31. Parallel Programming II

Gemeinsamer Speicher, Nebenläufigkeit , 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]

31.1 Gemeinsamer Speicher, Nebenläufigkeit

Gemeinsam genutzte Resourcen (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.

Konsistenz des Zustands

Gemeinsamen Zustand verwalten: 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 $\Rightarrow$ **Synchronisationsbedarf**

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)

Ungünstige Verschachtelung (Bad Interleaving)

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

	Thread 1	Thread 2
t ↓	`int b = getBalance();`	
		`int b = getBalance();`
		`setBalance(b-amount);`
	`setBalance(b-amount);`	

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

Verlockende Fallen

Auch FALSCH:

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

Annahmen über Atomizität von Operationen sind fast immer falsch

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 Ausschlus sicherstellen.

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!

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, sogenannte **Read-Modify-Write** (RMW) Operationen, welche atomar lesen und abhängig vom Leseergebis schreiben können.
- Ohne RMW Operationen ist der Algorithmus nichttrivial und benötigt zumindest atomaren Zugriff auf Variablen von primitivem Typ.

31.2 Gegenseitiger Ausschluss

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
```

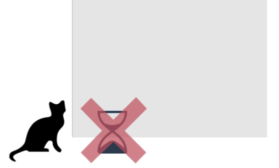
Anforderung an eine Mutex.

Korrektheit (Safety)

- Maximal ein Thread in der kritischen Region

Fortschritt (Liveness)

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



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?

RAII Ansatz

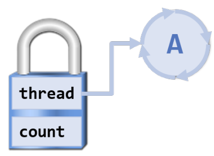
```
class BankAccount {
  int balance = 0;
  std::mutex m;
public:
  ...
  void withdraw(int amount) {
    std::lock_guard<std::mutex> guard(m);
    int b = getBalance();
    setBalance(b - amount);
  } // Destruction of guard leads to unlocking m
};
```

Was ist mit getBalance / setBalance?

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 freigeben



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);
  }
};
```

31.3 Race Conditions

Wettlaufsituation (Race Condition)

- Eine **Wettlaufsituation** (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); ...}
};
```

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!

Stack s von Threads 1 und 2 gemeinsam genutzt. Beide Threads rufen peek() auf

	Thread 1	Thread 2
t ↓	`int value = s.pop();`	
		`int value = s.pop();`
	`s.push(value);`	
		`s.push(value);`
		`return value;`
	`return value;`	

Elemente werden getauscht: die LIFO-Invariante hält nicht.

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 Wettlaufsitation: 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(){
    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.**

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 Resourcen anderweitig gut synchronisiert sind.

Genau hingeschaut

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

$\Longleftrightarrow$
sequentiell äquivalent

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

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

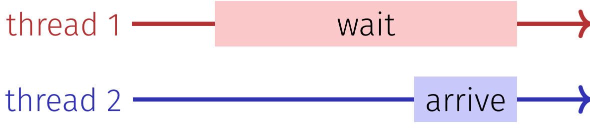
Beispiel: Selbstgemachtes Rendevouz

```
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?



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

arrive:
movl $0x2, x
```

if equal

Mit Optimierung

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

arrive
movl $0x2, x
```

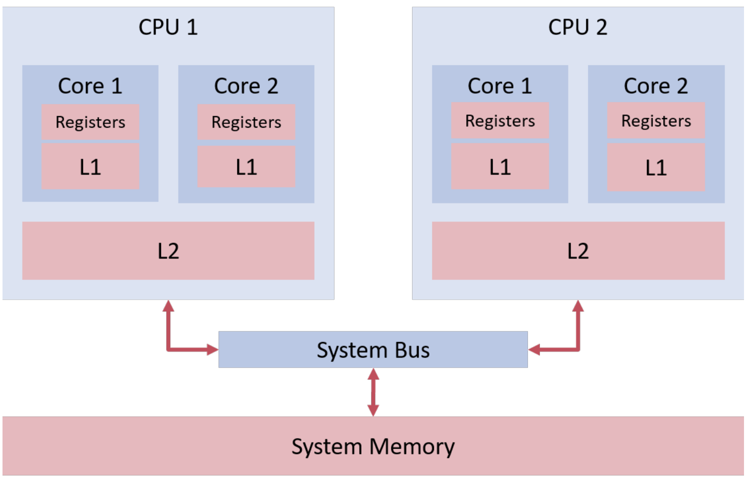
always

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.

Speicherhierarchien



Speicherhierarchien

Registers

L1 Cache

L2 Cache

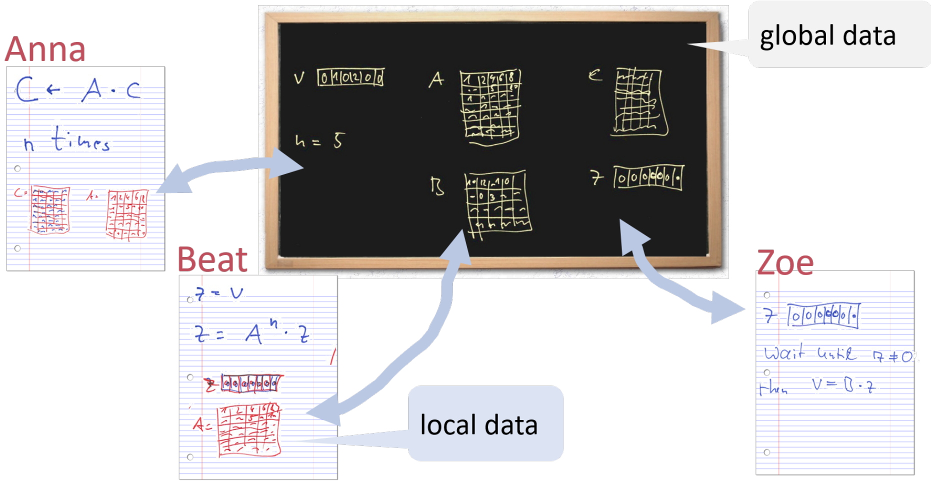
...

System Memory

schnell,kleine Latenz,hohe Kosten, geringe Kapazität

langsam, hohe Latenz, geringe Kosten, hohe Kapazität

Eine Analogie



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.

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
  }
};
```

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
  }
};
```