

## 28. Parallel Programming II

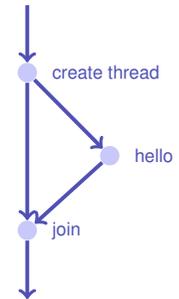
C++ Threads, Gemeinsamer Speicher, Nebenläufigkeit, Exkurs: Lock Algorithmus (Peterson), Gegenseitiger Ausschluss Race Conditions [C++ Threads: Williams, Kap. 2.1-2.2], [C++ Race Conditions: Williams, Kap. 3.1] [C++ Mutexes: Williams, Kap. 3.2.1, 3.3.3]

### C++11 Threads

```
#include <iostream>
#include <thread>

void hello(){
    std::cout << "hello\n";
}

int main(){
    // create and launch thread t
    std::thread t(hello);
    // wait for termination of t
    t.join();
    return 0;
}
```



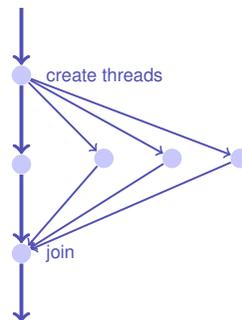
832

833

### C++11 Threads

```
void hello(int id){
    std::cout << "hello from " << id << "\n";
}

int main(){
    std::vector<std::thread> tv(3);
    int id = 0;
    for (auto & t:tv)
        t = std::thread(hello, ++id);
    std::cout << "hello from main \n";
    for (auto & t:tv)
        t.join();
    return 0;
}
```



834

### Nichtdeterministische Ausführung!

Eine Ausführung:

```
hello from main
hello from 2
hello from 1
hello from 0
```

Andere Ausführung:

```
hello from 1
hello from main
hello from 0
hello from 2
```

Andere Ausführung:

```
hello from main
hello from 0
hello from hello from 1
2
```

835

## Technisches Detail

Um einen Thread als Hintergrundthread weiterlaufen zu lassen:

```
void background();

void someFunction(){
    ...
    std::thread t(background);
    t.detach();
    ...
} // no problem here, thread is detached
```

## 28.2 Gemeinsamer Speicher, Nebenläufigkeit

## Mehr Technische Details

- Beim Erstellen von Threads werden auch Referenzparameter kopiert, ausser man gibt explizit `std::ref` bei der Konstruktion an.
- Funktoren oder Lambda-Expressions können auch auf einem Thread ausgeführt werden
- In einem Kontext mit Exceptions sollte das `join` auf einem Thread im `catch`-Block ausgeführt werden

Noch mehr Hintergründe im Kapitel 2 des Buches *C++ Concurrency in Action*, Anthony Williams, Manning 2012. Auch online bei der ETH Bibliothek erhältlich.

836

837

## Gemeinsam genutzte Ressourcen (Speicher)

- Bis hier: fork-join Algorithmen: Datenparallel oder Divide und Conquer
- Einfache Struktur (Datenunabhängigkeit der Threads) zum Vermeiden von Wettlaufsituationen (*race conditions*)
- Funktioniert nicht mehr, wenn Threads gemeinsamen Speicher nutzen müssen.

838

839

## Konsistenz des Zustands

Gemeinsamer Zustand: Hauptschwierigkeit beim nebenläufigen Programmieren.

Ansätze:

- Unveränderbarkeit, z.B. Konstanten
- Isolierte Veränderlichkeit, z.B. Thread-lokale Variablen, Stack.
- Gemeinsame veränderliche Daten, z.B. Referenzen auf gemeinsamen Speicher, globale Variablen

840

## Schütze den gemeinsamen Zustand

- Methode 1: Locks, Garantiere exklusiven Zugriff auf gemeinsame Daten.
- Methode 2: lock-freie Datenstrukturen, garantiert exklusiven Zugriff mit sehr viel feinerer Granularität.
- Methode 3: Transaktionsspeicher (hier nicht behandelt)

841

## Kanonisches Beispiel

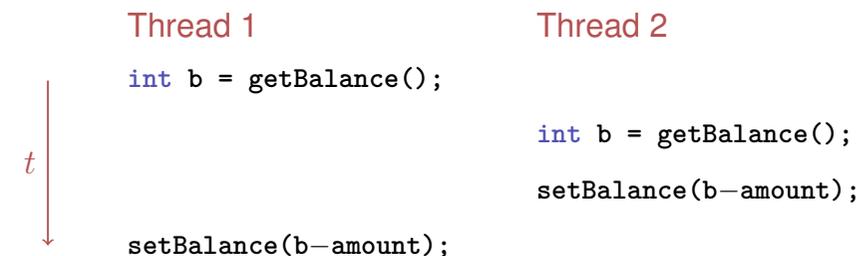
```
class BankAccount {
    int balance = 0;
public:
    int getBalance(){ return balance; }
    void setBalance(int x) { balance = x; }
    void withdraw(int amount) {
        int b = getBalance();
        setBalance(b - amount);
    }
    // deposit etc.
};
```

(korrekt bei Einzelthreadausführung)

842

## Ungünstige Verschachtelung (Bad Interleaving)

Paralleler Aufruf von `withdraw(100)` auf demselben Konto



843

## Verlockende Fallen

FALSCH:

```
void withdraw(int amount) {
    int b = getBalance();
    if (b==getBalance())
        setBalance(b - amount);
}
```

Bad interleavings lassen sich fast **nie mit wiederholtem Lesen** lösen

844

## Verlockende Fallen

Auch FALSCH:

```
void withdraw(int amount) {
    setBalance(getBalance() - amount);
}
```

**Annahmen über Atomizität von Operationen sind fast immer falsch**

845

## Gegenseitiger Ausschluss (Mutual Exclusion)

Wir benötigen ein Konzept für den gegenseitigen Ausschluss

*Nur ein Thread* darf zu einer Zeit die Operation *withdraw* *auf demselben Konto* ausführen.

Der Programmierer muss den gegenseitigen Ausschluss sicherstellen.

846

## Mehr verlockende Fallen

```
class BankAccount {
    int balance = 0;
    bool busy = false;
public:
    void withdraw(int amount) {
        while (busy); // spin wait
        busy = true;
        int b = getBalance();
        setBalance(b - amount);
        busy = false;
    }

    // deposit would spin on the same boolean
};
```

funktioniert nicht!

847

## Das Problem nur verschoben!

Thread 1

```
while (busy); //spin  
  
busy = true;  
  
int b = getBalance();  
  
setBalance(b - amount);
```

Thread 2

```
while (busy); //spin  
  
busy = true;  
  
int b = getBalance();  
setBalance(b - amount);
```

t

## Wie macht man das richtig?

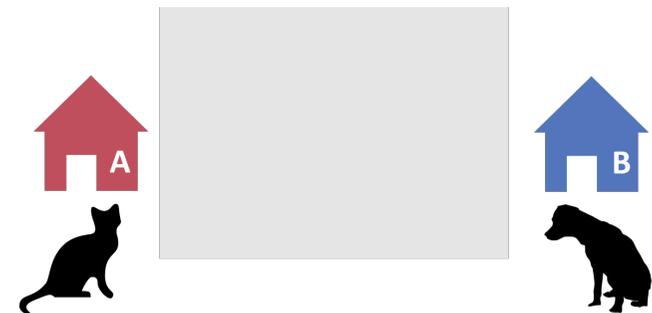
- Wir benutzen ein *Lock* (eine Mutex) aus Bibliotheken
- Eine Mutex verwendet ihrerseits Hardwareprimitiven, *Read-Modify-Write* (RMW) Operationen, welche atomar lesen und abhängig vom Leseergebnis schreiben können.
- Ohne RMW Operationen ist der Algorithmus nichttrivial und benötigt zumindest atomaren Zugriff auf Variablen von primitivem Typ.

848

849

## 28.3 Exkurs: Lock Algorithmus

## Alice Katze und Bobs Dog



850

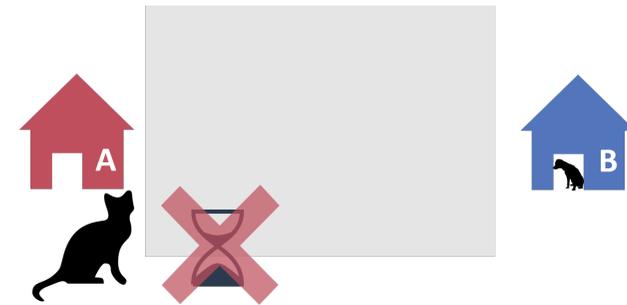
851

## Gefordert: Gegenseitiger Ausschluss



852

## Gefordert: Kein grundloses Aussperren



853

## Arten der Kommunikation

- Transient: Parteien kommunizieren zur selben Zeit



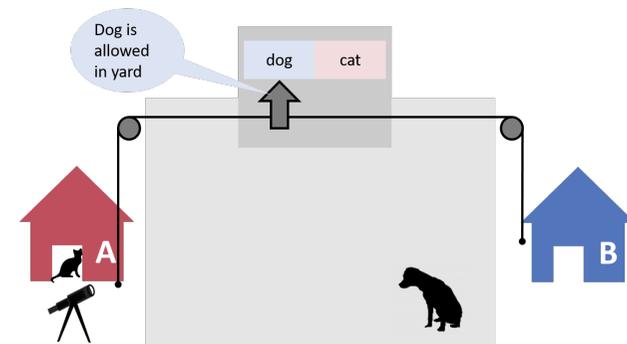
- Persistent: Parteien kommunizieren zu verschiedenen Zeiten



Gegenseitiger Ausschluss: Persistente Kommunikation

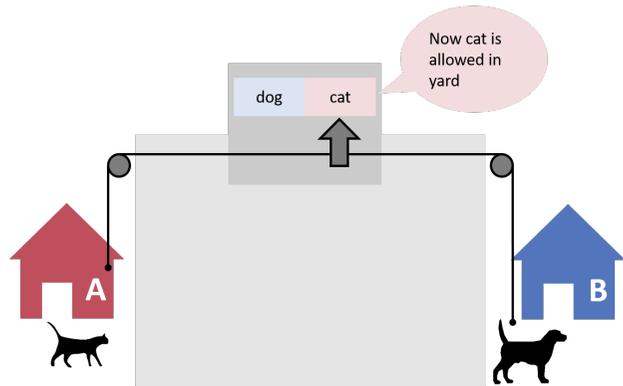
854

## Erste Idee



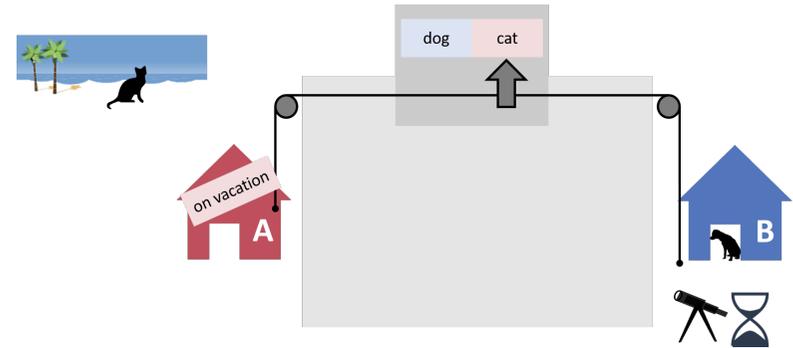
855

## Zugriffsprotokoll



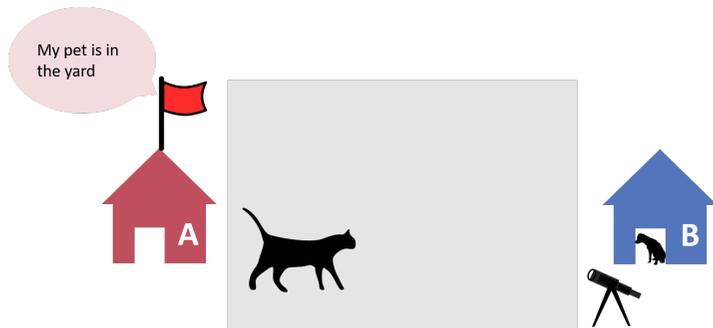
856

## Problem!



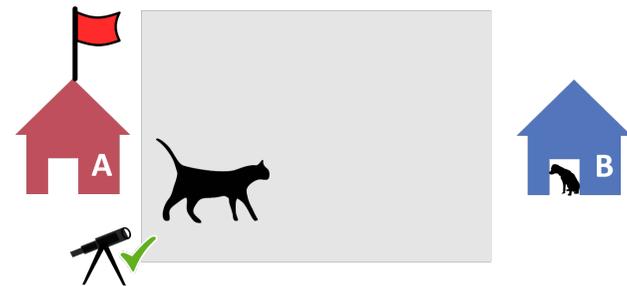
857

## Zweite Idee



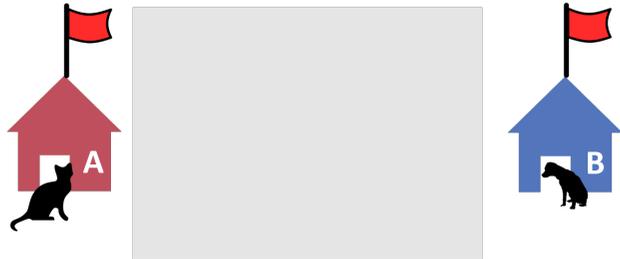
858

## Zugriffsprotokoll 2.1



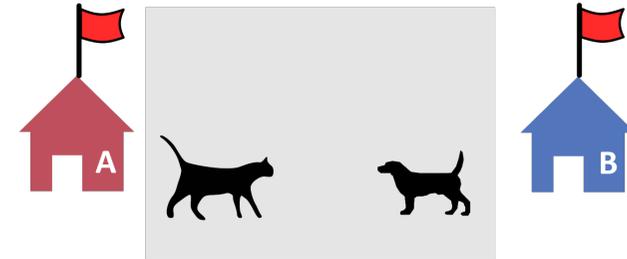
859

## Anderes Szenario



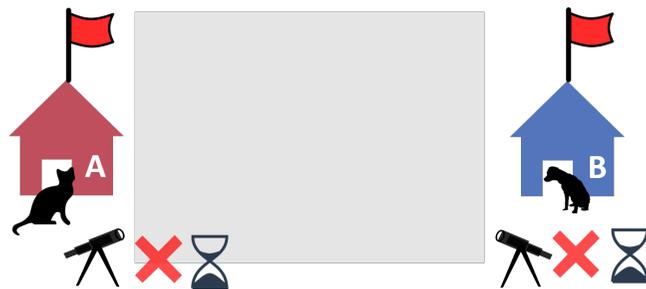
860

## Problem: Kein gegenseitiger Ausschluss



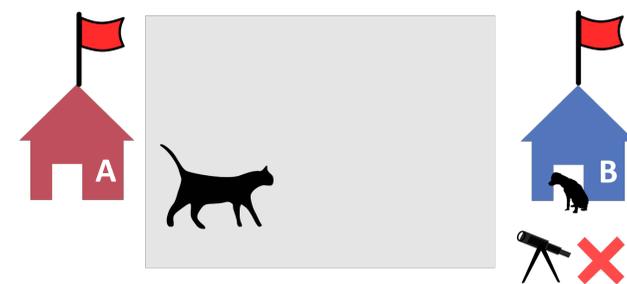
861

## Die Fahnen zweimal prüfen: Deadlock



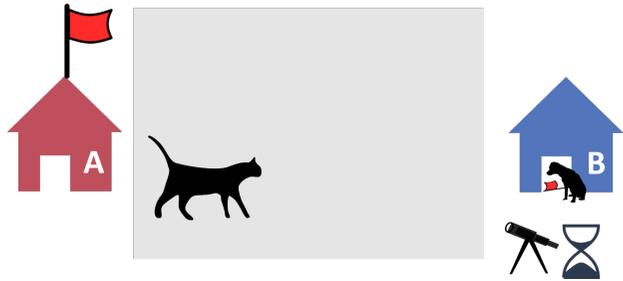
862

## Zugriffsprotokoll 2.2



863

## Zugriffsprotokoll 2.2: beweisbar korrekt



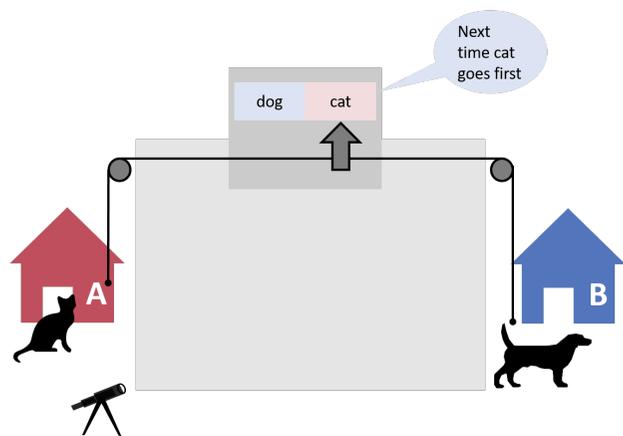
## Minor Problem: Starvation



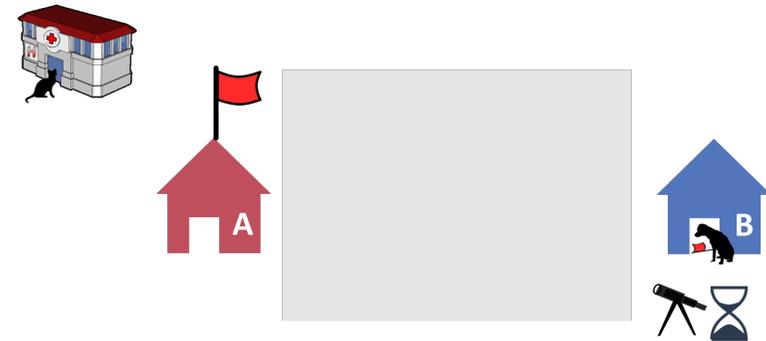
864

865

## Lösung



## Das generelles Problem mit Locking bleibt



866

867

## Der Algorithmus von Peterson<sup>42</sup>

für zwei Prozesse ist beweisbar korrekt und frei von Starvation.

`non-critical section`

```
flag[me] = true // I am interested
victim = me // but you go first
// spin while we are both interested and you go first:
while (flag[you] && victim == me) {};
```

`critical section`

```
flag[me] = false
```

Der Code setzt voraus, dass der Zugriff auf `flag` / `victim` atomar, linearisiert oder sequentiell konsistent ist, eine Anforderung, welche – wie wir weiter unten sehen – für normale Variablen nicht unbedingt gegeben ist. Das Peterson-Lock wird auf moderner Hardware nicht eingesetzt.

<sup>42</sup>nicht prüfungsrelevant

868

## 28.4 Gegenseitiger Ausschluss

869

## Kritische Abschnitte und Gegenseitiger Ausschluss

**Kritischer Abschnitt** (Critical Section)

Codestück, welches nur durch einen einzigen Thread zu einer Zeit ausgeführt werden darf.

**Gegenseitiger Ausschluss** (Mutual Exclusion)

Algorithmus zur Implementation eines kritischen Abschnitts

```
acquire_mutex(); // entry algorithm\
... // critical section
release_mutex(); // exit algorithm
```

870

## Anforderung an eine Mutex.

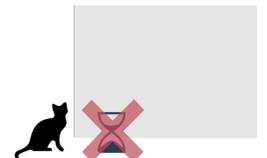
Korrektheit (Safety)

- Maximal ein Prozess in der kritischen Region



Fortschritt (Liveness)

- Das Betreten der kritischen Region darf nur endliche Zeit dauern, wenn kein Thread in der kritischen Region verweilt.



871

## Fast Korrekt

```
class BankAccount {
    int balance = 0;
    std::mutex m; // requires #include <mutex>
public:
    ...
    void withdraw(int amount) {
        m.lock();
        int b = getBalance();
        setBalance(b - amount);
        m.unlock();
    }
};
```

Was, wenn eine Exception auftritt?

872

## RAII Ansatz

```
class BankAccount {
    int balance = 0;
    std::mutex m;
public:
    ...
    void withdraw(int amount) {
        std::lock_guard<std::mutex> guard(m);
        int b = getBalance();
        setBalance(b - amount);
    } // Destruction of guard leads to unlocking m
};
```

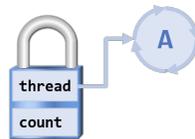
Was ist mit getBalance / setBalance?

873

## Reentrante Locks

Reentrant Lock (rekursives Lock)

- merkt sich den betroffenen Thread;
- hat einen Zähler
  - Aufruf von lock: Zähler wird inkrementiert
  - Aufruf von unlock: Zähler wird dekrementiert. Wenn Zähler = 0, wird das Lock freigegeben



874

## Konto mit reentrantem Lock

```
class BankAccount {
    int balance = 0;
    std::recursive_mutex m;
    using guard = std::lock_guard<std::recursive_mutex>;
public:
    int getBalance(){ guard g(m); return balance;
    }
    void setBalance(int x) { guard g(m); balance = x;
    }
    void withdraw(int amount) { guard g(m);
        int b = getBalance();
        setBalance(b - amount);
    }
};
```

875

## 28.5 Race Conditions

### Wetlaufsituation (Race Condition)

- Eine *Wetlaufsituation* (Race Condition) tritt auf, wenn das Resultat einer Berechnung vom Scheduling abhängt.
- Wir unterscheiden *bad interleavings* und *data races*
- Bad Interleavings können auch unter Verwendung einer Mutex noch auftreten.

876

877

### Beispiel: Stack

Stack mit korrekt synchronisiertem Zugriff:

```
template <typename T>
class stack{
    ...
    std::recursive_mutex m;
    using guard = std::lock_guard<std::recursive_mutex>;
public:
    bool isEmpty(){ guard g(m); ... }
    void push(T value){ guard g(m); ... }
    T pop(){ guard g(m); ...}
};
```

### Peek

Peek Funktion vergessen. Dann so?

```
template <typename T>
T peek (stack<T> &s){
    T value = s.pop();
    s.push(value);
    return value;
}
```

nicht Thread-sicher!

Code trotz fragwürdigem Stil in sequentieller Welt korrekt. Nicht so in nebenläufiger Programmierung!

878

879

## Bad Interleaving!

Initial leerer Stack  $s$ , nur von Threads 1 und 2 gemeinsam genutzt.

Thread 1 legt einen Wert auf den Stack und prüft, dass der Stack nichtleer ist. Thread 2 liest mit `peek()` den obersten Wert.

	Thread 1	Thread 2
$t$	<code>s.push(5);</code>	
	<code>assert(!s.isEmpty());</code>	<code>int value = s.pop();</code>
		<code>s.push(value);</code>
		<code>return value;</code>

880

## Die Lösung

Peek muss mit demselben Lock geschützt werden, wie die anderen Zugriffsmethoden.

881

## Bad Interleavings

Race Conditions in Form eines Bad Interleavings können also auch auf hoher Abstraktionsstufe noch auftreten.

Betrachten nachfolgend andere Form der Wettlaufsituation: Data Race.

## Wie ist es damit?

```
class counter{
    int count = 0;
    std::recursive_mutex m;
    using guard = std::lock_guard<std::recursive_mutex>;
public:
    int increase(){
        guard g(m); return ++count;
    }
    int get(){
        return count;
    }
}
```

*nicht Thread-sicher!*

882

883

## Warum falsch?

Es sieht so aus, als könne hier nichts schiefgehen, da der Update von count in einem "winzigen Schritt" geschieht.

Der Code ist trotzdem falsch und von Implementationsdetails der Programmiersprache und unterliegenden Hardware abhängig.

Das vorliegende Problem nennt man *Data-Race*

Moral: *Vermeide Data-Races, selbst wenn jede denkbare Form von Verschachtelung richtig aussieht. Mache keine Annahmen über die Anordnung von Speicheroperationen.*

## Etwas formaler

*Data Race* (low-level Race-Conditions) Fehlerhaftes Programmverhalten verursacht durch ungenügend synchronisierten Zugriff zu einer gemeinsam genutzten Resource, z.B. gleichzeitiges Lesen/Schreiben oder Schreiben/Schreiben zum gleichen Speicherbereich.

*Bad Interleaving* (High Level Race Condition) Fehlerhaftes Programmverhalten verursacht durch eine unglückliche Ausführungsreihenfolge eines Algorithmus mit mehreren Threads, selbst dann wenn die gemeinsam genutzten Ressourcen anderweitig gut synchronisiert sind.

884

885

## Genau hingeschaut

```
class C {
  int x = 0;
  int y = 0;
public:
  void f() {
    (A) x = 1;
    (B) y = 1;
  }
  void g() {
    (C) int a = y;
    (D) int b = x;
    assert(b >= a);
  }
}
```

Kann das schiefgehen

Es gibt keine Verschachtelung zweier f und g aufrufender Threads die die Bedingung in der Assertion falsch macht:

- A B C D ✓
- A C B D ✓
- A C D B ✓
- C A B D ✓
- C C D B ✓
- C D A B ✓

**Es kann trotzdem schiefgehen!**

## Ein Grund: Memory Reordering

*Daumenregel:* Compiler und Hardware dürfen die Ausführung des Codes so ändern, dass die *Semantik einer sequentiellen Ausführung* nicht geändert wird.

```
void f() {
  x = 1;
  y = x+1;
  z = x+1;
}
```

↔  
sequentiell äquivalent

```
void f() {
  x = 1;
  z = x+1;
  y = x+1;
}
```

886

887

## Die Software-Perspektive

Moderne Compiler geben keine Garantie, dass die globale Anordnung aller Speicherzugriffe der Ordnung im Quellcode entsprechen

- Manche Speicherzugriffe werden sogar komplett wegoptimiert
- Grosses Potential für Optimierungen – und Fehler in der nebenläufigen Welt, wenn man falsche Annahmen macht

## Beispiel: Selbstgemachtes Rendezvous

```
int x; // shared
```

```
void wait(){  
    x = 1;  
    while(x == 1);  
}
```

```
void arrive(){  
    x = 2;  
}
```

Angenommen Thread 1 ruft wait auf, später ruft Thread 2 arrive auf. Was passiert?



888

889

## Kompilation

Source

```
int x; // shared
```

```
void wait(){  
    x = 1;  
    while(x == 1);  
}
```

```
void arrive(){  
    x = 2;  
}
```

Ohne Optimierung

```
wait:  
movl $0x1, x  
test: ←  
mov x, %eax  
cmp $0x1, %eax  
je test
```

*if equal*

Mit Optimierung

```
wait:  
movl $0x1, x  
test: ←  
jmp test
```

*always*

```
arrive  
movl $0x2, x
```

890

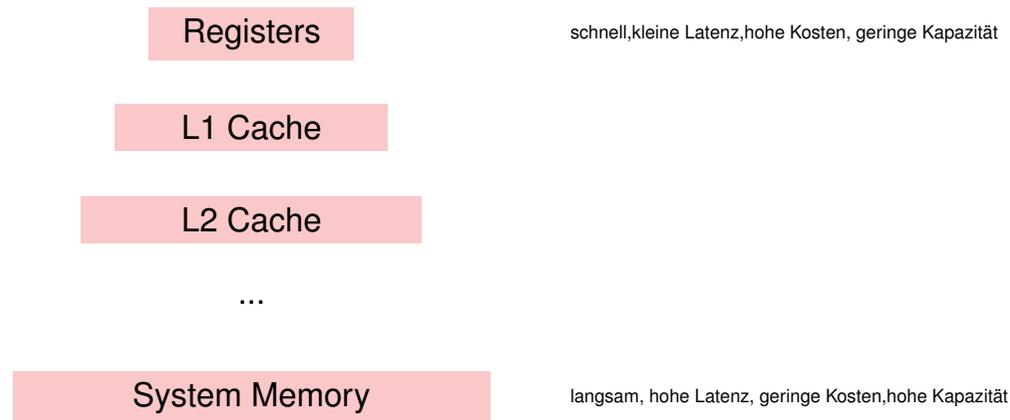
891

## Hardware Perspektive

Moderne Prozessoren erzwingen nicht die globale Anordnung aller Instruktionen aus Gründen der Performanz:

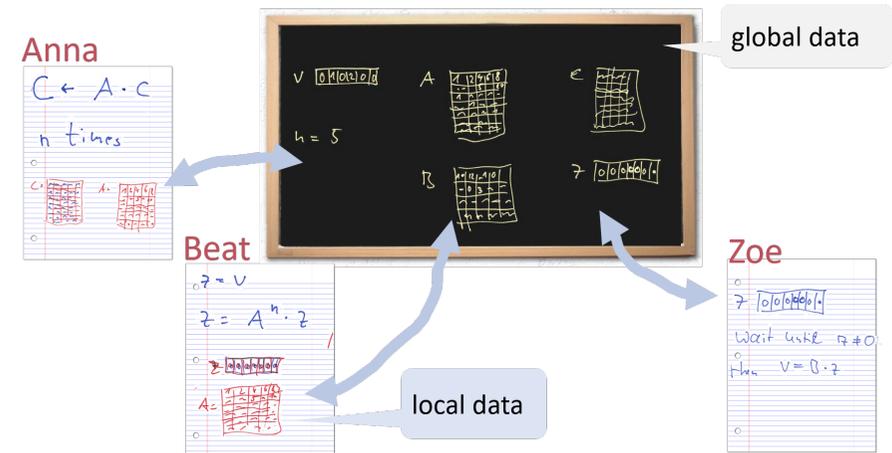
- Die meisten Prozessoren haben einen Pipeline und können Teile von Instruktionen simultan oder in anderer Reihenfolge ausführen.
- Jeder Prozessor(kern) hat einen lokalen Cache, der Effekt des Speicherns im gemeinsamen Speicher kann bei anderen Prozessoren u.U. erst zu einem späteren Zeitpunkt sichtbar werden.

## Speicherhierarchie



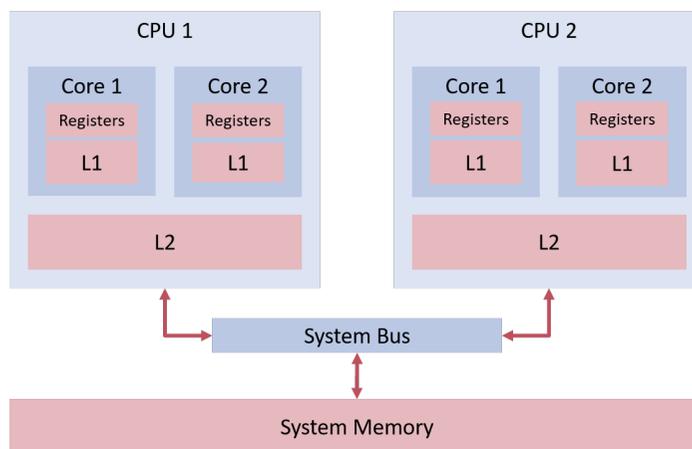
892

## Eine Analogie



893

## Schematisch



894

## Speichermodelle

Wann und ob Effekte von Speicheroperationen für Threads sichtbar werden, hängt also von Hardware, Laufzeitsystem und der Programmiersprache ab.

Ein *Speichermodell* (z.B. das von C++) gibt Minimalgarantien für den Effekt von Speicheroperationen.

- Lässt Möglichkeiten zur Optimierung offen
- Enthält Anleitungen zum Schreiben Thread-sicherer Programme

C++ gibt zum Beispiel *Garantien, wenn Synchronisation mit einer Mutex verwendet* wird.

895

## Repariert

```
class C {
    int x = 0;
    int y = 0;
    std::mutex m;
public:
    void f() {
        m.lock(); x = 1; m.unlock();
        m.lock(); y = 1; m.unlock();
    }
    void g() {
        m.lock(); int a = y; m.unlock();
        m.lock(); int b = x; m.unlock();
        assert(b >= a); // cannot fail
    }
};
```

896

## Atomic

Hier auch möglich:

```
class C {
    std::atomic_int x{0}; // requires #include <atomic>
    std::atomic_int y{0};
public:
    void f() {
        x = 1;
        y = 1;
    }
    void g() {
        int a = y;
        int b = x;
        assert(b >= a); // cannot fail
    }
};
```

897