

# 28. Parallel Programming II

Shared Memory, Concurrency, Excursion: lock algorithm (Peterson), Mutual Exclusion 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]

## **28.1 Shared Memory, Concurrency**

# Sharing Resources (Memory)

- Up to now: fork-join algorithms: data parallel or divide-and-conquer
- Simple structure (data independence of the threads) to avoid race conditions
- Does not work any more when threads access shared memory.

# Managing state

Managing state: Main challenge of concurrent programming.

Approaches:

- Immutability, for example constants.
- Isolated Mutability, for example thread-local variables, stack.
- Shared mutable data, for example references to shared memory, global variables

# Protect the shared state

- Method 1: locks, guarantee exclusive access to shared data.
- Method 2: lock-free data structures, exclusive access with a much finer granularity.
- Method 3: transactional memory (not treated in class)

# Canonical Example

```
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.
};
```

(correct in a single-threaded world)

# Bad Interleaving

Parallel call to `withdraw(100)` on the same account

Thread 1

```
int b = getBalance();
```

```
setBalance(b-amount);
```

Thread 2

```
int b = getBalance();
```

```
setBalance(b-amount);
```



# Tempting Traps

WRONG:

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

Bad interleavings cannot be solved with a repeated reading



# Tempting Traps

also WRONG:

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

Assumptions about atomicity of operations are almost always wrong

# Mutual Exclusion

We need a concept for mutual exclusion

*Only one thread* may execute the operation withdraw *on the same account* at a time.

The programmer has to make sure that mutual exclusion is used.

# More Tempting Traps

```
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
};
```

# More Tempting Traps


```
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
};
```

does not work!

# Just moved the problem!

## Thread 1



```
while (busy); //spin  
  
busy = true;  
  
int b = getBalance();  
  
setBalance(b - amount);
```

## Thread 2

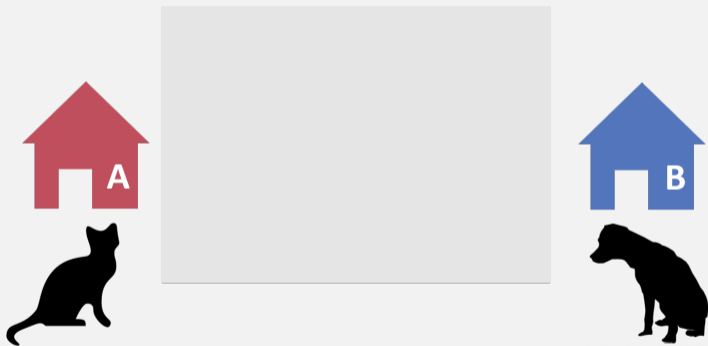
```
while (busy); //spin  
  
busy = true;  
  
int b = getBalance();  
setBalance(b - amount);
```

# How ist this correctly implemented?

- We use *locks* (mutexes) from libraries
- They use hardware primitives, *Read-Modify-Write* (RMW) operations that can, in an atomic way, read and write depending on the read result.
- Without RMW Operations the algorithm is non-trivial and requires at least atomic access to variable of primitive type.

## **28.2 Excursion: lock algorithm**

# Alice's Cat vs. Bob's Dog

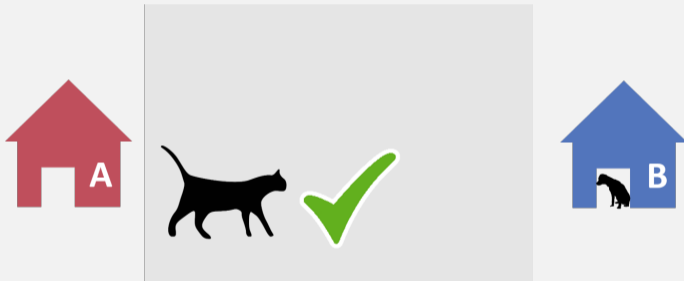




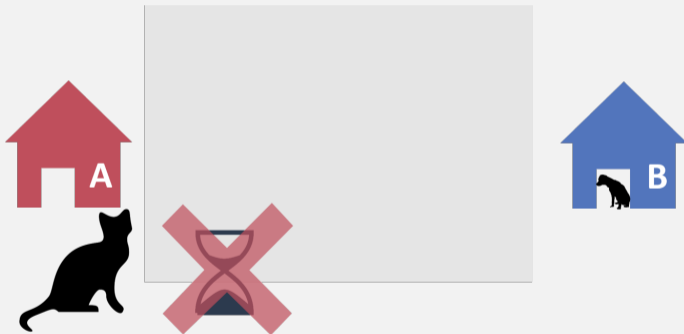
# Required: Mutual Exclusion



# Required: Mutual Exclusion



# Required: No Lockout When Free



# Communication Types

- Transient: Parties participate at the same time

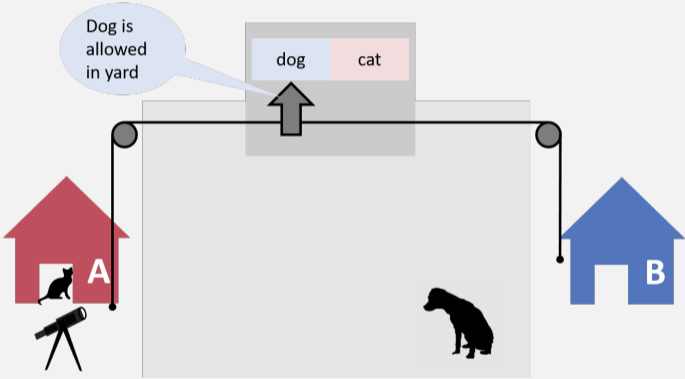


- Persistent: Parties participate at different times

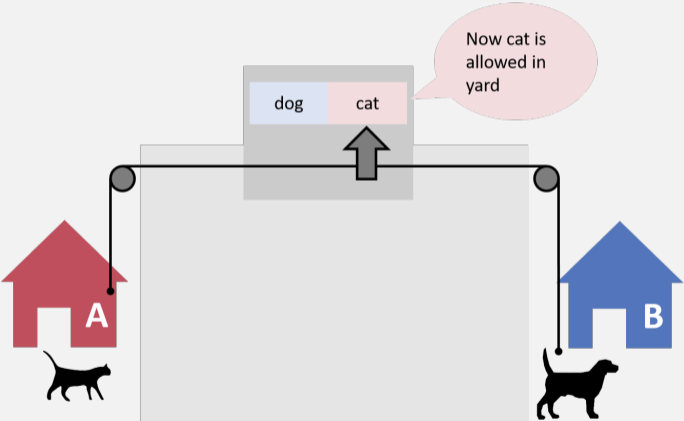


Mutual exclusion: persistent communication

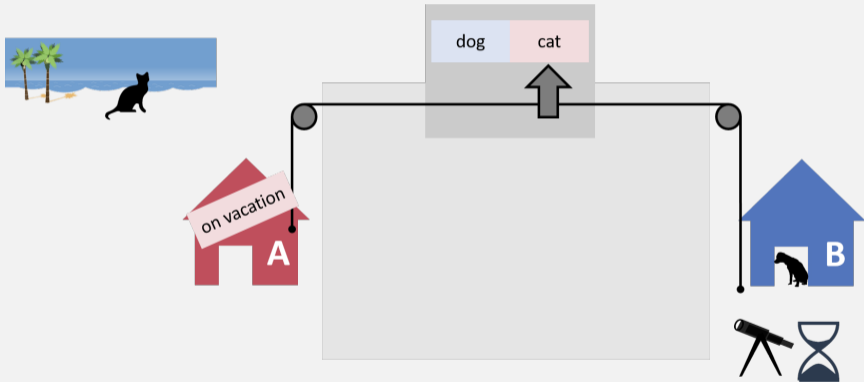
# Communication Idea 1



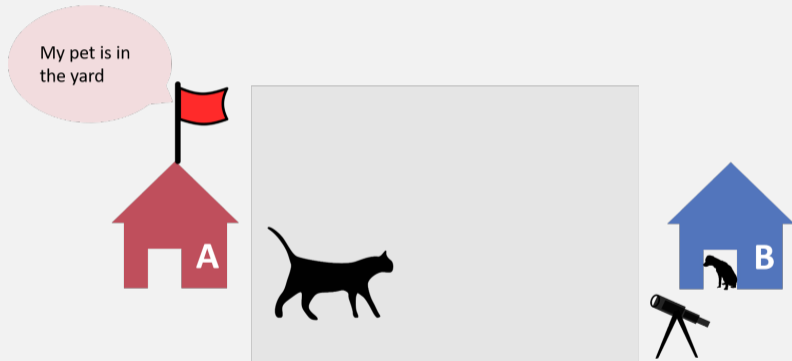
# Access Protocol



# Problem!

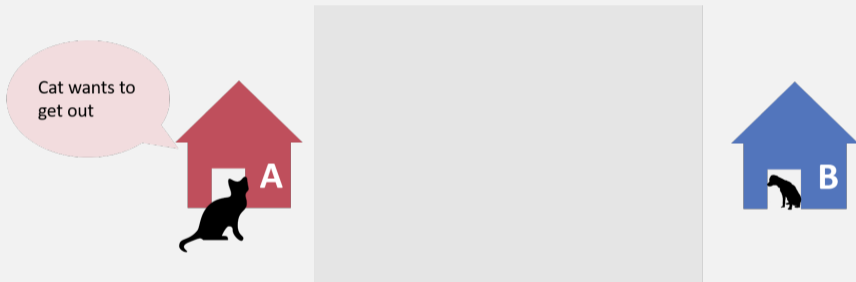


# Communication Idea 2

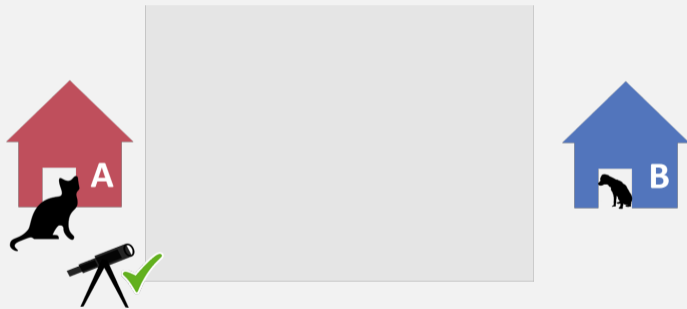




# Access Protocol 2.1



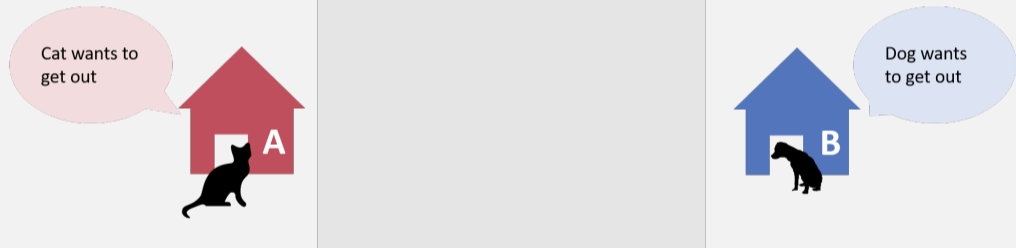
# Access Protocol 2.1



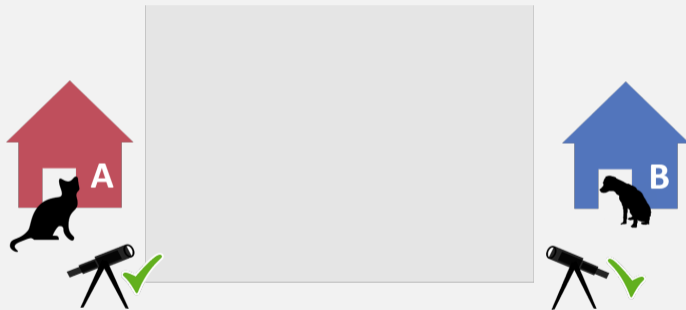
# Access Protocol 2.1



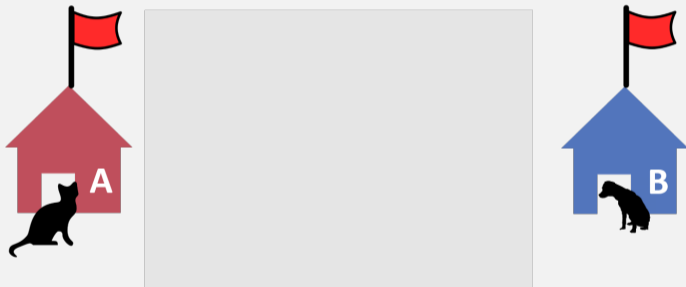
# Different Scenario



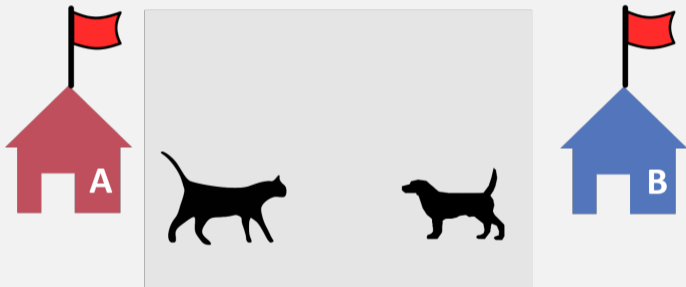
# Different Scenario



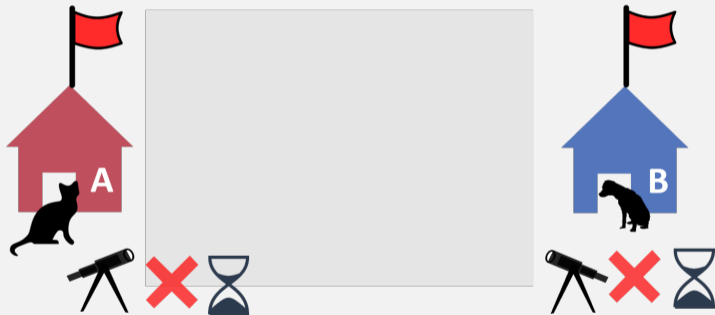
# Different Scenario



# Problem: No Mutual Exclusion

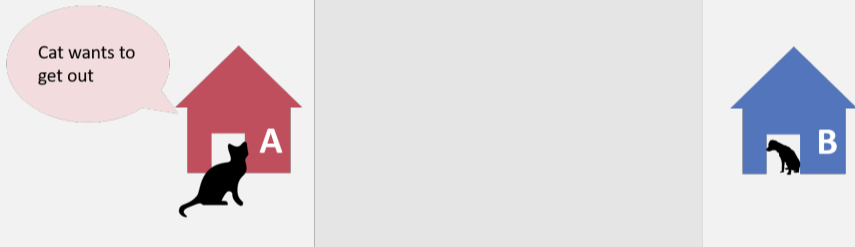


# Checking Flags Twice: Deadlock

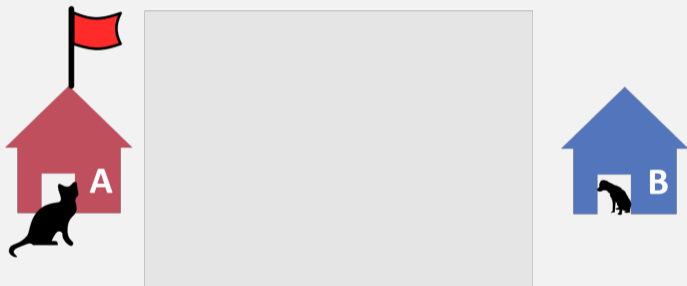




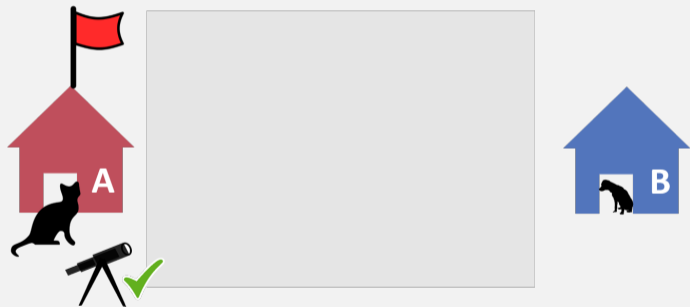
# Access Protocol 2.2



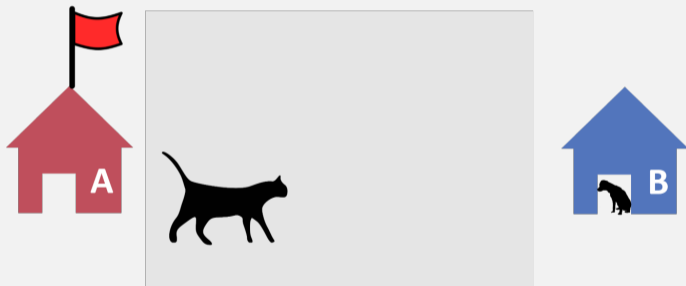
# Access Protocol 2.2



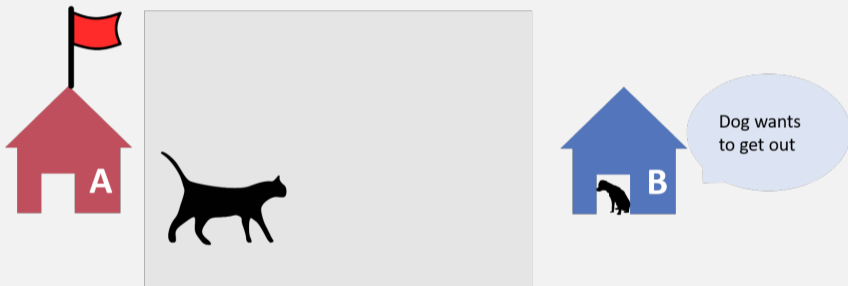
# Access Protocol 2.2



# Access Protocol 2.2



# Access Protocol 2.2



# Access Protocol 2.2



# Access Protocol 2.2



# Access Protocol 2.2:provably correct

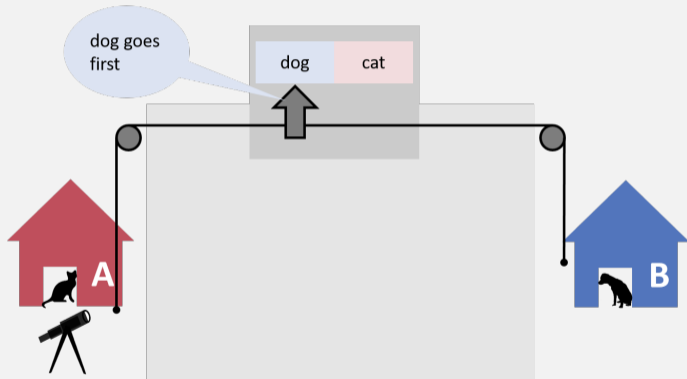




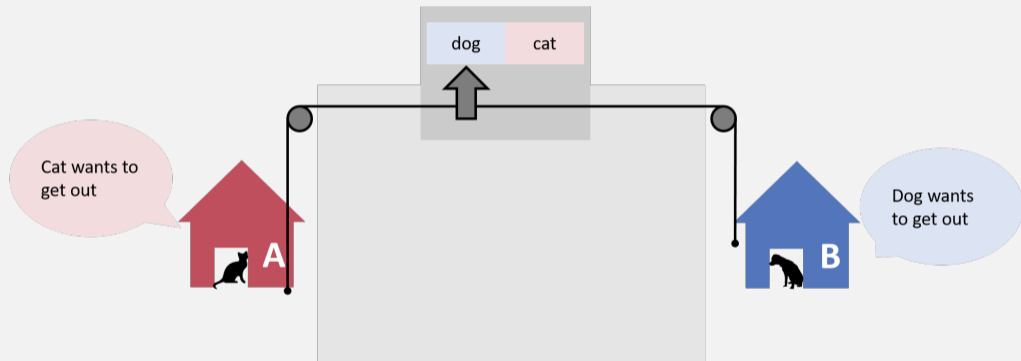
# Weniger schwerwiegend: Starvation



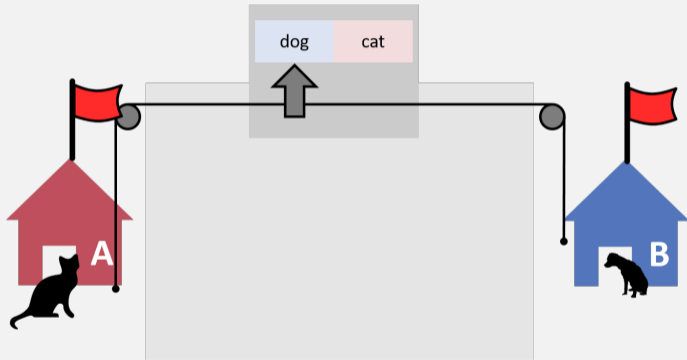
# Final Solution



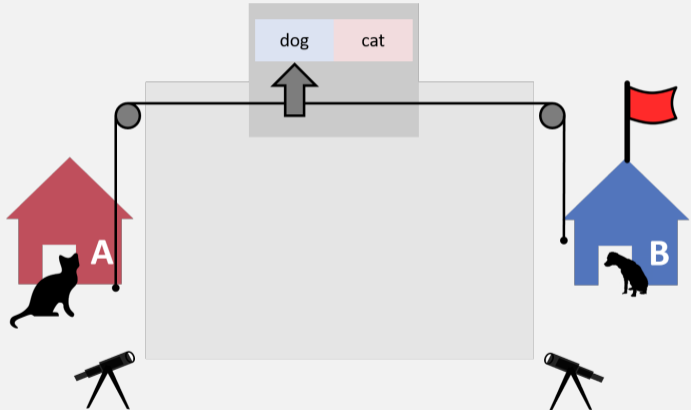
# Final Solution



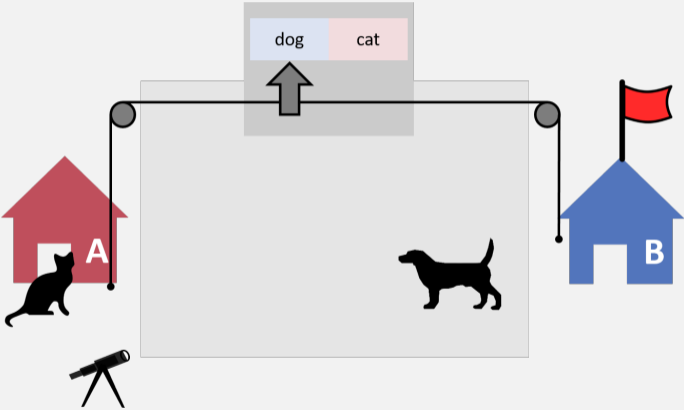
# Final Solution



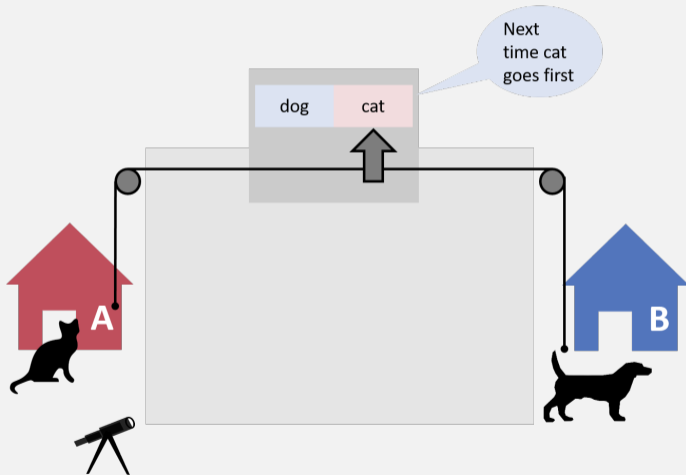
# Final Solution



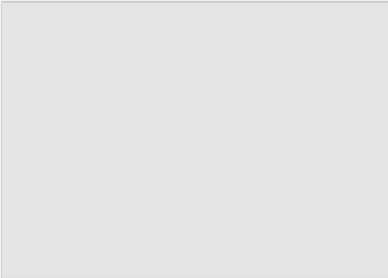
# Final Solution



# Final Solution



# General Problem of Locking remains





# Peterson's Algorithm<sup>54</sup>

for two processes is provable correct and free from 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
```

---

<sup>54</sup>not relevant for the exam

# Peterson's Algorithm<sup>54</sup>

for two processes is provable correct and free from 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
```

The code assumes that the access to flag / victim is atomic and particularly linearizable or sequential consistent. An assumption that – as we will see below – is not necessarily given for normal variables. The Peterson-lock is not used on modern hardware.

---

<sup>54</sup>not relevant for the exam

## 28.3 Mutual Exclusion

# Critical Sections and Mutual Exclusion

## *Critical Section*

Piece of code that may be executed by at most one process (thread) at a time.

## *Mutual Exclusion*

Algorithm to implement a critical section

```
acquire_mutex(); // entry algorithm\  
... // critical section\  
release_mutex(); // exit algorithm
```

# Required Properties of Mutual Exclusion

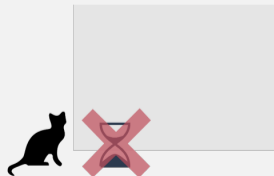
## Correctness (Safety)

- At most one process executes the critical section code



## Liveness

- Acquiring the mutex must terminate in finite time when no process executes in the critical section



# Almost Correct

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

# Almost Correct

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

What if an exception occurs?

# RAII Approach

```
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  
};
```



# RAII Approach

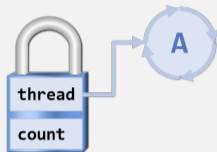
```
class BankAccount {
    int balance = 0;
    std::mutex m;
public:
    ...
    void withdraw(int amount) {
        std::lock_guard<std::mutex> guard(m);
        int b = getBalance();
        setBalance(b - amount);
    } // Destruction of guard leads to unlocking m
};
```

What about getBalance / setBalance?

# Reentrant Locks

## Reentrant Lock (recursive lock)

- remembers the currently affected thread;
- provides a counter
  - Call of lock: counter incremented
  - Call of unlock: counter is decremented. If counter = 0 the lock is released.



# Account with reentrant 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);
    }
};
```

## 28.4 Race Conditions

# Race Condition

- A *race condition* occurs when the result of a computation depends on scheduling.
- We make a distinction between *bad interleavings* and *data races*
- *Bad interleavings* can occur even when a mutex is used.

# Example: Stack

Stack with correctly synchronized access:

```
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

Forgot to implement peek. Like this?

```
template <typename T>
T peek (stack<T> &s){
    T value = s.pop();
    s.push(value);
    return value;
}
```

# Peek

Forgot to implement peek. Like this?

```
template <typename T>
T peek (stack<T> &s){
    T value = s.pop();
    s.push(value);
    return value;
}
```

not thread-safe!



# Peek

Forgot to implement peek. Like this?

```
template <typename T>
T peek (stack<T> &s){
    T value = s.pop();
    s.push(value);
    return value;
}
```

*not thread-safe!*

Despite its questionable style the code is correct in a sequential world. Not so in concurrent programming.

# Bad Interleaving!

Initially empty stack  $s$ , only shared between threads 1 and 2.

Thread 1 pushes a value and checks that the stack is then non-empty. Thread 2 reads the topmost value using `peek()`.

Thread 1

```
s.push(5);
```

```
assert(!s.isEmpty());
```

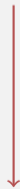
Thread 2

```
int value = s.pop();
```

```
s.push(value);
```

```
return value;
```

$t$



# The fix

Peek must be protected with the same lock as the other access methods

# Bad Interleavings

Race conditions as bad interleavings can happen on a high level of abstraction

In the following we consider a different form of race condition: data race.

# How about this?

```
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;
    }
}
```

# How about this?

```
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;
    }
}
```

*not thread-safe!*

# Why wrong?

It looks like nothing can go wrong because the update of count happens in a “tiny step”.

But this code is still wrong and depends on language-implementation details you cannot assume.

This problem is called *Data-Race*

Moral: *Do not introduce a data race, even if every interleaving you can think of is correct. Don't make assumptions on the memory order.*

# A bit more formal

*Data Race* (low-level Race-Conditions) Erroneous program behavior caused by insufficiently synchronized accesses of a shared resource by multiple threads, e.g. Simultaneous read/write or write/write of the same memory location

*Bad Interleaving* (High Level Race Condition) Erroneous program behavior caused by an unfavorable execution order of a multithreaded algorithm, even if that makes use of otherwise well synchronized resources.



# We look deeper

```
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);  
    }  
}
```

# We look deeper

```
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);  
    }  
}
```

Can this fail?

# We look deeper

```
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);  
    }  
}
```

There is no interleaving of f and g that would cause the assertion to fail:

- A B C D ✓
- A C B D ✓
- A C D B ✓
- C A B D ✓
- C C D B ✓
- C D A B ✓

**It can nevertheless fail!**

Can this fail?

# One Reason: Memory Reordering

*Rule of thumb:* Compiler and hardware allowed to make changes that do not affect the *semantics of a sequentially* executed program

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

$\iff$   
sequentially equivalent

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

# From a Software-Perspective

Modern compilers do not give guarantees that a global ordering of memory accesses is provided as in the sourcecode:

- Some memory accesses may be even optimized away completely!
- Huge potential for optimizations – and for errors, when you make the wrong assumptions

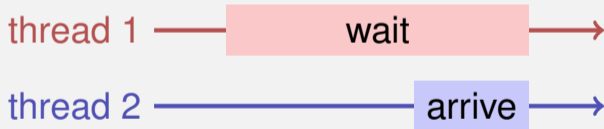
# Example: Self-made Rendezvous

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

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

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

Assume thread 1 calls wait, later thread 2 calls arrive. What happens?



# Compilation

## Source

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

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

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

## Without optimisation

```
wait:  
movl $0x1, x  
test: ←  
mov x, %eax  
cmp $0x1, %eax  
je test ← if equal
```

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

## With optimisation

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

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

# Hardware Perspective

Modern multiprocessors do not enforce global ordering of all instructions for performance reasons:

- Most processors have a pipelined architecture and can execute (parts of) multiple instructions simultaneously. They can even reorder instructions internally.
- Each processor has a local cache, and thus loads/stores to shared memory can become visible to other processors at different times



# Memory Hierarchy

Registers

fast, low latency, high cost, low capacity

L1 Cache

L2 Cache

...

System Memory

slow, high latency, low cost, high capacity

# An Analogy

Anna

$C \leftarrow A \cdot C$   
n times

$C =$ 

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

 $A =$ 

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

Beat

$z = V$   
 $z = A^n \cdot z$

$z =$ 

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

$A =$ 

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

$V =$ 

0	1	0	2	0	1	0
---	---	---	---	---	---	---

 $A =$ 

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

 $C =$ 

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

$h = 5$

$B =$ 

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

 $z =$ 

0	0	0	0	0	0	0
---	---	---	---	---	---	---

global data

Zoe

$z =$ 

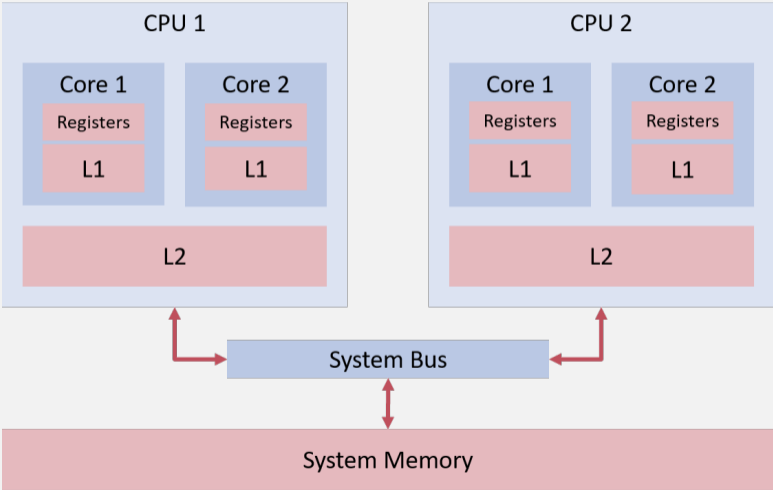
0	0	0	0	0	0	0
---	---	---	---	---	---	---

wait until  $z \neq 0$

then  $V = B \cdot z$

local data

# Schematic



# Memory Models

When and if effects of memory operations become visible for threads, depends on hardware, runtime system and programming language.

# Memory Models

When and if effects of memory operations become visible for threads, depends on hardware, runtime system and programming language.

A *memory model* (e.g. that of C++) provides minimal guarantees for the effect of memory operations

- leaving open possibilities for optimisation
- containing guidelines for writing thread-safe programs

# Memory Models

When and if effects of memory operations become visible for threads, depends on hardware, runtime system and programming language.

A *memory model* (e.g. that of C++) provides minimal guarantees for the effect of memory operations

- leaving open possibilities for optimisation
- containing guidelines for writing thread-safe programs

For instance, C++ provides *guarantees when synchronisation with a mutex* is used.

# Fixed

```
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

Here also possible:

```
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
    }
};
```