

6. Concurrency and Distribution

Overview:

6.1 Thread-based concurrency

6.2 Message-based concurrency

6.3 Distributed OO-programming

The object-oriented paradigm supports concurrency and distribution:

- Objects can work concurrently, communicating by messages (active objects).
- Objects can have different locations distributed to multiple computers.

That is, objects could be the basis for concurrency.

In practice, however,

- local concurrency is expressed by threads (Ausführungsstränge), i.e. „objects are executed“,
- non-local concurrency is handled by mechanisms for *remote method invocation* (*entfernter Methoden-Aufruf*).

6.1 Thread-based Concurrency

Most modern OO-languages use threads to express concurrent behavior. We discuss here the thread-model of Java.

Explanation: (Thread)

A **thread** is an abstract entity that

- is created at program start or by special statements,
- can terminate, wait, block, ...
- executes statements of a program (\rightarrow *action*),
- leads to a sequence of actions modifying *the* state,
- can communicate with other threads
- have a local state and can access global state,
- can run interleaved or in parallel with other threads.



Remark:

Thread-models essentially differ in how they realize the above six aspects in the programming language and its implementation model.



Java-Threads:

- Threads are represented by objects of class Thread. This allows to create and control them like any other object.
- The special method “start” starts new threads and returns immediately.

```
interface Runnable {  
    void run( );  
}
```

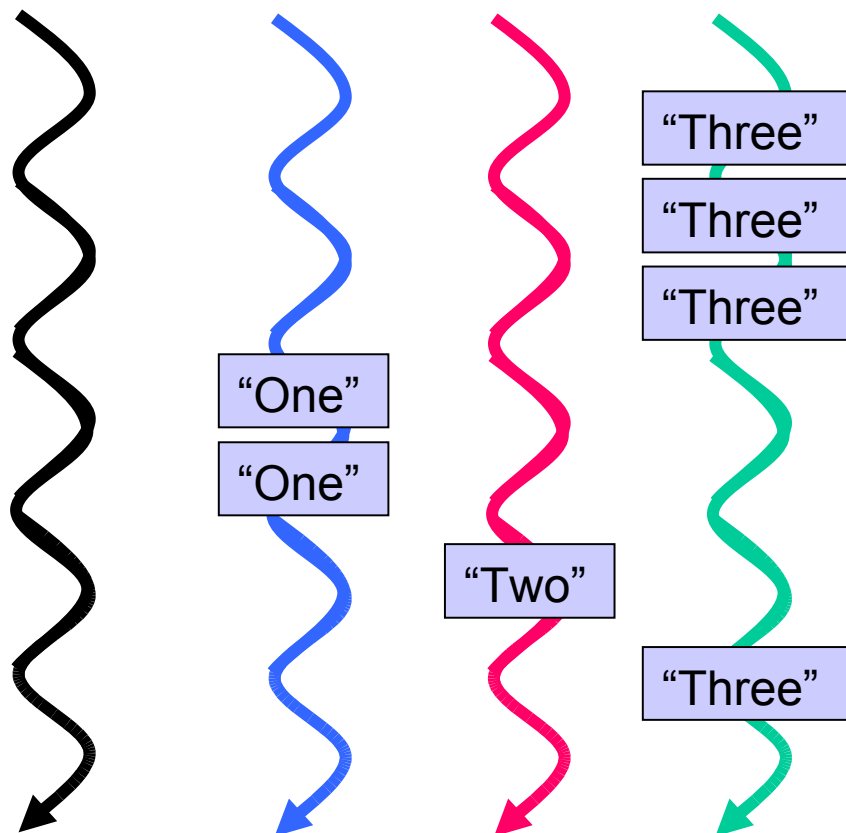
```
class Thread implements Runnable  
{  
    Thread( Runnable target ) { ... }  
    ...  
    void run( ) { ... }  
    native void start( );  
    void interrupt( ) { ... }  
    ...  
}
```

Example:

(Java-Threads)

```
class Printer implements Runnable {  
    String val;  
    Printer( String s ) { val = s; }  
    void run( ) {  
        while( true )  
            System.out.println( val );  
    }  
}
```

```
new Thread( new Printer("One") ).start();  
new Thread( new Printer("Two") ).start();  
new Thread( new Printer("Three") ).start();
```



Semantical Aspects of Threads

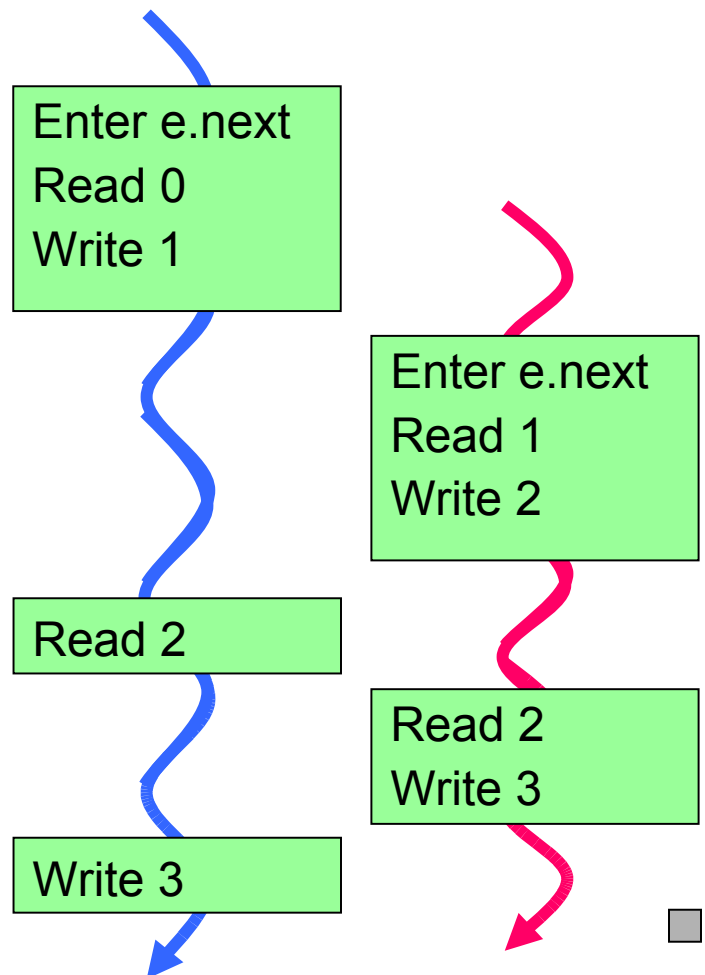
The language semantics should describe the allowed behaviors of a multi-threaded program. Two problems:

- weak semantics to enable optimizations by the compiler and runtime system
- *insufficient* synchronization by the user

Access to **common resources**, such as variables can lead to unwanted behavior.

Example:

```
class Even {  
    private int x;  
    void next( ) {  
        x++; x++;  
    }  
}
```



Explanation: (Shared Variables)

Variables that can be accessed from different threads are called *shared* (heap memory/variables). They are denoted by x, y, \dots in the following. Shared are in Java:

- instance fields
- static fields
- array elements

Local variables, parameters are not shared. They denoted by $r1, r2, \dots$ in the following.

Two accesses to the same variable are said to be ***conflicting*** if at least one is a write.



Examples: (*Incorrectly Synchronized Programs*)

A. Original code:

Thread 1 executes:	Thread 2 executes:
1. $r2 = x;$ 2. $y = 1;$	1. $r1 = y;$ 2. $x = 2;$

Initially: $x == y == 0.$

Can this result in $r2 == 2$ and $r1 == 1$ on termination?

It can! *Compiler* may transform the statements to:

Thread 1 executes:	Thread 2 executes:
<ol style="list-style-type: none">1. <code>y = 1;</code>2. <code>r2 = x;</code>	<ol style="list-style-type: none">1. <code>r1 = y;</code>2. <code>x = 2;</code>

Initially: `x == y == 0`.

Can result in `r2 == 2` and `r1 == 1`.

B. Original code:

Thread 1 executes:	Thread 2 executes:
<ol style="list-style-type: none">1. <code>r1 = x;</code>2. <code>r2 = r1.a;</code>3. <code>r3 = y;</code>4. <code>r4 = r3.a;</code>5. <code>r5 = r1.a;</code>	<ol style="list-style-type: none">1. <code>r6 = x;</code>2. <code>r6.a = 3;</code>

Initially: `x == y == 0`, `x.a == 0`.

Can this result in `r2 == r5 == 0` and `r4 == 3` on termination?

It can! *Compiler* may transform the statements to:

Thread 1 executes:	Thread 2 executes:
1. r1 = x;	1. r6 = x;
2. r2 = r1.a;	2. r6.a = 3;
3. r3 = y;	
4. r4 = r3.a;	
5. r5 = r2;	

Initially: $x == y == 0$, $x.a == 0$.

Can result in $r2 == r5 == 0$ and $r4 == 3$ on termination.



Synchronization in Java

Synchronization

- restricts the freedom of the compiler and
- guards access to statements.

Java supports two synchronization mechanisms:

- locks on objects: Every object has a lock. Every lock or unlock action of a thread T is a *synchronization action* of T.
- volatile instance variables: Every read and write of a volatile variable is a synchronization action.

(Further *synchronization actions* are related to the start and the termination of a thread.

Conceptual Memory Model for Java:

The precise thread semantics of Java is very complex.

We consider a *simplified* explanation.

(cf. old memory model of Java Lang. Spec., Sect. 17)

There is a **main memory** that is shared by all threads. It contains the master copy of every shared variable.

Every thread has a **working memory** in which it keeps its own working copy of variables that it must use or assign. As the thread executes a program, it operates on these working copies.

There are rules about when a thread is permitted or required to transfer the contents of its working copy of a variable into the master copy or vice versa. E.g.:

- Lock/unlock actions by T cause synchronization of working memory of T with main memory.
- Access to a volatile variable causes synchronization with main memory.

The actions are atomic (indivisible) and have the following meaning:

Thread actions:

use(*a*): vm-instruction gets value of variable *a* from working memory.

assign(*a*): vm-instruction sets variable *a* in working memory.

load(*a*): sets variable *a* in working memory after a read(*a*) by the memory memory.

store(*a*): provides value of variable *a* from working memory for a write into main memory.

Actions of main memory:

read(*a*): provides the value of variable *a* from main memory for a paired load action of a thread .

write(*a*): puts the given value *v* into main memory variable *a* (paired with a store(*a*)).

Joint actions:

lock(*X*): causes a thread to lay one claim on the lock for *X*.

unlock(*X*): causes a thread to release one claim on the lock for *X*.

There are a number of rules that restrict possible concurrent executions of actions, for example:

- all locks and unlocks are performed in some total sequential order
- reads and loads for a variable occur as pairs
- stores and writes for a variable occur as pairs
- new variables are created in main memory, i.e. a thread must perform a load or assign action before using the new variable.

Despite these rules, there is still a lot of freedom for a virtual machine to implement the thread-model.

Example: (Working memory)

```
class SoWas {
    private int v = 1, w = 1;
    void krk() {
        v = v + 1 ;
        w = v ;
    }
}
```

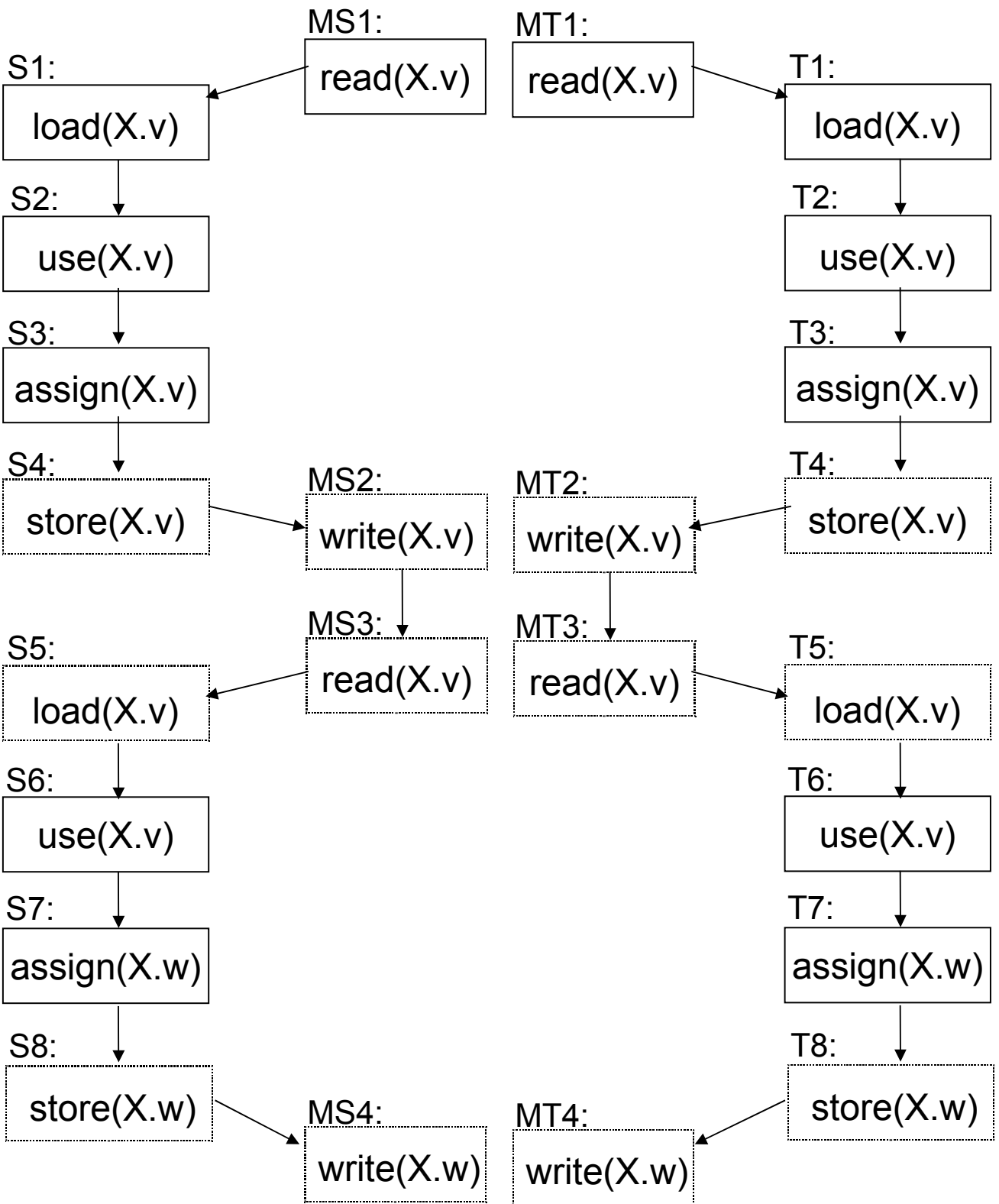
Consider two threads having a reference to the same SoWas-object X with $X.v == 1$ and $X.w == 1$.

What can happen if both threads execute `krk`?

Thread S:

Main memory:

Thread T:



Locking in Java:

An object *X* is either unlocked (lock count 0) or locked by one thread *T* a number of times (lock count *N*).

If *T* has locked *X*, we say *T* owns the lock of *X*.

- If *X* is not locked and *T* wants to lock *X*, *X* is locked with *T* as owner of the lock.
- If *T* owns the lock of *X* and wants to lock *X* (again), the lock count of *X* is incremented by 1.
- If *T* owns the lock of *X* and thread *S* ($\neq T$) wants to lock *X*, *S* is blocked until the lock is relinquished by *T* (i.e. lock count is 0).

Java does not provide explicit lock/unlock operations. Locking is done by the synchronization statement:

```
synchronized ( Expression ) Block
```

The expression has to yield an object *X*.

The executing thread tries to lock *X*. At the end of the block an unlock operation is performed on *X*.

To simplify notation one can write for example:

```
synchronized void mm() Block
```

instead of

```
void mm() { synchronized( this ) Block }
```

Volatile versus Synchronized Blocks:

```
class Test {
    static int i = 0, j = 0;
    static void one(){ i++; j++;}
    static void two(){
        System.out.println( i, j );
    }
}
```

Thread 1 executes method `one`.

Thread 2 executes method `two`.

Can `j` be greater than `i` in an output?

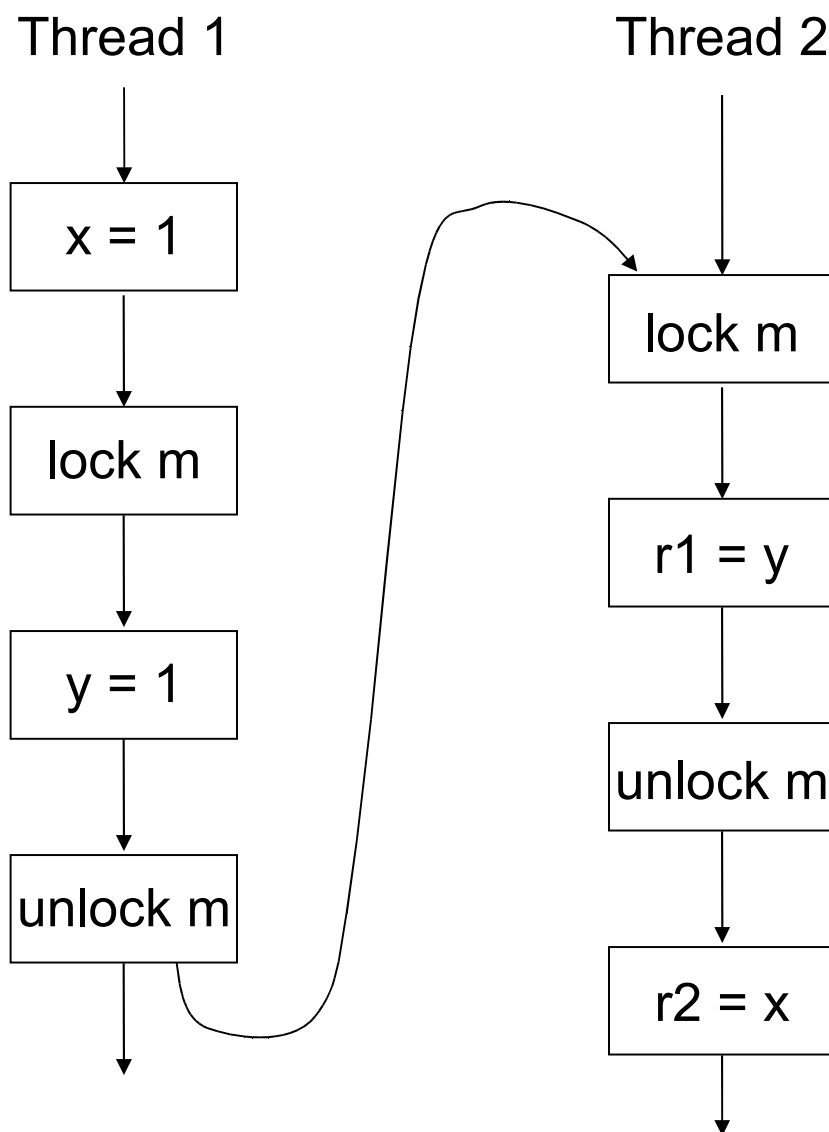
```
class Test {
    static int i = 0, j = 0;
    static synchronized void one(){ i++; j++;}
    static synchronized void two(){
        System.out.println( i, j );
    }
}
```

```
class Test {
    static volatile int i = 0, j = 0;
    static void one(){ i++; j++;}
    static void two(){
        System.out.println( i, j );
    }
}
```

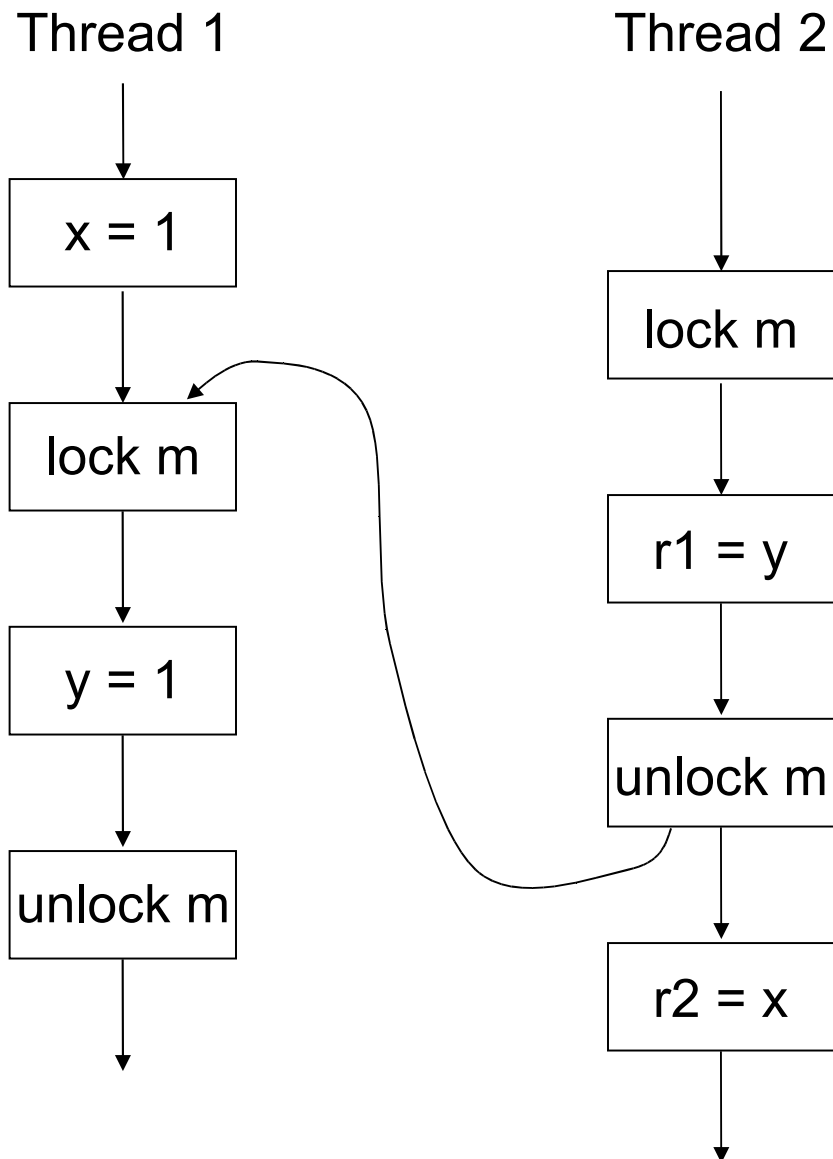
Correct Synchronization:

Let P be a program and $EX(P)$ an execution of P . The thread semantics defines a happens-before relation on the actions of $EX(P)$. We illustrate that relation by example.

Example: (Happens-before relation)



All conflicting accesses to x and y are ordered w.r.t. the happens-before relation.



The conflicting accesses to x are not ordered w.r.t. the happens-before relation.



Explanation: (Correctly synchronized)

A program execution is **sequentially consistent** iff

- all individual actions are totally ordered,
- the order is consistent with the program order,
- each action is atomic, and
- each action is immediately visible to all threads.

A program execution contains a **data race** iff there are two conflicting accesses that are not ordered by the happens-before relation.

A program is **correctly synchronized** iff all sequentially consistent executions are free of data races.



Remark:

- The behavior of incorrectly synchronized programs may be very different from the expectations (see above).
- The programmer is responsible to guarantee that programs are correctly synchronized.



Critical Sections

Similar to concurrent procedural programming, data races are avoided by using mutual exclusion on critical sections:

```
class Even {
    private int x;
    synchronized void next( ) {
        x++; x++;
    }
}
```

Unlike procedural programming, mutual exclusion is only w.r.t. threads locking the same object:

Example: (Object related mutual exclusion)

The following class Even does not guarantee that x is always even outside `next`:

```
class Even {
    private static int x = 0;
    synchronized void next( ) {
        x = x+1; x++;
    }
}
```



Designing Synchronization:

Declaring methods as synchronized is in general not sufficient to achieve well-behavior of concurrent programs.

Problems:

- related variables that are modified by several methods
- cooperation of threads
- fairness/starvation
- deadlock prevention

Locking and Encapsulation:

Usually, it does not suffice to lock only one object. Sometimes, a transactional behavior is needed.

Example: (Insufficient synchronization)

Consider a bank with offices sharing the accounts:

```
class Bank {
    protected Account[] accounts;
    class Account { int bal = 0; }

    Bank() { accounts = new Account[3];
            accounts[1] = new Account();
            accounts[2] = new Account();
    } }
}
```

```

class BankOffice extends Bank {

    BankOffice( Bank centralOffice ) {
        accounts = centralOffice.accounts;
    }

    synchronized void
    deposite( int accno, int amount ) {

        accounts[accno].bal += amount ;
    }

    synchronized boolean
    transfer( int from, int to, int amount ) {

        if( accounts[from].bal >= amount ) {
            int newBal = accounts[from].bal - amount;
            // possible interrupt: Thread.yield();
            accounts[from] = newBal;
            accounts[to] += amount;
            return true;
        }
        return false;
    }

    synchronized void printBalance12() {
        System.out.
            println("Account [1]: "+accounts[1].bal+
                "\t Account [2]: "+accounts[2].bal );
    }
}

```

```

public class BankTest {
    static Bank b0 = new Bank();
    static BankOffice b1 = new BankOffice( b0 );
    static BankOffice b2 = new BankOffice( b0 );

    public static void main( String[] argv ){

        b1.deposit( 1, 100 );
        b1.printBalance12();

        b2.deposit( 2, 100 );
        b2.printBalance12();

        Thread t1 = new Thread() {
            public void run(){
                while( true ) {
                    b1.transfer( 1, 2, 20 );
                    b1.printBalance12();
                }
            }
        };

        Thread t2 = new Thread() {
            public void run(){
                while( true ) {
                    b2.transfer( 2, 1, 50 );
                    b2.printBalance12();
                    yield();
                    b2.transfer( 1, 2, 30 );
                    b2.printBalance12();
                }
            }
        };

        t1.start();
        t2.start();
    }
}

```



The above example shows that synchronizing methods is not sufficient.

A lock or several locks are needed that protect all needed resources.

Example: (Multiple locks)

Instead of the solution above one can use a lock for each account:

```
boolean
transfer( int from, int to, int amount ) {
    synchronized( accounts[from] ) {
        synchronized( accounts[to] ) {
            ...
        } }
}
```




Notice, however, that multiple locks create deadlock problems! For example, the above method transfer easily lead to a deadlock, if transfers from a to b and b to a are interleaved.

Like in conventional concurrent programming locks should be obtained in a well-defined order.


Example: (Order on Locks)

To avoid the deadlock problem with transfer, we order the locks on accounts according to their account number:

```
boolean
transfer( int from, int to, int amount ) {
    // requires from != to
    Object one, two;
    if( from < to ) {
        one = accounts[from]; two = accounts[to];
    } else {
        one = accounts[to]; two = accounts[from];
    }
    synchronized( one ) {
        synchronized( two ) {
            ...
        }
    }
}
```



Remark:

- Fairness is difficult to achieve in languages like Java.
 - A special wait/notify-mechanism supports cooperation between objects.
- 

6.2 Message-based Concurrency

To avoid the disadvantages of thread-based concurrency, many alternative approaches are investigated in the literature.

Explanation: Message-based concurrency)

We speak of message-based concurrency if

- the state space is separated into disjoint components
- the components only communicate over messages
- synchronization is defined in terms of sending and receiving messages.



Remark:

Message-based concurrency can be combined with multi-threading.



Overview:

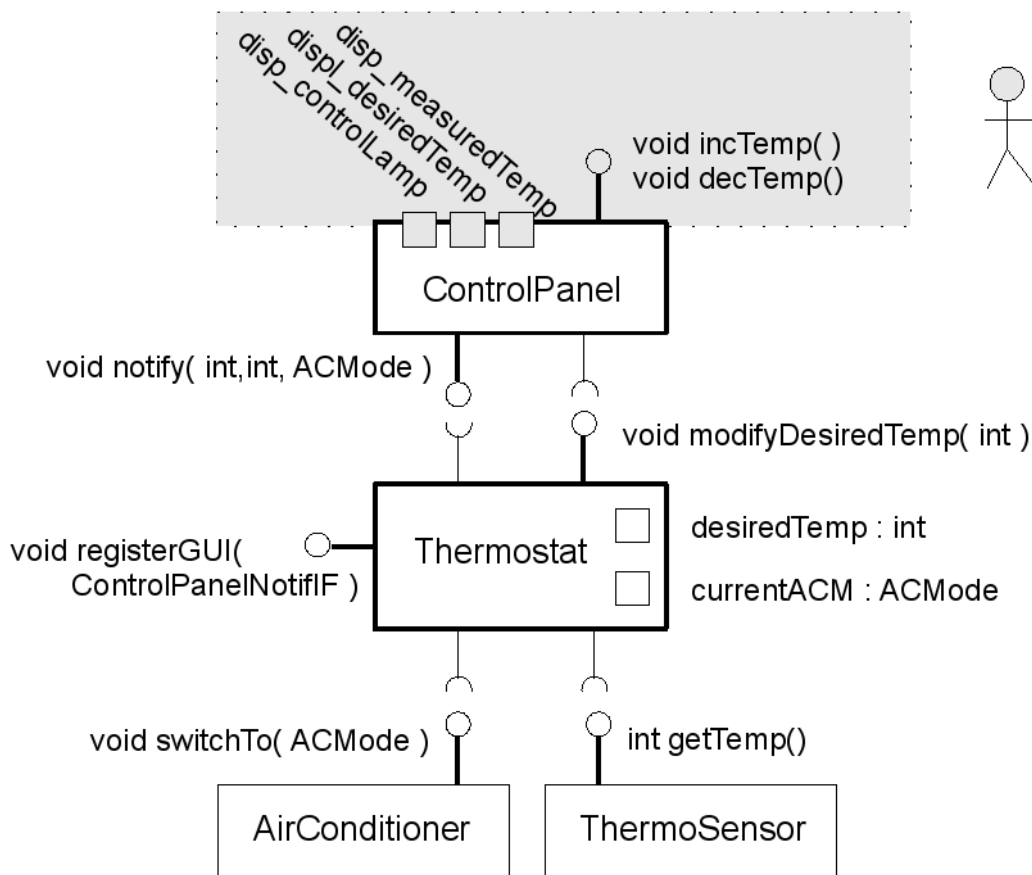
We consider:

- asynchronous message passing
 - synchronous message passing
- and discuss implementations in Java.

Example: (ThermoControl systems)

Four components:

- AirConditioner: interface to switch its mode of operation (OFF, HEATING, COOLING)
- ThermoSensor: sensor providing current temperature
- Thermostat: active component that
 - reads current temperature
 - compares it to desired temperature
 - switches mode of operation of AirConditioner if necessary
 - notifies ControlPanel if changes have occurred
- ControlPanel (active component): displays current status and allows to in- and decrement desired temp.



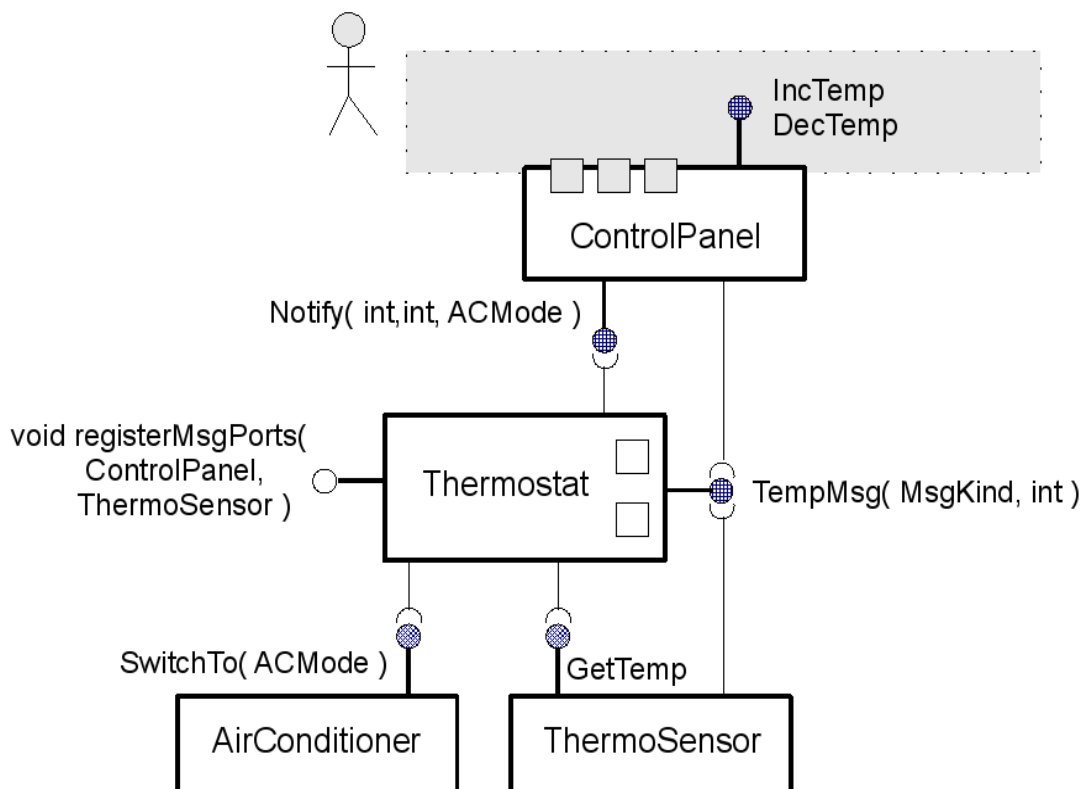
Asynchronous Messages

Communication via asynchronous messages is similar to communication by email:

- message queues at the receiver site
- receiver determines when she reads messages or whether messages are read at all

All communication partners have to be active components.

In the figure, message queues are represented by hatched connectors.



All synchronization happens at the message queues. They can be implemented by library classes, i. e. application independent expert code.

Example: (Use of message queues)

Possible implementation of ThermoSensor using

`java.util.concurrent.LinkedBlockingQueue:`

```
class ThermoSensor
    extends LinkedBlockingQueue<GetTemp>
    implements Runnable
{
    BlockingQueue<TempMsg> outmess;
    double measuredTemperature; // set by environment

    ThermoSensor(BlockingQueue<TempMsg> outmess){
        super(); this.outmess = outmess;
        new Thread( this ).start();
    }
    public void run() {
        while( true ){
            try {
                take();
                outmess.put(
                    new TempMsg(MsgKind.currentTemp,
                                (int)measuredTemperature));
            } catch( InterruptedException ie ) {}
        }
    }
}
```



Discussion:

Message queues play the role of connectors between components.

If threads are only component-local, data races and issues of thread-safety are avoided.

Components have better control over incoming messages/methods.

Disadvantage: Indirection caused by messages cause loss of efficiency.

Disadvantage: Closer coordination between components is more difficult to achieve.



Programming message passing systems:

Support by libraries and frameworks (e.g. Java Message Service JMS)

Language extensions for message support (JCoBox)

As basic mechanism in the language (Erlang)

JCoBox

extends Java with new concurrency concept:

- CoBox classes: `@CoBox` annotation
hierarchical structure the heap similar to ownership
- cooperative task scheduling:
 - one active task per cobox
 - active task has to yield control (not interrupted by scheduler)
- **asynchronous method calls:** `x!m(e)`
 - immediately return to the caller
 - create new task in cobox of receiver
- **futures:** `Future<ResultType> f;`
provide a location for results of asynchronous method calls; reading the result:
 - `f.await();` // suspends until result is available
 - `f.get();` // blocks the current task

Example: (Simple CoBox program)

```
@CoBox // the @CoBox annotation
class Ping {
    Pong pong;
    boolean stopped;
    Ping(Pong p) {
        pong = p;
        this!go(); // asynchronous self-call
                  // starts internal task
    }
    void go() {
        stopped = false;
        while (! stopped) {
            Fut<String> fut = pong!ping("Hello");
            String answer = fut.await();
                // allowing stop calls to be executed
            System.out.println(answer);
        } }
    void stop() { stopped = true; }
}

@CoBox
class Pong {
    String ping(String s) { return s+" World"; }
}

class Main {
    public static void main(String... args) {
        // at this point we are in the ROOT cobox
        Pong pong = new Pong();
        Ping ping = new Ping(pong);
        JCoBox.sleep(5, TimeUnit.SECONDS);
        pong.stop();//equivalent to: pong!stop().get();
    }
}
```



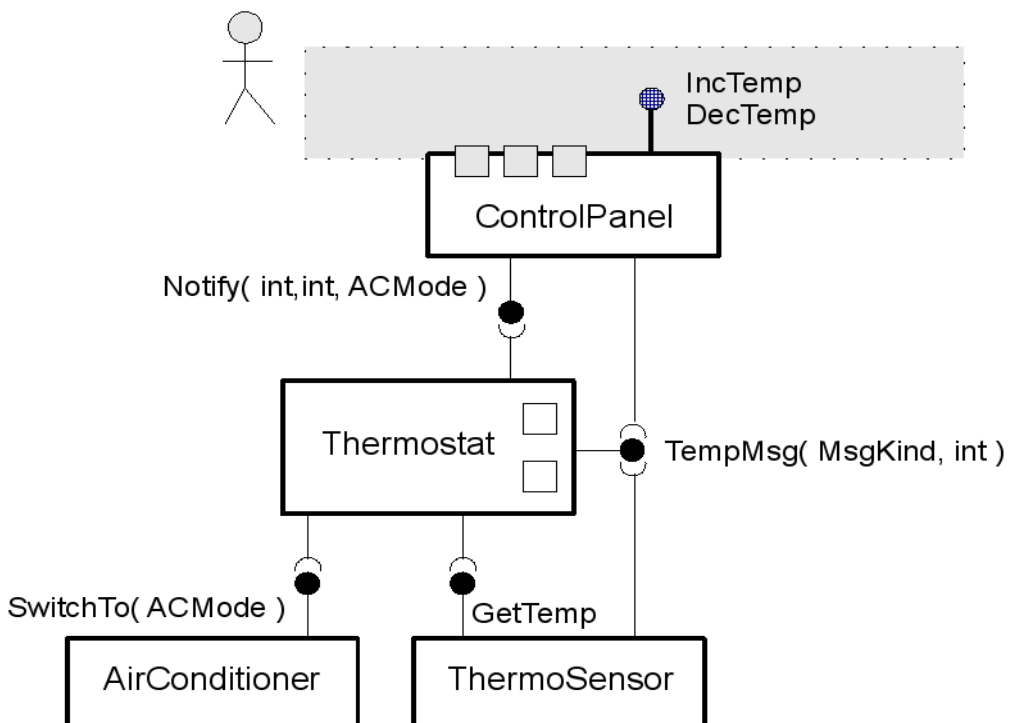
Synchronous Messages

Synchronous message passing is based on a rendezvous between sender and receiver:

- Messages are only sent if both sender and receiver are ready to perform the communication.
- If only one is ready, the other one is blocked.

All communication partners have to be active components.

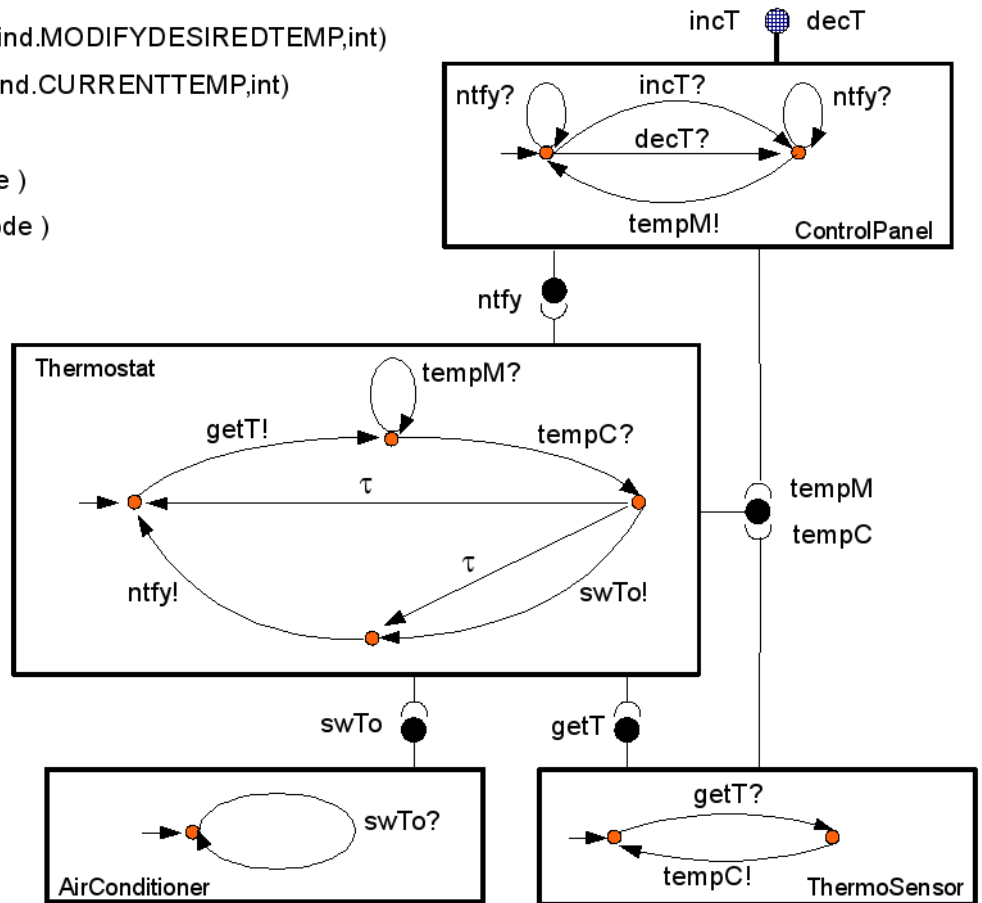
In the figure, connectors represent synchronous channels.



Behavior is modeled by the following interface automata:

```

tempM = TempMsg(MsgKind.MODIFYDESIREDTEMP,int)
tempC = TempMsg(MsgKind.CURRENTTEMP,int)
getT = GetTemp
swTo = SwitchTo( ACMode )
ntfy = Notify( int,int, ACMode )
incT = IncTemp
decT = DecTemp
    
```



Transitions with send messages $m!$ may only be taken synchronously with corresponding receive messages $m?$ in a component that is linked by a channel for m .

Remarks:

Synchronous communication has the advantages of asynchronous communication and simplifies to express close coordination.

Disadvantage: Danger of deadlocks is higher.

Synchronous communication underlies many communication models:

- Programming languages, e.g. Ada tasks
- Calculi:
 - Communicating sequential processes CSP
 - Calculus of communicating systems CCS
 - π -calculus
- Interface automata

In Java, synchronous communication can be achieved by using class `SynchronousQueue<E>` of package `java.util.concurrent` in which each put-operation must wait for a corresponding take-operation, and vice versa.

A synchronous queue does not have any internal capacity, not even a capacity of one.



6.3 Distributed OO-Programming

Distributed programming is about programs that run in different OS-processes/on different machines.

Central to distributed programming are the means of communication. Most OO-languages or frameworks support:

- communication over sockets and streams
- remote method invocation, a synchronous communication technique

Some support in addition:

- events, signals
- asynchronous messages
- group and multicast communication

Notice:

Distributed programs are usually concurrent programs.

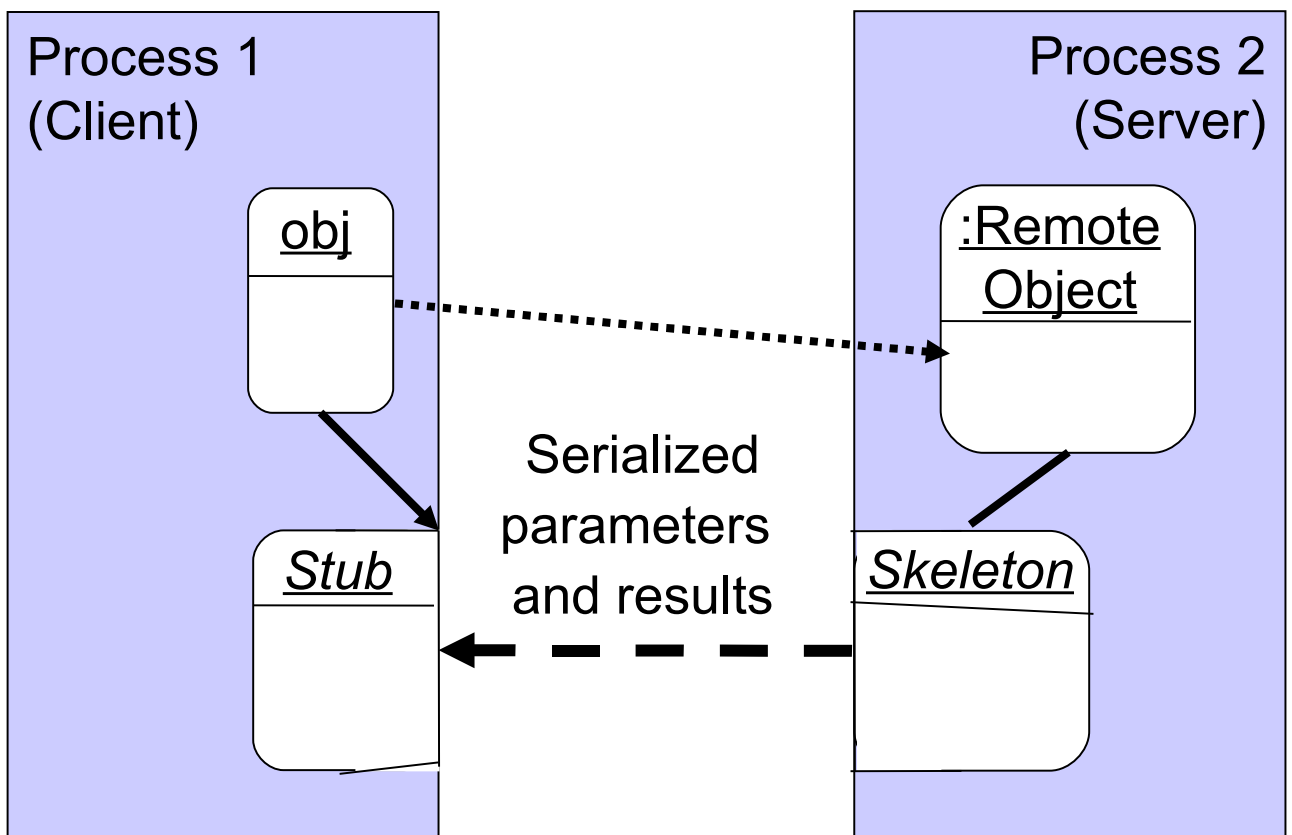
Remote Method Invocation in Java

- Methods of objects in other processes (remote objects) can be invoked, similar to methods on local objects
- Without additional code, only one thread can invoke remote methods, others are blocked.

Overview:

- Realization: Stubs and skeletons
- Remote interfaces and their implementations
- Binding and lookup of remote objects
- Invoking remote methods
- Parameter passing

Relalization: Stubs and Skeletons:



- Remote objects are represented locally by stubs
- Stubs and skeletons provide communication
- Code for stubs and skeletons is automatically generated by the Java compiler

Remote Interfaces and their Implementation:

- Methods that are available remotely must be specified in an interface that extends Remote:

```
interface Remote { }
```

- Implementations of remote objects extend UnicastRemoteObject (or similar classes)
- Constructors may throw exception
- Almost identical to local implementations

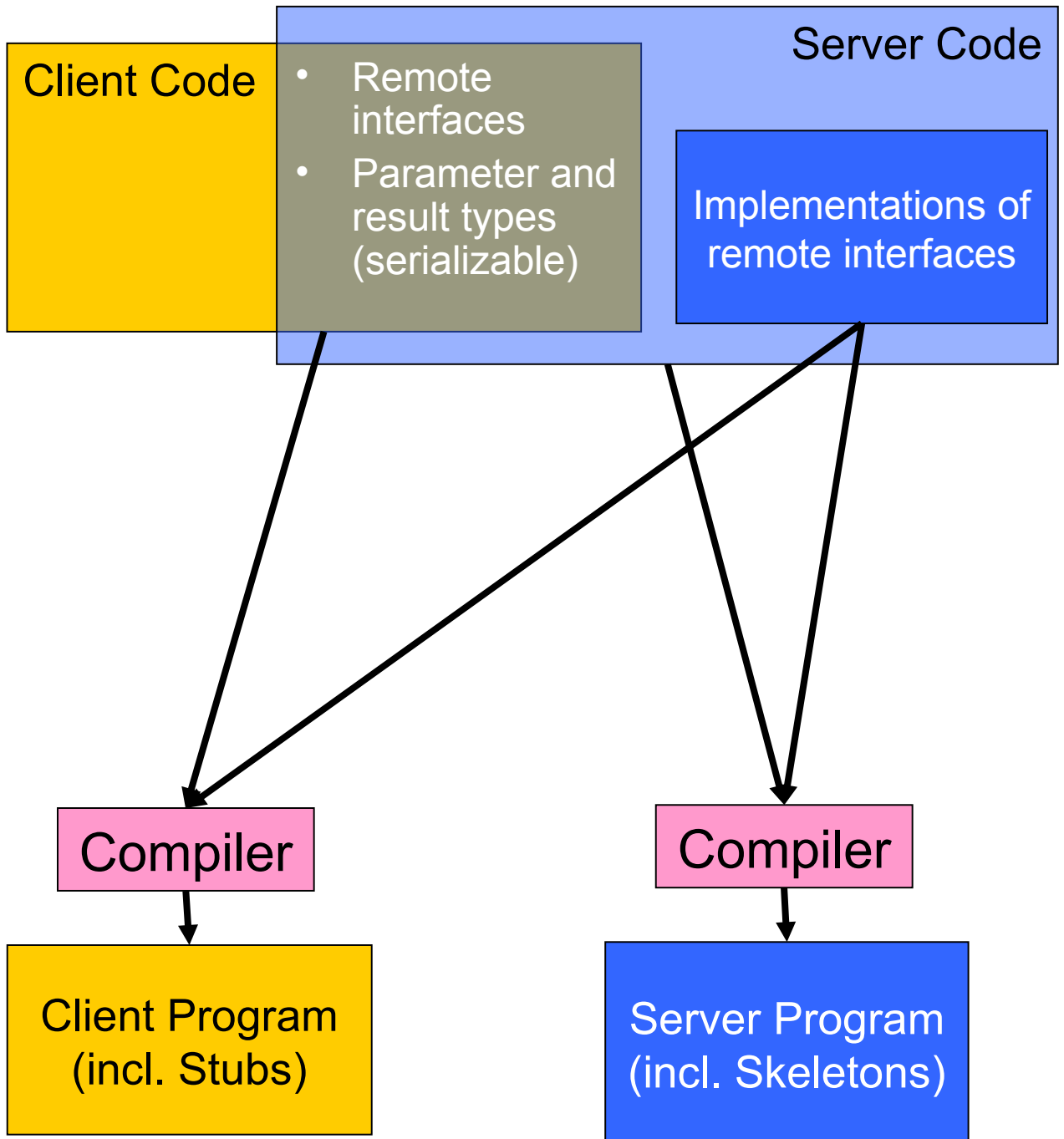
Example: (Remote object implementation)

Buffer that can be accessed remotely:

```
interface Buffer extends Remote {  
    void put( Prd p ) throws RemoteException;  
    Prd get( ) throws RemoteException;  
}
```

```
class BufferImpl extends UnicastRemoteObject  
    implements Buffer {  
    // fields identical to local solution;  
    BufferImpl() throws RemoteException { }  
    synchronized void put( Prd p )  
        { // identical to local solution }  
    synchronized Prd get( )  
        { // identical to local solution }  
}
```

Programming Infrastructure:



Binding and Lookup of Remote Objects:

- References to remote objects are obtained through a **name service**
- **Name server** (rmiregistry) must run on server site
 - Offers service at a certain port
 - Communication with name server is enabled by API
- Process of remote object binds remote object to a name.
- Potential invoking object gets reference through method lookup using an URL.

```
class Naming {
    static void rebind(String name, Remote obj)
        throws ... { ... }
    static Remote lookup( String name )
        throws ... { ... }
    ...
}
```

Example: (Binding and Lookup)

Buffer server binds a buffer to name „buffer“:

```
class BufferServer {
    static void main( ... ) throws Exception {
        Naming.rebind( "buffer",
            new BufferImpl() );
    }
}
```

Producer looks up and links to the remote buffer object:

```
class Producer extends Thread {
    ...
    static void main( ... ) throws Exception {
        String url = "rmi://monkey/buffer";
        Buffer b = (Buffer) Naming.lookup(url);
        new Producer( b ).start( );
    }
}
```



Invoking Remote Methods:

Remote references can be used like a local reference, in particular to invoke a method.

Example: (Invocation of remote methods)

```
class Producer extends Thread {
    Buffer buf;

    Producer( Buffer b ) { buf = b; }

    void run( ) {
        while ( true )
            try {
                buf.put( new Prd( ) );
            } catch( Exception e ) { ... }
    }
}
```

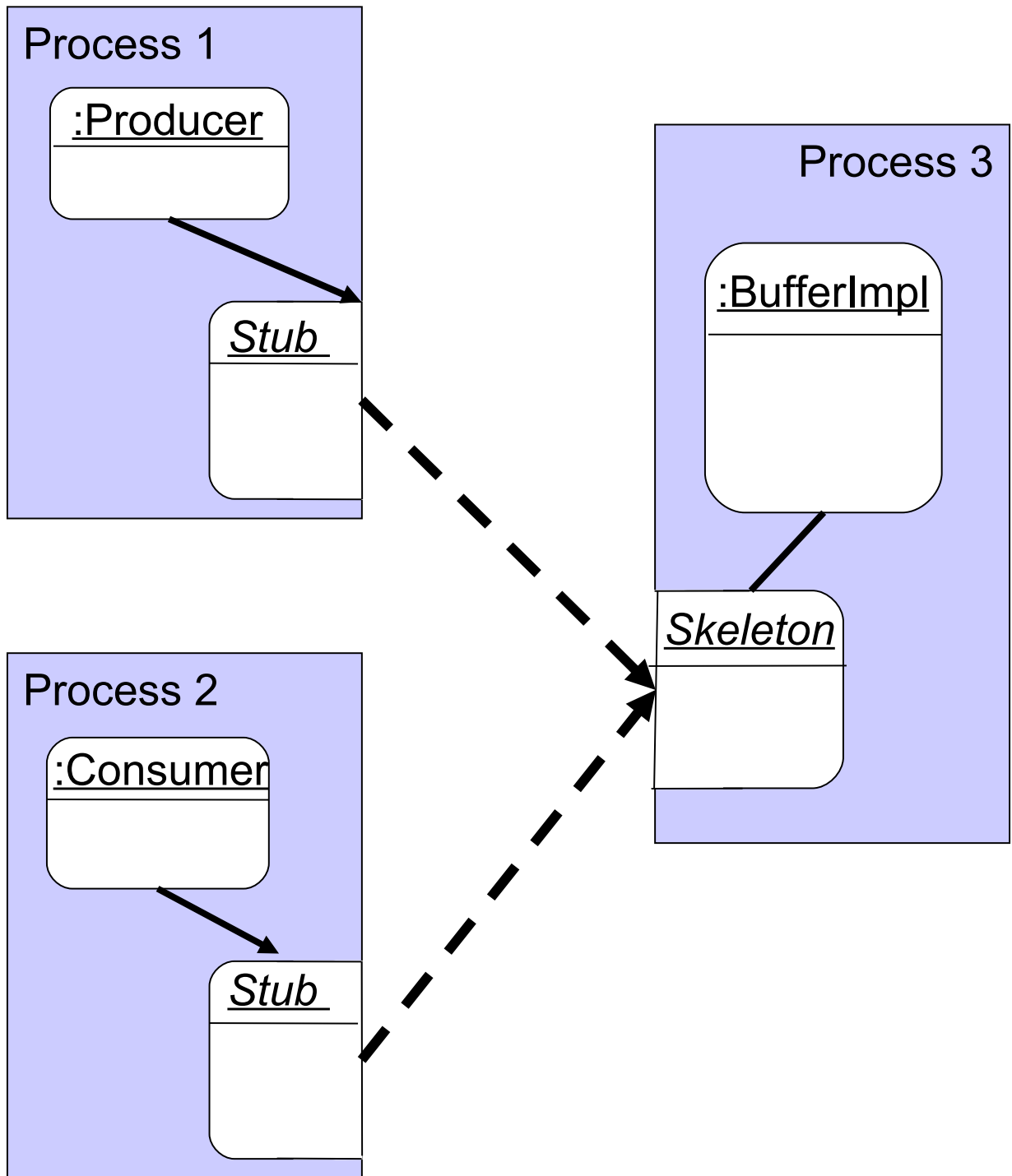


Remark:

- Remote interfaces can be used to invoke methods of remote objects
- Communication is transparent except for
 - Error handling
 - Problems of serialization
- Coding is almost identical to local solutions



Process Interaction:



Summary: Using RMI in Java:

- Define interface of remote object (extends Remote)
- Define implementation of remote object (extends UnicastRemoteObject)
- Start name server (rmiregistry)
- Server program registers remote objects at registry
- Client programs retrieve remote reference through URL (name of computer and name of remote object)

Parameter Passing:

Parameter passing is essentially done by serialization, however:

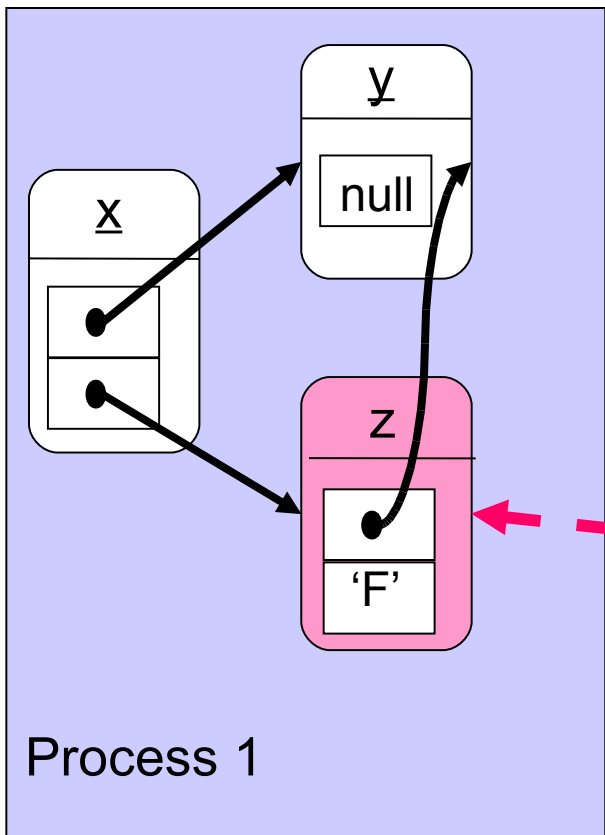
- all parameters are serialized as if they form a connected object structure (duplicates are copied only once)
- references to remote objects (more precisely to the stubs) are handled by
 - using the reference to the remote object, if it belong to the process of receiver object
 - creating a new stub on the remote side, otherwise

Example: (Passing parameters)

- Parameters of *one* remote method invocation are serialized together
- Aliases do not lead to duplicate objects

Call:

```
remoteObj.m( x, x, y, z );
```



Formal parameters:

