Parallele Programmierung
Spezifische Synchronisationsprimitiven

Vorlesung 3

Prof. Dr. Luc Bläser

Letzte Vorlesung - Quiz

Drehkreuz

```
public class Turnstile {
  private boolean open = false;

  public synchronized void pass()
         throws InterruptedException {
     while (!open) { wait(); }
     open = false;
  }

  public synchronized void open() {
     open = true;
     notify(); // or notifyAll() ?
  }
}
```

Reicht ein notify()?
Braucht es die wait()-Schlaufe?

Wann reicht ein Single Notify?

1. Nur eine semantische Bedingung (Uniform Waiters)
   - Bedingung interessiert **jeden** wartenden Thread
2. Bedingung gilt jeweils nur für einen (One-In/One-Out)
   - Nur ein **einziger** wartender Thread kann weitermachen

- Fairness-Problem in Java: Weckt beliebigen Thread
  - Grund für notifyAll() – garantiert aber auch keine Fairness

Monitor: Diskussion

- Vorteil: Sehr mächtiges Konzept
  - Jede Synchronisation realisierbar
  - Objekt-orientiert
- Nachteil: Nicht immer optimal
  - Effizienzprobleme (notifyAll() bei mehreren Conditions)
  - Fairnessprobleme (Signal and Continue, kein FIFO)

Alternative: Weitere spezifische Synchronisationsprimitiven

Inhalt Heute

- Spezifische Synchronisationsprimitiven
  - Semaphor
  - Lock & Condition
  - Read-Write Lock
  - Latch
  - Barriere
  - Exchanger

Lernziele

- Diverse spezifische Synchronisationsprimitiven kennenlernen
- Anwendungsfälle und Einschränkungen dieser verstehen
- Fallspezifisch Synchronisationsprimitiven auswählen und beurteilen können

©wikipedia.org

Semaphor-Konzept

- E.W. Dijkstra, 1965
  - Semaphor (gr. Zeichen): Signalmast

Rekapitulation «Betriebssysteme»: Was macht ein Semaphor?

Funktion eines Semaphors

Vergabe einer beschränkten
Anzahl freier Ressourcen

- Freie Zugstrecke für einen
- Freie Parkplätze für mehrere
- Kuchenstücke für mehrere / einen

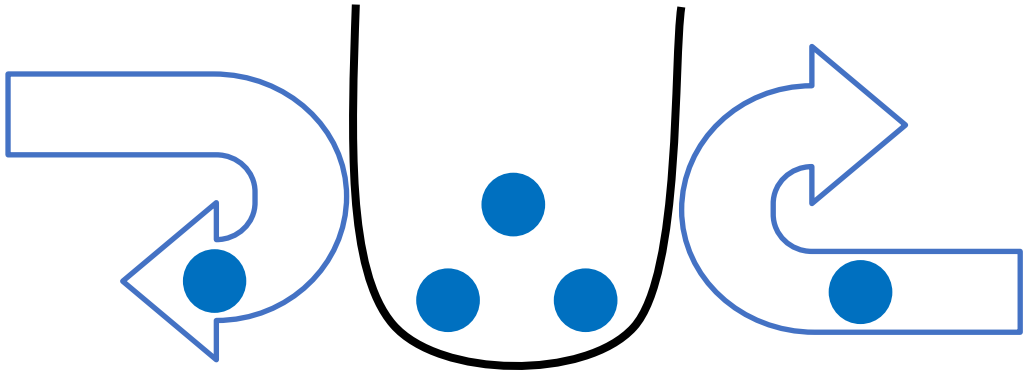
Semaphor

- Objekt mit Zähler
  - Zähler = Anzahl noch freier Ressourcen
- Methode acquire()
  - Beziehe freie Ressource
  - Warten, falls keine verfügbar (Zähler <= 0)
  - Sonst Zähler dekrementieren
- Methode release()
  - Ressource freigeben
  - Zähler inkrementieren

Semaphor Veranschaulichung

```
var s = new Semaphore(3);
```

Initialisierung mit 3 freien Ressourcen



`s.acquire()`

- Bezieht freie Ressource
- Wartet, wenn keine verfügbar

`s.release()`

- Gibt Ressource frei
- Benachrichtigt Wartende

Semaphor: Nachbau mit Monitor

```
public class Semaphore {
  private int value;

  public Semaphore(int initial) {
    value = initial;
  }

  public synchronized void acquire()
    throws InterruptedException {
    while (value <= 0) { wait(); }
    value--;
  }

  public synchronized void release() {
    value++;
    notify();
  }
}
```

Hier nicht garantiert fair

Arten von Semaphoren

- Allgemeine Semaphore (Zähler zwischen 0 bis N)
  - `new Semaphore(N)`
  - Bis zu N Threads können gleichzeitig akquiriert haben
  - Für Quotas, Service Throttling etc.
- Binäre Semaphore (Zähler nur 0 oder 1)
  - `new Semaphore(1)`
  - Für gegenseitigen Ausschluss (1 = offen, 0 = geschlossen)

- In Java kann Zähler auch negativ initialisiert werden
  - Nicht so in anderen Systemen (Windows, .NET)

Faire Semaphore

- `new Semaphore(N, true)`
  - Benutzt FIFO-Warteschlange für Fairness
  - Langsamer als unfaire Variante
- Default ist unfair
  - `new Semaphore(N)`

Semaphore: Anwendung

```
class BoundedBuffer<T> {
  private Queue<T> queue = new LinkedList<>();
  private Semaphore upperLimit = new Semaphore(CAPACITY, true);
  private Semaphore lowerLimit = new Semaphore(0, true);

  public void put(T item) throws InterruptedException {
    upperLimit.acquire();
    synchronized (queue) { queue.add(item); }
    lowerLimit.release();
  }

  public T get() throws InterruptedException {
    T item;
    lowerLimit.acquire();
    synchronized (queue) { item = queue.remove(); }
    upperLimit.release();
    return item;
  }
}
```

Was bewirken die Semaphore?
Wieso braucht es das synchronized?

Nur mit Semaphoren

```
class BoundedBuffer<T> {
  private Queue<T> queue = new LinkedList<>();
  private Semaphore upperLimit = new Semaphore(Capacity, true);
  private Semaphore lowerLimit = new Semaphore(0, true);
  private Semaphore mutex = new Semaphore(1, true);

  public void put(T item) throws InterruptedException {
    upperLimit.acquire();
    mutex.acquire(); queue.add(item); mutex.release();
    lowerLimit.release();
  }

  public T get() throws InterruptedException {
    lowerLimit.acquire();
    mutex.acquire(); T item = queue.remove(); mutex.release();
    upperLimit.release();
    return item;
  }
}
```

Mutual exclusion

Semaphore: In anderen Systemen

- Synonyme Operationen
  - acquire: decrement, wait, P («passeren», «probeer»)
  - release: increment, signal, V («vrijgave», «verhoog»)
- Gibt es auch im Betriebssystem
  - Für Inter-Prozess Synchronisation
- In Java: Intra-Prozess Synchronisation
  - Benutzt intern keine Betriebssystem-Semaphore (zu teuer)
  - Sondern übliche Thread-Synchronisation via Heap

Semaphore: Multi-Acquire/Release

- Mehr als eine Ressource anfordern und freigeben
  - `acquire(int permits)`
    - Wartet, solange Zähler < permits ist
    - Zähler -= permits
  - `release(int permits)`
    - Zähler += permits

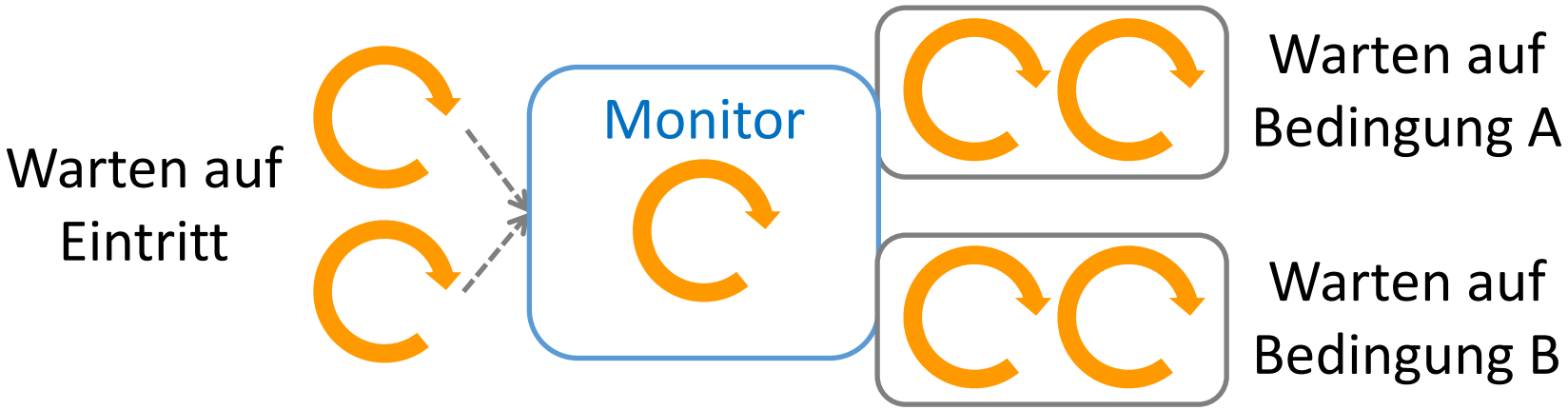
Semaphore: Diskussion

- Sehr mächtig
  - Beliebige Synchronisation implementierbar
- Aber relativ low-level
  - Beim Buffer wollen wir ineffiziente notifyAll() vermeiden
  - Signalisierung spezifischer Bedingungen gewünscht
    - Nicht leer ⇔ lowerLimit > 0
    - Nicht voll ⇔ upperLimit > 0

=> Weitere Synchronisationsprimitiven

Lock & Conditions

Monitor mit mehreren Wartelisten für verschiedene Bedingungen



Lock & Conditons: Hintergrund

- Unabhängig vom eingebauten Java Monitor
  - Kein synchronized, wait(), notify(), notifyAll()
- Spezifische Synchronisationsprimitiven über API
  - Kann benutzt werden, um eigenen Art Monitor zu bauen

Package `java.util.concurrent.locks`

Lock & Conditons Primitiven

- Lock-Objekt: Sperre für Eintritt in Monitor
  - Äussere Warteliste
- Condition-Objekt: Wait & Signal für bestimmte Bedingung
  - Innere Warteliste
- Mehrere Conditions pro Lock (d.h. Monitor) möglich

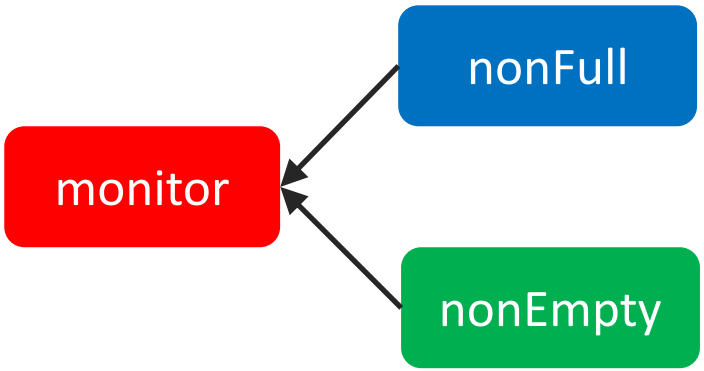
Buffer mit Lock & Conditions (1)

```
class BoundedBuffer<T> {
  private Queue<T> queue = new LinkedList<>();


  private Lock monitor = new ReentrantLock(true);
  private Condition nonFull = monitor.newCondition();
  private Condition nonEmpty = monitor.newCondition();

  …
```

Fairness aktivieren (Default false)



Buffer mit Lock & Conditions (2)

```
public void put(T item) throws InterruptedException {
  monitor.lock();
  try {
    while (queue.size() == Capacity) { nonFull.await(); }
    queue.add(item);
    nonEmpty.signal();
  } finally { monitor.unlock(); }
}

public T get() throws InterruptedException {
  monitor.lock();
  try {
    while (queue.size() == 0) { nonEmpty.await(); }
    T item = queue.remove();
    nonFull.signal();
    return item;
  } finally { monitor.unlock(); }
}
}
```

Schlaufe wegen Überholproblem & Spurious Wakeup

Gezieltes Einzel-Signal für diese Bedingung

Wieso finally?

Read-Write Locks

- Gegenseitiger Ausschluss ist unnötig streng für rein lesende Abschnitte
  - Erlaube parallele Lese-Zugriffe (Reader)
  - Gegenseitiger Ausschluss bei Schreib-Zugriffen (Writer)

Parallel	Read	Write
Read	Ja	Nein
Write	Nein	Nein

Read-Write Lock: Verwendung

Fairer Lock

```
var rwLock = new ReentrantReadWriteLock(true);

rwLock.readLock().lock();
// read-only accesses
rwLock.readLock().unlock();

rwLock.writeLock().lock();
// write (and read) accesses
rwLock.writeLock().unlock();
```

Shared Lock

Exclusive Lock

Falls schreibender Zugriff in Abschnitt involviert => Write Lock

Read-Write Lock: Beispiel

```
class NameDatabase {
  private Collection<String> names = new HashSet<>();
  private ReadWriteLock rwLock = new ReentrantReadWriteLock(true);

  public boolean exists(String pattern) {
    rwLock.readLock().lock();
    try {
      return names.stream().anyMatch(
                name -> name.matches(pattern));
    } finally {
      rwLock.readLock().unlock();
    }
  }
  public void insert(String name) {
    rwLock.writeLock().lock();
    try {
      names.add(name);
    } finally {
      rwLock.writeLock().unlock();
    }
  }
}
```

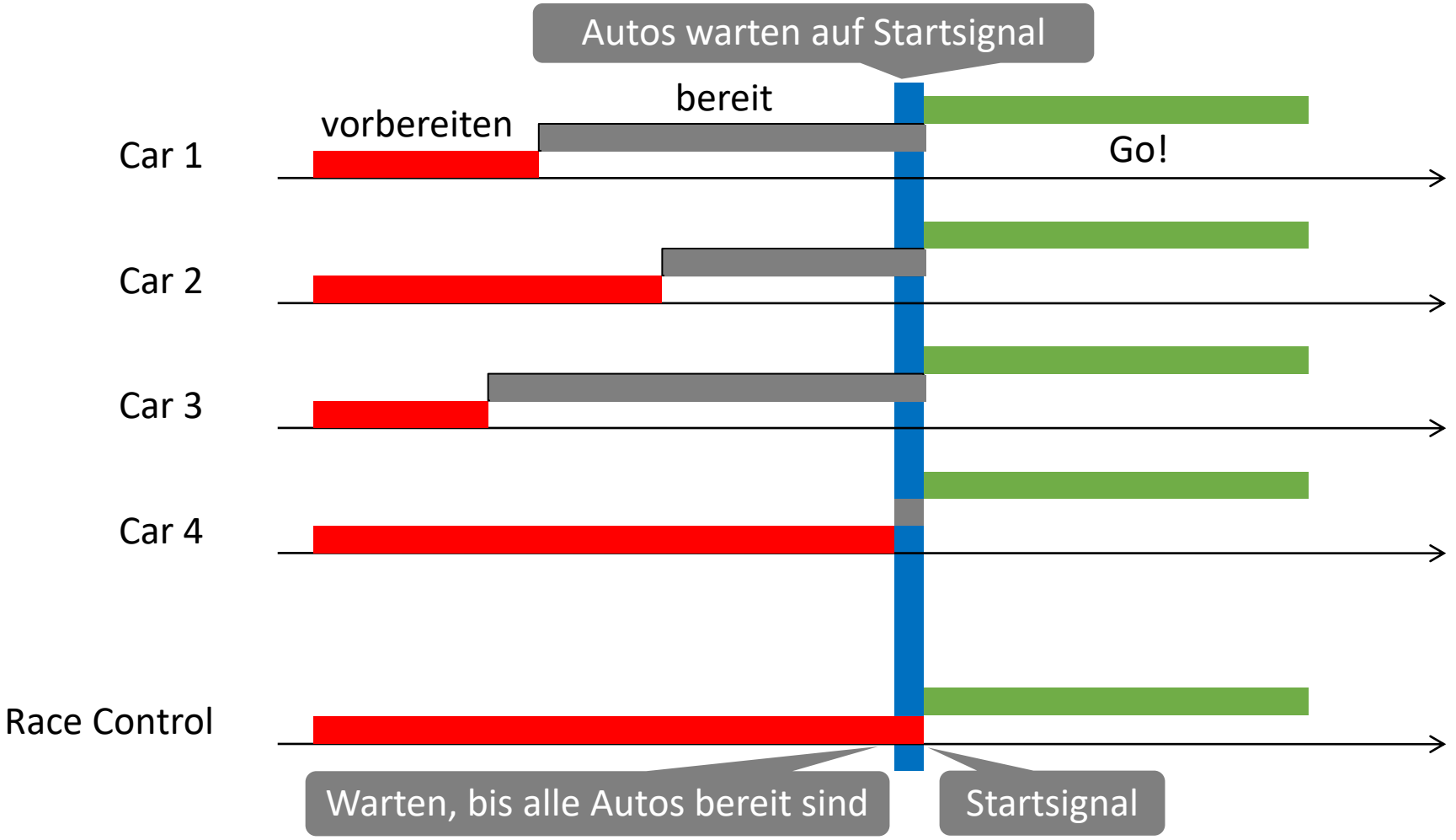
Read-only

Update

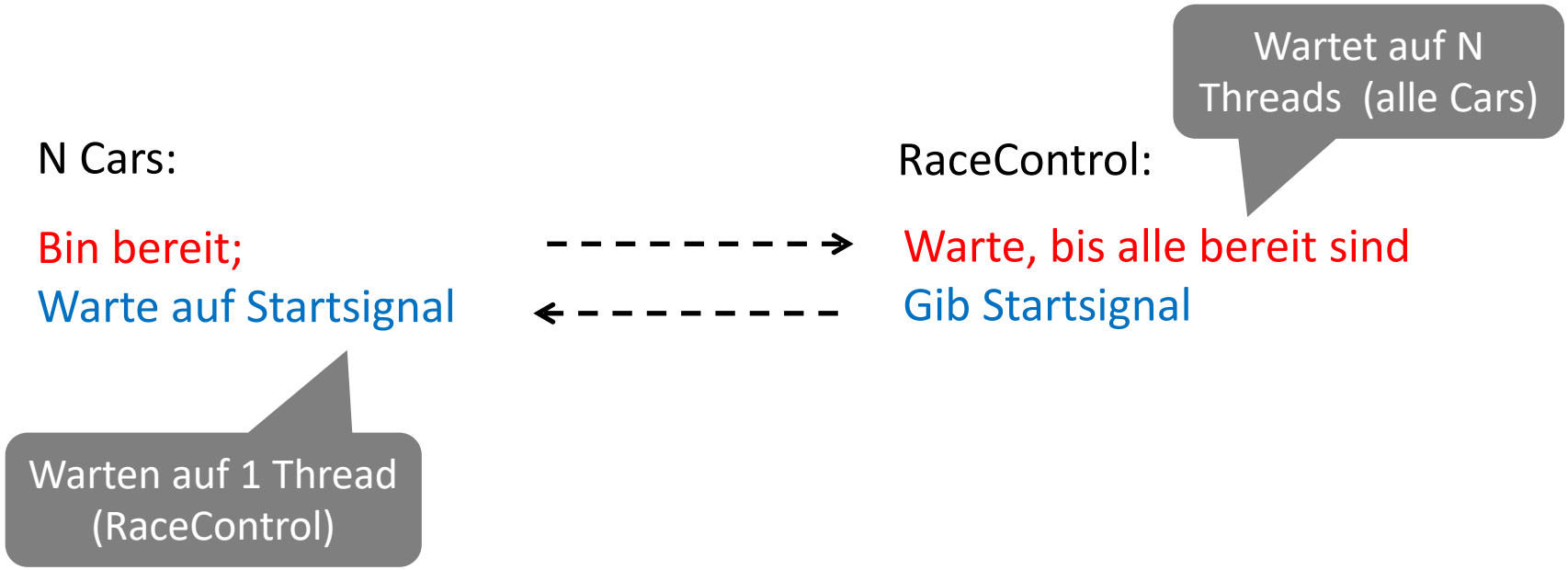
Zwischenstand

- Bisherige Synchronisationsprimitiven
  - Schutz von Shared Ressourcen bei Multi-Threading
  - Monitor, Semaphore, Lock&Condition, Read-Write Lock
- Weitere Synchronisationsprimitiven
  - Zeitlicher Synchronisationspunkt von mehreren Threads
  - CountDownLatch, CyclicBarrier, (Semaphore)

Beispiel Autorennen

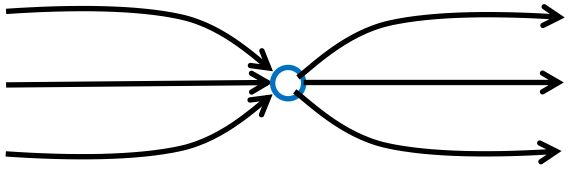


Beispiel Logik



Synchronisationspunkt

- Anzahl Threads warten auf eine Bedingung
  - Nach Erfüllung der Bedingung laufen **alle** weiter



Beispiele:
- Autos warten auf Startsignal
- Player warten alle aufeinander

Count Down Latch

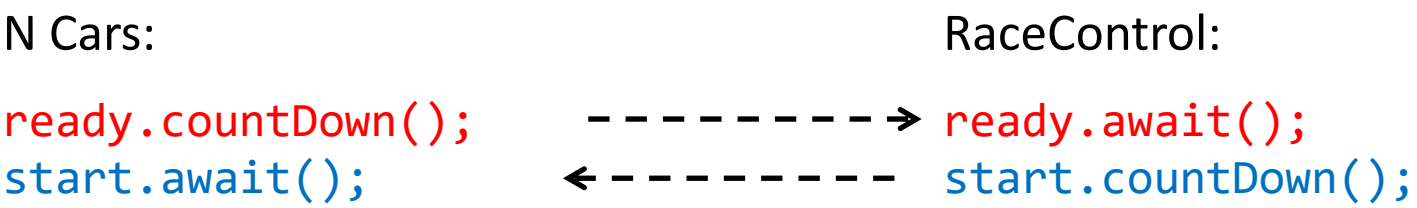
- Synchronisationsprimitive mit Count Down Zähler
- Threads können warten, bis Zähler <= 0 wird
  - await(): Warten, bis dass Count Down 0 ist
- Threads können runterzählen
  - countDown(): Zähler um 1 dekrementieren
- Latches sind nur einmalig verwendbar
  - Kein countUp()

Beispiel mit CountDown Latch

Warte auf N cars

```
var ready = new CountDownLatch(N);
var start = new CountDownLatch(1);
```

Einer gibt Signal



Analoge Lösung mit Semaphoren

```
var ready = new Semaphore(0);
var start = new Semaphore(0);
```

Warte auf N Cars

N Cars:

```
ready.release();
start.acquire();
```

RaceControl:

```
ready.acquire(N);
start.release(N);
```

Erlaube N Cars

Welche Variante ziehen Sie vor?

Cyclic Barrier

- Treffpunkt für fixe Anzahl Threads
  - Threads warten, bis alle angekommen sind
  - Warten am Treffpunkt mit await()
- Anzahl treffender Threads muss vorgegeben sein
  - Fix beim Konstruktor
- Cyclic Barrier ist wiederverwendbar
  - Threads können sich in mehreren Runden bei der gleichen Barriere synchronisieren

Beispiel mit Cyclic Barrier

Anzahl sich treffender Threads

```
var start = new CyclicBarrier(N);
```

N Cars:

```
start.await();
```

Autos fahren direkt los, sobald alle da sind

Brauche kein Race Control mehr

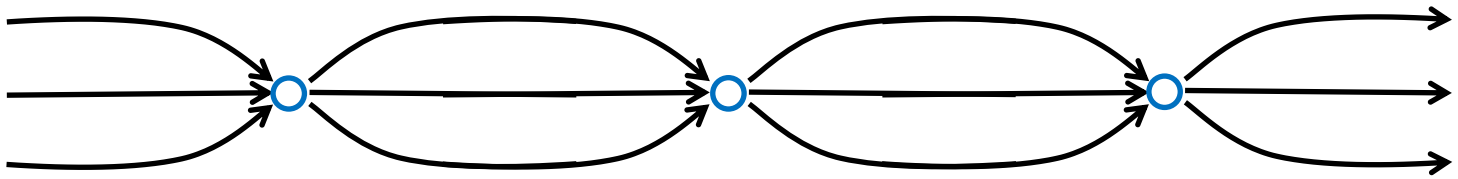
Wiederholte Barriere

```
var gameRound = new CyclicBarrier(N);
```

N Players:

```
while (true) {
  gameRound.await();
  // play concurrently with others
}
```

Barriere schliesst sich automatisch für nächste Runde



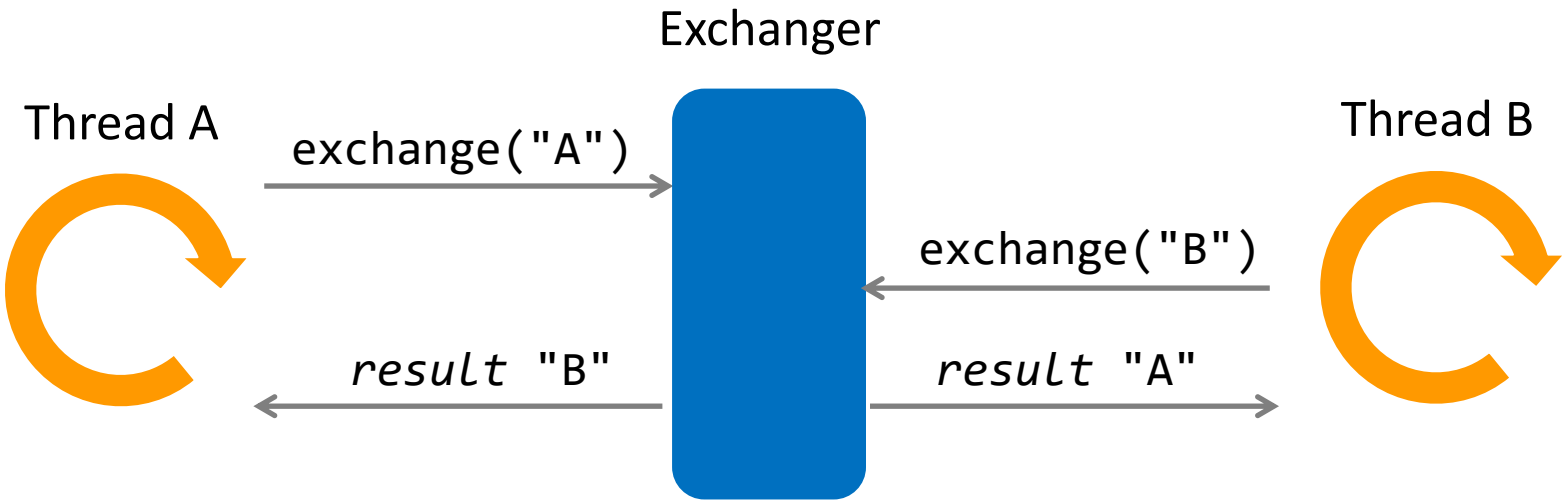
Rendez-Vous

- Barriere mit Informationsaustausch
  - Spezialfall: Nur 2 Parteien
- Zwei Threads treffen sich und tauschen Objekte aus
  - Ohne Austausch: `new CyclicBarrier(2)`
  - Mit Austausch: `Exchanger.exchange(something)`

Exchanger

- `V exchange(V x)`
  - Blockiert, bis anderer Thread auch exchange() aufruft
  - Liefert Argument x des jeweils anderen Threads

Exchanger


Exchanger
Thread A
exchange("A")
exchange("B")
Thread B
result "B"
result "A"

Exchanger Beispiel

```
var exchanger = new Exchanger<Integer>();
for (int count = 0; count < 2; count++) {
  new Thread(() -> {
    for (int in = 0; in < 5; in++) {
      try {
        int out = exchanger.exchange(in);
        System.out.println(
          Thread.currentThread().getName() + " got " + out);
      } catch (InterruptedException e) { }
    }
  }).start();
}
```

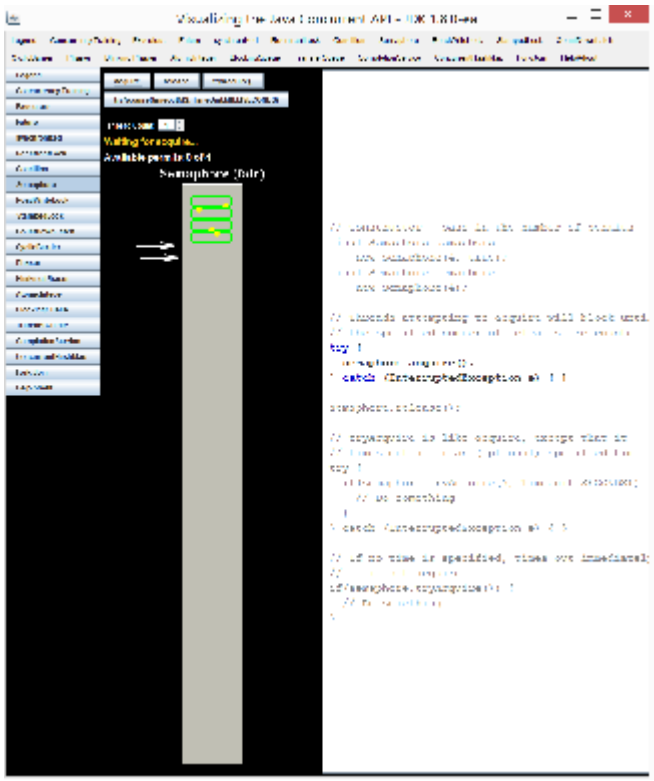
Welche Ausgabe erwarten Sie?

Rückblick: Lernziele

- Diverse spezifische Synchronisationsprimitiven kennenlernen
- Anwendungsfälle und Einschränkungen dieser verstehen
- Fallspezifisch Synchronisationsprimitiven auswählen und beurteilen können

Selbststudium

- Animation der Synchronisationsprimitiven
  - Interaktiv in visueller Simulation
- Skripteserver -> Materialien
  - javaConcurrentAnimated.jar



Weitere Lektüre (fakultativ)

- B. Goetz et al. Java Concurrency in Practice, Addison Wesley, 2006
  - Abschnitte 5.5, 14.3, 14.6
- D. Lea. Concurrent Programming in Java: Design Principles and Patterns 2nd Ed., Addison Wesley, 2000
  - Abschnitte 3.4, 4.4.3