

29. Parallel Programming III

Verklemmung (Deadlock) und Verhungern (Starvation)
Producer-Consumer, Konzept des Monitors, Condition Variables

Verklemmung (Deadlock) Motivation

```
class BankAccount {  
    int balance = 0;  
    std::recursive_mutex m;  
    using guard = std::lock_guard<std::recursive_mutex>;  
public:  
    ...  
    void withdraw(int amount) { guard g(m); ... }  
    void deposit(int amount){ guard g(m); ... }  
  
    void transfer(int amount, BankAccount& to){  
        guard g(m);  
        withdraw(amount);  
        to.deposit(amount);  
    }  
};
```

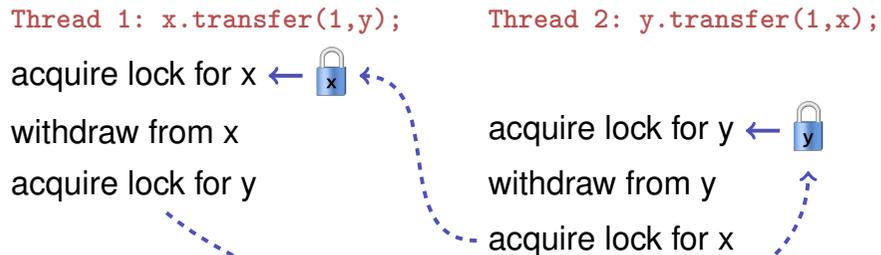
Problem?

907

908

Verklemmung (Deadlock) Motivation

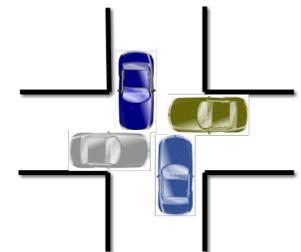
Betrachte BankAccount Instanzen x und y



909

Deadlock

Deadlock: zwei oder mehr Prozesse sind gegenseitig blockiert, weil jeder Prozess auf einen anderen Prozess warten muss, um fortzufahren.



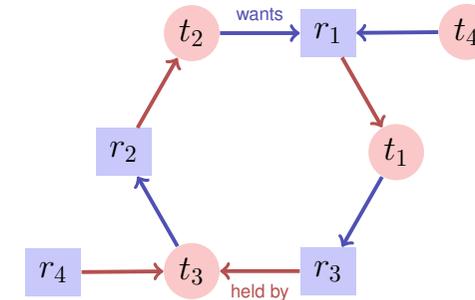
910

Threads und Ressourcen

- Grafisch: Threads t und Ressourcen (Locks) r
- Thread t versucht Resource a zu bekommen: $t \rightarrow a$
- Resource b wird von Thread q gehalten: $s \leftarrow b$

Deadlock – Erkennung

Ein Deadlock für Threads t_1, \dots, t_n tritt auf, wenn der gerichtete Graph, der die Beziehung der n threads und Ressourcen r_1, \dots, r_m beschreibt, einen Kreis enthält.



911

912

Techniken

- **Deadlock Erkennung** findet die Zyklen im Abhängigkeitsgraph. Deadlock kann normalerweise nicht geheilt werden: Freigeben von Locks resultiert in inkonsistentem Zustand.
- **Deadlock Vermeidung** impliziert, dass Zyklen nicht auftreten können
 - Grobere Granularität "one lock for all"
 - Zwei-Phasen-Locking mit Retry-Mechanismus
 - Lock-Hierarchien
 - ...
 - **Anordnen der Ressourcen**

Zurück zum Beispiel

```
class BankAccount {
    int id; // account number, also used for locking order
    std::recursive_mutex m; ...
public:
    ...
    void transfer(int amount, BankAccount& to){
        if (id < to.id){
            guard g(m); guard h(to.m);
            withdraw(amount); to.deposit(amount);
        } else {
            guard g(to.m); guard h(m);
            withdraw(amount); to.deposit(amount);
        }
    }
};
```

913

914

C++11 Stil

```
class BankAccount {  
    ...  
    std::recursive_mutex m;  
    using guard = std::lock_guard<std::recursive_mutex>;  
public:  
    ...  
    void transfer(int amount, BankAccount& to){  
        std::lock(m,to.m); // lock order done by C++  
        // tell the guards that the lock is already taken:  
        guard g(m,std::adopt_lock); guard h(to.m,std::adopt_lock);  
        withdraw(amount);  
        to.deposit(amount);  
    }  
};
```

915

Übrigens...

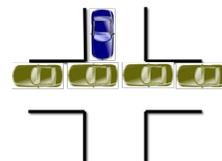
```
class BankAccount {  
    int balance = 0;  
    std::recursive_mutex m;  
    using guard = std::lock_guard<std::recursive_mutex>;  
public:  
    ...  
    void withdraw(int amount) { guard g(m); ... }  
    void deposit(int amount){ guard g(m); ... }  
  
    void transfer(int amount, BankAccount& to){  
        withdraw(amount);  
        to.deposit(amount);  
    }  
};
```

Das hätte auch funktioniert. Allerdings verschwindet dann kurz das Geld, was inakzeptabel ist (kurzzeitige Inkonsistenz!)

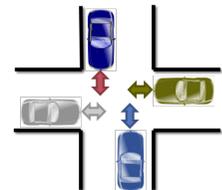
916

Starvation und Livelock

Starvation: der wiederholte, erfolglose Versuch eine zwischenzeitlich freigegebene Ressource zu erhalten, um die Ausführung fortzusetzen.

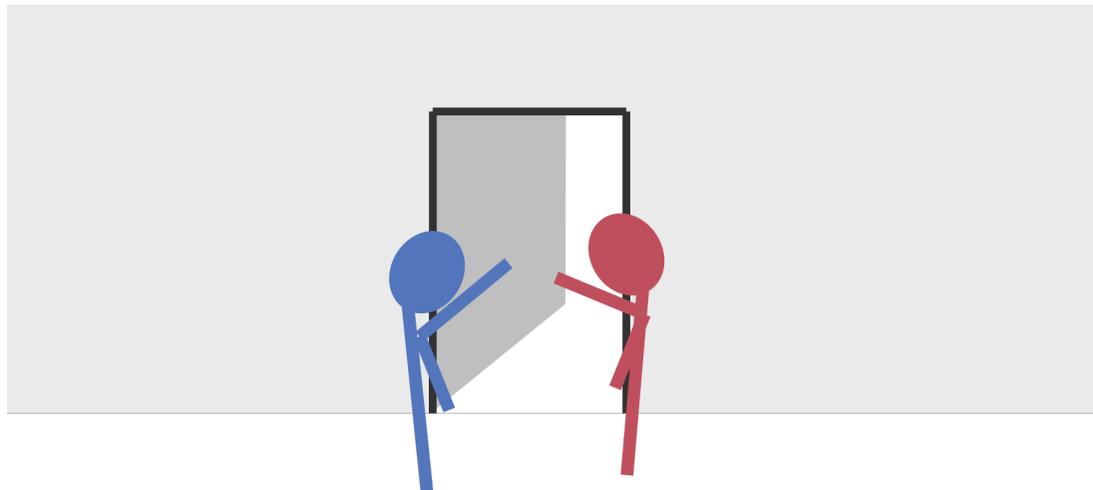


Livelock: konkurrierende Prozesse erkennen einen potentiellen Deadlock, machen aber keinen Fortschritt beim Auflösen des Problems.



917

Politelock



918

Produzenten-Konsumenten Problem

Zwei (oder mehr) Prozesse, Produzenten und Konsumenten von Daten, sollen mit Hilfe einer Datenstruktur entkoppelt werden.
Fundamentale Datenstruktur für den Bau von Software-Pipelines!



Sequentielle Implementation (unbeschränkter Buffer)

```
class BufferS {
    std::queue<int> buf;
public:
    void put(int x){
        buf.push(x);
    }

    int get(){
        while (buf.empty()){} // wait until data arrive
        int x = buf.front();
        buf.pop();
        return x;
    }
};
```

nicht Thread-sicher

919

920

Wie wärs damit?

```
class Buffer {
    std::recursive_mutex m;
    using guard = std::lock_guard<std::recursive_mutex>;
    std::queue<int> buf;
public:
    void put(int x){ guard g(m);
        buf.push(x);
    }
    int get(){ guard g(m);
        while (buf.empty()){}
        int x = buf.front();
        buf.pop();
        return x;
    }
};
```

Deadlock

921

Ok, so?

```
void put(int x){
    guard g(m);
    buf.push(x);
}
int get(){
    m.lock();
    while (buf.empty()){
        m.unlock();
        m.lock();
    }
    int x = buf.front();
    buf.pop();
    m.unlock();
    return x;
}
```

Ok, das geht, verschwendet aber CPU Zeit!

922

Besser?

```
void put(int x){
    guard g(m);
    buf.push(x);
}
int get(){
    m.lock();
    while (buf.empty()){
        m.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
        m.lock();
    }
    int x = buf.front(); buf.pop();
    m.unlock();
    return x;
}
```

Ok, etwas besser. Limitiert aber die Reaktivität!

Moral

Wir wollen das Warten auf eine Bedingung nicht selbst implementieren müssen.

Dafür gibt es bereits einen Mechanismus: *Bedingungsvariablen (condition variables)*.

Das zugrunde liegende Konzept nennt man *Monitor*.

923

924

Monitor

Monitor Abstrakte Datenstruktur, die mit einer Menge Operationen ausgestattet ist, die im gegenseitigen Ausschluss arbeiten und synchronisiert werden können.

Erfunden von C.A.R. Hoare und Per Brinch Hansen (cf. Monitors – An Operating System Structuring Concept, C.A.R. Hoare 1974)

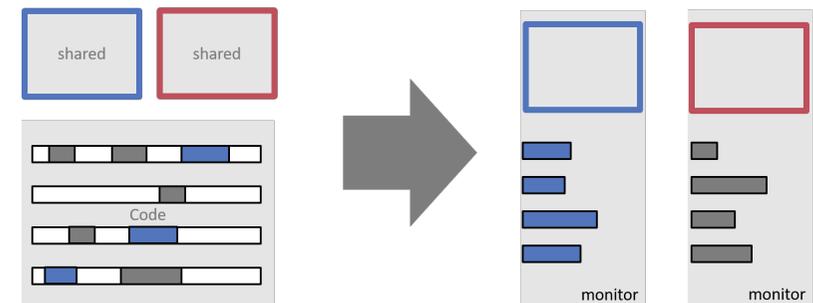


C.A.R. Hoare,
*1934



Per Brinch Hansen
(1938-2007)

Monitors vs. Locks



925

926

Monitor und Bedingungen

Ein Monitor stellt, zusätzlich zum gegenseitigen Ausschluss, folgenden Mechanismus bereit:

Warten auf Bedingungen: Ist eine Bedingung nicht erfüllt, dann

- Gib das Lock auf
- Warte auf die Erfüllung der Bedingung
- Prüfe die Erfüllung der Bedingung wenn ein Signal gesendet wird

Signalisieren: Thread, der die Bedingung wahr machen könnte:

- Sende Signal zu potentiell wartenden Threads

Bedingungsvariablen

```
#include <mutex>
#include <condition_variable>
...

class Buffer {
    std::queue<int> buf;

    std::mutex m;
    // need unique_lock guard for conditions
    using guard = std::unique_lock<std::mutex>;
    std::condition_variable cond;
public:
    ...
};
```

927

928

Bedingungsvariablen

```
class Buffer {
...
public:
    void put(int x){
        guard g(m);
        buf.push(x);
        cond.notify_one();
    }
    int get(){
        guard g(m);
        cond.wait(g, [&]{return !buf.empty();});
        int x = buf.front(); buf.pop();
        return x;
    }
};
```

929

Technische Details

- Ein Thread, der mit `cond.wait` wartet, läuft höchstens sehr kurz auf einem Core. Danach belastet er das System nicht mehr und "schläft".
- Der Notify (oder Signal-) Mechanismus weckt schlafende Threads auf, welche nachfolgend ihre Bedingung prüfen.
 - `cond.notify_one` signalisiert *einen* wartenden Threads.
 - `cond.notify_all` signalisiert *alle* wartende Threads. Benötigt, wenn wartende Threads potentiell auf *verschiedene* Bedingungen warten.

930

Technische Details

- In vielen anderen Sprachen gibt es denselben Mechanismus. Das Prüfen von Bedingungen (in einem Loop!) muss der Programmierer dort selbst implementieren.

Java Beispiel

```
synchronized long get() {
    long x;
    while (isEmpty())
        try {
            wait ();
        } catch (InterruptedException e) { }
    x = doGet();
    return x;
}

synchronized put(long x){
    doPut(x);
    notify ();
}
```

931

Übrigens: mit bounded Buffer..

```
class Buffer {
    ...
    CircularBuffer<int,128> buf; // from lecture 6
public:
    void put(int x){ guard g(m);
        cond.wait(g, [&]{return !buf.full();});
        buf.put(x);
        cond.notify_all();
    }
    int get(){ guard g(m);
        cond.wait(g, [&]{return !buf.empty();});
        cond.notify_all();
        return buf.get();
    }
};
```

932