Beispiel

x = 0;

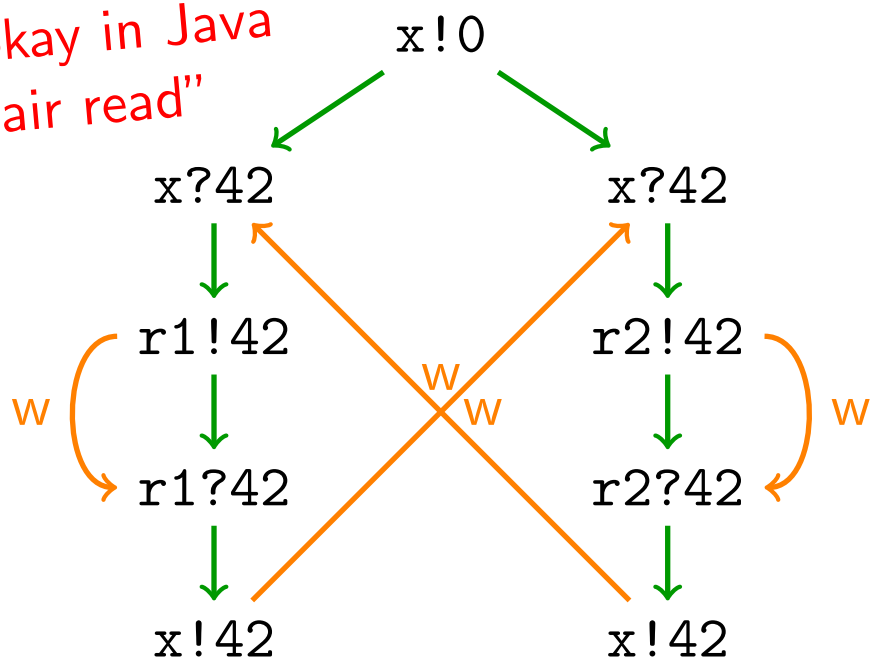
r1 = x;
x = r1;

r2 = x;
x = r2;

- $\rightarrow \subseteq \leq$ ordne ein wenig mehr, wenn gewünscht
- für alle $x?v \in \mathcal{I}$ gilt:
  - $\neg\,(w(x?v) \geq x?v)$ lies nicht aus der Zukunft
  - es gibt kein $x!v' \neq w(x?v)$ so dass $w(x?v) \leq x!v' \leq x?v$

lies nur die aktuellsten Werte

schwach konsistent
nicht okay in Java
"Thin air read"



Schwache Konsistenz

- schwach konsistent und nicht okay in Java heißt: Die Java Semantik versucht (leider wenig erfolgreich), Thin air reads zu vermeiden, aber alle denkbaren Optimierungen zu erlauben.
- schwach konsistent und okay in Java heißt: "es gibt jemand, der kennt jemand, der hat von jemand gehört, der ne Compileroptimierung kennt, die das kann."

Fences

Verbessert sich die Situation, wenn wir Locks bzw. Fences benutzen?

Eine `fence`-Operation garantiert, dass für alle Operationen nach `fence` gilt:
Es sind alle Speicheroperationen "sichtbar", die in den Threadordnungen vor dem `fence` liegen.

Schreibe alle Werte des Agenten (liegen in irgendeinem Cache) in den RAM und sage den anderen Agenten, dass sie ihre Werte neu aus dem RAM lesen müssen!

Eingezäunter Speicher

Wir erweitern $\mathcal{I}$ um $fence$-Operationen.

Definition: Schwache Konsistenz mit $fence$-Anweisungen

Eine mit einem Programm P mit $fence$-Ausdrücken kompatible Ausführung $(\mathcal{I}, \rightarrow, w)$ ist schwach konsistent, wenn es eine partielle Ordnung $\leq$ auf $\mathcal{I}$ gibt, so dass gilt:

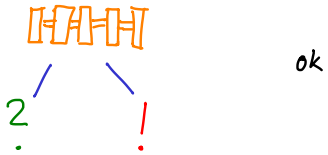
- $\rightarrow \subseteq \leq$
- für alle $x?v \in \mathcal{I}$ gilt:
  - $\neg(w(x?v) \geq x?v)$
  - es gibt kein $x!v' \neq w(x?v)$ so dass $w(x?v) \leq x!v' \leq x?v$
- für alle $fence$-Operationen in P und alle Leseoperationen $x?v \in \mathcal{I}$ gilt:

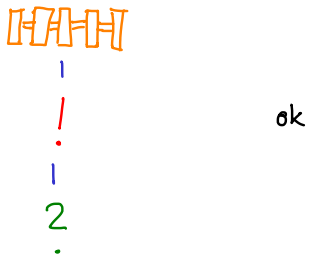
$$w(x?v) \geq fence \Longrightarrow x?v \geq fence$$

bis hier alles wie gehabt

Also: wenn das Schreiben, das $x?v$ begründet, nach dem $fence$ stattfand, dann muss auch $x?v$ nach dem $fence$ sein.

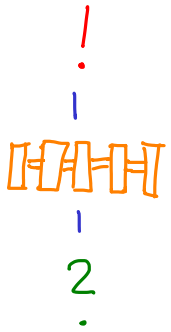
Fences





$$w(x?v) \geq fence$$
$$\Longrightarrow x?v \geq fence$$

Fences



ok

nicht ok

nicht ok

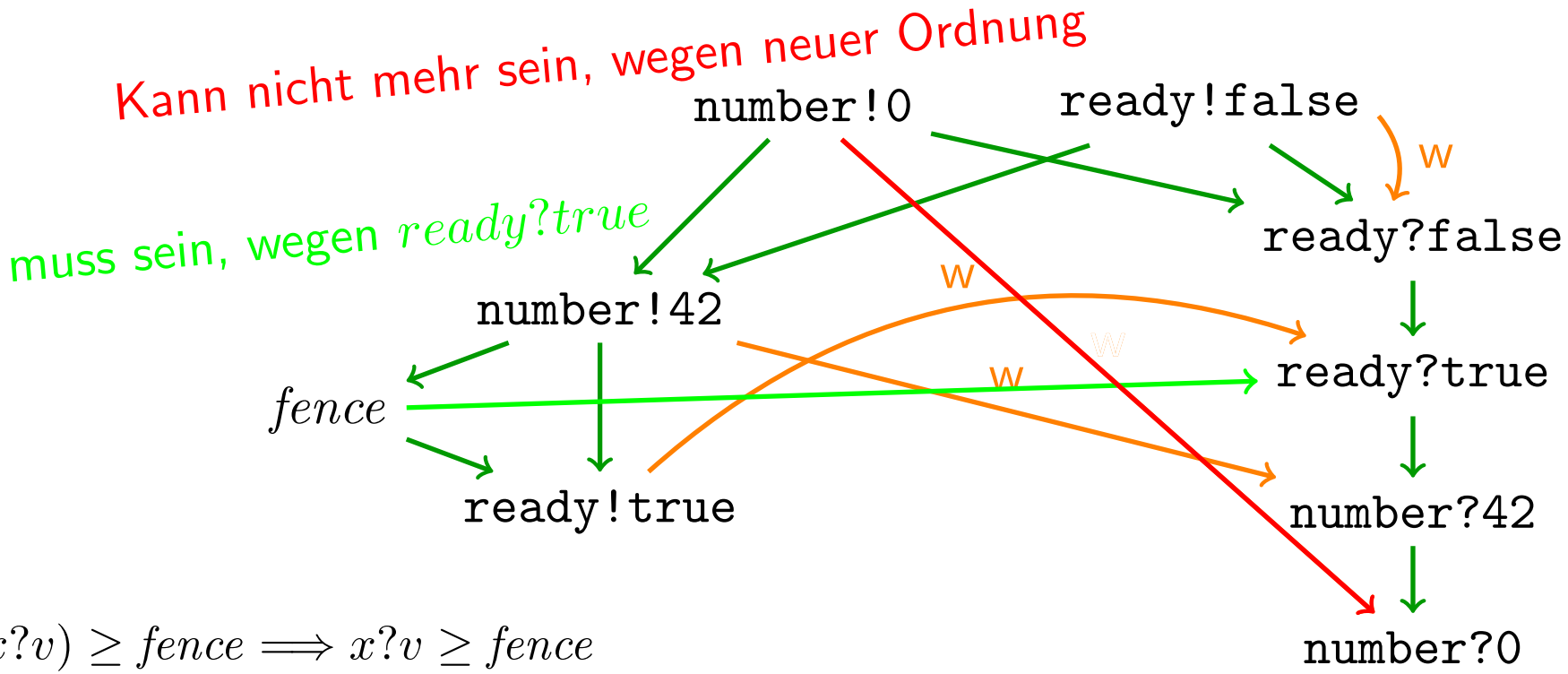
$$w(x?v) \geq fence$$
$$\Longrightarrow x?v \geq fence$$

Schwache Konsistenz mit Fences

```
private static boolean ready = false;
private static long number = 0;
```

```
number = 42;
ready = true;
```

```
while (!ready)
Thread.yield();
System.out.println(number);
System.out.println(number);
```



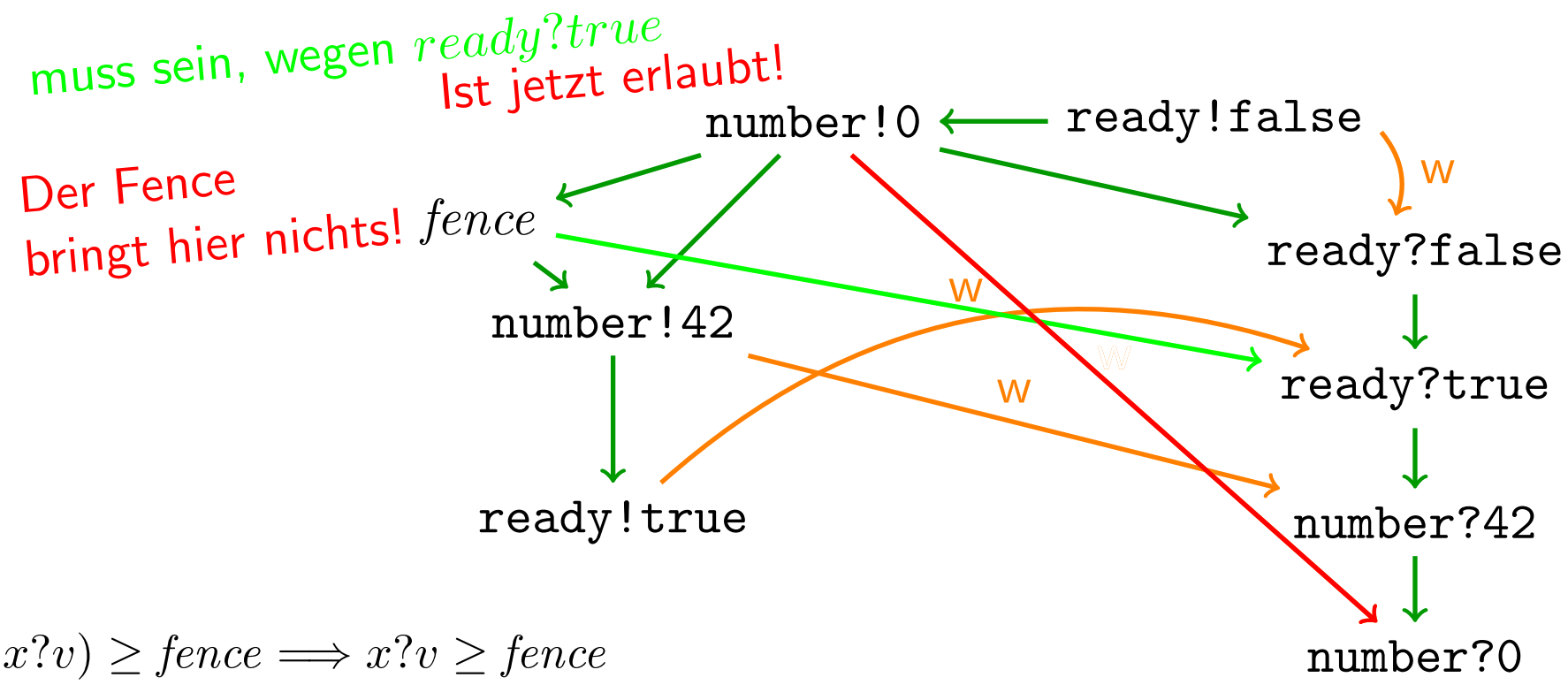
$w(x?v) \geq fence \Longrightarrow x?v \geq fence$

Schwache Konsistenz mit Fences

```
private static boolean ready = false;
private static long number = 0;
```

```
number = 42;
ready = true;
```

```
while (!ready)
Thread.yield();
System.out.println(number);
System.out.println(number);
```



$$w(x?v) \geq fence \Longrightarrow x?v \geq fence$$

Fences

Was haben nun Fences mit Locks zu tun?

Locks verwenden die `fence`-Operation, um sicherzustellen, dass bei Eintritt in den kritischen Abschnitt auch alle im letzten kritischen Abschnitt geschriebenen Variablen auf dem neuesten Stand sind (wie im Beispiel oben mit ready!true ist $fence$ ein Teil des `unlock`-Befehls!)

Locks ergeben allerdings eine noch viel stärkere Einschränkung für die erlaubten Ausführungen.

Happens-Before Konsistenz

Wir erweitern $\mathcal{I}$ um $lock/unlock$-Operationen.

Definition: Happens-Before Konsistenz

Eine mit einem Programm P (mit Lockanweisungen) kompatible Ausführung $(\mathcal{I}, \rightarrow, w)$ ist *happens-before konsistent*, wenn es eine partielle Ordnung $\leq$ auf $\mathcal{I}$ gibt, so dass gilt:

- $\rightarrow \subseteq \leq$
- für alle $x?v \in \mathcal{I}$ gilt:
  - $\neg(w(x?v) \geq x?v)$
  - es gibt kein $x!v' \neq w(x?v)$ so dass $w(x?v) \leq x!v' \leq x?v$

bis hier alles wie gehabt

- für jedes Lock/Unlock Paar $lock(l)_1$, $unlock(l)_1$, $lock(l)_2$, $unlock(l)_2$ über dem selben Lock gilt:

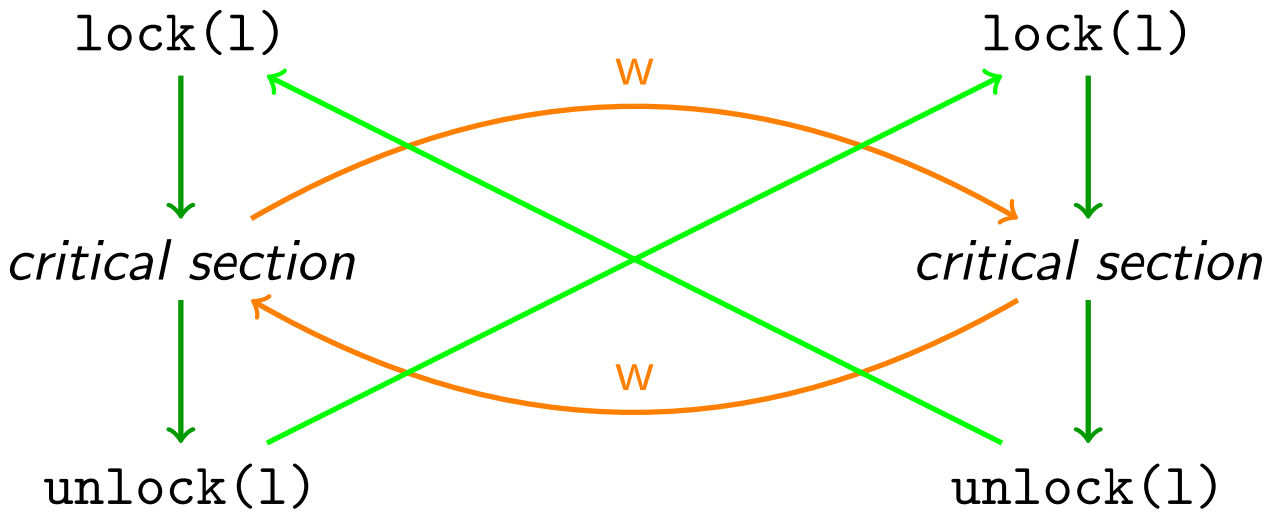
$$\text{entweder } unlock(l)_1 \leq lock(l)_2 \text{ oder } unlock(l)_2 \leq lock(l)_1.$$

Also: erzwinge eine Ordnung zwischen zwei aufeinanderfolgenden Ausführungen des kritischen Abschnitts. $\Rightarrow$ Der nachfolgende Thread hat eine aktualisierte Sicht auf den Speicher!

Intuition Happens-Before-Konsistenz

entweder $unlock(l)_1 \leq lock(l)_2$ oder $unlock(l)_2 \leq lock(l)_1$

Wo muss jetzt
die Ordnung $\geq$ verändert werden?



Zusammenfassung

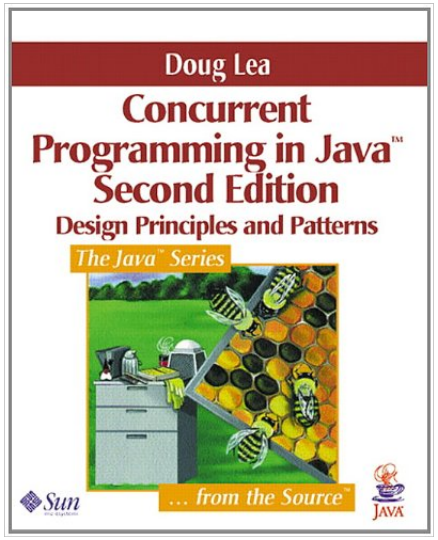
- Was bringt uns das JAVA-Speichermodell?
- Wir lernen: die Semantik des gemeinsamen Speichers in Java ist eine komplizierte Teilmenge von schwacher Konsistenz, die allerhand unintuitives Verhalten zulässt
- Compiler halten sich u.U. nicht an die dokumentierte Semantik, daher ist sie wenig nützlich
- Man sollte gemeinsamen Speicher in Java nicht benutzen, ausser man nutzt Locks, synchronized, etc.

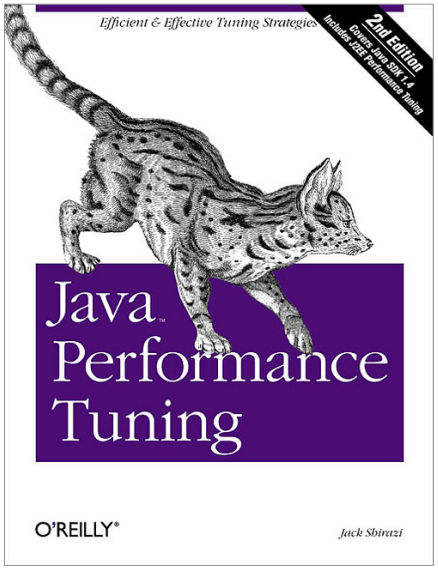
Zusammenfassung

- Das Speichermodell von Java basiert (u.a.) auf der Happens-Before-Konsistenz und versucht diverse Phänomene auszuschliessen.
  Es ist nicht implementierbar (es gibt keinen Compiler, der es implementiert).
  $\Rightarrow$ Schützen Sie Ihren nebenläufigen Code mit Locks, um Happens-Before-Beziehungen zu erhalten!
- Java ist unfair: Es gibt keine Garantie, dass jemals mehr als ein Thread ausgeführt wird.
- Ihre Hardware zerlegt vielleicht double und long. $\Rightarrow$ "very very thin air"
- Spurious wakeups: Wartende Threads können jederzeit aktiviert werden.

$\Rightarrow$ `wait` immer in eine While-Schleife packen!

- Schreiben Sie nur Programme, die keine data-races enthalten.
- Verwenden Sie `volatile` nur, wenn Sie genau wissen, was Sie tun! Also nicht.





- "From the point of view of other threads that might be 'spying' on a thread by concurrently running unsynchronized methods, almost anything can happen."
- "The only useful constraint is that the relative orderings of synchronized methods and blocks, are always preserved."
- "Note however that volatile has been incompletely implemented in most JVMs. Using volatile may not help to achieve the results you desire (yes this is a JVM bug, but its been low priority until recently)."
- http://www.javaperformancetuning.com/tips/volatile.shtml

  Last Updated: 2015-07-01

Programmierprojekt

Regularien:

- Ihre Lösung der Aufgabe fassen Sie bitte mit allen erforderlichen Dokumenten in ein Archiv im `zip`- oder `tgz`-Format zusammen.
- Diese können Sie als Abgabe im *mCMS* spätestens am Montag, *17. August, 23:59 Uhr* einreichen.

Programmierprojekt

Ihre Lösung muss die folgenden Dokumente umfassen:

- Ihre Implementierung, die in Form von kompilierendem *Java-8*- oder alternativ *Go*-Code verfasst ist. Achten Sie unbedingt darauf, dass Ihr Code angemessen kommentiert und insgesamt selbsterklärend ist.
- Ein Dokument (in pdf, nicht handschriftlich, maximal 2 Seiten – Diagramme und Zeichungen ausgenommen), in welchem Sie Ihren Lösungsansatz, insbsondere in Hinblick auf die Nebenläufigkeit des eigentlichen Problems, darlegen.

Programmierprojekt

Ihre Abgabe wird von einem Tutor begutachtet und bewertet. Folgende Bewertungen Ihrer Abgabe sind möglich:

- Ihre Lösung ist bereits korrekt (sowohl funktional als auch in Sinne der Nebenläufigkeit), dann haben Sie das Projekt bestanden und diesen Teil der Klausurzulassung erhalten. Die Mitglieder der besten Gruppen, die das Projekt an dieser Stelle bestanden haben, erhalten einen Bonus von jeweils 0,3 auf die Endnote.
- Ihre Lösung hat noch funktionale und/oder nebenläufige Fehler, zeigt aber, dass Sie sich ausführlich mit der Aufgabenstellung beschäftigt haben. In diesem Fall wird Ihnen die Möglichkeit eingeräumt nachzuarbeiten und die Fehler zu beseitigen. Details werden nach dem 17. August bekanntgegeben.
- Ihre Lösung zeigt nicht, dass Sie sich ausführlich mit der Aufgabenstellung beschäftigt haben. In diesem Fall haben Sie das Projekt nicht bestanden und damit die Klausurzulassung nicht erhalten. Sie können den Kurs "'Nebenläufige Programmierung"' dann nicht mehr bestehen.

Programmierprojekt

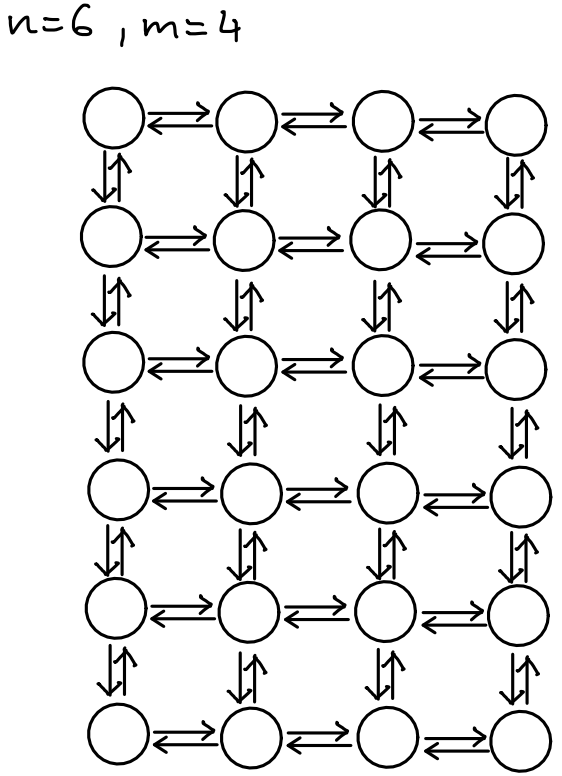
- Desweiteren ist es für alle verpflichtend, bis zum Montag, 20. Juli einen Entwurf (Meilenstein 1) des oben beschriebenen Dokuments einzureichen.
- Dieser Entwurf muss dokumentieren, dass Sie sich hinreichend mit der Aufgabenstellung beschäftigt haben.
- Tut sie das nicht, ist der *Meilenstein 1* nicht bestanden, damit dürfen Sie keine Lösung der Aufgabe (Meilenstein 2, 17. August) mehr abgeben und haben das Projekt und damit den Kurs "'Nebenläufige Programmierung"' nicht bestanden.
- Das gilt natürlich auch, wenn Sie die Abgabe überhaupt nicht machen.
- Zeigt Ihr Entwurf, dass Sie sich hinreichend mit der Aufgabenstellung beschäftigt haben, werden Sie – sofern nötig – an einem anschliessenden individuell zu vereinbarenden Besprechungstermin mit Ihrem Tutor teilnehmen.
- Auch hier gilt: Nehmen Sie diesen Termin nicht wahr, ist der *Meilenstein 1* (und damit NP) nicht bestanden.

Programmierprojekt

- Wir werden bis zum 21. Juli 2015 wie gewohnt Office-Hours abhalten, um Ihre projektrelevanten Fragen zu klären.
- Falls es Fragen organisatorischer Art gibt, oder es andere nicht inhaltliche Probleme gibt, die sich nicht offensichtlich im Forum klären lassen, können Sie auch gerne eine E-Mail an Thilo schreiben.
- Das schlussendliche Bestehen des praktischen Projekts (*Meilenstein 2*) ist notwendige Voraussetzung zum Bestehen der Vorlesung.

Programmierprojekt

Wir betrachten einen Graphen (V, E), der eine 2-dimensionale Gitterstruktur der Grösse $n \times m$ beschreibt.
Jedem Knoten $v \in V$ wird ein Wert zwischen 0 und 1 zugeordnet. Dies kann als ein Schwarz-Weiss-Bild mit $n \times m$ Pixeln gesehen werden, dessen (normierte) Grauwerte bekannt sind.

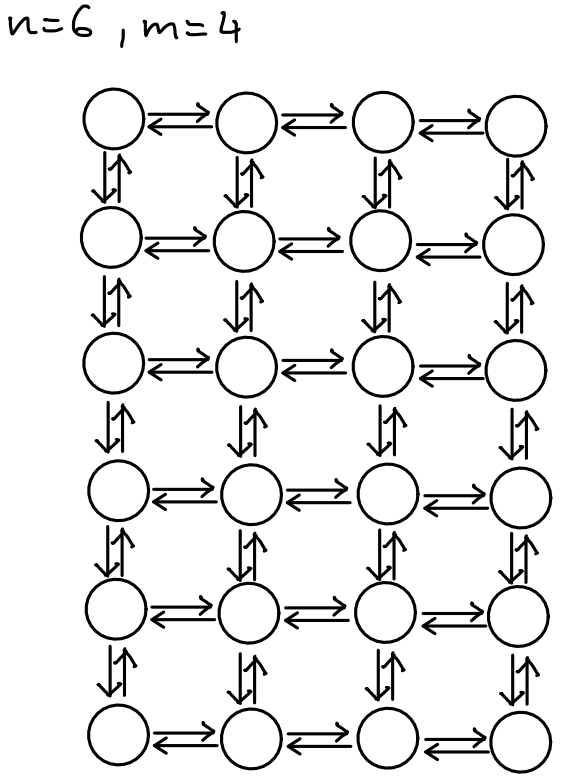


Programmierprojekt

Nun starten wir einen dynamischen Osmoseprozess, der iterativ die (Grau-)Werte verändert. In jedem Iterationsschritt "propagiert" jeder Knoten einen gewissen Anteil seines Wertes an die (direkten) Nachbarknoten.

Jeder Knoten hat 4 ausgehende Transitionen zu seinen 4 direkten Nachbarknoten (links, rechts, oben, unten), sofern er nicht am Rand liegt. Randknoten haben nur Transitionen zu Nachbarknoten.

Video



Osmose-Iteration

- Wir nehmen an, dass jede Kante $(v, w) \in E$ des Graphen beschriftet ist mit einer Übergangsrate $q_{vw} \geq 0$, so dass $\sum_{w, w \neq v} q_{vw} \leq 1$ für alle Knoten v gilt.
- Sei nun $p_v(i)$ der Wert des Kontens v vor der $(i+1)$-ten Iteration (der initiale Wert ist $p_v(0)$).
- Dann ergibt sich $p_v(i+1)$ durch

$$p_v(i+1) = p_v(i) - \sum_{w, w \neq v} p_v(i) \cdot q_{vw} + \sum_{w, w \neq v} p_w(i) \cdot q_{wv},$$

wobei $i \in \{0, 1, \ldots\}$.

Für $i \to \infty$ konvergieren die Werte $p_v(i)$ zu einem Limit, das dem finalen Schwarz-Weiss-Bild entspricht. Inflow = Outflow!

Beschreibung eines Osmoseprozesses

- Um auch sehr grosse Osmoseprozesse beschreiben zu können, verwenden wir zur Spezifikation sogenannte *guarded commands*. Ein *command* beschreibt eine Menge von Kanten im Graphen und hat die Form

  `guard : rate -> update`

- Sei x die Zeile und y die Spalte eines Knotens in der Gitterstruktur, dann ist `guard` ein Boolscher Ausdruck über x und y (z.B. $x > 0 \wedge y > 0$). Falls dieser zu `true` evaluiert, dann gibt es eine Transition von $v = (x, y)$ zum Knoten `update` mit der Übergangsrate `rate`.
- Hierbei ist `update` eine Funktion, die mit (x, y) als Argument einen Nachfolgeknoten $w = (x', y')$ berechnet (z.B. $x' = x + 1, y' = y$).
- Die Übergangsrate `rate` ist ebenfalls eine Funktion die mit (x, y) als Argument eine reelle Zahl $q_{vw} \geq 0$ berechnet (z.B. $c \cdot x \cdot y$, für eine feste positive Zahl c). Achtung: Hier muss sichergestellt sein, dass $\sum_w q_{vw} \leq 1$ ist.
- Wir werden uns bei der initialen Bedingung $p(0)$ auf solche Vektoren beschränken, deren Einträge allen Knoten den Wert Null zuordnen bis auf einen Knoten, genannt Anfangsknoten, v_0, der den Wert 1 hat.

Dynamische Datenstruktur

- Wir nehmen an, dass die Grösse der Gitterstruktur es nicht zulässt, dass die Übergangsraten als Matrix im Speicher zur Verfügung steht. Weiterhin ist es möglich, dass manche Knoten niemals einen positiven Wert bekommen.
- Daher arbeiten wir mit einer dynamischen Datenstruktur für den Graphen, die anfangs nur den Knoten v_0 enthält und in jedem Schritt nur solche Knoten hinzufügt, deren Wert sich von 0 auf eine positive Zahl ändert.
- (Optional können auch Knoten, deren Wert auf 0 sinkt wieder entfernt werden, bzw. kann der Speicherplatz für neu explorierte Knoten verwendet werden.)
- Z.B. ArrayList

Sequentielle Lösung

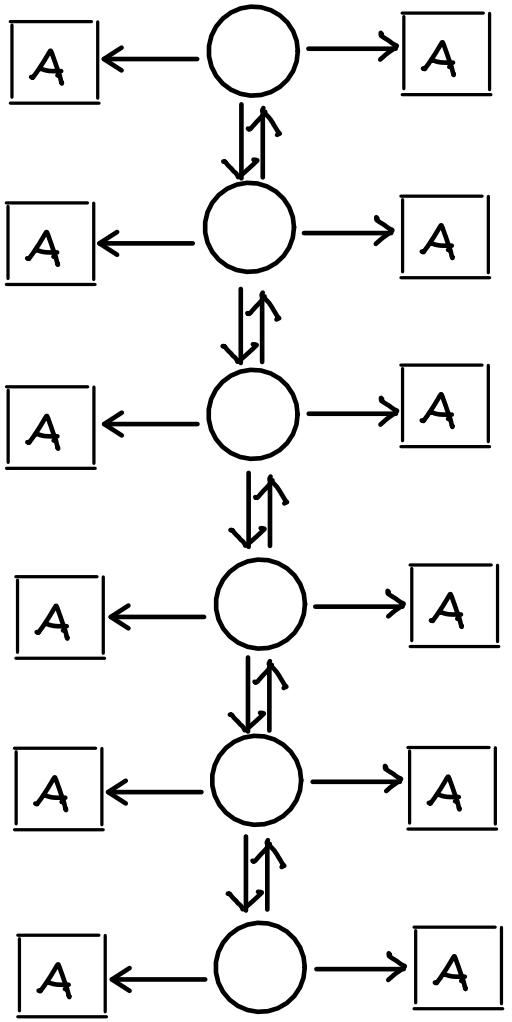
Gegeben: guarded commands, v_0 und ϵ
Gesucht: die Werte $p_v(i)$ für alle $v \in V$ und i.

Wir nehmen Konvergenz an, sobald $||p(i) - p(i+1)||_2 < \epsilon$.

Angenommen, wir definieren eine Datenstruktur `Knoten` mit dem Einträgen `double Wert`, `double Akkumulator` und vier Referenzen auf die möglichen Nachfolgerknoten.

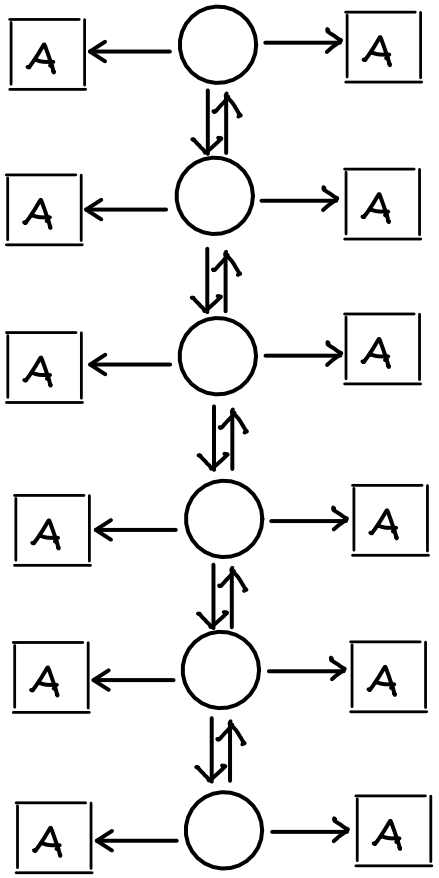
Nebenläufige Lösung

- Das Ziel des Projekts ist es, die obige Berechnung in JAVA oder in GO zu implementieren und zu parallelisieren.
- Achtung: eine gute Approximation reicht aus (tradeoff zwischen Effizienz und Genauigkeit sollte mit Parametern steuerbar sein!)
- Unterteilen Sie die Gitterstruktur in m Spalten und lassen Sie einen Thread nur maximal den flow einer Spalte der Gitterstruktur zur gleichen Zeit bearbeiten.
- Z.B. Sie entwerfen eine Datenstruktur, in der Sie Akkumulatorfelder für jeden horizontalen Nachbarn anlegen.



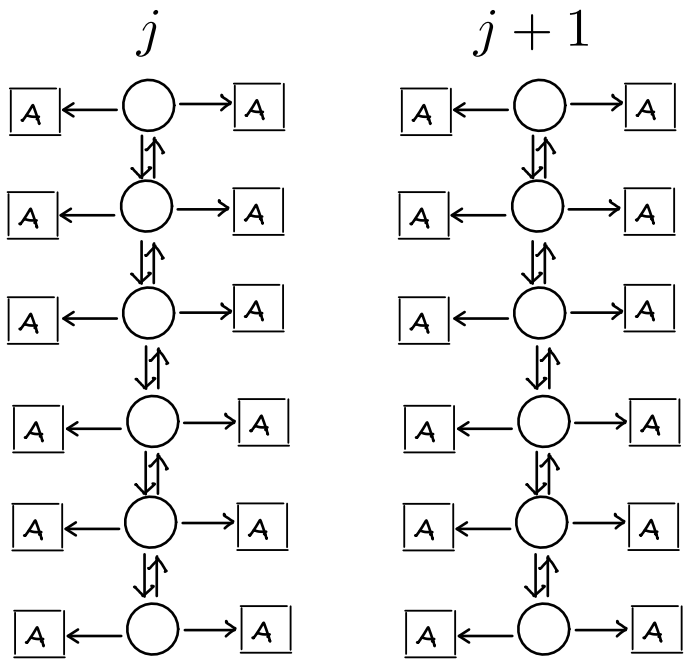
Nebenläufige Lösung

- Lassen Sie die Werte einer Spalte für eine bestimmte Anzahl von internen/lokalen Iterationsschritten propagieren - inklusive dem "Befüllen" der Akkumulatorfelder, die zu horizontalen Nachbarn (also anderen Spalten) gehören.
- Geben Sie erst nach dieser bestimmten Anzahl von lokalen Iterationsschritten den outflow zu den anderen Spalten weiter (z.B. nach 100 Schritten.)
- Dies resultiert in einer Approximation der Lösung, die je nach Dynamik des Prozesses und der gewählten lokalen Anzahl von Iterationen variiert.



Nebenläufige Lösung

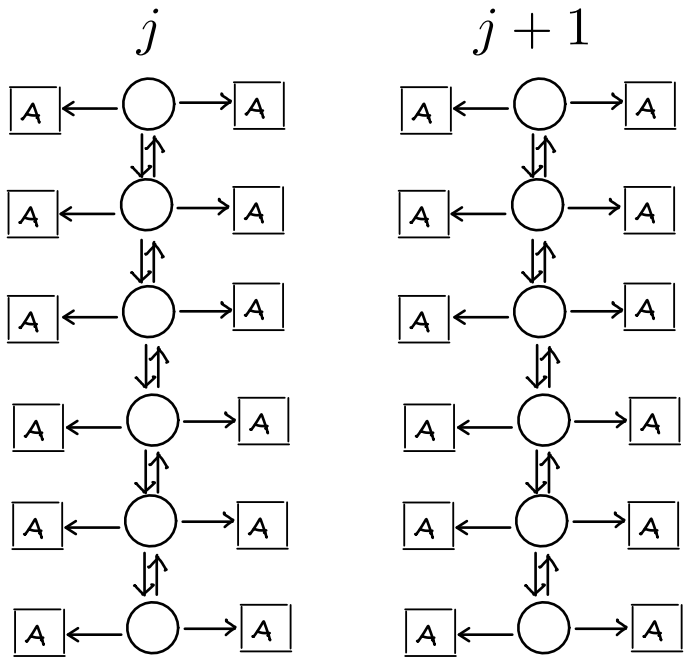
- Angenommen in Spalte j wird während einer Reihe von lokalen Iterationen für einen Knoten v der Nachbarspalte $j+1$ der Anteil x an inflow akkumuliert. Jetzt muss synchronisiert werden:
- Synchronisieren beim Propagieren: Sei i die Gesamtzahl von Iterationen bzgl. Spalte j und k die Gesamtzahl von Iterationen bzgl. Spalte $j+1$. Dann sollte der flow x nur dann zum Wert des Knotens v addiert werden, wenn $i = k$.



Nebenläufige Lösung

Betrachten Sie folgenden Probleme im Zusammenhang mit Ihrer Implementierung:

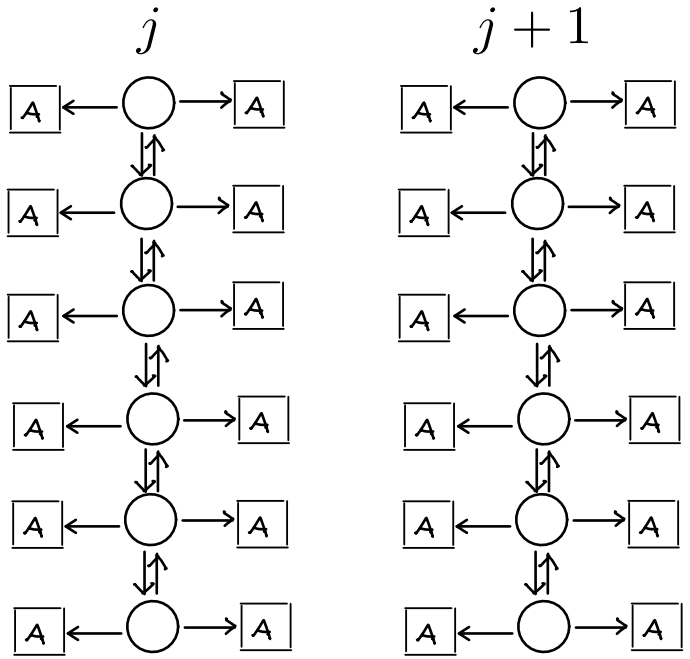
- Propagieren zwischen Spalten: Stellen Sie sicher, dass die Übergabe von Masse, die horizontal propagiert wird und daher von einem Thread an einen anderen übergeben werden muss, korrekt gehandhabt wird.
- Hierbei können Sie sich entscheiden, ob Sie message passing oder gemeinsame Variablen verwenden. Im letzteren Fall sind Data Races oder ähnliche negative shared-memory Phänomene zu vermeiden.



Nebenläufige Lösung

Betrachten Sie folgenden Probleme im Zusammenhang mit Ihrer Implementierung:

- Terminierung und globale Konvergenz:
  Überlegen Sie sich eine Lösung, um globale Konvergenz zu erkennen und die Terminierung aller Threads sicherzustellen.



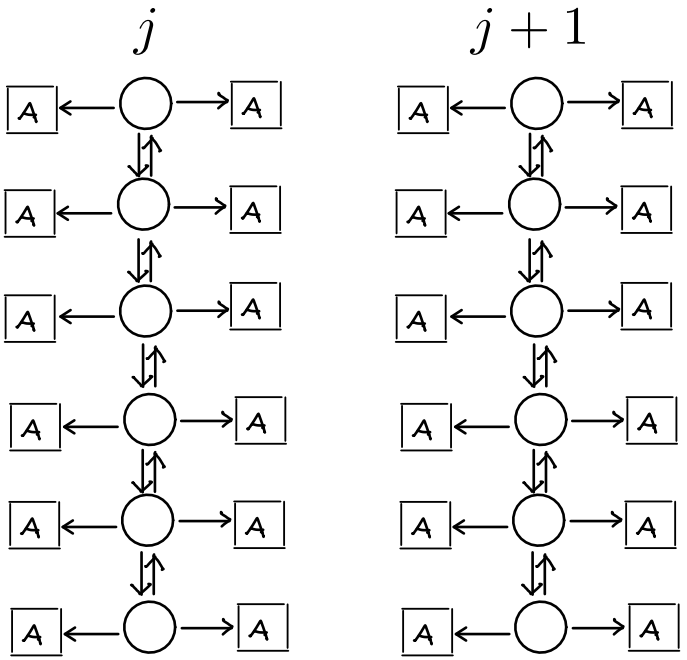
Nebenläufige Lösung

Betrachten Sie folgenden Probleme im Zusammenhang mit Ihrer Implementierung:

- Achtung: Es kann zu divergentem/zyklischen Verhalten kommen, wenn nur selten horizontal propagiert wird. Um dies zu verhindern, ist es sinnvoll, die horizontal propagierten Werte zu überwachen.
- Falls für den gesamten horizontalen "Flow" zwischen allen Spalten gilt

  inflow $\approx$ outflow

  dann sollte z.B. nach jedem (internen) Schritt horizontal propagiert werden und auf globale Konvergenz überprüft werden (hier darf beliebig optimiert werden!).



Nebenläufige Lösung

Wie bekomme ich einen Bonus?

- Angenommen, Ihre Implementierung hat die korrekte Semantik im Hinblick auf die berechneten Ergebnisse und auf mögliche Nebenläufigkeits-Probleme.
- Beeindrucken Sie uns damit, wie sie es gelöst haben:
- Elegant!!
- Effizient (schnell, geringer Speicherbedarf, skaliert)
- Genau (ohne langsam zu sein)

Was wir bereitstellen:

- Parser für die guarded commands + Ausgabe zum Plotten
- Beispiele

Bei Problemen: Forum, Office Hours, Thilo

Hilfsmittel in JAVA

Java stellt verschiedene Monitore zur Verfügung.

Beispiel: Ein Monitor in JAVA für Vektoren:

```
List<String> list = Collections.synchronizedList(new
ArrayList<String>());
```

- Monitormethoden wie `get`, `add`, `size`, etc.
- performanter für primitive Datentypen wie double: `gnu.trove.list.array.TDoubleArrayList`

  `http://trove4j.sourceforge.net/javadocs/gnu/trove/list/array/TDoubleArrayList.html`

  Aber bitte mit Locks!

Hilfsmittel in JAVA

Alternative zu synchronized collections: Concurrent collections

- Erlauben mehrfachen Zugriff, indem Kopien erstellt werden oder indem die Struktur in "nebenläufige Teile" aufgeteilt wird.
- Beispiel: ConcurrentHashMap erlaubt beliebig lesende und eine begrenzte Zahl von schreibenden Threads
- statt Locking der gesamten Struktur: mehrere Locks schützen disjunkte Abschnitte
- Methoden wie `size()` liefern nur approximative Ergebnisse
- Beispiel: CopyOnWriteArrayList
- Kopiert die ArrayList bei jeder Modifikation (z.B. add) statt zu Locken
- Kopieren ist teuer, daher nur sinnvoll, wenn hautsächlich iteriert und gelesen wird.

Hilfsmittel in JAVA

Ein Hinweis zum Thema Client-Side-Locking:

```
class AListHelper <E> {
 public List<E> list = Collections.synchronizedList(new ArrayList<E>());

 public boolean putIfAbsent(E x){
   boolean absent = !list.contains(x);
   if (absent)
     list.add(x);
   return absent;
 }
}
```

Reparatur?

```
class AListHelper <E> {
 public List<E> list = Collections.synchronizedList(new ArrayList<E>());

 public synchronized boolean putIfAbsent(E x){
   boolean absent = !list.contains(x);
   if (absent)
     list.add(x);
   return absent;
 }
}
```


Hilfsmittel in JAVA

Ein Hinweis zum Thema Client-Side-Locking:

```
class AListHelper <E> {
 public List<E> list = Collections.synchronizedList(new ArrayList<E>());

 public boolean putIfAbsent(E x){
   boolean absent = !list.contains(x);
    if (absent)
      list.add(x);
    return absent;
 }
}
```

So ist es korrekt!

```
class AListHelper <E> {
 public List<E> list = Collections.synchronizedList(new ArrayList<E>());

 public boolean putIfAbsent(E x){
   synchronized ( list ) {
     boolean absent = !list.contains(x);
      if (absent)
        list.add(x);
      return absent;
   }
 }
}
```


Hilfsmittel in JAVA

Ein Hinweis zum Thema Hand-Over-Hand-Locking:
Die Block-Struktur der sychronized-Blöcke ist in manchen Situationen nicht zweckmässig.

Beispiel: Ein Thread möchte in einer Collection

- nach einem bestimmten Element suchen (exklusiver Zugriff auf ganze Collection)
- es anschliessend aufwändig bearbeiten (exklusiver Zugriff auf Element)

Statt mit synchronized nun beide Locks zu halten, kann man folgendes machen:

```
collectionLock.lock();
//suche Element
elementLock.lock();
collectionLock.unlock();
//bearbeite Element
elementLock.unlock();
```

Explizite Locks werden kreuzweise angefordert und freigegeben. **Alle** Threads müsse das explizite Lock benutzen.

Hilfsmittel in JAVA

Das `this`-escape-Problem:

```
public class MyClass{
  String name;

  public MyClass(String s)
  {
    if (s==null)
    {
      throw new IllegalArgumentException();
    }
    OtherClass.method(this);
    name= s;
  }

  public getName(){
    return name;
  }
}
```

Wie publiziert ein Thread den Zugriff auf eine gemeinsame Resource?

Niemals im Konstruktor, denn der kann nicht als `synchronized` deklariert werden (da er nur von dem einen erzeugenden Thread ausgeführt wird).
In dem Beispiel können andere Threads das Objekt verändern, obwohl es noch nicht fertig erstellt ist!

Thread Pools

Ziel: organisiere die Ausführung verschiedener "Tasks" (implementieren Runnable)

1. Definiere (verschiedene) Tasks
2. Statt nun Threads mit jeweils einem Task zu starten:

   Wähle eine "Executor" (`Executor executor = ?;`) und übergebe ein Runnable instance (z.B. einen Task) mit executor.execute(task) zur Ausführung
3. Häufigster Executor: `ThreadPoolExecutor`
4. Vorteile: Resourcen/Strukturen (z.B. Threads) können wiederverwendet werden nach Ausführung von Tasks; Systemauslastung kann angepasst werden

```
class Task implements Runnable
{
  private String name;

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

  public String getName() {
    return name;
  }

  public void run()
  {
    try
    {
      System.out.println("My name: " + name);
    }
    catch (InterruptedException e){}
  }
}
```


Thread Pools

`newCachedThreadPool()`

"Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. "

`newSingleThreadExecutor()`

"Creates an Executor that uses a single worker thread operating off an unbounded queue."

Bitte lesen Sie dazu
`http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html`

```
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class BasicThreadPoolExecutorExample {
  public static void main(String[] args)
  {
    //Use when you have a reasonable number of threads
    //or when they have a short duration.
    ThreadPoolExecutor executor = (ThreadPoolExecutor)
        Executors.newCachedThreadPool();
    for (int i = 0; i <= 5; i++){
      Task task = new Task("Task " + i);
      System.out.println("A new task has been
        added : " + task.getName());
      executor.execute(task);
    }
    executor.shutdown();
  }
}
```


Viel Erfolg bei Klausur und Projekt!