Speichermodelle

Kenan Gündogan

Friedrich-Alexander-Universität Erlangen-Nürnberg

10.01.2017

Motivation

➔ Dekker-Algorithmus

```
1  static int a = 0,  b = 0;
2  static int r1 = 0, r2 = 0;
3
4  static void *thread1(void *param) {
5     a  = 2017;
6     r1 = b;
7     return NULL;
8  }
9
10 static void *thread2(void *param) {
11    b  = 2017;
12    r2 = a;
13    return NULL;
14 }
```

Ist das Resultat **r1 = r2 = 0** möglich?

Motivation

→ Dekker-Algorithmus

```
1  static int a = 0,  b = 0;
2  static int r1 = 0, r2 = 0;
3
4  static void *thread1(void *param) {
5     a  = 2017;
6     r1 = b;
7     return NULL;
8  }
9
10 static void *thread2(void *param) {
11    b  = 2017;
12    r2 = a;
13    return NULL;
14 }
```

Ist das Resultat **`r1 = r2 = 0`** möglich?

Ja!

Motivation

➔ Rekked-Algorithmus

```
1  static int a = 0,  b = 0;
2  static int r1 = 0, r2 = 0;
3
4  static void *thread1(void *param) {
5     r1 = b;
6     a  = 2017;
7     return NULL;
8  }
9
10 static void *thread2(void *param) {
11    r2 = a;
12    b  = 2017;
13    return NULL;
14 }
```

Ist das Resultat **`r1 = r2 = 2017`** möglich?

Motivation

➔ Rekked-Algorithmus

```
1  static int a = 0,  b = 0;
2  static int r1 = 0, r2 = 0;
3
4  static void *thread1(void *param) {
5     r1 = b;
6     a  = 2017;
7     return NULL;
8  }
9
10 static void *thread2(void *param) {
11    r2 = a;
12    b  = 2017;
13    return NULL;
14 }
```

Ist das Resultat **`r1 = r2 = 2017`** möglich?

x86, IBM Mainframe: ***Nein!***

ARM, PowerPC: ***Ja!***

➔ Architekturabhängig

Gliederung

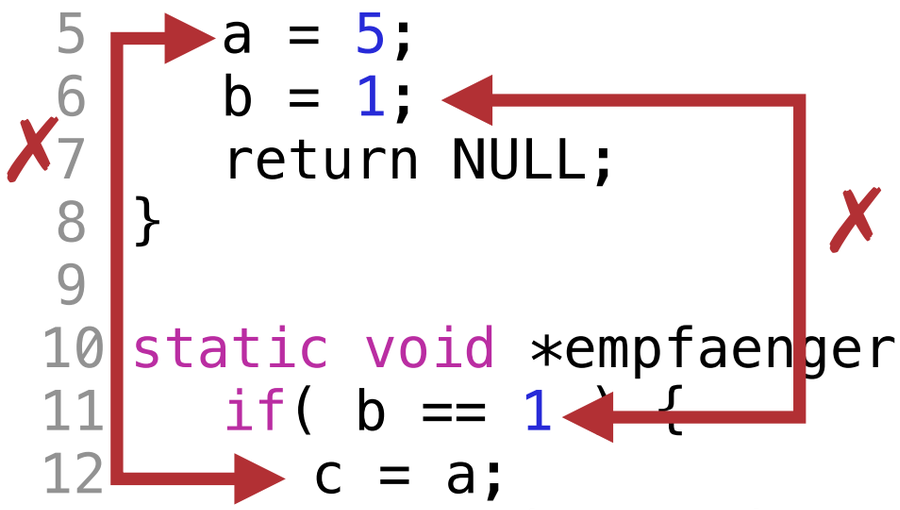
1. Motivation ✓
2. **Problem**: Manipulierung der Instruktionsreihenfolge
3. **Lösung**: Speichermodelle
    1. *Relaxed Consistency*
    2. *Release-Acquire Consistency*
    3. *Sequentielle Konsistenz (Sequential Consistency)*
4. Technische Realisierung
5. Zusammenfassung

Einschub: Datenwettlauf *(Data Race)*

```
1  static int a = 0, b = 0;
2  static int c = -1;
3
4  static void *sender(void *param) {
5     a = 5;
6     b = 1; //Signalvariable
7     return NULL;
8  }
9
10 static void *empfaenger(void *param) {
11    if( b == 1 ) {
12       c = a;
13       assert(c == 5);
14    }
15    return NULL;
16 }
```

Einschub: Datenwettlauf *(Data Race)*

```
1  static int a = 0, b = 0;
2  static int c = -1;
3
4  static void *sender(void *param) {
5     a = 5;
6     b = 1;
7     return NULL;
8  }
9
10 static void *empfaenger(void *param) {
11    if( b == 1 ) {
12       c = a;
13       assert(c == 5);
14    }
15    return NULL;
16 }
```



✗ Datenwettlauf - Definition:

1. Gleichzeitiger Zugriff mehrerer Aktivitätsfäden auf den selben gemeinsam genutzten Speicherbereich, wobei min. einer der Fäden schreibend zugreift
2. Die Fäden verwenden keinerlei Mechanismen zur Serialisierung der Zugriffe

Manipulierung der Instruktionsreihenfolge

Optimierungstechniken - Ausschnitt:

- *Out-of-Order Execution*
- Spekulative Ausführung von Bedingungen
- Lokale Puffer (*Caches*)
- *Register Promotion*

Manipulierung der Instruktionsreihenfolge

Optimierungstechniken - Ausschnitt:

- *Out-of-Order Execution*
- Spekulative Ausführung von Bedingungen
- Lokale Puffer (*Caches*)
- *Register Promotion*

➔ Hardware/Kompilierer können die Reihenfolge der Instruktionen verändern!

Manipulierung der Instruktionsreihenfolge

```
1  static atomic_int a = 0, b = 0;
2  static int        c = -1;
3
4  static void *sender(void *param) {
5     atomic_store_explicit(&a, 5, memory_order_relaxed);
6
7     atomic_store_explicit(&b, 1, memory_order_relaxed);//Signalvariable
8     return NULL;
9  }
10
11 static void *empfaenger(void *param) {
12    if( atomic_load_explicit(&b, memory_order_relaxed) == 1 ) {
13       c = atomic_load_explicit(&a, memory_order_relaxed);
14       assert(c == 5);
15    }
16    return NULL;
17 }
```

Kann die Zusicherung **c == 5** scheitern?

Manipulierung der Instruktionsreihenfolge

```
1  static atomic_int a = 0, b = 0;
2  static int        c = -1;
3
4  static void *sender(void *param) {
5     atomic_store_explicit(&a, 5, memory_order_relaxed);
6
7     atomic_store_explicit(&b, 1, memory_order_relaxed); //Signalvariable
8     return NULL;
9  }
10
11 static void *empfaenger(void *param) {
12    if( atomic_load_explicit(&b, memory_order_relaxed) == 1 ) {
13       c = atomic_load_explicit(&a, memory_order_relaxed);
14       assert(c == 5);
15    }
16    return NULL;
17 }
```

Fall 1	
	if(b==1)
	return
a = 5	
b = 1	

Manipulierung der Instruktionsreihenfolge

```
1  static atomic_int a = 0, b = 0;
2  static int        c = -1;
3
4  static void *sender(void *param) {
5     atomic_store_explicit(&a, 5, memory_order_relaxed);
6
7     atomic_store_explicit(&b, 1, memory_order_relaxed); //Signalvariable
8     return NULL;
9  }
10
11 static void *empfaenger(void *param) {
12    if( atomic_load_explicit(&b, memory_order_relaxed) == 1 ) {
13       c = atomic_load_explicit(&a, memory_order_relaxed);
14       assert(c == 5);
15    }
16    return NULL;
17 }
```

Fall 1	
	if(b==1)
	return
a = 5	
b = 1	

Fall 2	
a = 5	
b = 1	
	if(b==1)
	c = a

Manipulierung der Instruktionsreihenfolge

```
1  static atomic_int a = 0, b = 0;
2  static int        c = -1;
3
4  static void *sender(void *param) {
5     atomic_store_explicit(&a, 5, memory_order_relaxed);
6
7     atomic_store_explicit(&b, 1, memory_order_relaxed); //Signalvariable
8     return NULL;
9  }
10
11 static void *empfaenger(void *param) {
12    if( atomic_load_explicit(&b, memory_order_relaxed) == 1 ) {
13       c = atomic_load_explicit(&a, memory_order_relaxed);
14       assert(c == 5);
15    }
16    return NULL;
17 }
```

Fall 1	
	if(b==1)
	return
a = 5	
b = 1	

Fall 2	
a = 5	
b = 1	
	if(b==1)
	c = a

Fall 3	
b = 1	
a = 5	
	if(b==1)
	c = a

Manipulierung der Instruktionsreihenfolge

```
1  static atomic_int a = 0, b = 0;
2  static int        c = -1;
3
4  static void *sender(void *param) {
5     atomic_store_explicit(&a, 5, memory_order_relaxed);
6
7     atomic_store_explicit(&b, 1, memory_order_relaxed); //Signalvariable
8     return NULL;
9  }
10
11 static void *empfaenger(void *param) {
12    if( atomic_load_explicit(&b, memory_order_relaxed) == 1 ) {
13       c = atomic_load_explicit(&a, memory_order_relaxed);
14       assert(c == 5);
15    }
16    return NULL;
17 }
```

Fall 1	
	if(b==1)
	return
a = 5	
b = 1	

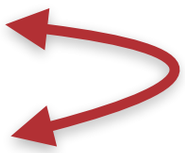
Fall 2	
a = 5	
b = 1	
	if(b==1)
	c = a

Fall 3	
b = 1	
a = 5	
	if(b==1)
	c = a

Fall 4	
b = 1	
	if(b==1)
	c = a
a = 5	

Manipulierung der Instruktionsreihenfolge

```
1  static atomic_int a = 0, b = 0;
2  static int        c = -1;
3
4  static void *sender(void *param) {
5     atomic_store_explicit(&a, 5, memory_order_relaxed);
6
7     atomic_store_explicit(&b, 1, memory_order_relaxed);
8     return NULL;
9  }
10
11 static void *empfaenger(void *param) {
12    if( atomic_load_explicit(&b, memory_order_relaxed) == 1 ) {
13       c = atomic_load_explicit(&a, memory_order_relaxed);
14       assert(c == 5);
15    }
16    return NULL;
17 }
```



Fall 4	
b = 1	
	if(b==1)
	c = a
a = 5	

➔ **Inkorrekte Signalübermittlung!**

Gliederung

1. Motivation ✓
2. **Problem**: Manipulierung der Instruktionsreihenfolge ✓
3. **Lösung**: Speichermodelle
    1. *Relaxed Consistency*
    2. *Release-Acquire Consistency*
    3. *Sequentielle Konsistenz (Sequential Consistency)*
4. Technische Realisierung
5. Zusammenfassung

Speichermodelle

- **Schnittstelle** zwischen Programmierer & Rechensystem
- Funktionalitäten:
  - **Programmierbarkeit**
  - **Portabiltät**
  - ***Performance***
- Seit C11/C++11 drei Speichermodelle verfügbar
- Verschiedene Speichermodelle sind unterschiedlich stark bzw. schwach

1. Relaxed Consistency

→ `memory_order_relaxed`

- Schwächstes Speichermodell
- Garantien:
  - **Atomarität**
- Typischer Anwendungsfall:
  - Implementierung einer Zählervariablen, die nebenläufig inkrementiert wird
- Auf alle Funktionen der `stdatomic.h` Bibliothek anwendbar

Gliederung

1. Motivation ✓
2. **Problem**: Manipulierung der Instruktionsreihenfolge ✓
3. **Lösung**: Speichermodelle ✓
    1. *Relaxed Consistency* ✓
    2. *Release-Acquire Consistency*
    3. *Sequentielle Konsistenz (Sequential Consistency)*
4. Technische Realisierung
5. Zusammenfassung

2. Release-Acquire Consistency

- Stärker als *Relaxed Consistency*
- Garantien:
  - **Atomarität**
  - Speichermodell bietet drei Konsistenzen an:
    - **`memory_order_release`** (Freigebende Semantik)
    - **`memory_order_acquire`** (Akquirierende Semantik)
    - **`memory_order_acq_rel`** (Freigebende & Akquirierende Semantik)
- Etabliert eine **Happens-Before-Beziehung**

Einschub: Happens-Before-Beziehung

- **A happens-before B ⇏ A geschieht vor B**
- **A geschieht vor B ⇏ A happens-before B**
- **A happens-before B → Speicheroperationen von A treten als sichtbarer Seiteneffekt für B auf**

2. Release-Acquire Consistency

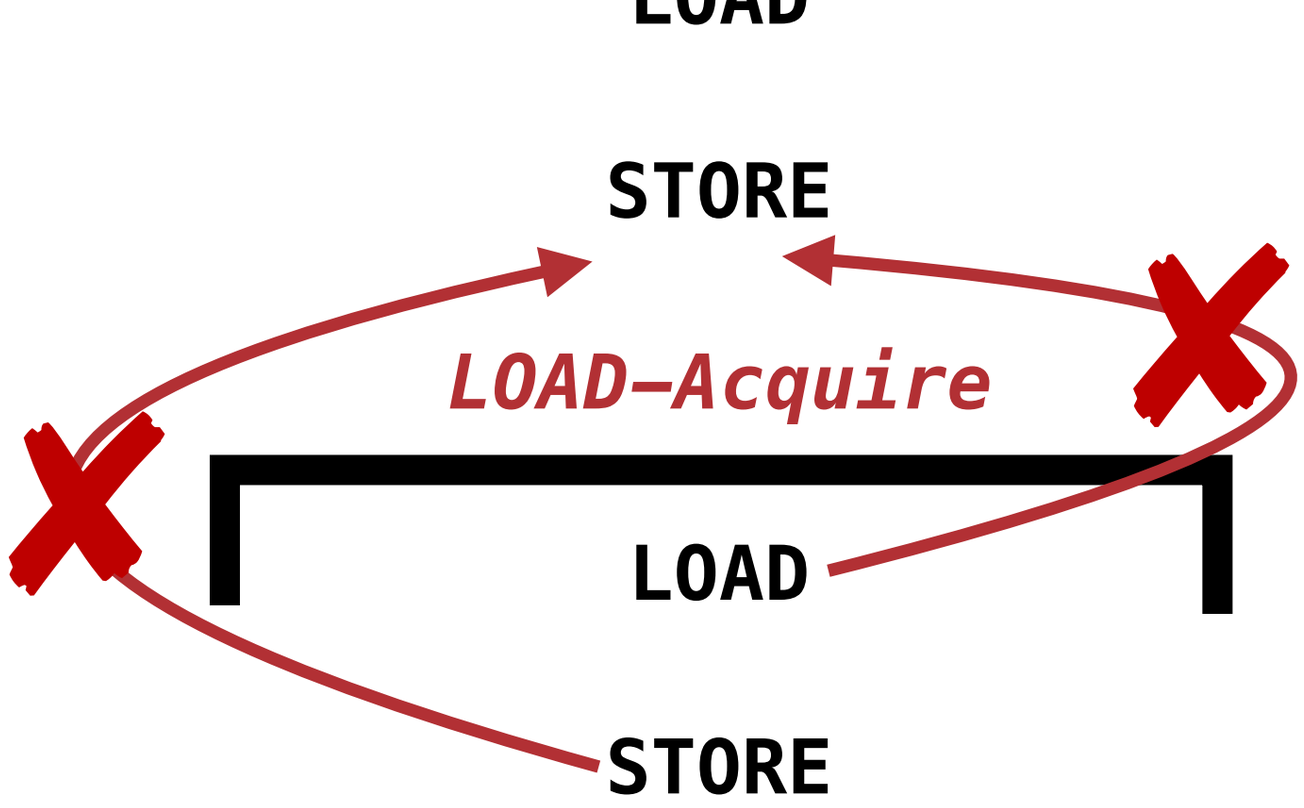
→ `memory_order_acquire` *(Erwerbende Semantik)*

- Für reine **Ladeoperationen** und **Lesen-Ändern-Schreibende** Operationen

2. Release-Acquire Consistency

→ `memory_order_acquire` *(Erwerbende Semantik)*

- Für reine **Ladeoperationen** und **Lesen-Ändern-Schreibende** Operationen
- Unterhalb der *Acquire*-Operation befindliche Speicherzugriffe können **nicht** über die *Acquire*-Operation gezogen werden:



2. Release-Acquire Consistency

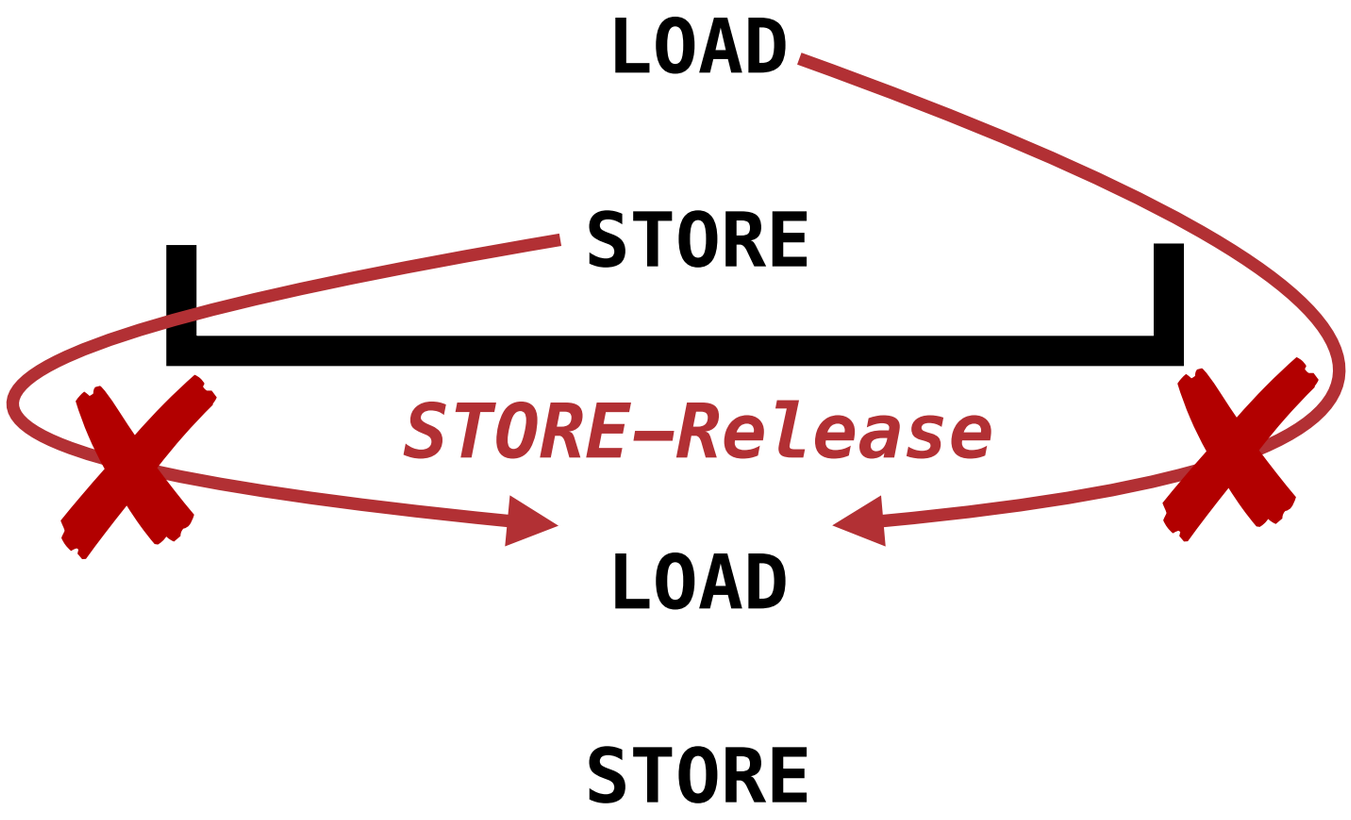
→ `memory_order_release` *(Freigebende Semantik)*

- Für reine **Schreiboperationen** und **Lesen-Ändern-Schreibende** Operationen

2. Release-Acquire Consistency

→ `memory_order_release` *(Freigebende Semantik)*

- Für reine **Schreiboperationen** und **Lesen-Ändern-Schreibende** Operationen
- Oberhalb der *Release*-Operation befindliche Speicherzugriffe können **nicht** unter die *Release*-Operation gezogen werden:



2. Release-Acquire Consistency

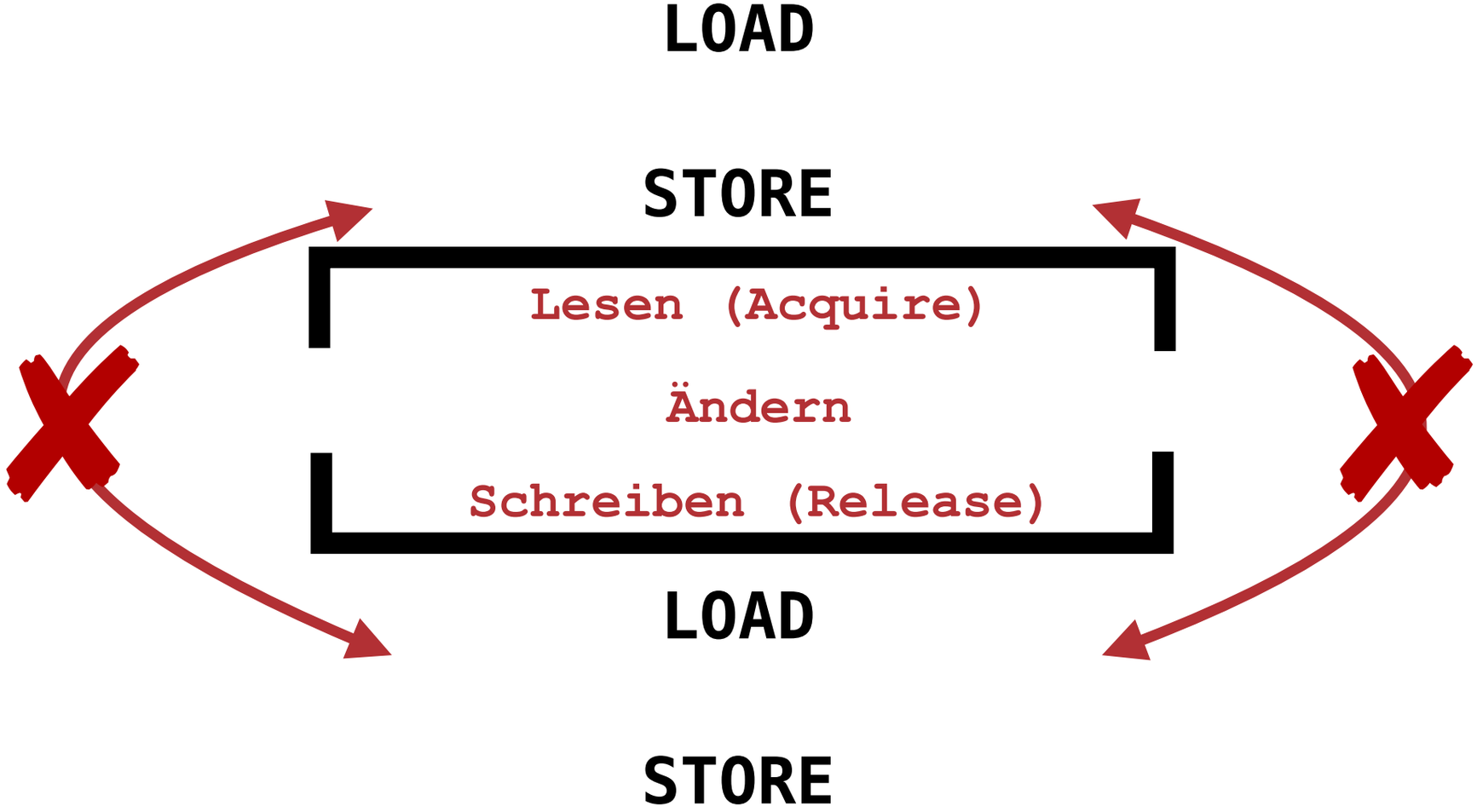
➔ `memory_order_acq_rel` *(Akquirierende-Freigebende Semantik)*

- Nur für **Lesen-Ändern-Schreibende** Operationen

2. Release-Acquire Consistency

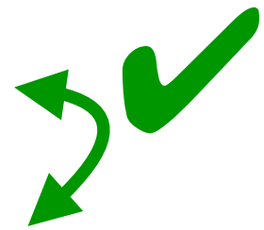
➔ `memory_order_acq_rel` *(Akquirierende-Freigebende Semantik)*

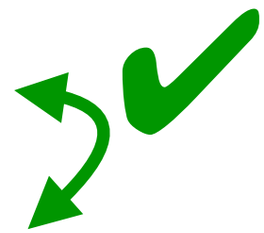
- Nur für **Lesen-Ändern-Schreibende** Operationen
- Speicherzugriffe können **nicht** über die *Acquire-Release-Operation* hinweg verschoben werden:



Beispiel: Dekker - Acquire-Release

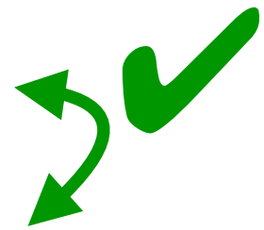
```
1  static atomic_int a = 0,  b = 0;
2  static int        r1 = 0, r2 = 0;
3
4  static void *t1(void *param) {
5
6      atomic_store_explicit(&a, 1, memory_order_release);
7      r1 = atomic_load_explicit(&b, memory_order_acquire);
8      return NULL;
9  }
10
11 static void *t2(void *param) {
12
13     atomic_store_explicit(&b, 1, memory_order_release);
14     r2 = atomic_load_explicit(&a, memory_order_acquire);
15     return NULL;
16 }
```

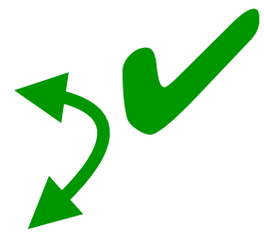




Beispiel: Dekker - Acquire-Release

```
1  static atomic_int a = 0,  b = 0;
2  static int        r1 = 0, r2 = 0;
3
4  static void *t1(void *param) {
5
6      atomic_store_explicit(&a, 1, memory_order_release);
7      r1 = atomic_load_explicit(&b, memory_order_acquire);
8      return NULL;
9  }
10
11 static void *t2(void *param) {
12
13     atomic_store_explicit(&b, 1, memory_order_release);
14     r2 = atomic_load_explicit(&a, memory_order_acquire);
15     return NULL;
16 }
```





r1 = 0 & r2 = 1	
a = 1	
r1 = b	
	b = 1
	r2 = a

r1 = 1 & r2 = 0	
	b = 1
	r2 = a
a = 1	
r1 = b	

r1 = 1 & r2 = 1	
a = 1	**b = 1**
r1 = b	**r2 = a**

r1 = 0 & r2 = 0	
r1 = b	**r2 = a**
a = 1	**b = 1**

Beispiel: Rekked - Acquire-Release

```
1 static atomic_int a = 0,  b = 0;
2 static int        r1 = 0, r2 = 0;
3
4 static void *t1(void *param) {
5   r1 = atomic_load_explicit(&b, memory_order_acquire);
6
7   atomic_store_explicit(&a, 1, memory_order_release);
8   return NULL;
9 }
10
11 static void *t2(void *param) {
12   r2 = atomic_load_explicit(&a, memory_order_acquire);
13
14   atomic_store_explicit(&b, 1, memory_order_release);
15   return NULL;
16 }
```

Beispiel: Rekked - Acquire-Release

```
1  static atomic_int a = 0,  b = 0;
2  static int        r1 = 0, r2 = 0;
3
4  static void *t1(void *param) {
5      r1 = atomic_load_explicit(&b, memory_order_acquire);
6
7      atomic_store_explicit(&a, 1, memory_order_release);
8      return NULL;
9  }
10
11 static void *t2(void *param) {
12     r2 = atomic_load_explicit(&a, memory_order_acquire);
13
14     atomic_store_explicit(&b, 1, memory_order_release);
15     return NULL;
16 }
```

r1 = 0 & r2 = 1	
r1 = b	
a = 1	
	r2 = a
	b = 1

r1 = 1 & r2 = 0	
	r2 = a
	b = 1
r1 = b	
a = 1	

r1 = 0 & r2 = 0	
r1 = b	**r2 = a**
a = 1	**b = 1**

r1 = 1 & r2 = 1	
a = 1	**b = 1**
r1 = b	**r2 = a**

Nicht möglich

Gliederung

1. Motivation ✓
2. **Problem**: Manipulierung der Instruktionsreihenfolge ✓
3. **Lösung**: Speichermodelle ✓
    1. *Relaxed Consistency* ✓
    2. *Release-Acquire Consistency* ✓
    3. *Sequentielle Konsistenz (Sequential Consistency)*
4. Technische Realisierung
5. Zusammenfassung

3. Sequentielle Konsistenz

→ `memory_order_seq_cst`

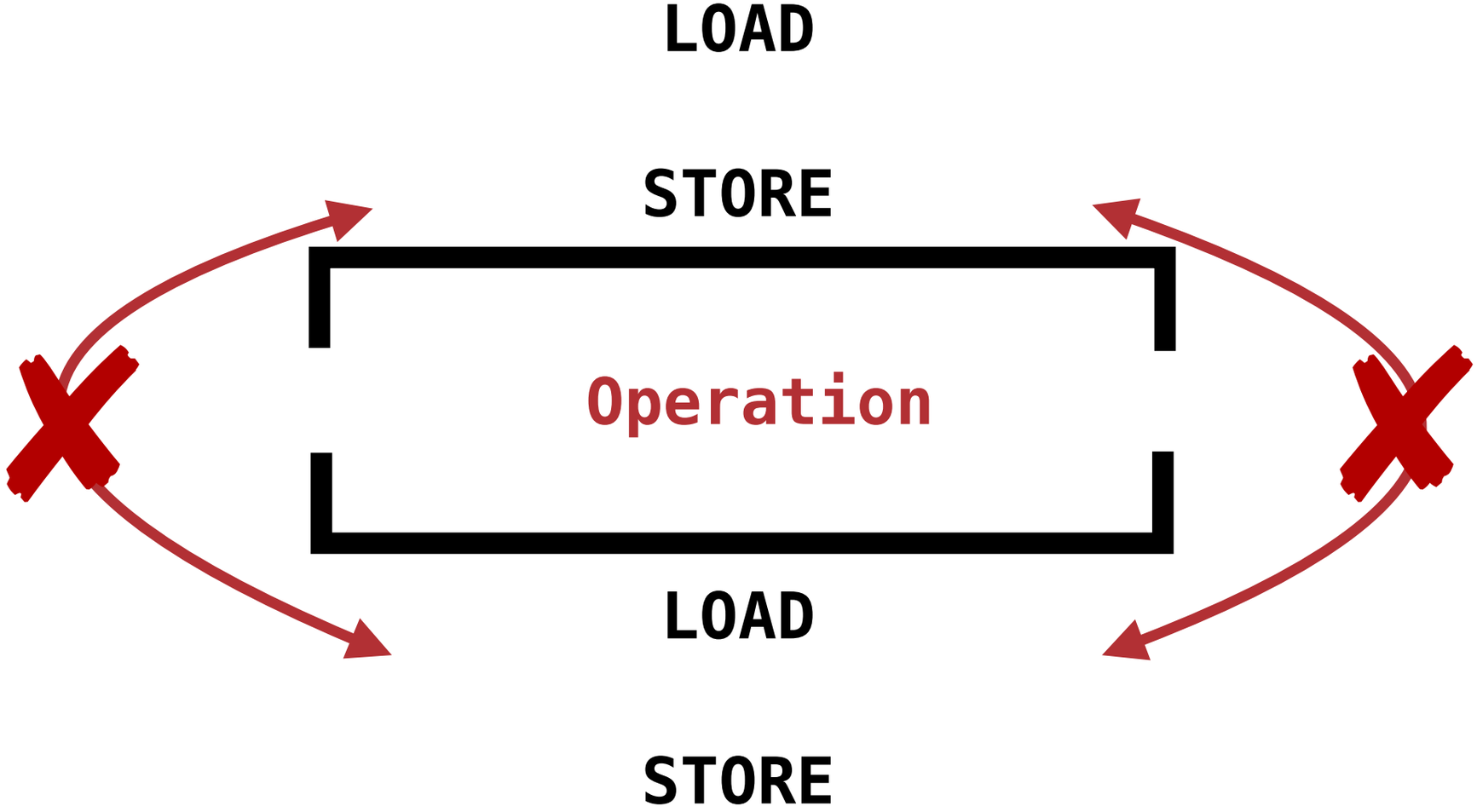
- Standard aller atomaren Funktionen der `stdatomic.h` Bibliothek

3. Sequentielle Konsistenz

→ `memory_order_seq_cst`

- Standard aller atomaren Funktionen der `stdatomic.h` Bibliothek

- Speicherzugriffe können **nicht** über die sequentiell konsistente Operation hinweg verschoben werden:



Beispiel: Dekker - Sequentiell konsistent

```
1  static atomic_int a = 0,  b = 0;
2  static int        r1 = 0, r2 = 0;
3
4  static void *t1(void *param) {
5     atomic_store_explicit(&a, 1, memory_order_seq_cst);
6  ------------------------------------------------------
7     r1 = atomic_load_explicit(&b, memory_order_seq_cst);
8     return NULL;
9  }
10
11 static void *t2(void *param) {
12    atomic_store_explicit(&b, 1, memory_order_seq_cst);
13 ------------------------------------------------------
14    r2 = atomic_load_explicit(&a, memory_order_seq_cst);
15    return NULL;
16 }
```

r1 = 0 & r2 = 1	
a = 1	
r1 = b	
	b = 1
	r2 = a

r1 = 1 & r2 = 0	
	b = 1
	r2 = a
a = 1	
r1 = b	

r1 = 1 & r2 = 1	
a = 1	**b = 1**
r1 = b	**r2 = a**

Gliederung

1. Motivation ✓
2. **Problem**: Manipulierung der Instruktionsreihenfolge ✓
3. **Lösung**: Speichermodelle ✓
    1. *Relaxed Consistency* ✓
    2. *Release-Acquire Consistency* ✓
    3. *Sequentielle Konsistenz (Sequential Consistency)* ✓
4. Technische Realisierung
5. Zusammenfassung

Technische Realisierung

- Gewährleistung der Konsistenzen über Barrieren

Technische Realisierung

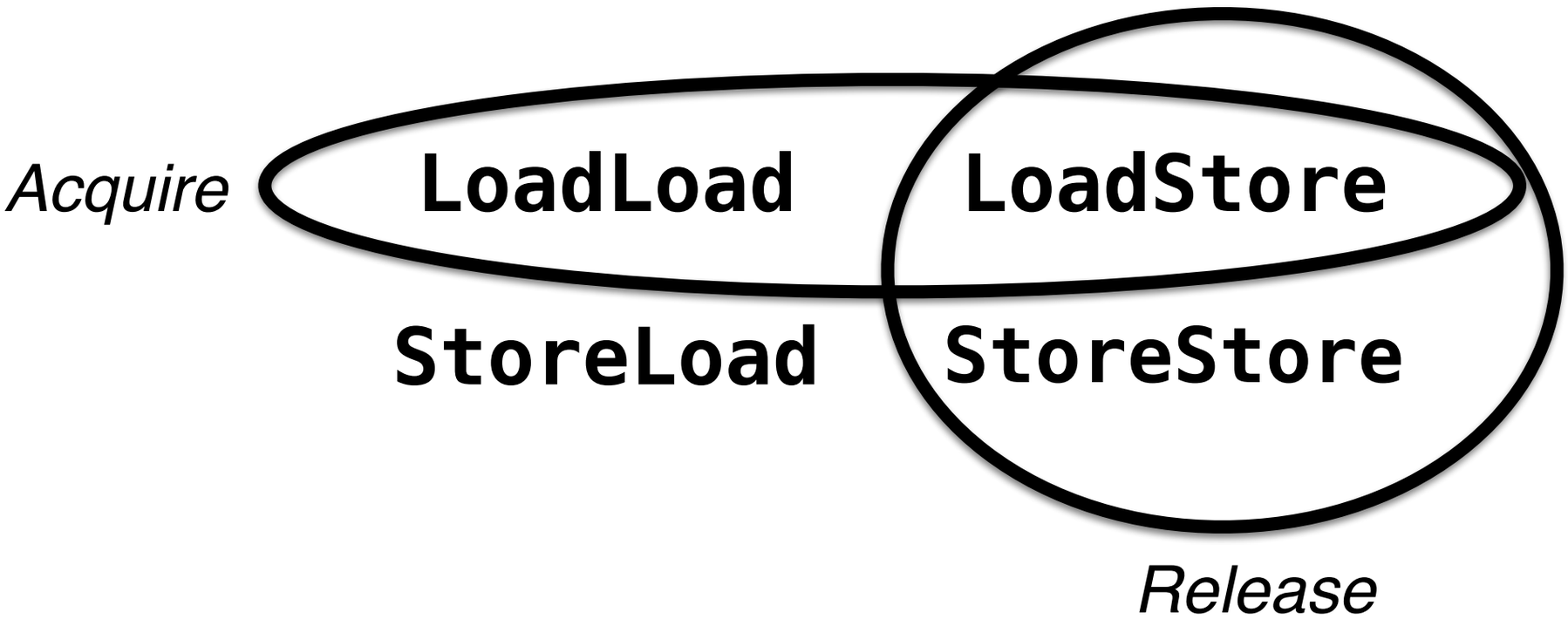
- Gewährleistung der Konsistenzen über Barrieren
- **Acquire-Release-Speichermodell** benötigt drei Barrieretypen:

`LoadLoad` **`LoadStore`**

`StoreLoad` **`StoreStore`**

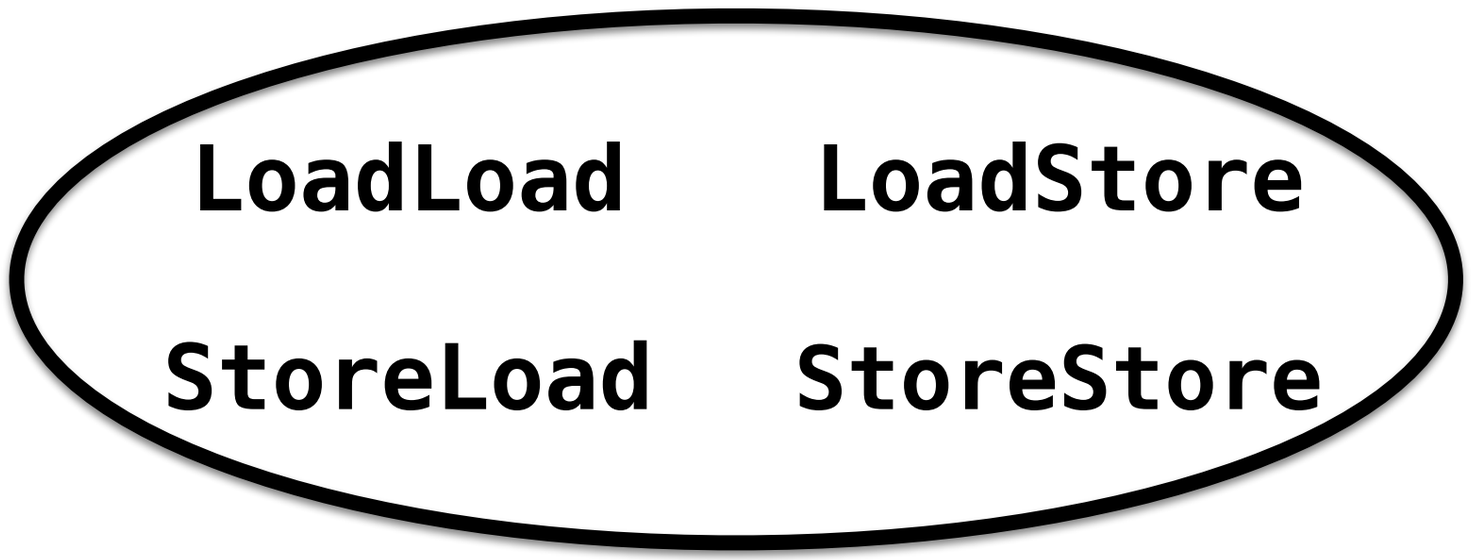
Technische Realisierung

- Gewährleistung der Konsistenzen über Barrieren
- **Acquire-Release-Speichermodell** benötigt drei Barrieretypen:



Technische Realisierung

- Gewährleistung der Konsistenzen über Barrieren
- **Sequentiell konsistentes Speichermodell** benötigt alle Barrieretypen:



Gliederung

1. Motivation ✓
2. **Problem**: Manipulierung der Instruktionsreihenfolge ✓
3. **Lösung**: Speichermodelle ✓
    1. *Relaxed Consistency* ✓
    2. *Release-Acquire Consistency* ✓
    3. *Sequentielle Konsistenz (Sequential Consistency)* ✓
4. Technische Realisierung ✓
5. Abschluss

Abschluss

- C11/C++11 garantiert Speichermodelle nur, wenn Programm frei von **Datenwettläufen** ist ➔ **Data-Race-Free Speichermodell**
- Falls **Datenwettlauf** vorhanden ➔ **Undefinded Behavior**
- Geeignet für Sprachen mit Augenmerk auf Performance (C/C++)
- Nicht geeignet für sichere Sprachen wie z.B. Java