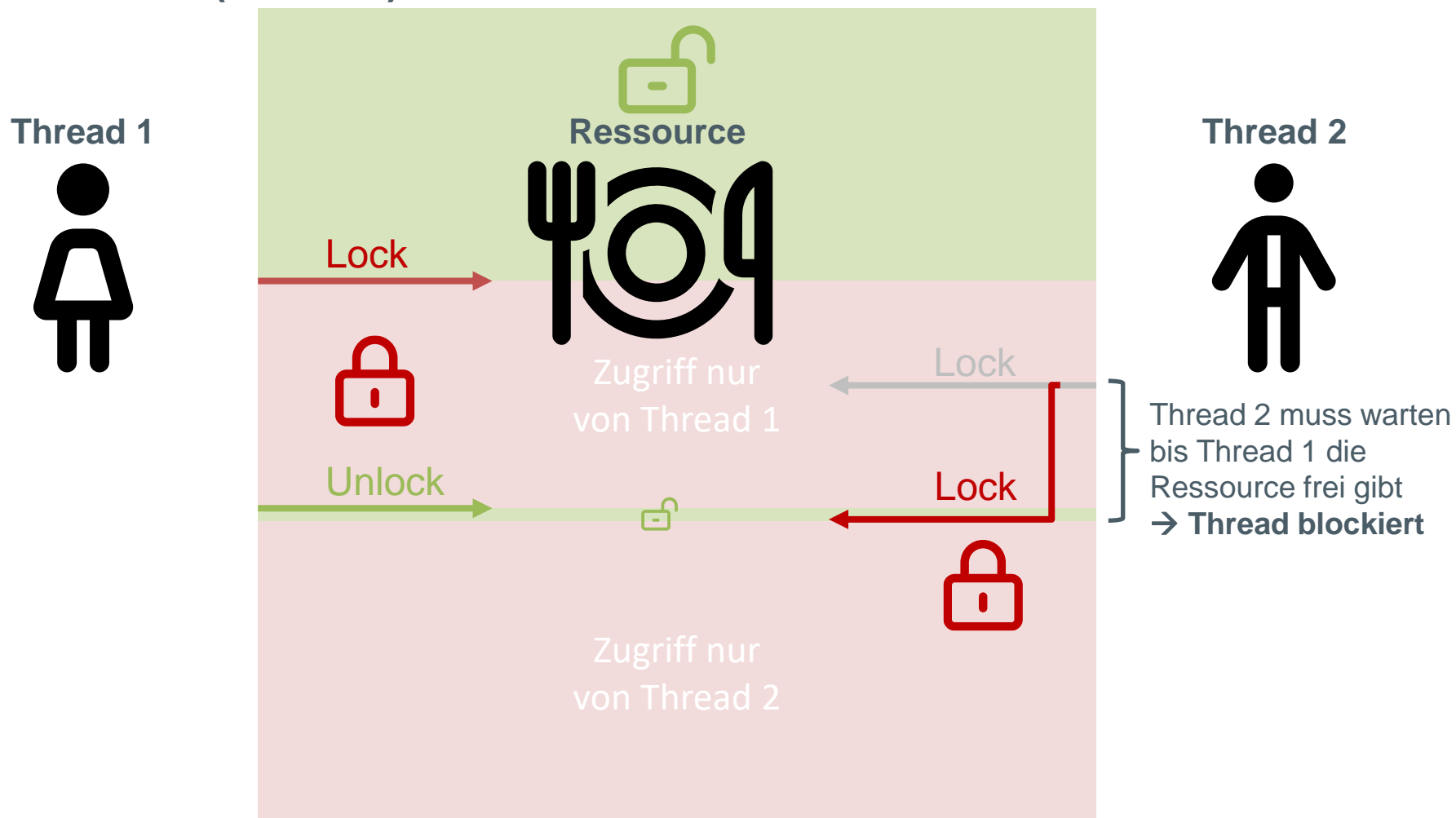
Problem:

Während ein Thread mit der Ressource interagiert, kann ein anderer Thread die Ressource verändern.

→ *Threads können miteinander in Konflikt geraten*

→ *Schwer zu debuggende Fehlerquellen (z.B. Race Conditions)*

Grundidee von Mutex ("Lock")



Mutex mit **synchronized**

@Override

```
public void run() {
    while (true) {
        try {
            synchronized (mutexObj) {
                System.out.print(
                    String.format("[%s] Mutex locked", name));
                Thread.sleep(1000);
                System.out.println(" + released");
            }
            Thread.sleep((int) (Math.random() * 100));
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

synchronized(mutexObj) sperrt das Objekt **mutexObj** für andere Threads, solange der Block bearbeitet wird

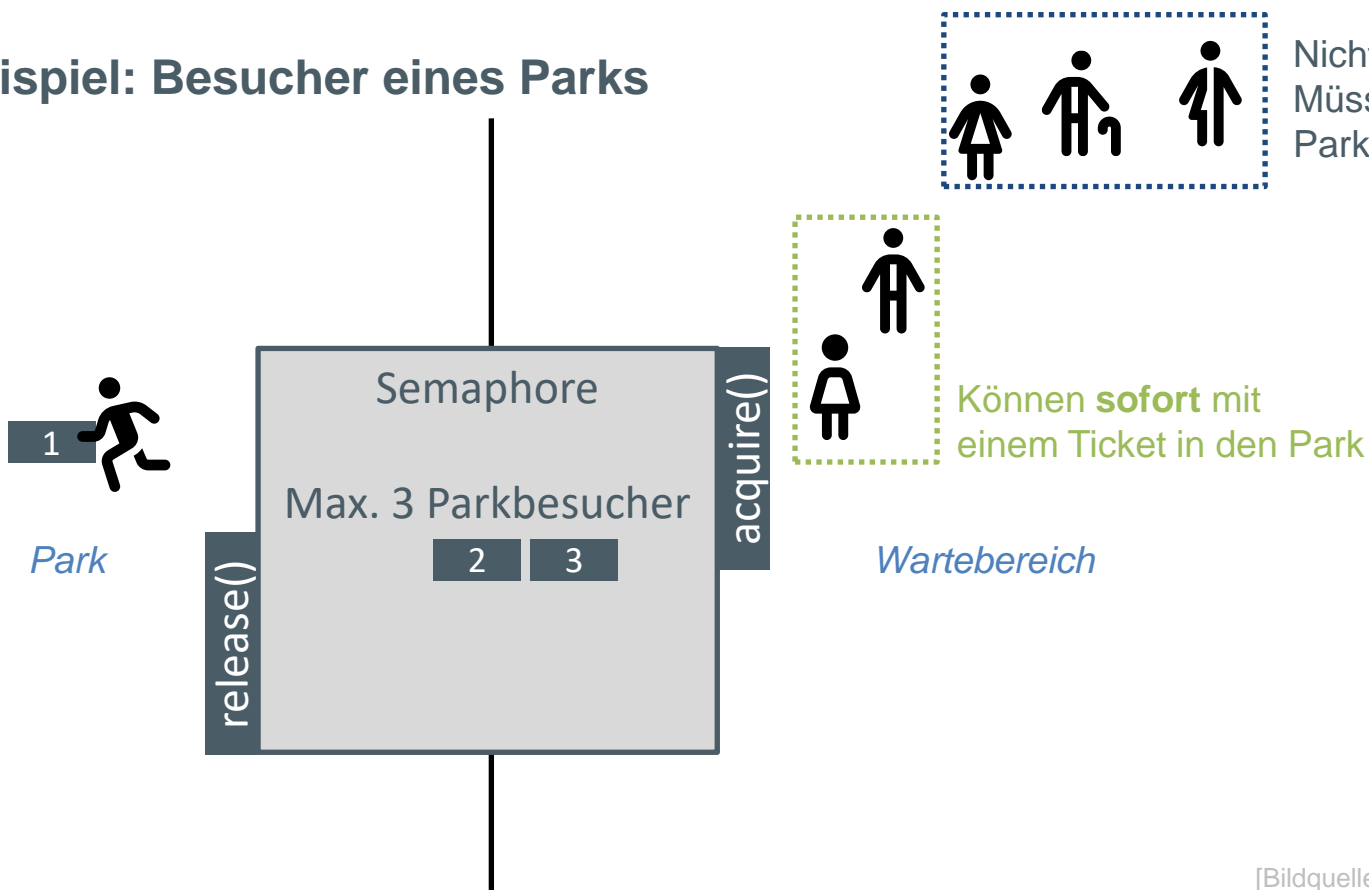
Andere Threads warten bei einem **synchronized(mutexObj)**-Aufruf bis das Objekt vom sperrenden Thread wieder freigegeben wurde

Alle Objekte können als Mutex verwendet werden

Grundidee von Semaphoren

Erlaubt eine Beschränkung auf N-zeitgleiche Ressourcen-Zugriffe

Beispiel: Besucher eines Parks



[Bildquelle rechts: <https://scrumandkanban.co.uk/is-it-ever-okay-to-fully-load-a-system/>]

Semaphoren (limitieren die Anzahl gleichzeitiger Zugriffe)

// Erlaubt zwei zeitgleiche Zugriffe

```
private static Semaphore semaphore = new Semaphore(2);
```

Eine über mehrere Threads geteilte Semaphore, welche 2 zeitgleiche Zugriffe ermöglicht (muss nicht statisch sein)

```
@Override
```

```
public void run() {
```

```
    while (true) {
```

```
        try {
```

```
            semaphore.acquire();
```

Holt sich ein "Zugriffs-Ticket", falls welche vorhanden sind.
Wartet ansonsten bis es freie Tickets gibt (blockiert den Thread!)

```
            System.out.println(String.format("%s acquired semaphore [%d]",  
                                             name, semaphore.availablePermits()));
```

```
            Thread.sleep(10000);
```

```
            semaphore.release();
```

Gibt ein "Zugriffs-Ticket" zurück

```
            System.out.println(String.format("%s released semaphore [%d]",  
                                             name, semaphore.availablePermits()));
```

```
            Thread.sleep((int) (Math.random() * 100));
```

```
        } catch (InterruptedException e) {}
```

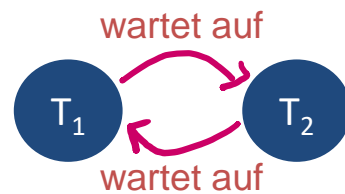
```
    }
```

```
}
```

Deadlocks und das Philosophenproblem (von Edsger W. Dijkstra)

Ein Deadlock ist ein Zustand in dem mehrere Threads/Prozesse aufeinander warten

→ Das Programm ist “verklemmt”



Das Philosophenproblem:

Fünf Philosophen essen an einem runden Tisch.

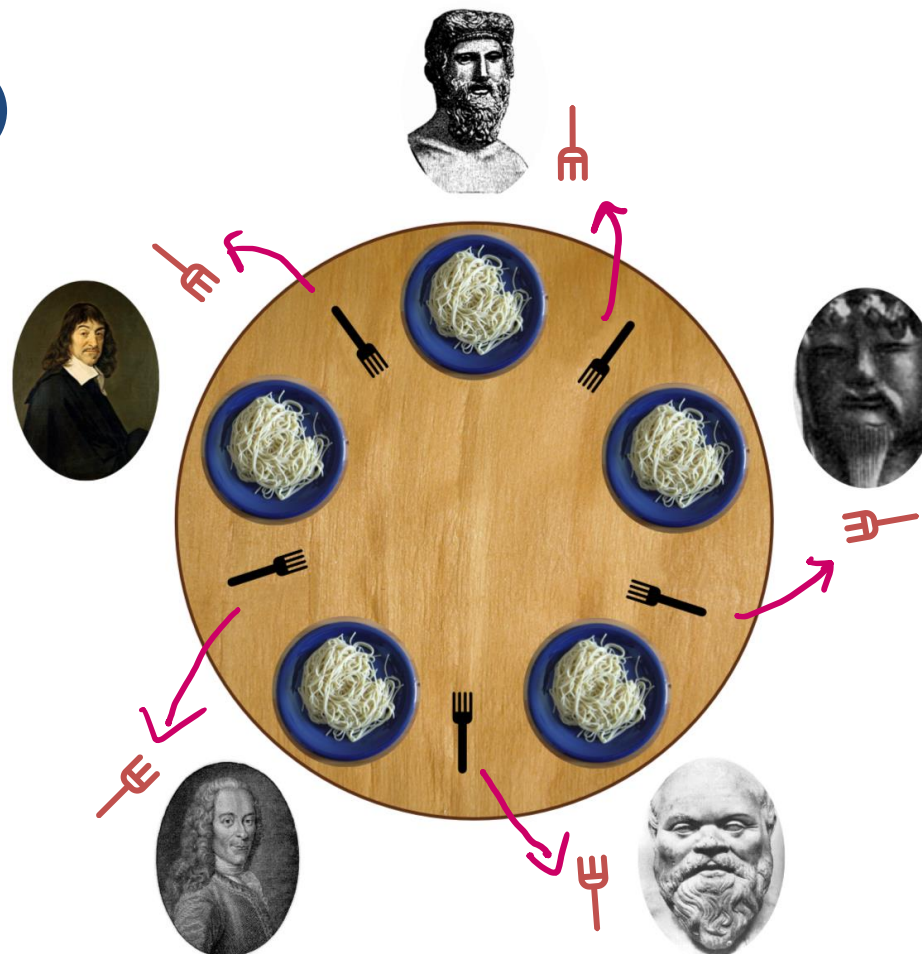
Jeder Philosoph braucht **zwei Gabeln** zum Essen.

Wenn ein Philosoph **hungrig** ist, greift er zuerst nach der linken und danach zur rechten Gabel

- Wenn es **keine zweite Gabel** gibt, wartet er philosophierend mit der linken Gabel in der Hand
- Wenn er **satt** ist, legt er beide Gabeln zurück

Solange nur 1-4 Philosophen hungrig sind OK

Wenn alle fünf gleichzeitig hungrig sind → **Deadlock**



[Bildquelle: https://de.wikipedia.org/wiki/Philosophenproblem#/media/Datei:An_illustration_of_the_dining_philosophers_problem.png]

Fragen?



[Bildquelle: <https://www.reddit.com/r/LitterboxComics/comments/m5ompw/deadlock/>]