# Verteilte Systeme

...für C++ Programmierer

Synchronisation

by

Dr. Günter Kolousek

# Threads und gemeinsame Daten

- Ein (Haupt)Vorteil von Threads ist…

# Threads und gemeinsame Daten

- Ein (Haupt)Vorteil von Threads ist…
  Zugriff auf gemeinsame Daten

# Threads und gemeinsame Daten

- Ein (Haupt)Vorteil von Threads ist…
  Zugriff auf gemeinsame Daten
- Regelung des Zugriffes auf gemeinsame Ressourcen von kritischen Abschnitten (engl. critical sections)

# Threads und gemeinsame Daten

- Ein (Haupt)Vorteil von Threads ist…
  Zugriff auf gemeinsame Daten
- Regelung des Zugriffes auf gemeinsame Ressourcen von kritischen Abschnitten (engl. critical sections)
  - $\rightarrow$ Race conditions (…Wettkampfbedingung, Gleichzeitigkeitsbedingung)
    - Ergebnis einer Operation hängt von der zeitlich verschränkten Ausführung mit Operationen ab
  - $\rightarrow$ Synchronisation um wechselseitigen Ausschluss (engl. mutual exclusion) zu erreichen
  - $\rightarrow$ zustandsabhängige Steuerung
  - nicht verteilt vs. verteilt

# Problematik

```cpp
#include <iostream>  // problem.cpp
#include <thread>
using namespace std;
int balance{15};

void withdraw(int amount, bool& success) {
    if (balance >= amount) {
        balance -= amount;
        success = true;
    } else {
        success = false;
    }
}
```

# Problematik – 2

```
int main() {
    bool success1{};
    bool success2{};
    thread t1{withdraw, 10, ref(success1)};
    thread t2{withdraw, 6, ref(success2)};
    t1.join();
    t2.join();
    cout << balance << ' ' << success1 << ' ';
    cout << success2 << endl;
}
```

Erwartetes Ergebnisse wären: `9 0 1` oder `5 1 0`

# Problematik – 2

```
int main() {
    bool success1{};
    bool success2{};
    thread t1{withdraw, 10, ref(success1)};
    thread t2{withdraw, 6, ref(success2)};
    t1.join();
    t2.join();
    cout << balance << ' ' << success1 << ' ';
    cout << success2 << endl;
}
```

Erwartetes Ergebnisse wären: `9 0 1` oder `5 1 0`

allerdings geht auch: `-1 1 1`

# Problematik – 2

```
int main() {
    bool success1{};
    bool success2{};
    thread t1{withdraw, 10, ref(success1)};
    thread t2{withdraw, 6, ref(success2)};
    t1.join();
    t2.join();
    cout << balance << ' ' << success1 << ' ';
    cout << success2 << endl;
}
```

Erwartetes Ergebnisse wären: `9 0 1` oder `5 1 0`

allerdings geht auch: `-1 1 1`

oder z.B. auch: `9 1 1` (!)

# Problematik – 3

| t1 | t2 | balance |
|---|---|---|
| `15 >= 10` | | 15 |
| `balance - 10` | | 15 |
| | `15 > 6` | 15 |
| | `balance - 6` | 15 |
| `balance = 5` | | 5 |
| | `balance = 9` | 9 |
| | `success = true` | |
| `success = true` | | |

# Race Conditions

- mehrere Threads greifen zumindest schreibend auf gemeinsame Ressource zu
  - Write/Write
  - Read/Write
- data race
  - C++ Begriff für Race Condition, die sich auf ein einziges Speicherobjekt bezieht
  - $\rightarrow$ undefiniertes Verhalten!

# Race Conditions – 2

- Write/Write
  mind. 2 Threads schreiben

```
void double() {
    x = x * 2;  // write
}

void halve() {
    x = x / 2;  // write
}
```

# Race Conditions – 3

- Read/Write
  ein Thread liest, einer schreibt

```
void calc_sides(double r, double phi) {
    a = r * sin(phi);  // write
    b = r * cos(phi);
}

void calc_area() {
    A = (a * b) / 2;  // read
}
```

# Lösung

- Erreichung eines wechselseitigen Ausschlusses (engl. mutual exclusion)
- durch Synchronisation
- verschiedene Synchronisationsmechanismen existieren
- in C++ wird hauptsächlich der Synchronisationmechanismus ”Mutex” verwendet!
  - Klasse `mutex`
    - `lock()` und `unlock()`
    - wenn schon gelockt und `lock()`:
      $\rightarrow$ anderer Thread: blockiert
      $\rightarrow$ gleicher Thread: undefiniertes Verhalten
    - `unlock()` nur vom gleichen Thread, ansonsten undefiniertes Verhalten
  - Klasse `recursive_mutex`
- lösen lediglich der data race Situation:
  - `atomic<int> balance{15};` (aus `<atomic>`)

# Mutex

```
#include <iostream>  // mutex.cpp
#include <thread>
#include <mutex>
using namespace std;
int balance{15};
mutex m;
void withdraw(int amount, bool& success) {
    m.lock();
    if (balance >= amount) {
        balance -= amount;
        success = true;
    } else {
        success = false;
    }
    m.unlock();
}
```

# Mutex und Exceptions

Was wäre wenn eine Exception vor `unlock()` …?

# Mutex und Exceptions

Was wäre wenn eine Exception vor `unlock()`…?
abfangen… ist aber auch mühsam (und fehleranfällig)

# Mutex und Exceptions

Was wäre wenn eine Exception vor `unlock()`...?
abfangen... ist aber auch mühsam (und fehleranfällig)

```
#include <iostream>  // mutex2.cpp
#include <thread>
#include <mutex>
using namespace std;
int balance{15};
mutex m;
void withdraw(int amount, bool& success) {
    lock_guard<mutex> guard{m};
    if (balance >= amount) {
        balance -= amount;
        success = true;
    } else { success = false; } }
```

# Begriffe

- Synchronisation

  *Synchronisation (griechisch: syn ≡ „zusammen“, chrónos ≡ „Zeit“) bezeichnet das zeitliche Aufeinander-Abstimmen von Vorgängen, Uhren und Zeitgebern. Synchronisation sorgt dafür, dass Vorgänge gleichzeitig (synchron) oder in einer bestimmten Reihenfolge ablaufen.* *Wikipedia*

# Begriffe

- Synchronisation

  > *Synchronisation (griechisch: syn ≡ „zusammen“, chrónos ≡ „Zeit“) bezeichnet das zeitliche Aufeinander-Abstimmen von Vorgängen, Uhren und Zeitgebern. Synchronisation sorgt dafür, dass Vorgänge gleichzeitig (synchron) oder in einer bestimmten Reihenfolge ablaufen.* *Wikipedia*

  Synchronisation beschreibt ein Verfahren wie Prozesse oder Threads sich untereinander abstimmen, um Aktionen in einer bestimmten Reihenfolge auszuführen.

# Begriffe – 2

- Betriebsmittel, Ressource (engl. resource): Speicher, Dateien, I/O Kanäle, Netzwerkverbindungen, Locks, Prozessor, Bildschirm, Drucker
- Kritischer Abschnitt (engl. critical section): Programmcode von dem auf gemeinsam genutzte Ressourcen zugegriffen wird
- Wechselseitiger Ausschluss (engl. mutual exclusion): Verfahren, das anderen Prozessen (oder Threads) den Zutritt in kritischen Abschnitt verwehrt, solange ein Prozess (oder Thread) sich in solch einem befindet.

# Deadlock

```cpp
#include <iostream>  // deadlock.cpp
#include <thread>
#include <mutex>
using namespace std;
int main() {
    mutex m1{};
    mutex m2{};
    thread t1{[&]() { m1.lock(); m2.lock();  //...
                      m1.unlock(); m2.unlock(); }};
    thread t2{[&]() { m2.lock(); m1.lock();  //...
                      m2.unlock(); m1.unlock(); }};
    t1.join();
    t2.join();
}
```

# Deadlock

```cpp
#include <iostream>  // deadlock.cpp
#include <thread>
#include <mutex>
using namespace std;
int main() {
    mutex m1{};
    mutex m2{};
    thread t1{[&]() { m1.lock(); m2.lock();  //...
                      m1.unlock(); m2.unlock(); }};
    thread t2{[&]() { m2.lock(); m1.lock();  //...
                      m2.unlock(); m1.unlock(); }};
    t1.join();
    t2.join();
}
```

→ Lösung: Locken in gleicher Reihenfolge (geht aber nicht immer)!

# Deadlock – 2

- Deadlock: Eine Situation in der eine Gruppe von Prozessen (Threads) für immer blockiert ist, weil jeder der Prozesse auf Ressourcen wartet, die von einem anderem Prozess in der Gruppe gehalten werden.
- Achtung: Deadlocks auch ohne Locks möglich, z.B.

| t1 | t2 |
|---|---|
| `t2.join();` | |
| | `t1.join();` |

# Deadlock – 3

Notwendige Bedingungen, damit ein Deadlock entsteht (Coffman)

- Circular wait: Zwei oder mehr Prozesse bilden eine geschlossene Kette von Abhängigkeiten insoferne, dass ein Prozess auf die Ressource des nächsten Prozesses wartet.
- Hold and wait: Processe fordern neue Ressourcen an, obwohl sie den Zugriff auf andere Ressourcen behalten.
- Mutual exclusion: Der Zugriff auf die Ressourcen ist exklusiv
- No preemption: Ressourcen können Prozessen nicht entzogen werden.

# Vermeiden eines Deadlocks

… indem eine der Bedingungen nicht erfüllt ist!

- Circular wait: Ressourcen werden in gleicher Reihenfolge angeordnet und so vergeben (siehe oben).
- Hold and wait: Alle Ressourcen werden auf einmal zugeteilt (wenn frei) oder keine Ressourcen werden zugeteilt.
- Mutual exclusion: exklusiven Zugriff z.B. durch Spooling auflösen (z.B. Drucker)
- No preemption: Ressource wird Prozess entzogen und anderem Prozess zugeteilt.

# Funktion std::lock

```
#include <iostream>  // lock.cpp
#include <thread>
#include <mutex>
using namespace std;
int main() {
    mutex m1{};  mutex m2{};
    thread t1{[&]() { lock(m1, m2);  //...
              m1.unlock(); m2.unlock(); }};
    thread t2{[&]() { lock(m1, m2);  //...
              m2.unlock(); m1.unlock(); }};
    t1.join();  t2.join();
}
```

# lock & lock_guard

```cpp
#include <iostream>  // lock.cpp
#include <thread>
#include <mutex>
using namespace std;
int main() {
    mutex m1{};  mutex m2{};
    thread t1{[&]() { lock(m1, m2);
          lock_guard<mutex> lock1(m1, adopt_lock);
          lock_guard<mutex> lock2(m2, adopt_lock);
          /* ... */ }};
    thread t2{[&]() { lock(m1, m2);
          lock_guard<mutex> lock1(m1, adopt_lock);
          lock_guard<mutex> lock2(m2, adopt_lock);
          /* ... */ }};
    t1.join();  t2.join();
}
```

# scoped_lock (ab C++17)

Äquivalent zur vorhergehenden Lösung!

```
#include <iostream>  // scoped_lock.cpp
#include <thread>
#include <mutex>
using namespace std;
int main() {
    mutex m1{};  mutex m2{};
    thread t1{[&]() {
          scoped_lock sl(m1, m2);
          /* ... */ }};
    thread t2{[&]() {
          scoped_lock sl(m1, m2);
          /* ... */ }};
    t1.join();  t2.join();
}
```

# Auflösen eines Deadlocks

1. Erkennen des Deadlocks
2. Deadlock beseitigen
   - Ressource entziehen (siehe oben)
   - Prozess terminieren

# Konkrete Tipps zum Vermeiden

- Locks immer gleichzeitig (atomar) anfordern (`lock` verwenden)
  - nicht immer einfach/möglich!
- keinen weiteren Lock anfordern, wenn schon einer gehalten wird (nested locks)
- keine benutzerdefinierte Funktionen aufrufen, wenn ein Lock gehalten wird
  - dieser könnte einen weiteren Lock anfordern ($\rightarrow$ nested locks)
- Locks in gleicher Reihenfolge anfordern
  - nicht immer einfach/möglich!

# Interface von `mutex`

```
constexpr mutex() noexcept;
mutex(const mutex&) = delete;
void lock();  // system_error
bool try_lock();  // non blocking
void unlock();
```

- d.h. kein Kopieren eines `mutex` möglich
- gibt auch nicht blockierende Lösung mit `try_lock`

# Konkrete Tipps zum Vermeiden – 2

- Vor Eintritt eines Deadlocks einen ”Schritt zurück” (engl. backoff) und eine oder mehrere Locks freiwillig zurückgeben:

```
while (true) {
    m1.lock();
    if (m2.try_lock()) {
        break;
    }
    m1.unlock();
}

// critical section

m2.unlock();
m1.unlock();
```

# Interface von `lock_guard`

```
explicit lock_guard(mutex_type&);
// adopt... assume that mutex is already locked
lock_guard(mutex_type&, adopt_lock_t);
lock_guard(const lock_guard&) = delete;
```

- kann ebenfalls nicht kopiert werden
- gibt aber Lock **immer** frei!!

# Weitere Aspekte

- `lock()` kann von einem Thread nicht mehrmals aufgerufen werden → `recursive_mutex` (sinnvoll bei Methoden von Klassen)
- `lock()` hat kein Timeout → `timed_mutex`
  - außerdem: `recursive_timed_mutex`
- `mutex` und `lock_guard` können nicht in anderen Gültigkeitsbereich verschoben werden → `unique_lock`
- keine Differenzierung in lesende und schreibende Zugriffe
- keine Zugriffskontrolle: jeder kann `lock()` aufrufen

# Grundlegende Probleme

- Deadlock (siehe oben)
- Starvation (dt. verhungern)
  - Thread/Prozess wird andauernd Zugriff auf Ressource verweigert (und kann damit nicht fertig werden)
  - Beispiele
    - stark befahrene Vorrangstraße mit Kreuzung
    - Thread mit niedriger Priorität kommt nicht zur Ausführung…
- Livelock
  - kein Fortschritt von zwei abhängigen Threads, obwohl beide nicht blockiert sind
  - Bsp.: Eingangstüre und 2 Personen wollen eintreten, aber beide wollen jeweils dem Anderen den Vortritt geben

# Livelock

```
#include <thread>  // livelock
#include <mutex>
#include <functional>
using namespace std;
//using namespace std::literals;
void enter(mutex& me, mutex& other) {
    bool entered{};
    while (!entered) {
        me.lock();
      // simulate both reaching door simultaneously
        this_thread::sleep_for(500ms);
        if (other.try_lock()) {// should fail often
            me.unlock();   other.unlock();
            entered = true;
        } else me.unlock();  // ...the other first!
        } }
```

# Livelock - 2

```
int main() {
    mutex a_enters;
    mutex b_enters;

    thread friend_a{enter,
                    ref(a_enters), ref(b_enters)};
    thread friend_b{enter,
                    ref(b_enters), ref(a_enters)};

    friend_a.join();
    friend_b.join();
}
```