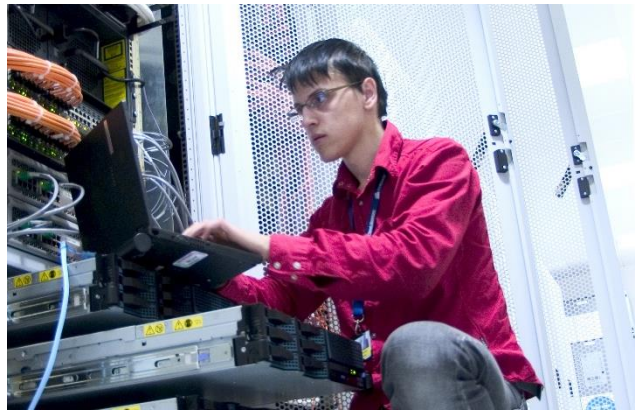
EMIL-VON-BEHRING GYMNASIUM

Verklemmung (Deadlock)



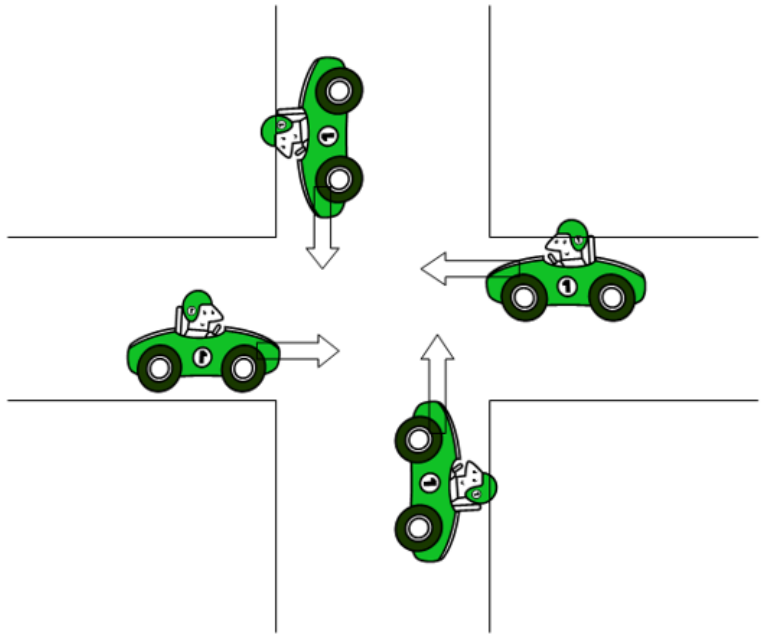
Verklemmung (Deadlock)

- 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 vorher sicherstellen, dass Verklemmungen gar nicht erst eintreten.

Verklemmung (Deadlock)

- Verklemmung im Straßenverkehr

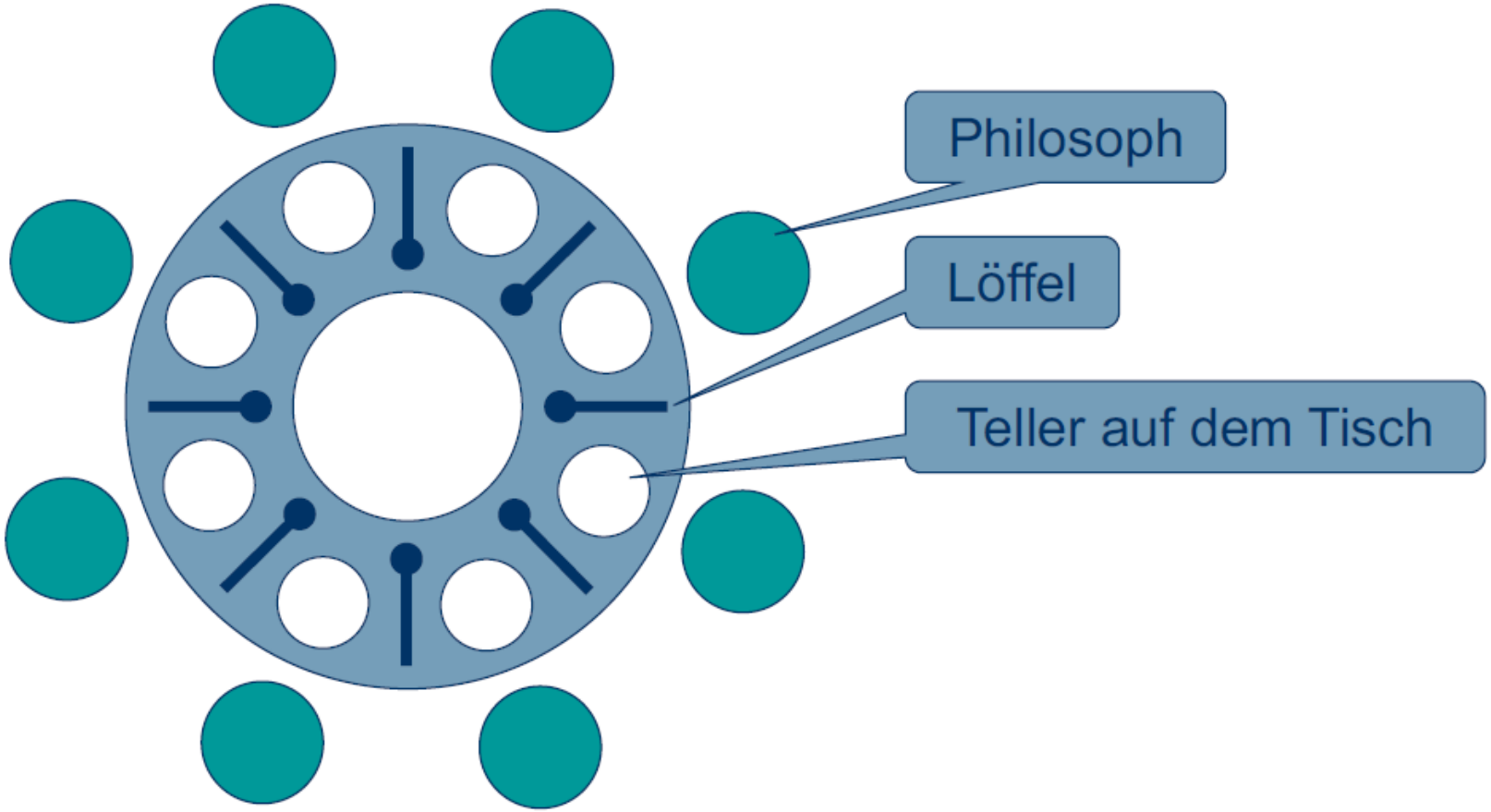
Es geht nicht weiter, weil die Kreuzung belegt ist. Zurück geht es aber auch nicht mehr, weil nachfolgende Fahrzeuge vorhanden sind.



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

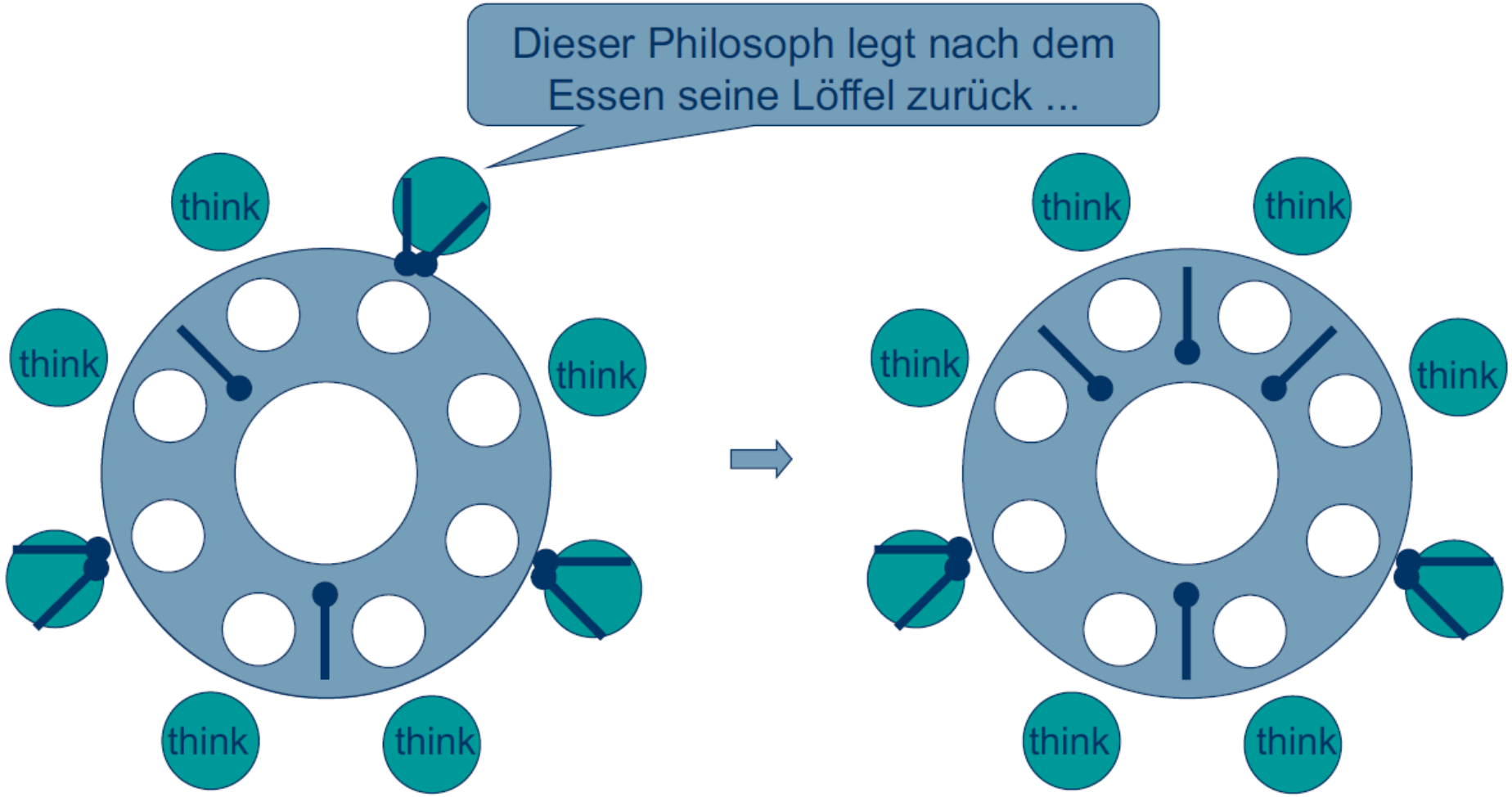
Verklemmung (Deadlock)

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



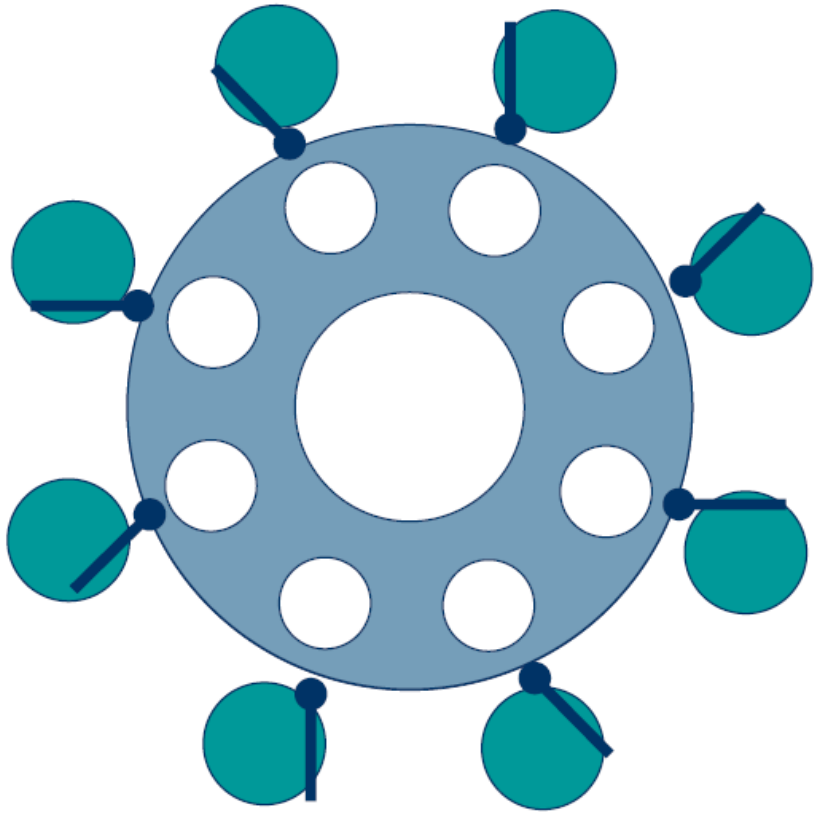
Verklemmung (Deadlock)

- 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.

Coffman-Bedingungen für eine Verklemmung

1. **Gegenseitiger Ausschluss**
   - Jedes Betriebsmittel (Marken, Löffel, Code-Abschnitte, Speicher, …) wird entweder von genau einem Thread belegt oder es ist verfügbar.
2. **Halten und Warten**
   - Ein Thread, der bereits Betriebsmittel belegt, kann weitere Betriebsmittel anfordern.
3. **Ununterbrechbarkeit**
   - Die von einem Thread belegten Betriebsmittel können nicht von außen entzogen werden; der Thread selbst muss sie freigeben.
4. **Zyklisches Warten**
   - Es gibt eine zyklische Kette von Threads, von denen jeder ein Betriebsmittel angefordert hat, das vom nächsten Thread der Kette belegt ist.

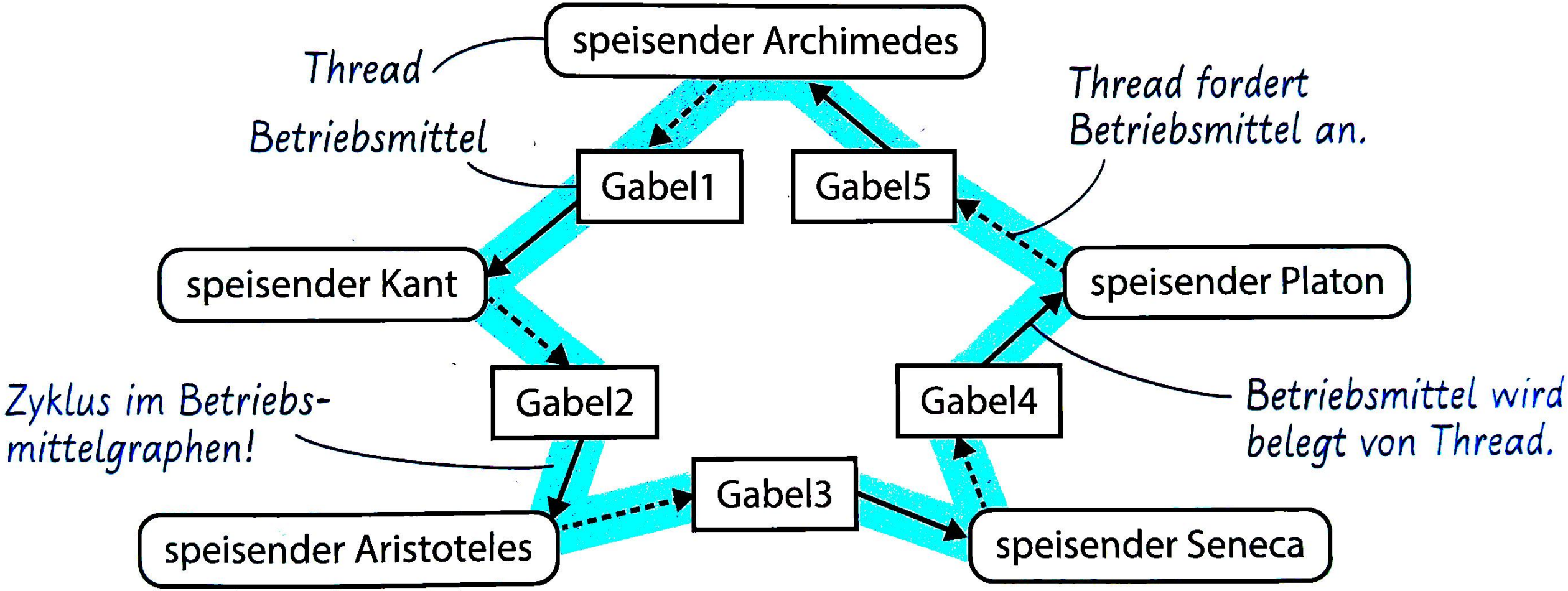
Deadlock (Verklemmung)

- Das Philosophenbeispiel erfüllt alle Coffman-Bedingungen und kann sich deshalb bei seiner Ausführung verklemmen!
- Man verhindert eine Verklemmung, indem man mindestens eine dieser Coffman-Bedingungen verhindert.
- Ideen?

Gegenmaßnahmen

- Wechselseitiger Ausschluss beseitigen
  - Die benötigten Betriebsmittel für alle Prozesse zugänglich zu machen, indem man den exklusiven Zugriff auflöst. → voll nebenläufiges Design (selten realisierbar)
- Hold and Wait verhindern
  - Jeder Prozess gibt zu Beginn an, welche Betriebsmittel er benötigt. Falls alle benötigten Betriebsmittel gleichzeitig frei sind, bekommt sie ein Prozess auf einmal zugeteilt.
- Unterbrechbarkeit einführen
  - Einem Prozess werden Betriebsmittel entzogen, um sie einem anderen zuzuteilen.
- Zyklisches Warten ausschließen
  - Die Betriebsmittel werden z. B. linear geordnet und nur in dieser definierten Reihenfolge angefordert oder zugeteilt.

Betriebsmittelzuteilungsgraph



- Abituraufgabe II 3a,b) https://www.isb.bayern.de/fileadmin/user_upload/Gymnasium/IlluPA/Inf/IlluPA_Informatik_Beispielaufgaben_gA.pdf

Exklusives Betriebsmittel mit der 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

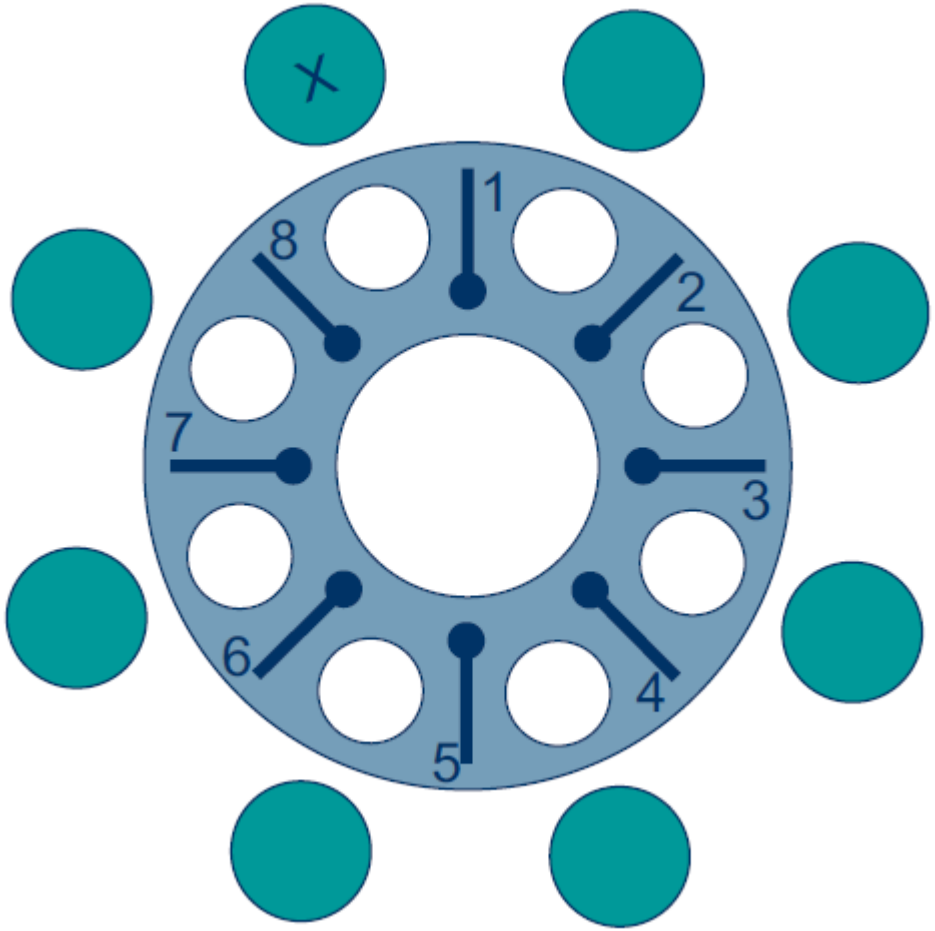
- Öffne das BlueJ-Projekt Philosophen Vorlage und verschaffe dir einen Überblick über die Klasse Gabel, Philosoph, PhilosophDeadlock und Verklemmung.
- Teste die Verklemmung des Programms durch das Ausführen der statischen Methode PhilosophenDeadlock(…) der Klasse Verklemmung.

Aufgabe: Gegenmaßnahme Halten und Warten verhindern

- Die notwendige Bedingung der iterativen Anforderung der Gabeln wird ausgehebelt, indem man die beiden Gabeln „gleichzeitig“ 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: Zyklisches Warten ausschließen

- 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: