

## 32. Parallel Programming III

---

Deadlock and Starvation Producer-Consumer, The concept of the monitor, Condition Variables [Deadlocks : Williams, Kap. 3.2.4-3.2.5] [Condition Variables: Williams, Kap. 4.1]

# 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);
    }
};
```

# 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?

# Deadlock Motivation

Suppose BankAccount instances **x** and **y**

**Thread 1: x.transfer(1,y);**

acquire lock for x

withdraw from x

acquire lock for y

**Thread 2: y.transfer(1,x);**

acquire lock for y


withdraw from y

acquire lock for x

# Deadlock Motivation

Suppose BankAccount instances **x** and **y**

**Thread 1:** `x.transfer(1,y);`

acquire lock for x ← 

withdraw from x

acquire lock for y

**Thread 2:** `y.transfer(1,x);`

acquire lock for y

withdraw from y

acquire lock for x

# Deadlock Motivation

Suppose BankAccount instances **x** and **y**


**Thread 1: x.transfer(1,y);**

acquire lock for x ← 

withdraw from x

acquire lock for y

**Thread 2: y.transfer(1,x);**

acquire lock for y ← 

withdraw from y

acquire lock for x

# Deadlock Motivation

Suppose BankAccount instances **x** and **y**

**Thread 1:** `x.transfer(1,y);`

acquire lock for x ← 

withdraw from x

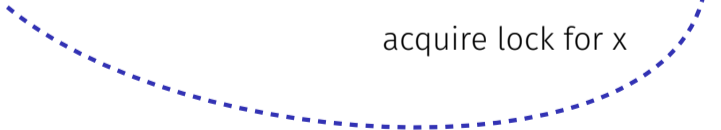
acquire lock for y

**Thread 2:** `y.transfer(1,x);`

acquire lock for y ← 

withdraw from y


acquire lock for x



# Deadlock Motivation

Suppose BankAccount instances **x** and **y**


**Thread 1:** `x.transfer(1,y);`

acquire lock for x ←  ←

withdraw from x

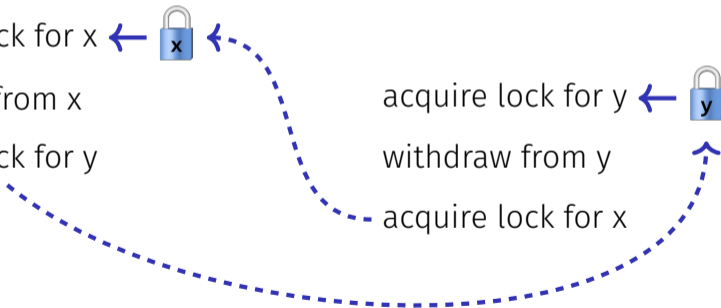
acquire lock for y

**Thread 2:** `y.transfer(1,x);`

acquire lock for y ←  ←

withdraw from y

acquire lock for x







# Deadlock

**Deadlock:** two or more processes are mutually blocked because each process waits for another of these processes to proceed.

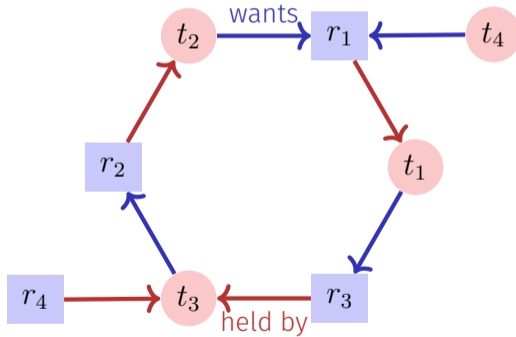


# Threads and Resources

- Grafically  $t$  and Resources (Locks)  $r$
- Thread  $t$  attempts to acquire resource  $a$ :   $t \rightarrow a$
- Resource  $b$  is held by thread  $q$ :   $s \leftarrow b$

# Deadlock – Detection

A deadlock for threads  $t_1, \dots, t_n$  occurs when the graph describing the relation of the  $n$  threads and resources  $r_1, \dots, r_m$  contains a cycle.



# Techniques

- **Deadlock detection** detects cycles in the dependency graph. Deadlocks can in general not be healed: releasing locks generally leads to inconsistent state
- **Deadlock avoidance** amounts to techniques to ensure a cycle can never arise
  - Coarser granularity “one lock for all”
  - Two-phase locking with retry mechanism
  - Lock Hierarchies
  - ...
  - **Resource Ordering**

# Back to the Example

```
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);
        }
    }
};
```

# C++11 Style

```
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);
    }
};
```

## By the way...

```
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);
    }
};
```

## By the way...

```
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);
    }
};
```

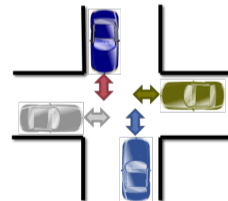
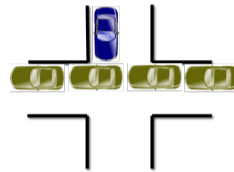
This would have worked here also. But then for a very short amount of time, money disappears, which does not seem acceptable (transient inconsistency!)



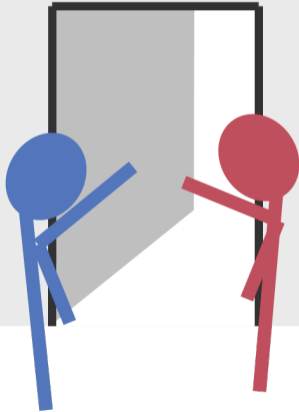
# Starvation und Livelock

**Starvation:** the repeated but unsuccessful attempt to acquire a resource that was recently (transiently) free.

**Livelock:** competing processes are able to detect a potential deadlock but make no progress while trying to resolve it.



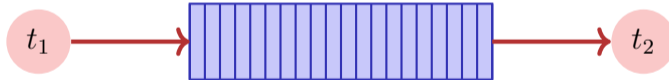
# Politelock



# Producer-Consumer Problem

Two (or more) processes, producers and consumers of data should become decoupled by some data structure.

Fundamental Data structure for building pipelines in software.



## Sequential implementation (unbounded 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;
    }
};
```

## Sequential implementation (unbounded 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;
    }
};
```

not thread-safe

# How about this?

```
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;
        }
    };
};
```

# How about this?

```
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

# Well, then this?

```
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;
}
```



# Well, then this?

```
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 this works, but it wastes CPU time.

# Better?

```
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;
}
```

# Better?

```
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 a little bit better, limits reactivity though.

We do not want to implement waiting on a condition ourselves.  
There already is a mechanism for this: **condition variables**.  
The underlying concept is called **Monitor**.

# Monitor

**Monitor** abstract data structure equipped with a set of operations that run in mutual exclusion and that can be synchronized.

Invented by C.A.R. Hoare and Per Brinch Hansen (cf. Monitors – An Operating System Structuring Concept, C.A.R. Hoare 1974)

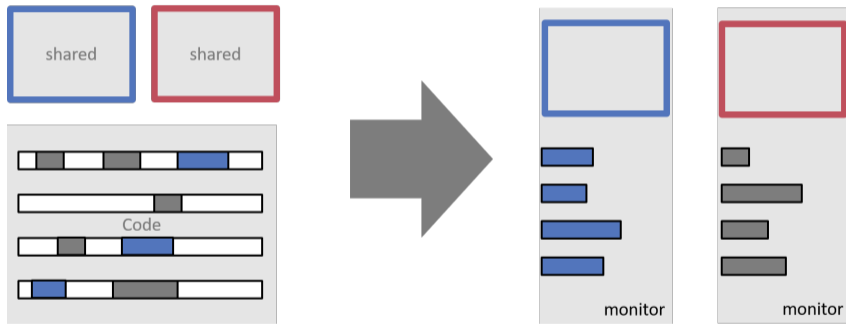


C.A.R. Hoare,  
\*1934



Per Brinch  
Hansen  
(1938-2007)

# Monitors vs. Locks



# Monitor and Conditions

Monitors provide, in addition to mutual exclusion, the following mechanism:

**Waiting on conditions:** If a condition does not hold, then

- Release the monitor lock
- Wait for the condition to become true
- Check the condition when a signal is raised

**Signalling:** Thread that might make the condition true:

- Send signal to potentially waiting threads

# Condition Variables

```
#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:
    ...
};
```



# Condition Variables

```
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;  
    }  
};
```

# Technical Details

- A thread that waits using **cond.wait** runs at most for a short time on a core. After that it does not utilize compute power and “sleeps”.
- The notify (or signal-) mechanism wakes up sleeping threads that subsequently check their conditions.
  - **cond.notify\_one** signals *one* waiting thread
  - **cond.notify\_all** signals *all* waiting threads. Required when waiting threads wait potentially on *different* conditions.

# Technical Details

- Many other programming languages offer the same kind of mechanism. The checking of conditions (in a loop!) has to be usually implemented by the programmer.

## Java Example

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

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

## By the way, using a 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();
    }
};
```