

28. Parallel Programming II

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]

914

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.

916

28.1 Gemeinsamer Speicher, Nebenläufigkeit

915

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

917

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)

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)

918

919

Ungünstige Verschachtelung (Bad Interleaving)

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

	Thread 1	Thread 2
	<code>int b = getBalance();</code>	<code>int b = getBalance();</code>
		<code>setBalance(b-amount);</code>
t	<code>setBalance(b-amount);</code>	

920

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

921

Verlockende Fallen

Auch FALSCH:

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

Annahmen über Atomizität von Operationen sind fast immer falsch

922

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.

923

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!

924

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

925

Wie macht man das richtig?

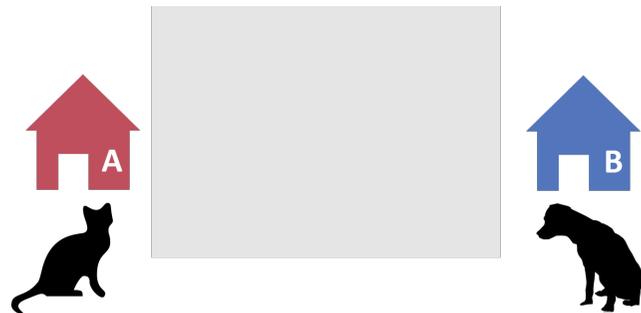
- Wir benutzen ein *Lock* (eine Mutex) aus Bibliotheken
- Eine Mutex verwendet ihrerseits Hardwareprimitiven, *Read-Modify-Write* (RMW) Operationen, welche atomar lesen und abhängig vom Leseergebnis schreiben können.
- Ohne RMW Operationen ist der Algorithmus nichttrivial und benötigt zumindest atomaren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

28.2 Exkurs: Lock Algorithmus

926

927

Alice Katze und Bobs Dog



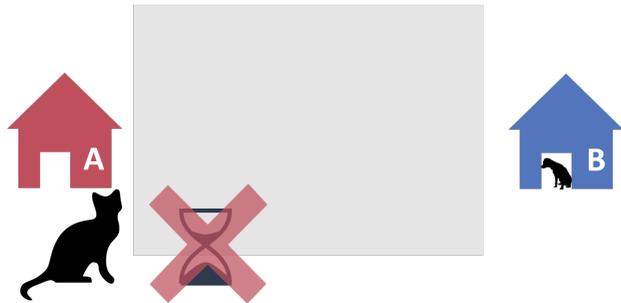
928

Gefordert: Gegenseitiger Ausschluss



929

Gefordert: Kein grundloses Aussperren



Arten der Kommunikation

- Transient: Parteien kommunizieren zur selben Zeit



- Persistent: Parteien kommunizieren zu verschiedenen Zeiten

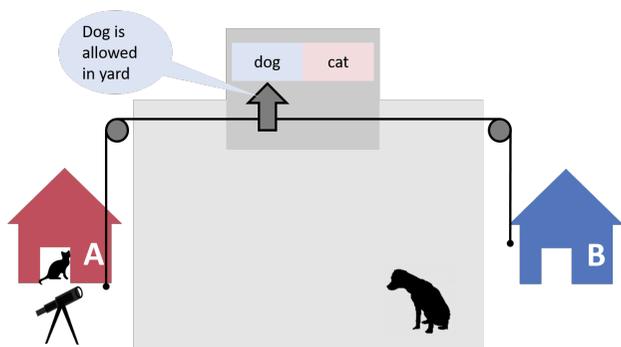


Gegenseitiger Ausschluss: Persistente Kommunikation

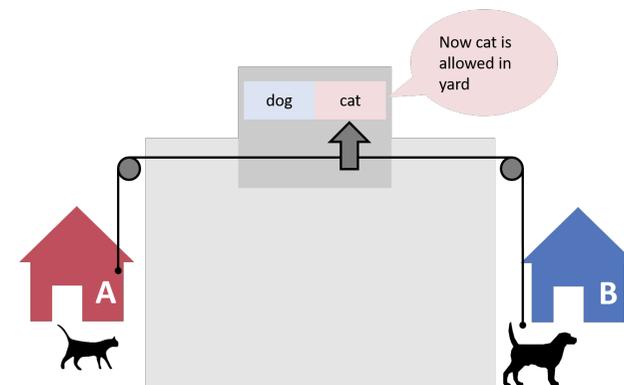
930

931

Erste Idee



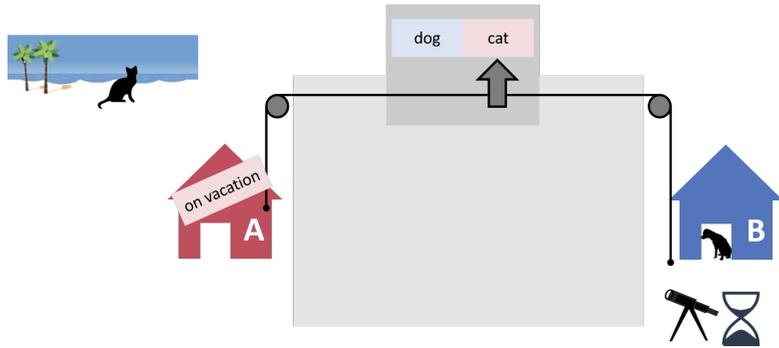
Zugriffsprotokoll



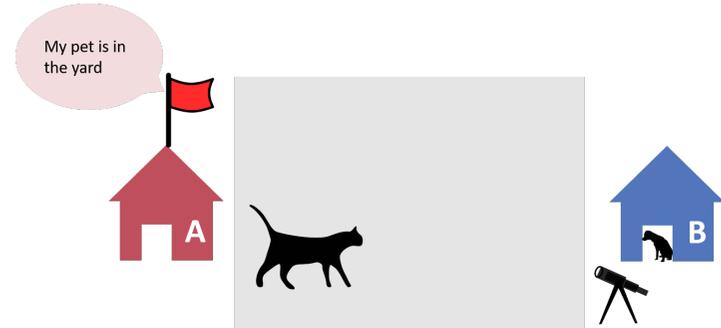
932

933

Problem!



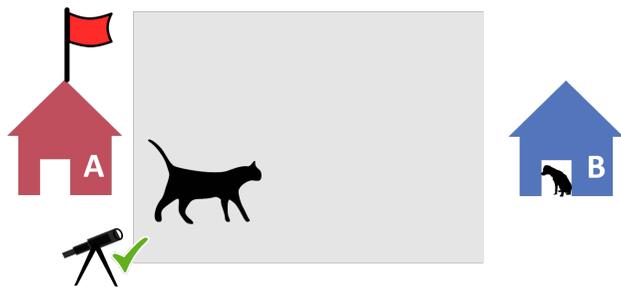
Zweite Idee



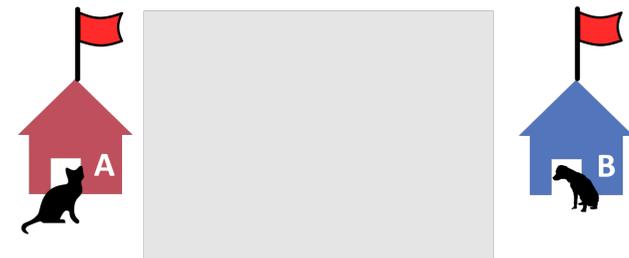
934

935

Zugriffsprotokoll 2.1



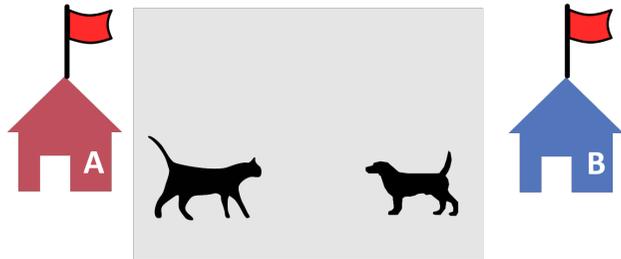
Anderes Szenario



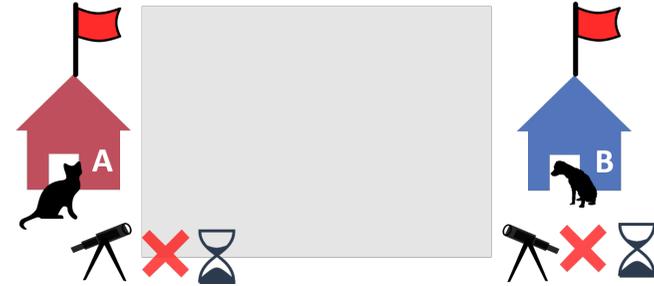
936

937

Problem: Kein gegenseitiger Ausschluss



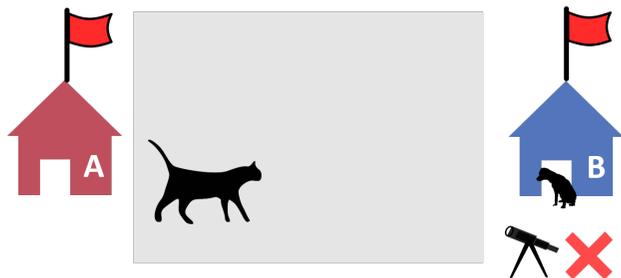
Die Fahnen zweimal prüfen: Deadlock



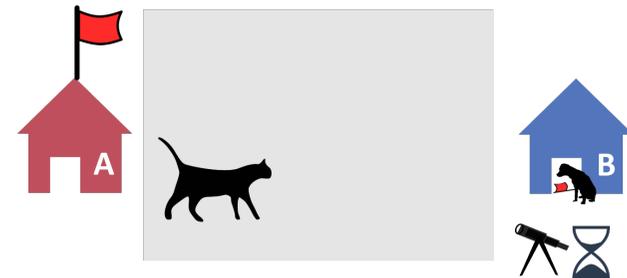
938

939

Zugriffsprotokoll 2.2



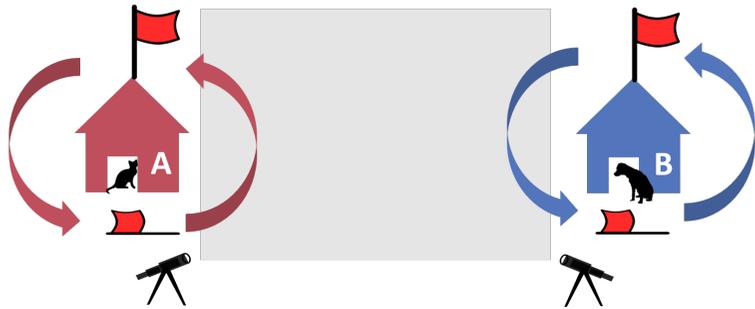
Zugriffsprotokoll 2.2: beweisbar korrekt



940

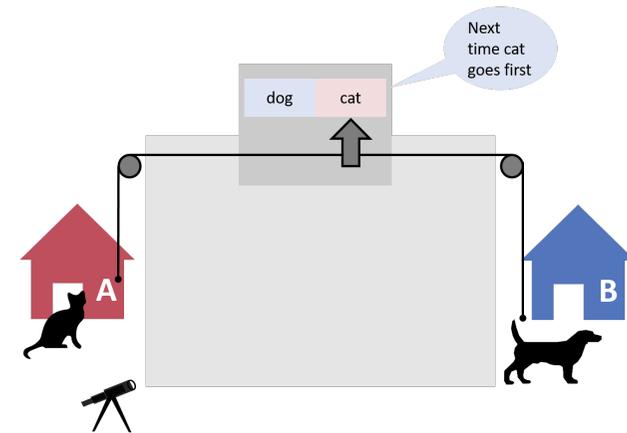
941

Minor Problem: Starvation



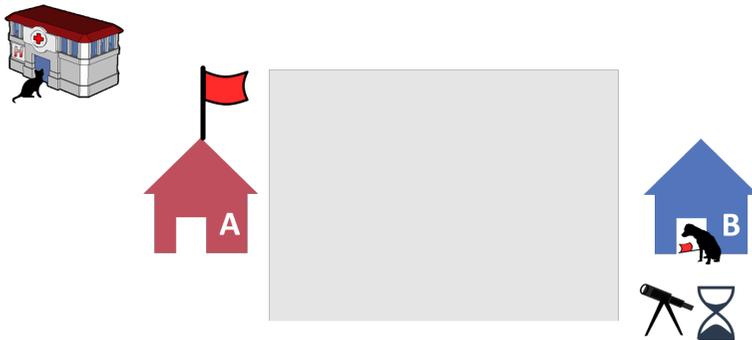
942

Lösung



943

Das generelles Problem mit Locking bleibt



944

Der Algorithmus von Peterson⁵⁵

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.

⁵⁵nicht prüfungsrelevant

945

Kritische Abschnitte und Gegenseitiger Ausschluss

28.3 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
```

946

947

Anforderung an eine Mutex.

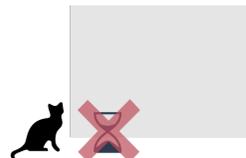
Korrektheit (Safety)

- 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.



948

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?

949

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
};
```

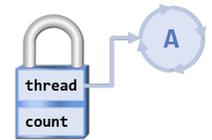
Was ist mit getBalance / setBalance?

950

Reentrante Locks

Reentrantes 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



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);
    }
};
```

952

28.4 Race Conditions

951

953

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.

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); ...}
};
```

954

955

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!

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
<pre>s.push(5);</pre>	
<pre>assert(!s.isEmpty());</pre>	<pre>int value = s.pop();</pre>
	<pre>s.push(value);</pre>
	<pre>return value;</pre>

t ↓

956

957

Die Lösung

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

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.

958

959

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!

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.*

960

961

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.

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!

962

963

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;
}
```

964

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

965

Beispiel: Selbstgemachtes Rendezvous

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

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

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



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

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
```

A red arrow points from the 'je test' instruction back to the 'test' instruction, with the text 'if equal' written next to it.

Mit Optimierung

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

A red arrow points from the 'jmp test' instruction back to the 'test' instruction, with the text 'always' written next to it.

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

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

966

967

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

Registers

schnell, kleine Latenz, hohe Kosten, geringe Kapazität

L1 Cache

L2 Cache

...

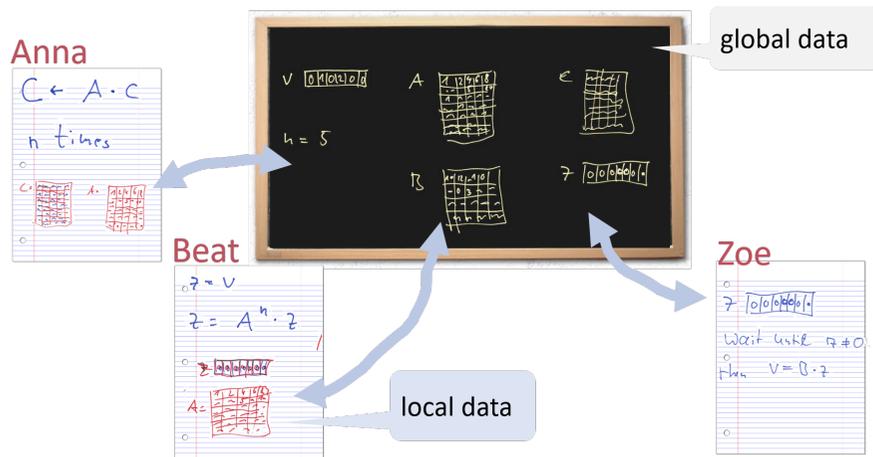
System Memory

langsam, hohe Latenz, geringe Kosten, hohe Kapazität

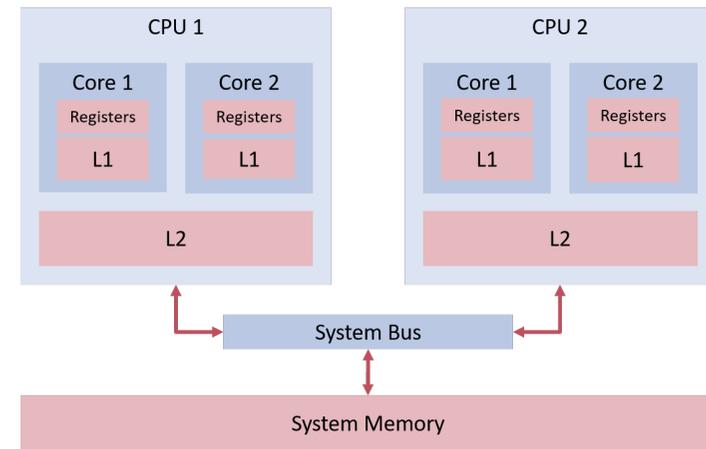
968

969

Eine Analogie



Schematisch



970

971

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.

972

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
    }
};
```

973

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  
    }  
};
```