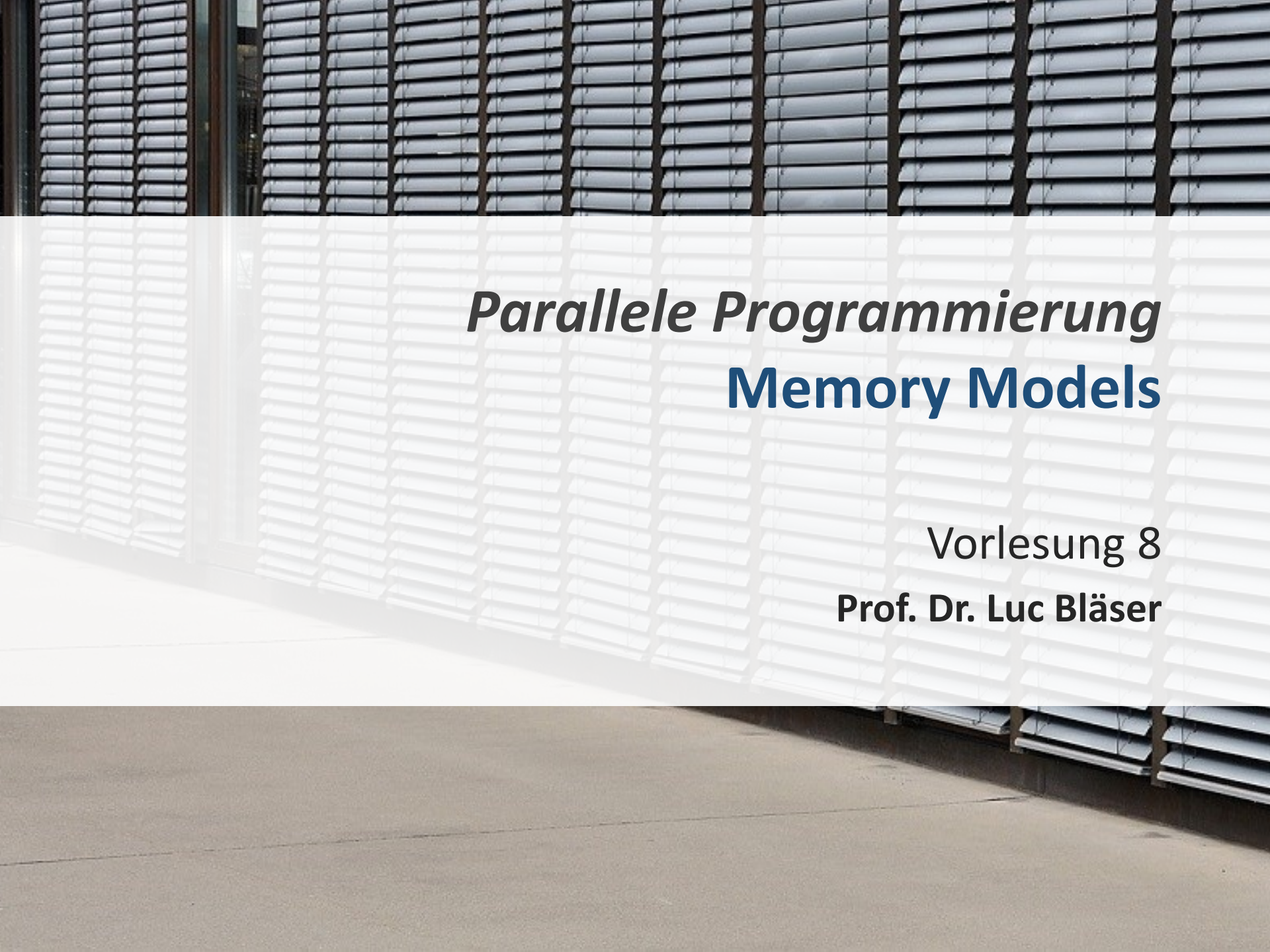


Parallele Programmierung
Memory Models

Vorlesung 8

Prof. Dr. Luc Bläser

Letzte Vorlesung - Quiz

`task.Wait()` versus `await task;`

Was ist der Unterschied?

Inhalt Heute

- Java Memory Model
- Lock-freie Datenstrukturen
- .NET Memory Model

Lernziele

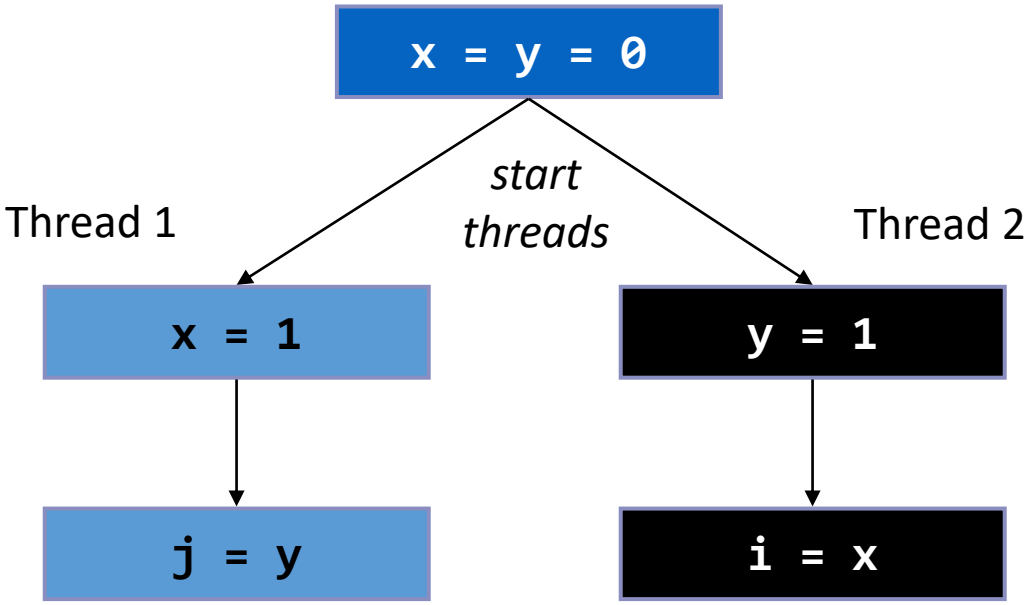
- Java und .NET Memory Model verstehen
- Atomare Instruktionen und volatile-Zugriffe einsetzen können
- Prinzip der lock-freien Datenstrukturen kennen

Lock-Freie Programmierung

- Korrekte nebenläufige Interaktionen ohne Locks
- Garantien des Speichermodells nutzen

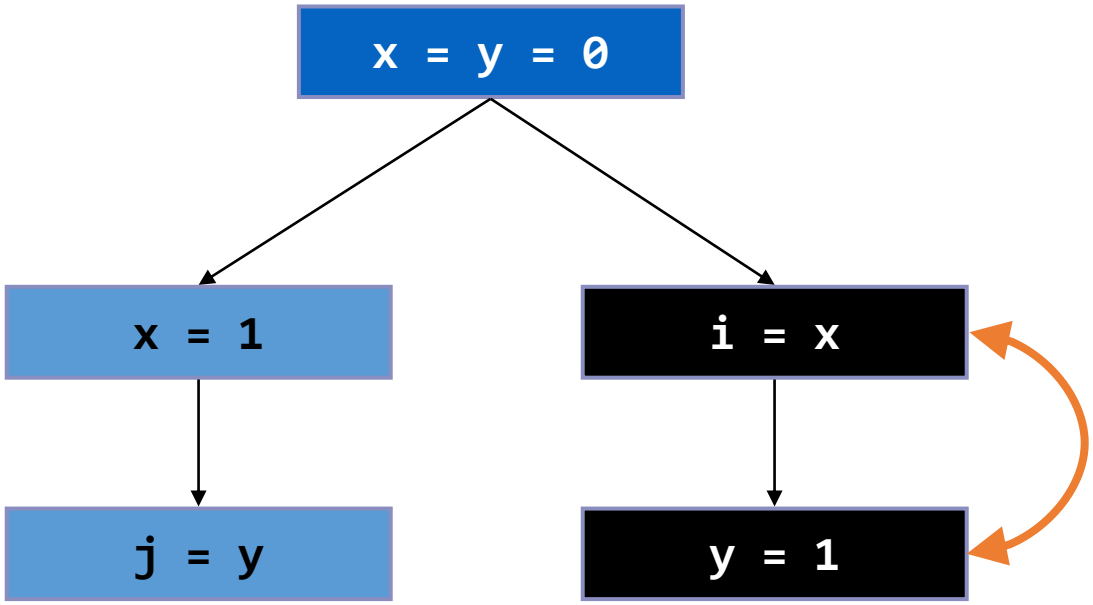
Ziel: Effiziente Synchronisation

Einstiegsbeispiel

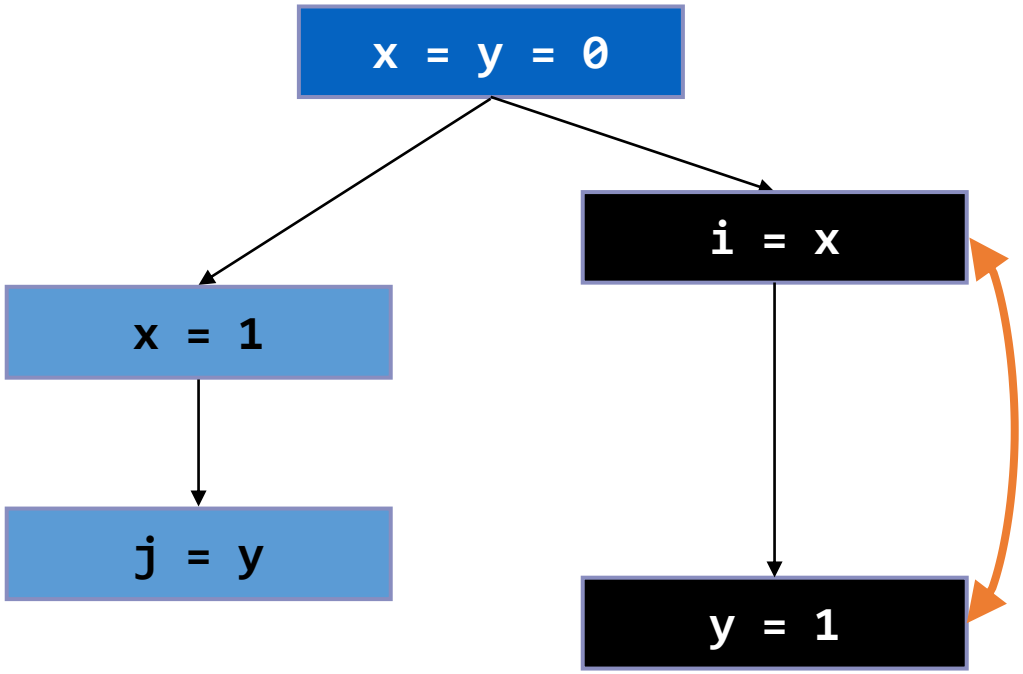


Was sind die möglichen Werte für i und j?

Umordnung



Ausführung



Result: `j = 0, i = 0`

Ursachen für Probleme

- Weak Consistency
  - Speicherzugriffe werden in verschiedenen Reihenfolgen von verschiedenen Threads gesehen
  - Ausnahme: Synchronisationen/Speicherbarrieren
- Optimierungen
  - Compiler, Laufzeitsystem und CPUs
  - Instruktionen werden umgeordnet/wegoptimiert

Keine sequentielle Konsistenz bei Nebenläufigkeit

Java Memory Model

Minimale spezifizierte Garantien

- Atomicity (Unteilbarkeit)
- Visibility (Sichtbarkeit)
- Ordering (Reihenfolge)

Java Atomicity Garantien

Einzelnes Lesen bzw. einzelnes Schreiben ist atomar für

- Primitive Datentypen bis 32 Bit
- Objekt-Referenzen
- long und double nur mit volatile Keyword atomar

Unteilbarkeit ≠ Sichtbarkeit

Unteilbarkeit heisst noch nicht Sichtbarkeit

- Nach Write sieht anderer Thread evtl. noch alten Wert
- Immer gültigen Wert: entweder alten oder neuen

Sichtbarkeit: Analyse

```
class Worker extends Thread {
  private boolean doRun = true;

  @Override
  public void run() {
    while (doRun) {
      …
    }
  }

  public void stopRequest() {
    doRun = false;
  }
}
```

Was ist hier das Problem?

Sichtbarkeit: Problematik

- Sieht Änderungen eines anderen Thread eventuell nicht oder viel später
- Z.B. wegen Compiler-/JIT-Optimierung

```
while (doRun) {
 …
}
```

```
      load doRun
      if_false end
loop:
      …
      goto loop
end:
```

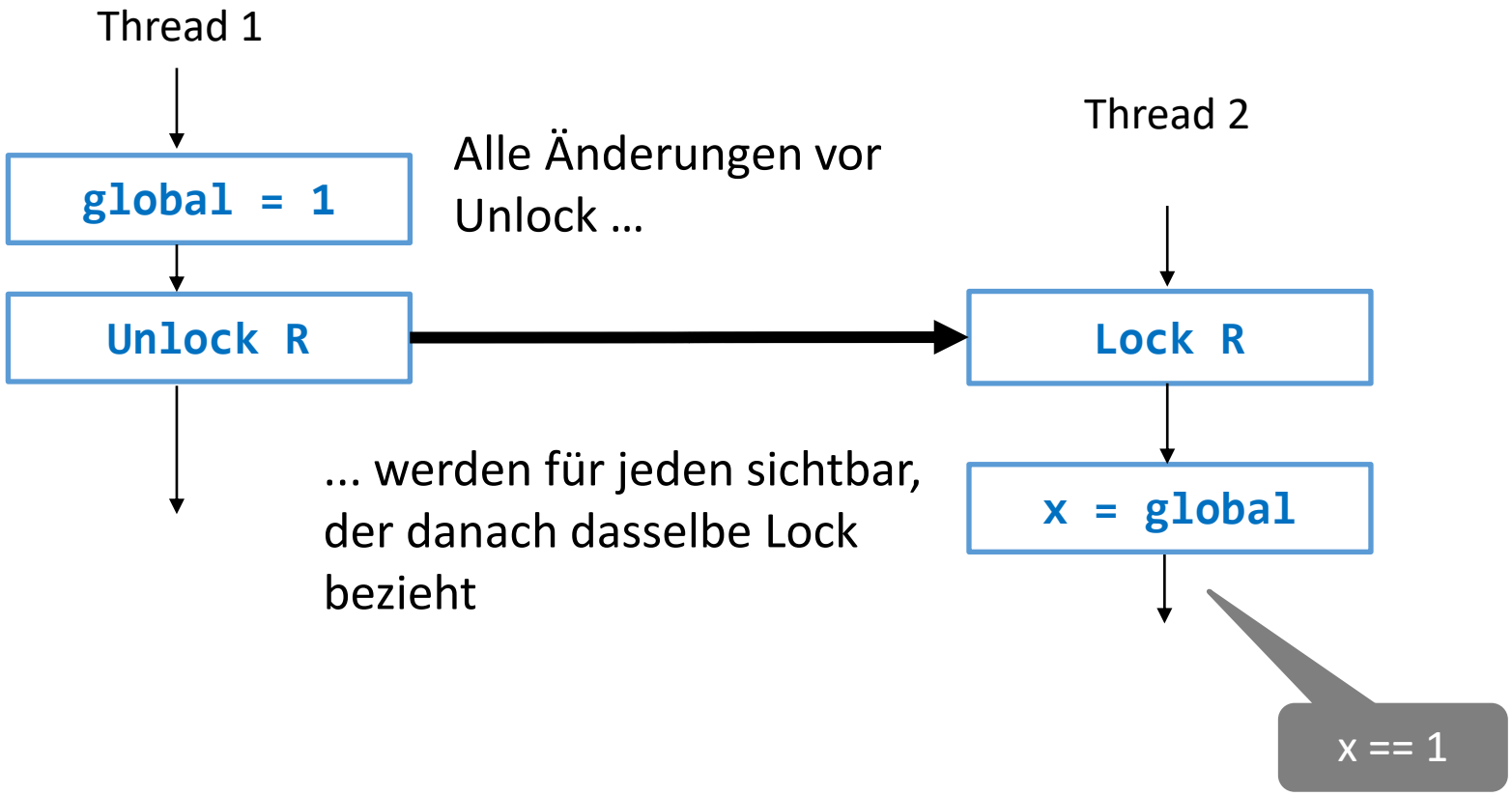
Endlosschlaufe

Java Visibility Garantien

Garantierte Sichtbarkeit zwischen den Threads

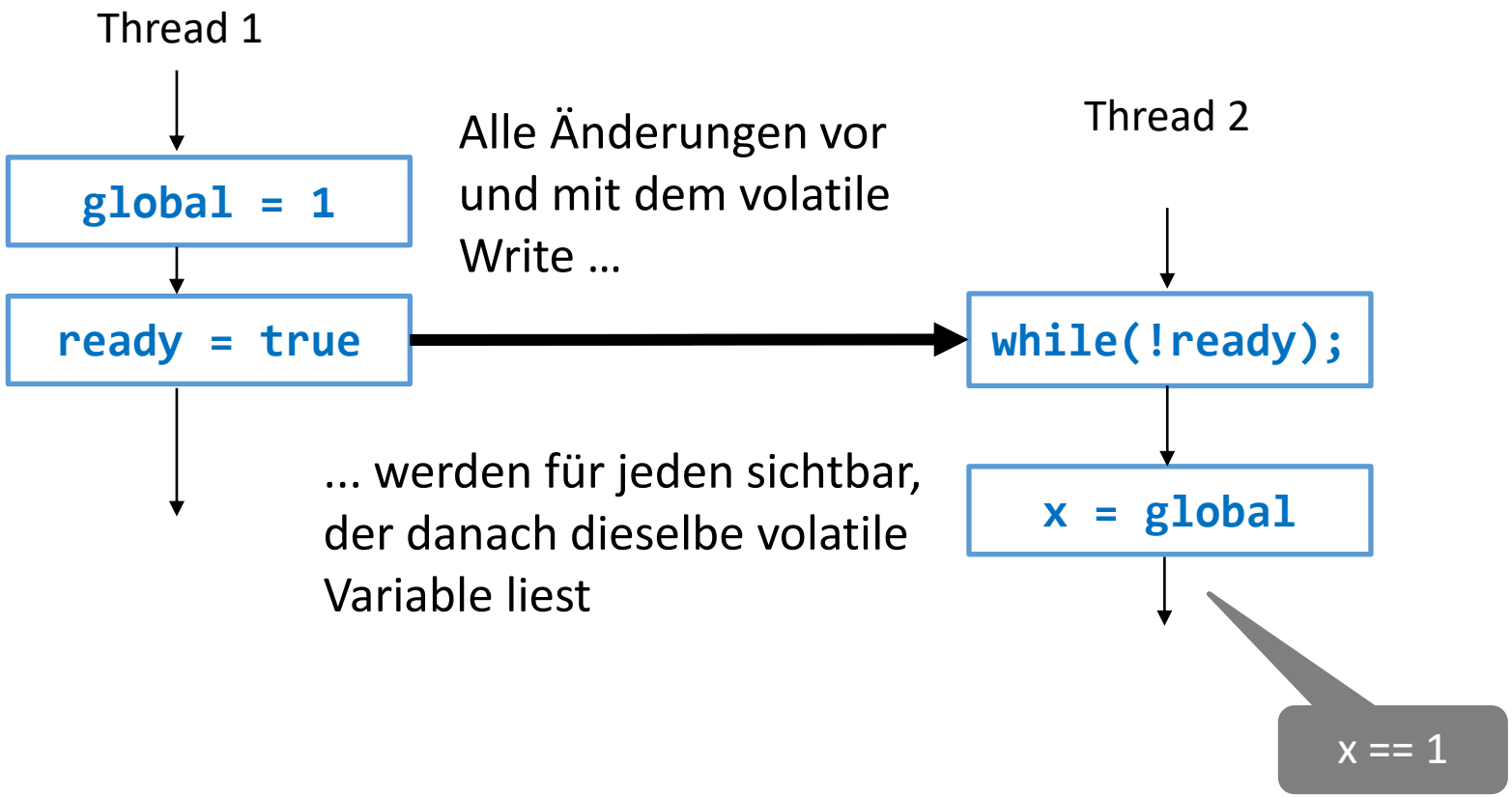
- Locks Release & Acquire
  - Änderungen vor Release werden bei Acquire sichtbar
- Volatile Variable
  - Änderungen bis zum Write werden beim Read sichtbar
- Thread/Task-Start und Join
  - Start: Eingabe an Thread; Join: Ausgabe vom Thread
- Initialisierung von final Variablen
  - Nach Ende des Konstruktors sichtbar

Sichtbarkeit mit Locks



Sichtbarkeit mit volatile

```
int global = 0;
volatile boolean ready = false;
```



Sichtbarkeit: Problem

```
class Worker extends Thread {
  private boolean doRun = true;

  @Override
  public void run() {
    while (doRun) {
      …
    }
  }

  public void stopRequest() {
    doRun = false;
  }
}
```

Wie lässt sich das korrigieren?

Sichtbarkeit – Lösung

```
class Worker extends Thread {
  private volatile boolean doRun = true;

  @Override
  public void run() {
    while (doRun) {
      …
    }
  }

  public void stopRequest() {
    doRun = false;
  }
}
```

Kein Data Race mehr, weil volatile

Java Ordering Garantien

- Program Order ("as-if-serial")
  - Sequentielles Verhalten jedes einzelnen Thread bleibt erhalten (andere Threads dürfen es aber anders sehen)
- Synchronization Order (Total Order)
  - Synchronisationsbefehle werden zueinander nie umgeordnet (Locking, volatile, Thread Start/Join)
- Happens-Before Relation (Partial Order)
  - Alles andere kann umgeordnet werden, ausser es gibt garantierte Sichtbarkeit unter den Threads

Beispiel: Rendez-Vous in Java

```
volatile boolean a = false, b = false;
```

Thread 1

```
a = true;
while (!b) { }
```

Thread 2

```
b = true;
while (!a) { }
```

Funktioniert dies in Java?

Java: Synchronization Order

```
volatile boolean a = false, b = false;
```

Thread 1

```
a = true;
while (!b) { }
```

Thread 2

```
b = true;
while (!a) { }
```

Richtig in Java

Keine Umordnungen in Java, weil alles volatile

Eigener Spin-Lock: Fehlversuch

```
public class SpinLock {
  private volatile boolean locked = false;

  public void acquire() {
    while (locked) {}
    locked = true;
  }

  public void release() {
    locked = false;
  }
}
```

Müsste atomar sein

Nicht korrekt ohne atomares Lesen & Schreiben

Spin-Lock mit atomarer Operation

```
public class SpinLock {
  private AtomicBoolean locked = new AtomicBoolean(false);

  public void acquire() {
    while (locked.getAndSet(true)) { }
  }

  public void release() {
    locked.set(false);
  }
}
```

Initialwert false

Lese alten Wert und setze neuen Wert atomar (Rückgabe = gelesener Wert)

Setze false und mache sichtbar

Atomares Compare and Set

- `boolean compareAndSet (boolean expect, boolean update)`
  - Setzt update, wenn Wert gleich expect ist (atomar)
  - Retourniert true bei erfolgreichem Update

```
if (current == expect) {
  current = update;
  return true;
} else {
  return false;
}
```

atomar

Atomic Klassen Übersicht

- Klassen für Boolean, Integer, Long und Referenzen
  - Auch für Array-Elemente
- Diverse atomare Operationen
  - `addAndGet()`, `getAndAdd()` etc.
  - Auch mit Lambda

```
AtomicInteger squares = new AtomicInteger(0);

squares.updateAndGet(x -> x * x);
```

Optimistische Synchronisation

Lese aktuellen Wert

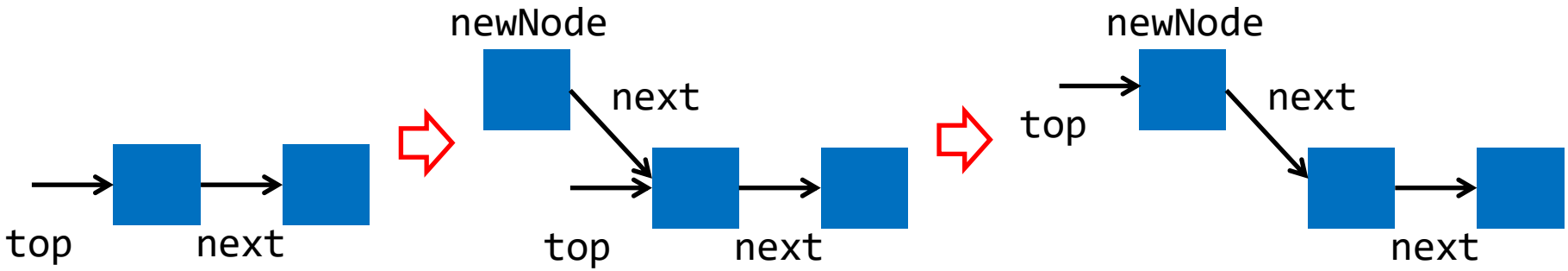
```
do {
  oldValue = var.get();
  newValue = calculateChanges(oldValue);
} while (!var.compareAndSet(oldValue, newValue));
```

Schreibe, falls gelesener Wert immer noch aktuell ist

Welches Problem sehen Sie hier?

Lock-Free Stack (Treiber 1986)

```
AtomicReference<Node<T>> top = new AtomicReference<>();
…
void push(T value) {
  Node<T> newNode = new Node<>(value);
  Node<T> current;
  do {
    current = top.get();
    newNode.setNext(current);
  } while (!top.compareAndSet(current, newNode));
}
```



Vorgefertigte Lock-freie Datenstrukturen

- `ConcurrentLinkedQueue<V>`
- `ConcurrentLinkedDeque<V>`
- `ConcurrentSkipListSet<V>`
- `ConcurrentHashMap<K, V>`
- `ConcurrentSkipListMap<K, V>`

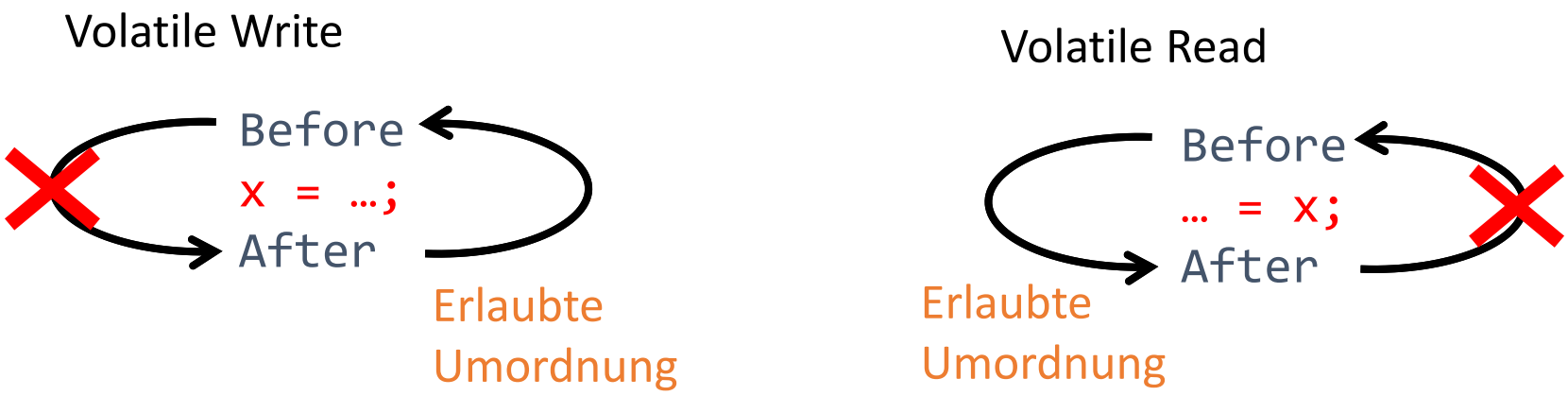
.NET Memory Model

- Unterschied zu Java Memory Model
  - Atomicity: long/double nicht mit volatile atomar
  - Visibility: Nicht definiert, implizit durch Ordering
  - Ordering: nur Half und Full Fences
- Atomare Instruktionen
  - `Interlocked` Klasse

.NET Volatile Half Fences

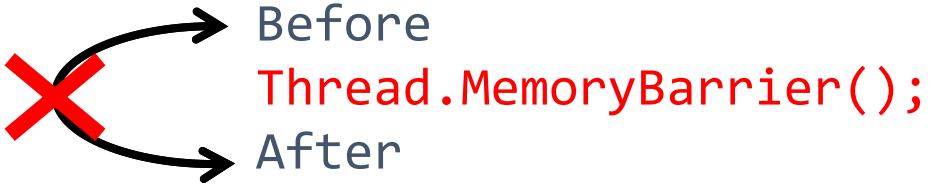
- Volatile Write: Release Semantik
  - Vorangehende Zugriffe bleiben davor
- Volatile Read: Acquire Semantik
  - Nachfolgende Zugriffe bleiben danach

```
volatile int x;
```



.NET Memory Barrier = Full Fence

- Verbietet Umordnung in beide Richtungen



Rendez-Vous in .NET

```
volatile bool a = false, b = false;
```

Thread 1

```
a = true;
while (!b) { }
```

Thread 2

```
b = true;
while (!a) { }
```

Ist dies garantiert korrekt in .NET?

.NET: Ungenügende Memory Fences

```
volatile bool a = false, b = false;
```

Thread 1

```
a = true;
while (!b) { }
```

Umordnung möglich

Thread 2

```
b = true;
while (!a) { }
```

Umordnung möglich

Inkorrekt

.NET: Reparatur mit Full Fence

```
volatile bool a = false, b = false;
```

Thread 1

```
a = true;
Thread.MemoryBarrier();
while (!b) { }
```

Thread 2

```
b = true;
Thread.MemoryBarrier();
while (!a) { }
```

Richtig in .NET

Rückblick: Lernziele

- Java und .NET Memory Model verstehen
- Atomare Instruktionen und volatile-Zugriffe einsetzen können
- Prinzip der lock-freien Datenstrukturen kennen

Literatur (Selbststudium)

Weitergehende Literatur (1)

- Java Language Specification, SE 16, 2021
  - Section 17.4: Java Memory Model
- C# Language Specification, Draft Version 6, 2017
  - Sections “atomicity of variable references”, “volatile fields”
- ECMA 335 CLI Specification, Version 6, 2017
  - I 12.6 Memory Model
- J. Manson, W. Pugh, S. V. Adve. The Java Memory Model, POPL, 2005, ACM Library.
  - Definition des Java 5 Memory Models (immer noch aktuell)
  - Hintergrund und Theorie

Weitergehende Literatur (2)

- B. Goetz et al. Java Concurrency in Practice, Addison Wesley, 2006
  - Kapitel 15 (Atomic & Non-Blocking)
  - Kapitel 16 (Java Memory Model)
- D. Lea. Synchronization and the Java Memory Model, 1996-1999
  - http://gee.cs.oswego.edu/dl/cpj/jmm.html

Weitergehende Literatur (3)

- Berühmte Non-Blocking Lock-Free Data Structures
  - Stack
    - R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.
  - Queue
    - M. M. Michael and M. L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing* (PODC '96). ACM, New York, NY, USA