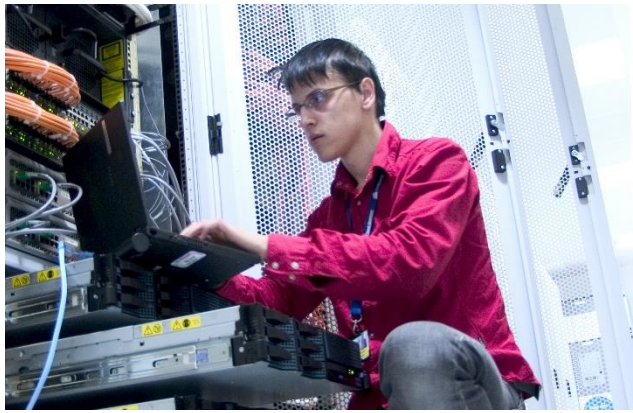
EMIL-VON-BEHRING GYMNASIUM

Verklemmung (Deadlock)



Verklemmung (Deadlock)

- Bei einer Verklemmung kann ein Aktivitätsträger nicht weiterarbeiten, weil er auf irgendetwas wartet, das nicht eintreten wird bzw. nicht verfügbar wird.
- *-> Schwerwiegender Fehler!*
- In Java kann eine solche Verklemmung (fast) nicht aufgelöst werden, nachdem sie eingetreten ist.
  -> Daher muss der Programmierer sicherstellen, dass Verklemmungen gar nicht erst eintreten.

Verklemmung (Deadlock)

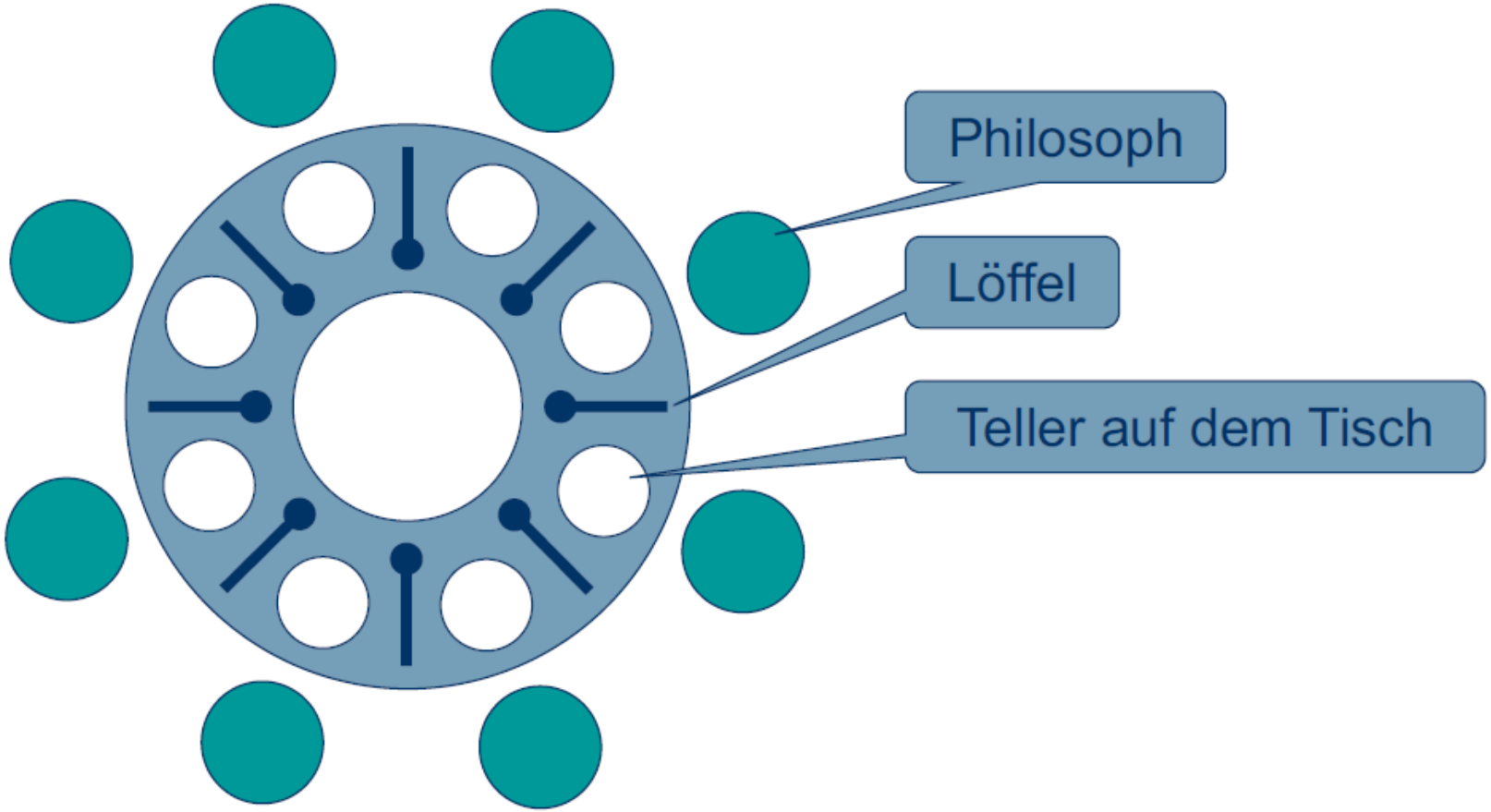
- Verklemmung im Straßenverkehr

Es geht nicht weiter, weil die Kreuzung belegt ist. Zurück geht es aber auch nicht mehr, weil nachfolgende Fahrzeuge vorhanden sind.

- Man spricht von einer zyklischen Wartesituation (deadly embrace).

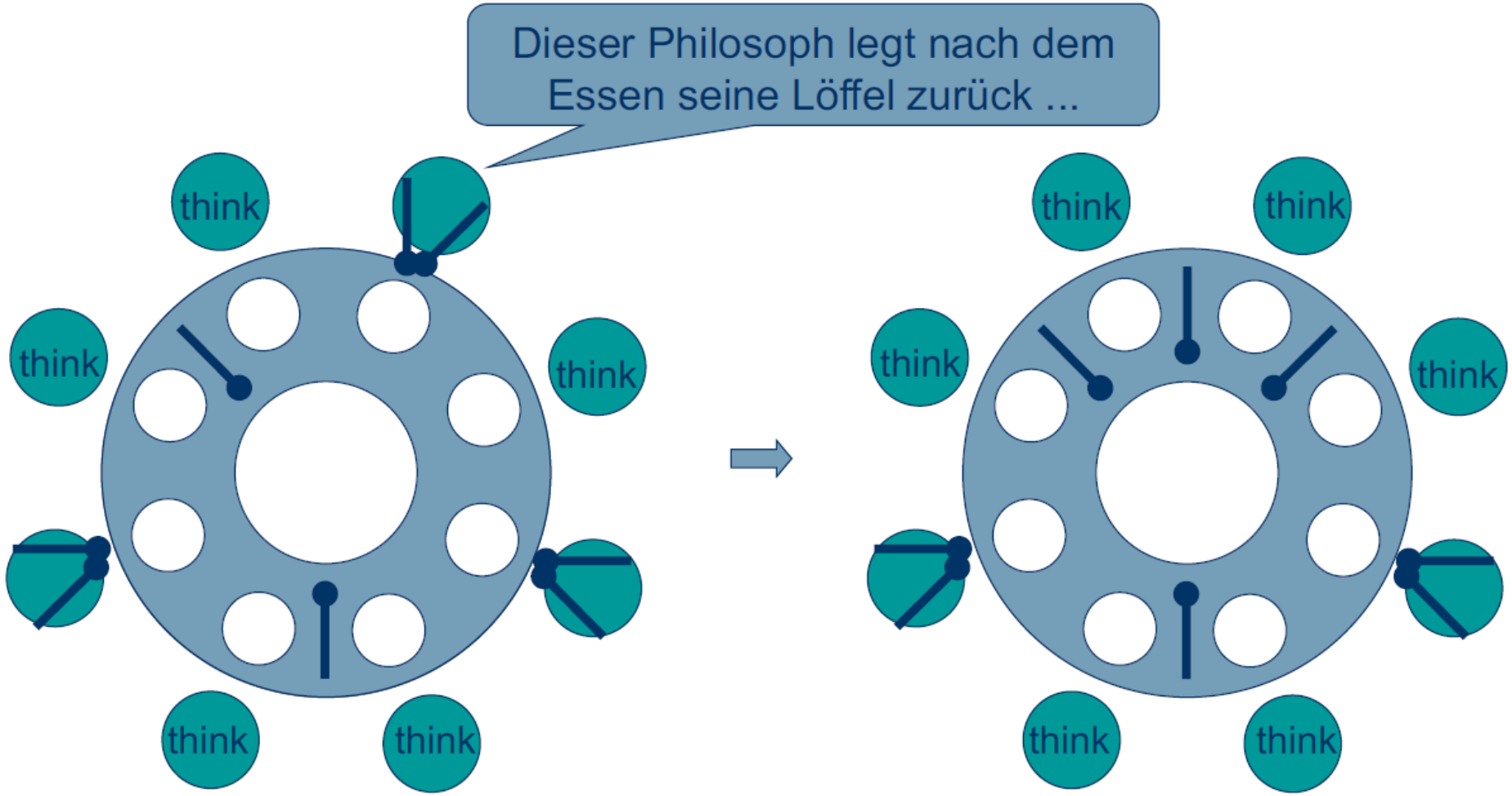
Verklemmung (Deadlock)

- Klassisches Problem für Verklemmung: Dinierende Philosophen
- Hauptaufgabe von Philosophen: denken und essen!



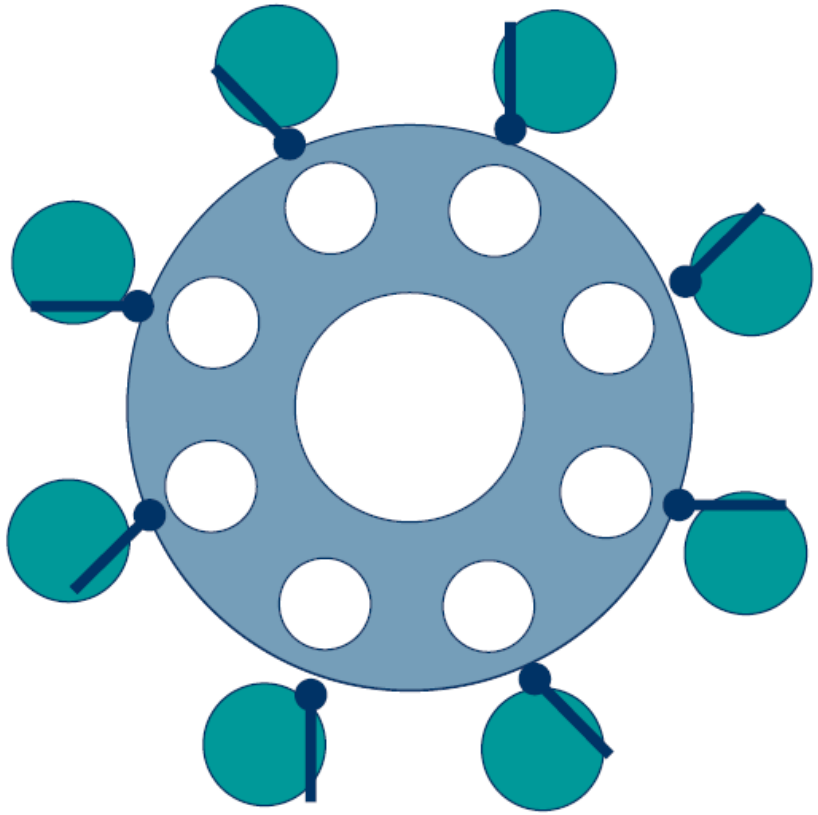
Verklemmung (Deadlock)

- Beispielablauf:



Verklemmung (Deadlock)

- Wenn aber alle Philosophen gleichzeitig den jeweils rechten Löffel greifen…



-> Verklemmung:

… dann wartet jeder Philosoph vergeblich und für immer auf den linken Löffel.

Notwendige Voraussetzungen für Verklemmungen

- Folgende drei notwendigen Bedingungen müssen erfüllt sein, damit es zu einer Verklemmung kommen ***kann***:

1. **Gegenseitiger Ausschluss**
   - Aktivitätsfäden belegen „Dinge“ (Marken, Code-Stücke, Löffel) exklusiv.
   - Wenn es keinen gegenseitigen Ausschluss gäbe, wenn also alle Aktivitätsfäden die „Dinge“ nebenläufig nutzen können, ohne auf diese warten zu müssen, kann es keine Verklemmung geben.
   - Im Abschnitt über Synchronisierung haben wir gesehen, dass es gegenseitigen Ausschluss geben muss, um Korrektheit zu erreichen.
     - -> Das Vorliegen dieser Voraussetzung für Verklemmungen kann also vom Programmierer fast gar nicht verhindert werden.

Notwendige Voraussetzungen für Verklemmungen

- Folgende drei notwendigen Bedingungen müssen erfüllt sein, damit es zu einer Verklemmung kommen ***kann***:

2. **Kein Entzug**
   - Keinem Aktivitätsfaden kann eine Belegung von außen entzogen werden.
   - Sonst könnten (drohende) Verklemmungen verhindert werden.
   - „Philosophen prügeln sich nicht um Löffel."
   - Java: komplizierte Lösung mit **interrupt** denkbar, um diese Voraussetzung zu verhindern.
   - (Datenbanksysteme: „Dinge" werden im Verklemmungsfall entzogen.)

Notwendige Voraussetzungen für Verklemmungen

- Folgende drei notwendigen Bedingungen müssen erfüllt sein, damit es zu einer Verklemmung kommen ***kann***:

3. **Iterative Anforderung**
   - Es werden weitere „Dinge“ zur exklusiven Nutzung angefordert, obwohl bereits „Dinge“ exklusiv belegt sind.
   - Wenn die Philosophen atomar 2 Löffel greifen würden, dann tritt keine Verklemmung auf.

- Das waren die notwendigen Bedingungen. Die zusätzliche Bedingung, sodass eine Verklemmung auftritt ist die zirkuläre Abhängigkeit (engl. deadly embrace).

Gegenmaßnahmen

Diese Möglichkeiten gibt es um Verklemmungen zu verhindern:

- Voll nebenläufiges Design: Die Thread besitzen gemeinsamen Strukturen, bei denen sie sich gegenseitig ausschließen könnten. (selten realisierbar)
- Angeforderte locks nach einer gewissen Zeit/Bedingung/… wieder frei geben
- Atomar alle locks anfordern, sodass kein anderer Thread die Anforderung unterbrechen kann.
- Globale Ordnung der locks: Diese Variante kommt sogar ohne Synchronisation aus.

Klasse ReentrantLock

- Das Paket **java.util.concurrent.locks** bietet Klassen, mit denen gegenseitige Ausschlüsse feiner kontrolliert werden können als mit **synchronized**-Code.
- Klasse ReentrantLock:
  - **lock()** erwirbt die Marke oder blockiert bis die Marke frei ist.
  - **unlock()** gibt die Marke wieder frei.
    - Beides mit den üblichen Auswirkungen auf die Sichtbarkeit (also analog zu synchronized)
  - **boolean tryLock()** liefert sofort **false** zurück, falls die Marke belegt ist; sonst Wirkung von **lock()**. Auch mit maximaler Wartezeit, ehe aufgegeben wird.
  - **boolean isLocked()**
- Ferner kann man herausfinden,
  - welches **Thread**-Objekt die Marke besitzt,
  - welche **Thread**-Objekte auf die Marke warten,
  - …

Aufgabe: Verklemmung

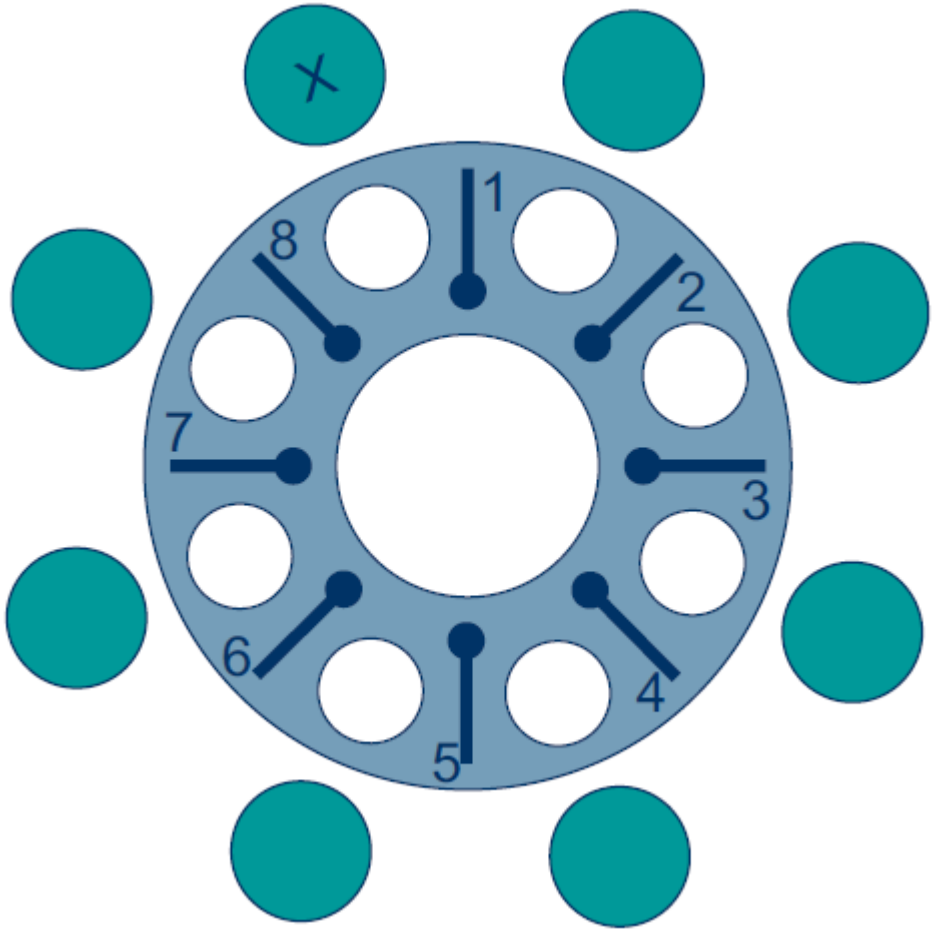
- Importiere die Klassen aus der .zip-Datei Verklemmung in deine Entwicklungsumgebung und verschaffe dir einen Überblick über die Klasse Verklemmung, Gabel, Philosoph und PhilosophDeadlock und teste die Verklemmung des Programms.

Aufgabe: Gegenmaßnahme Belegung auf einen Schlag

- Die notwendige Bedingung der iterativen Anforderung wird ausgehebelt, indem man die beiden Gabeln „gleichzeitig“ (d.h. für andere Threads nicht unterbrechbar) anfordert. -> synchronized
- Nutze anschließend die Gegenmaßnahme (Belegung auf einen Schlag), sodass sich die Klasse PhilosophGleichzeitigeBelegung nicht mehr verklemmt:
  - Nutze einen möglichst kleinen synchronized-Block
  - Überlege dir, welche Objekte sich zum Synchronisieren eigenen. Hier gibt es mehrere Möglichkeiten.

Aufgabe: Globale Ordnung

- Gabeln (= Marken) sind durchnummeriert. Philosophen greifen immer erst den Löffel mit der kleineren Nummer (= Erwerben entsprechend der globalen Ordnung).
  - Bis auf den Philosophen X greifen alle erst nach dem rechten, dann nach dem linken Löffel. Da X in der problematischen Situation den Löffel 1 nicht erhält, wird mindestens 1 Philosoph mit dem Essen beginnen können.
- Implementiere diese Variante in der Klasse PhilosophGlobaleOrdnung, sodass sich diese Variante nicht mehr verklemmt.



Beispiele für schlechte Synchronisation

- Am this-Objekt darf nicht synchronisiert werden, da sonst jeder Thread nur sich selbst synchronisiert!
- An den Gabeln synchronisieren löst das Problem auch nicht:

```
private final Object left
   = new Object();
private final Object right
   = new Object();

public void leftRight() {
   synchronized(left) {
      synchronized(right) {
         // eat
}  }  }

public void rightLeft() {
   synchronized(right) {
      synchronized(left) {
         // eat
}  }  }
```

Problematische zeitl. Verzahnung: