



TECHNISCHE
UNIVERSITÄT
DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

OPERATING-SYSTEM CONSTRUCTION

Material based on slides by Olaf
Spinczyk, Universität Osnabrück

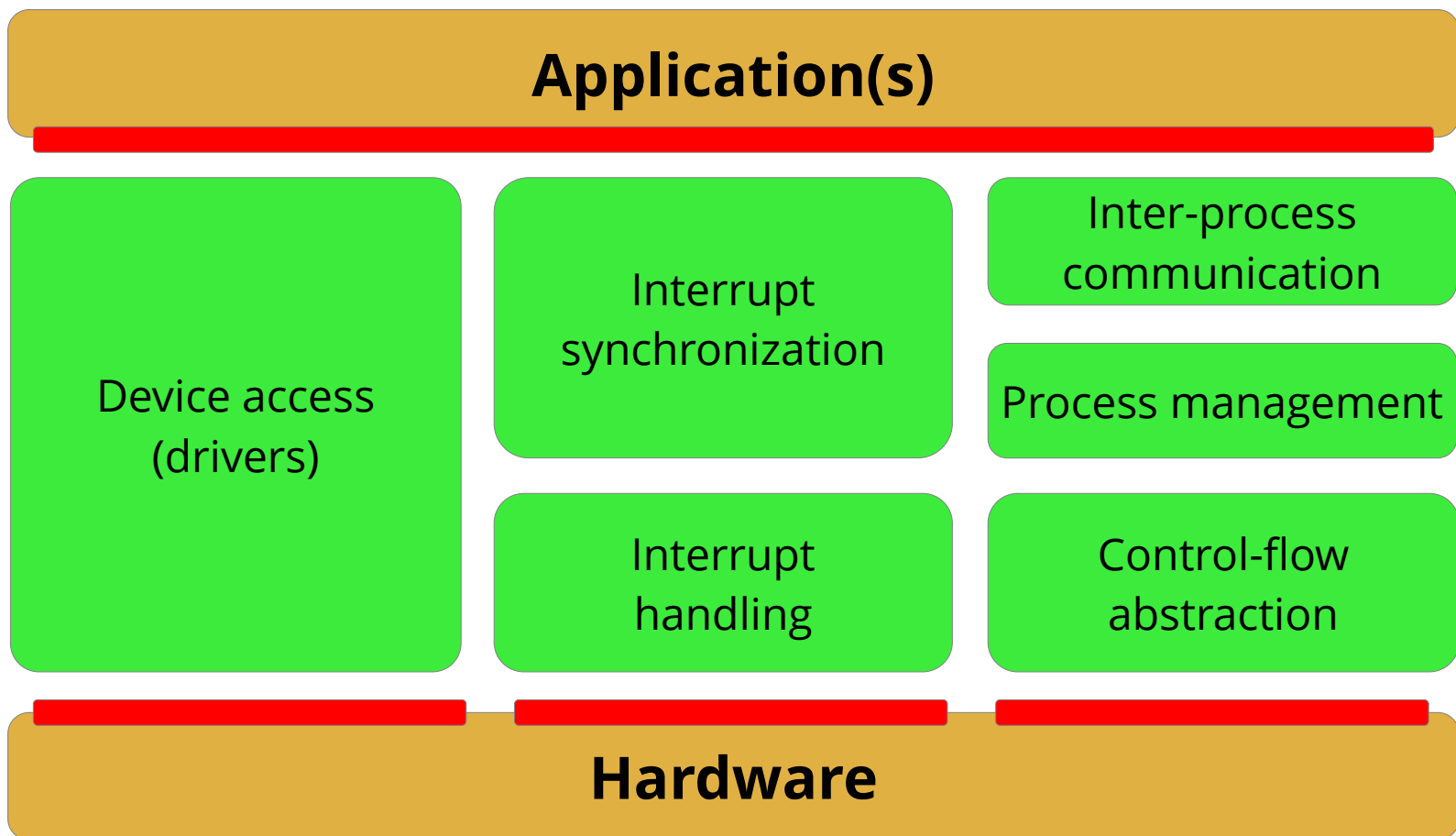
Inter-Process Communication (IPC)

<https://tud.de/inf/os/studium/vorlesungen/betriebssystembau>

HORST SCHIRMEIER

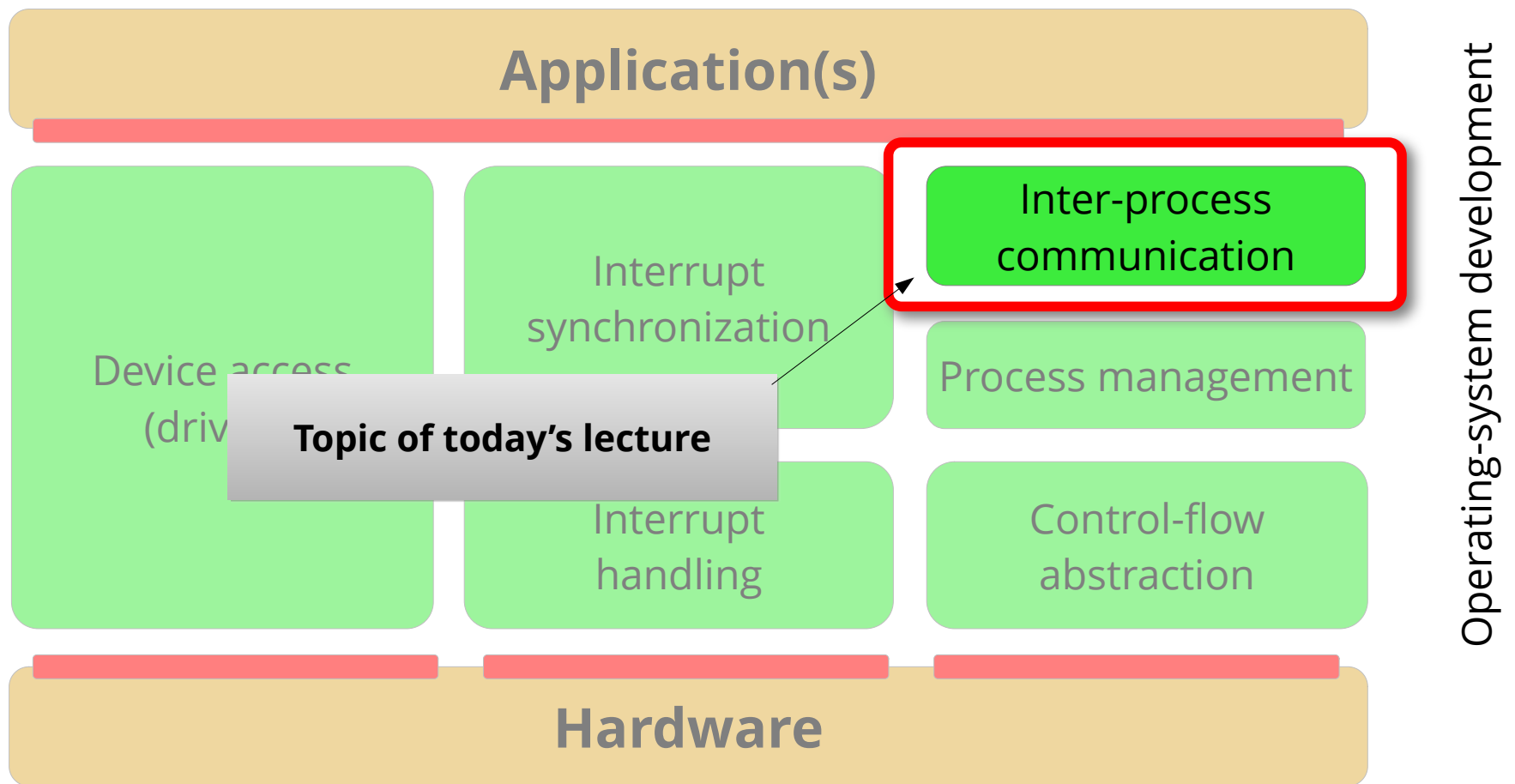
Overview: Lectures

Structure of the “OO-StuBS” operating system:



Overview: Lectures

Structure of the "OO-StuBS" operating system:



Agenda

- Communication and Synchronization
- IPC via Shared Memory
 - Semaphore, Monitor, Path Expressions
- IPC via Messages
 - Send/Receive
- Basic Abstractions in Operating Systems
- Duality of Concepts
- Summary

Agenda

- **Communication and Synchronization**
- IPC via Shared Memory
 - Semaphore, Monitor, Path Expressions
- IPC via Messages
 - Send/Receive
- Basic Abstractions in Operating Systems
- Duality of Concepts
- Summary

Communication and Synchronization

- ... are related through the principle of causality:

If **A** needs a piece of information from **B** to continue its work, **A** must *wait* until **B** supplies that information.

- Message-based communication (usually) implies synchronization (e.g. in **send()** and **receive()**)
- Synchronization primitives are a suitable basis for implementing communication primitives (e.g. **semaphore**)

Agenda

- Communication and Synchronization
- **IPC via Shared Memory**
 - Semaphore, Monitor, Path Expressions
- IPC via Messages
 - Send/Receive
- Basic Abstractions in Operating Systems
- Duality of Concepts
- Summary

IPC via Shared Memory

- **Use cases / constraints**

- Unprotected system (all processes in same address space)
- System with language-based memory protection
- Communication between threads in the same address space
- OS-supplied, MMU-based shared memory
(e.g. UNIX System V Shared Memory, see man page **shm_overview(7)**)
- Common kernel address space of isolated processes

- **Positive properties**

- Atomic memory accesses do not require additional synchronization
- Fast: zero-copy
- Simple IPC applications easy to implement
- Unsynchronized communication possible
- M:N communication simple

Semaphore – Simple Interactions

- Mutual exclusion

```
// Shared memory  
Semaphore mutex(1);  
SomeType shared;
```

```
void process_1() {  
    mutex.wait();  
    shared.access();  
    mutex.signal();  
}
```

```
void process_2() {  
    mutex.wait();  
    shared.access();  
    mutex.signal();  
}
```

- Unilateral synchronization

```
// Shared memory  
Semaphore elem(0);  
SomeQueue shared;
```

```
void producer() {  
    shared.put();  
    elem.signal();  
}
```

```
void consumer() {  
    elem.wait();  
    shared.get();  
}
```

- Resource-oriented synchronization

```
// Shared memory  
Semaphore resource(N); // N>1  
SomeResource shared;
```

otherwise identical to
mutual exclusion

Semaphore – more Complex Interactions

- Readers—writers problem
 - Writers need exclusive access to memory
 - Multiple readers may work simultaneously

Acquire (Reader)

- become active reader
- wait as long active writers exist
- become reading reader

Acquire (Writer)

- become active writer
- wait as long active readers exist
- become writing writer
- wait for writer mutex

READ

Shared
memory

WRITE

Release (Reader)

- stop being reader
- if no more reading readers exist
but waiting writers, wake them up

Release (Writer)

- release writer mutex
- stop being writer
- if no more active writers exist
but waiting readers, wake them up

Semaphore - Readers—Writers Problem

```
// Acquire (Reader)
mutex.p();
ar++; // active readers
if (aw==0) {
    rr++; // reading readers
    read.v();
}
mutex.v();
read.p();
```

READ

```
// Release (Reader)
mutex.p();
ar--; rr--;
while (rr==0 && ww<aw) {
    ww++;
    write.v();
}
mutex.v();
```

```
// Acquire (Writer)
mutex.p();
aw++; // active writers
if (rr==0) {
    ww++; // writing writers
    write.v();
}
mutex.v();
write.p();
w_mutex.p();
```

WRITE

```
// Release (Writer)
w_mutex.v();
mutex.p();
aw--; ww--;
while (aw==0 && rr<ar) {
    rr++;
    read.v();
}
mutex.v();
```

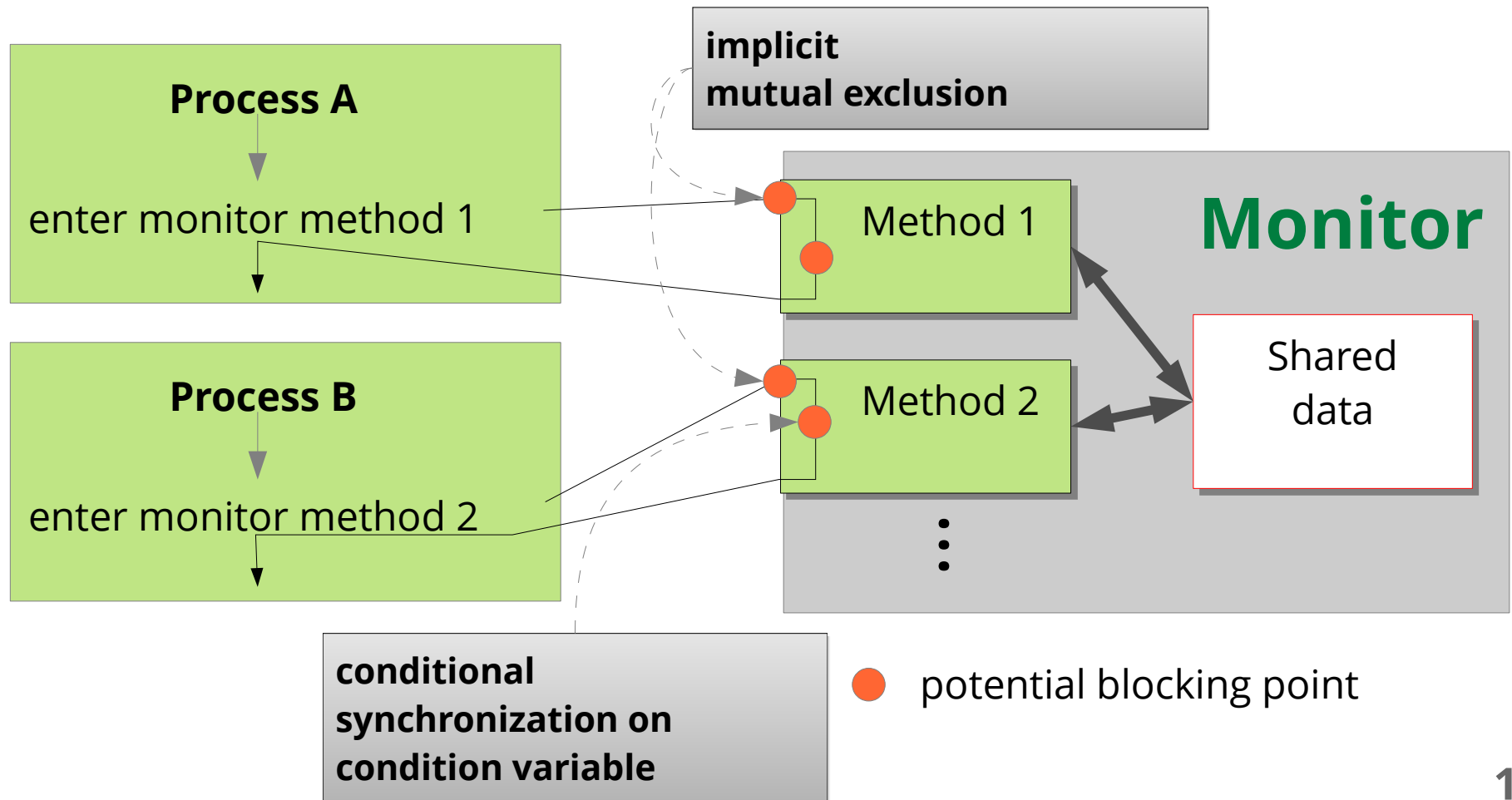
Semaphore – Discussion

- Extensions
 - Non-blocking P()
 - Timeout
 - Counter array
- Sources of errors (bugs!)
 - Semaphore use is **not enforced**
 - Cooperating processes depend on each other
 - **All** must comply with the protocol
 - Implementation effort
- ➔ Programming-language support
 - Enforces correct synchronization

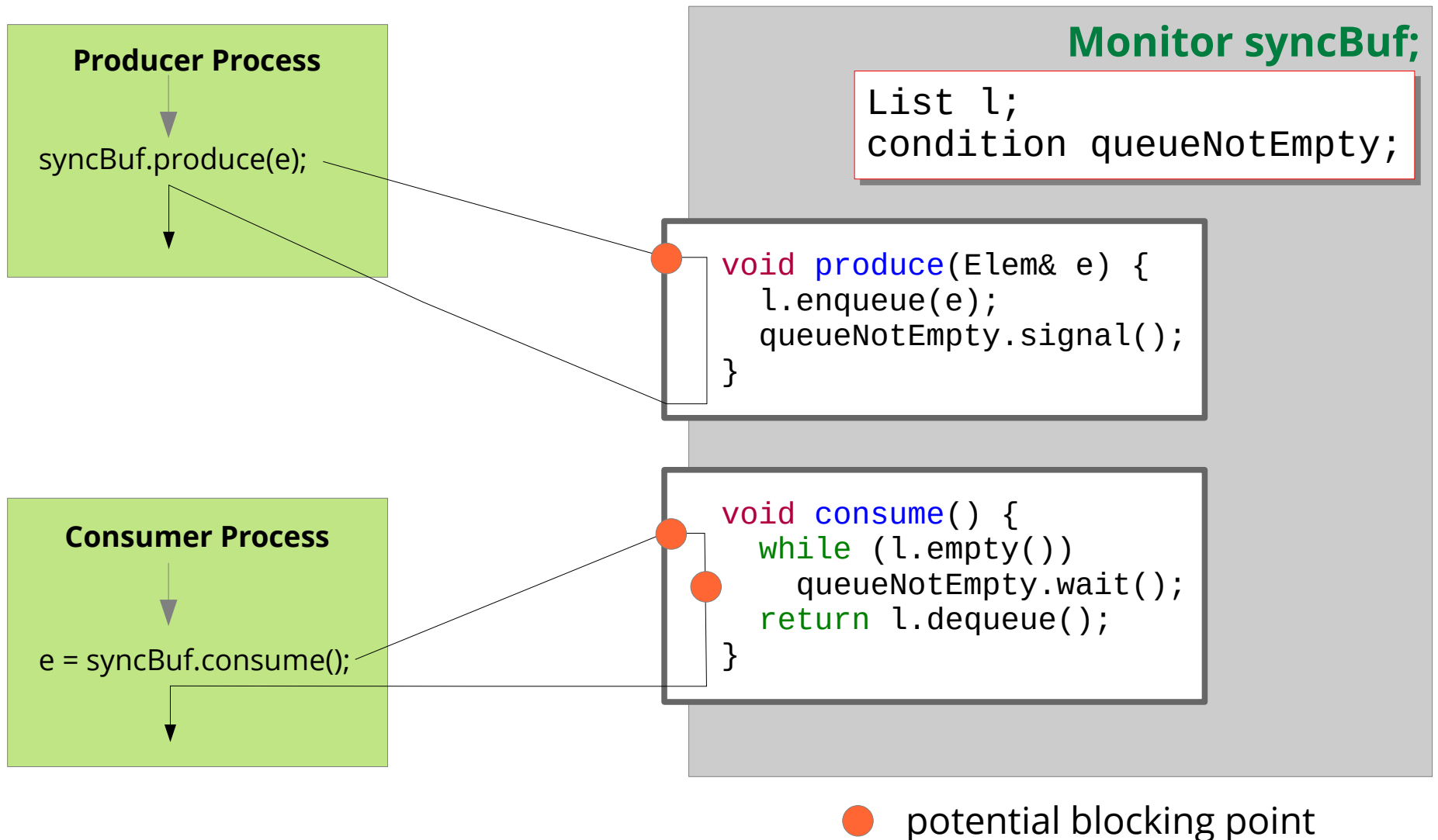
Monitors – Synchronized ADTs [1]

[1] C. A. R. Hoare, *Monitor – An Operating System Structuring Concept*, Communications of the ACM 17, 10, S. 549-557, 1974

- Idea: Couple abstract data type with synchronization properties



Monitors – Producer–Consumer



Monitors - Readers—Writers

SomeType
resource;

```
void read() {
    ...
}
```

Shared
data

```
void write() {
    ...
}
```

not a monitor!

Reader Process

```
rwMon.startRead();
resource.read();
rwMon.endRead();
```

Writer Process

```
rwMon.startWrite();
resource.write();
rwMon.endWrite();
```

```
void startRead() {
    if (aw>0)
        read.wait();
    rr++;
    read.signal();
}
```

```
void endRead() {
    rr--;
    if (rr==0)
        write.signal();
}
```

```
void startWrite() {
    aw++;
    if (busyW||rr>0)
        write.wait();
    busyW=true;
}
```

```
void endWrite() {
    busyW=false;
    aw--;
    if (aw==0)
        read.signal();
    else
        write.signal();
}
```

Monitor
rwMon;

condition read,write;

```
int rr,aw;
bool busyW;
```

Monitors – Implementation

- ... based on semaphores

Simple implementation that only supports a single condition variable

```
Semaphore mutex(1);
Semaphore s_signal(0);
Semaphore s_wait(0);
int c_signal = 0;
int c_wait = 0;
```

Monitor

```
void op() {
    mutex.p();
    // original op()
    ...
    cond.wait();
    ...
    cond.signal();
    ...
    // finished
    if (c_signal > 0)
        s_signal.v();
    else
        mutex.v();
}
```

```
void Cond::wait() {
    c_wait++;
    if (c_signal > 0)
        s_signal.v();
    else
        mutex.v();
    s_wait.p();
    c_wait--;
}
```

```
void Cond::signal() {
    if (c_wait > 0) {
        c_signal++;
        s_wait.v();
        s_signal.p();
        c_signal--;
    }
}
```


Monitors – Discussion

- Limits concurrency to **full** mutual exclusion
 - That’s why Java allows **synchronized** for individual methods
- Coupling of logical structure and synchronization not necessarily “natural”
 - see readers—writers example
 - Same problem: Just like with the semaphore, programmers must comply with a protocol
- ➔ Synchronization should be **separated** from data organization and methods.

Path Expressions [2]

- Idea: Flexible expressions describe permitted sequences of execution and the object-access degree of concurrency
- **path** *name1, name2, name3* **end**
 - Arbitrary order and arbitrarily concurrent execution of *name1-3*
- **path** *name1; name2* **end**
 - Before each execution of *name2* at least once *name1*
- **path** *name1 + name2* **end**
 - Alternative execution: either *name1* or *name2*
- **path** *N:(path expression)* **end**
 - max. N control flows are permitted to be in path expression

[2] R. H. Campbell and A. N. Habermann, *The Specification of Process Synchronization by Path Expressions*, Lecture Note in Computer Science 16, Springer, 1974

Path Expressions – Example

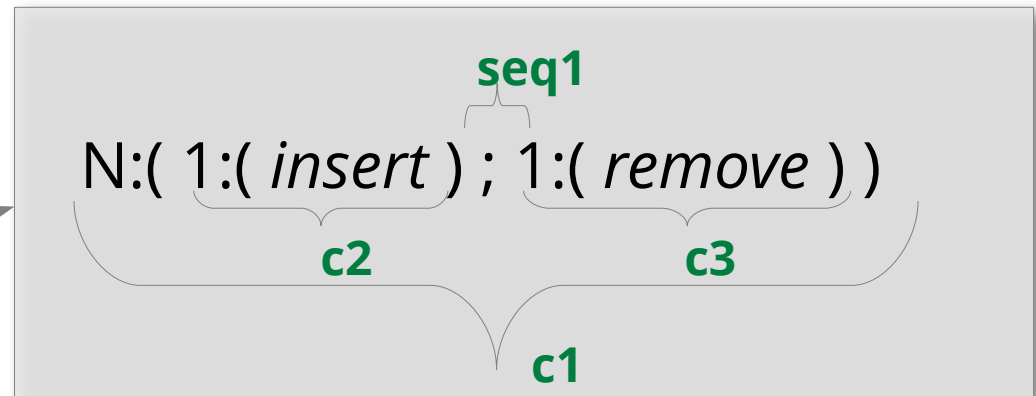
- Idea: Flexible expressions describe permitted sequences of execution and the degree of concurrency, e.g.:
- **path** 10:(1:(*insert*); 1:(*remove*)) **end**
- Synchronization of a 10-element buffer
 - Mutual exclusion during execution of *insert* and *remove*
 - At least one *insert* before each *remove*
 - Never more than 10 finalized *insert* operations

Path Expressions - Implementation (1)

- Transformation to a state machine
 - State transition at entry/exit into/from operation

- Example:

For each 'X:(..)' and ';' we introduce a counter.



```
int c1=0;
int c2=0;
int c3=0;
int seq1=0;
```

```
bool mayInsert () {
    return c1<N && c2<1;
}

void startInsert () {
    c1++; c2++;
}

void endInsert () {
    c2--; seq1++;
}
```

```
bool mayRemove () {
    return c1<N && seq1>0 && c3<1;
}

void startRemove () {
    c3++; seq1--;
}

void endRemove () {
    c3--; c1--;
}
```

Path Expressions - Implementation (2)

- Transforming the operations

For each operation we introduce a semaphore and a counter.

$N:(\underbrace{1:(insert)}_{sem1/csem1}; \underbrace{1:(remove)}_{sem2/csem2})$

```
Semaphore mutex(1);
int csem1=0;
Semaphore sem1(0);
int csem2=0;
Semaphore sem2(0);
```

```
void Insert() {
    mutex.p();
    if (!mayInsert()) {
        csem1++;
        mutex.v();
        sem1.wait();
    }
    startInsert();
    mutex.v();
    // [orig. insert code]
    mutex.p();
    endInsert();
    if (!wakeup())
        mutex.v();
}
```

```
bool wakeup() {
    if (csem1>0 &&
        mayInsert()) {
        csem1--;
        sem1.v();
        return true;
    }
    if (csem2>0 &&
        mayRemove()) {
        csem2--;
        sem2.v();
        return true;
    }
    return false;
}
```

Path Expressions – Discussion

- Advantages
 - More **complex interaction patterns** possible than with monitors
 - read + 1: write
 - Compliance with interaction protocols is **enforced**
 - Less bugs!
- Disadvantages
 - Synchronization behavior cannot depend on state variables or parameters
 - Extension: Path expressions with predicates
 - **Synchronization of the state machine itself** can become the bottleneck
 - **No support** for path expressions in common programming languages

Agenda

- Communication and Synchronization
- IPC via Shared Memory
 - Semaphore, Monitor, Path Expressions
- **IPC via Messages**
 - Send/Receive
- Basic Abstractions in Operating Systems
- Duality of Concepts
- Summary

IPC via Messages

- **Use cases/Constraints**
 - IPC across machine boundaries
 - Interaction of isolated processes
- **Positive properties**
 - Uniform paradigm for IPC with local and remote processes
 - Buffering and synchronization if necessary
 - Indirection allows for transparent protocol extensions
 - Encryption, error correction, ...
 - High-level language mechanisms such as OO messages or procedure calls can be mapped to IPC via messages (RPC, RMI)

Message-based Communication

- Already well-known from *“Betriebssysteme und Sicherheit”*:
Variations of send() and receive()
 - synchronous / asynchronous (blocking / non-blocking)
 - buffered / not buffered
 - direct / indirect addressing
 - fixed / variable message sizes
 - symmetric / asymmetric communication
 - with / without timeout
 - broadcast / multicast

Agenda

- Communication and Synchronization
- IPC via Shared Memory
 - Semaphore, Monitor, Path Expressions
- IPC via Messages
 - Send/Receive
- **Basic Abstractions in Operating Systems**
- Duality of Concepts
- Summary

Basic Abstractions

- Which basic IPC abstractions do operating systems offer?
 - **UNIX:** Sockets, System V Semaphore, messages, shared memory
 - **Windows NT/2000/...:** Shared memory, events, Semaphore, Mutant, sockets, asynchronous I/O, ...
 - **Mach:** Messages to *ports* and shared memory (with copy-on-write)
- System-internal abstractions
 - **Practically always:** Semaphore
 - Mutual exclusion & unilateral synchronization → very common use cases
 - **Microkernels** and **distributed operating systems:** Messages
 - Basis for message implementations: Synchronization primitives
 - **Monolithic systems:** Semaphore and shared memory

Agenda

- Communication and Synchronization
- IPC via Shared Memory
 - Semaphore, Monitor, Path Expressions
- IPC via Messages
 - Send/Receive
- Basic Abstractions in Operating Systems
- **Duality of Concepts**
- Summary

Duality – Messages in Shared Memory

- Semaphores + shared memory → **Mailbox** abstraction

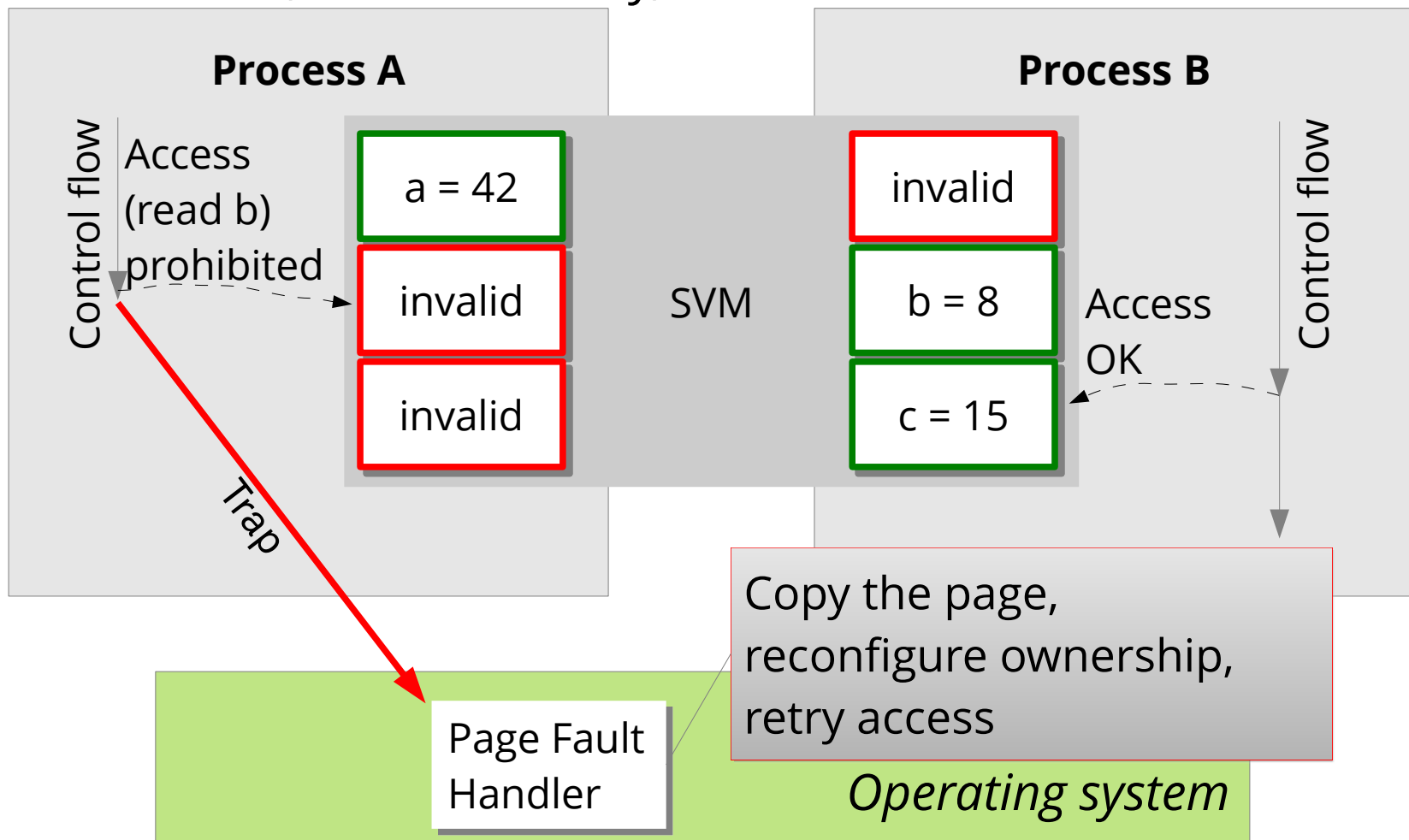
- Messages are not copied
 - Sender provides memory
- Receive may block
- Mailbox abstraction allows for M:N IPC

```
class Mailbox : public List {
    Semaphore mutex(1);
    Semaphore has_elem(0);
public:
    void send(Message *msg) {
        mutex.p();
        enqueue(msg); // from List
        mutex.v();
        has_elem.v();
    }
    Message *receive() {
        has_elem.p();
        mutex.p();
        Message *result = dequeue(); // List
        mutex.v();
        return result;
    }
};
```

Duality – Shared Memory with Messages

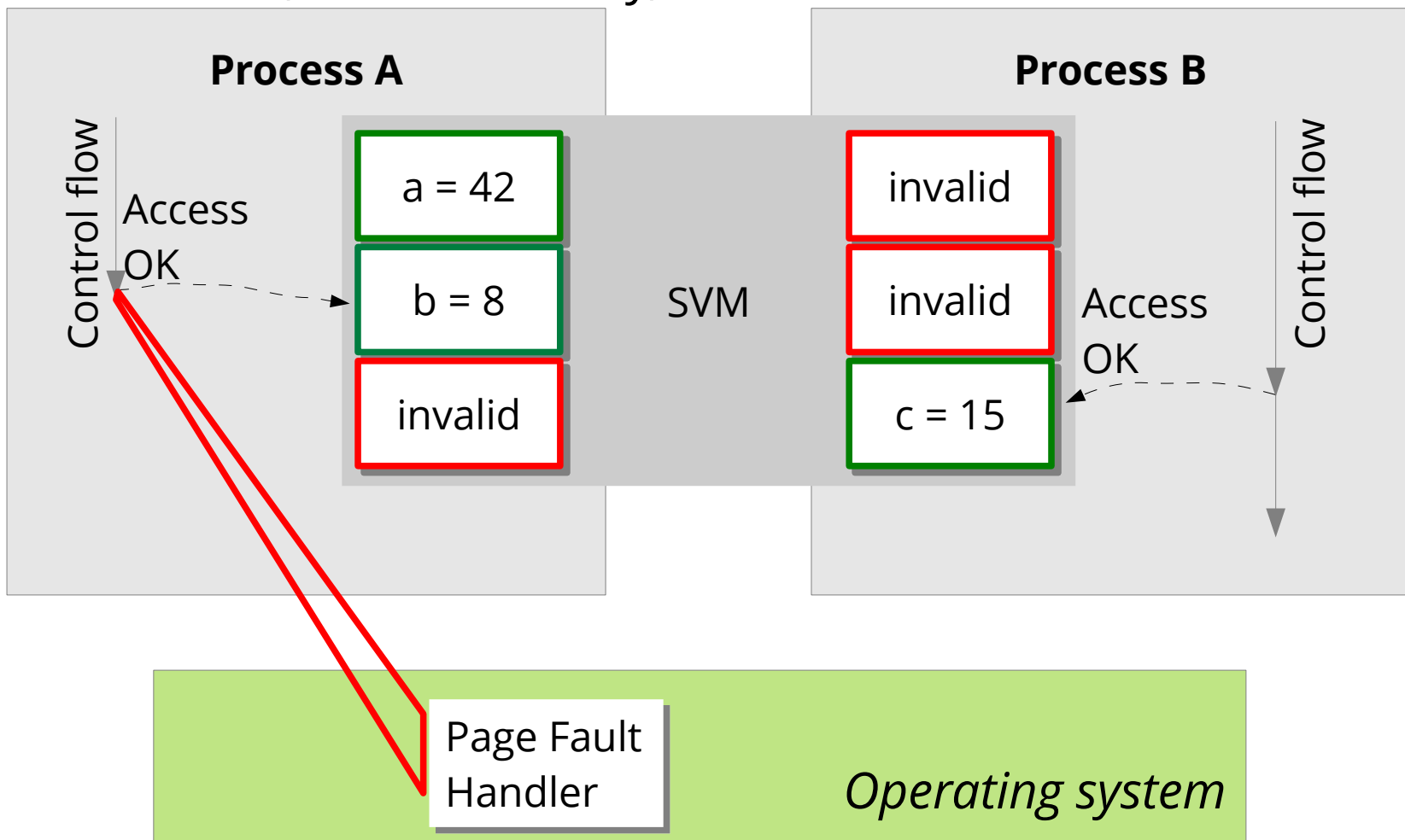
[3] K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*,

PhD Thesis, Yale University, 1986



Duality – Shared Memory with Messages

[3] K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*,
PhD Thesis, Yale University, 1986



Duality – Discussion SVM

- Distributed virtual shared memory allows ...
 - to apply the multiprocessor programming model on distributed systems
 - IPC via (virtual) shared memory in spite of isolated address spaces
- Problems:
 - Communication and trap-handling latency
 - “False sharing” – Page size does not match object size
- Approaches:
 - Weak consistency models, e.g.:
 - Not every access causes a trap, accept outdated values
 - Distribute changes asynchronously via broadcast / multicast

Duality – Active Objects

- Objects with control flow
- Suited for access synchronization in systems with message-based IPC

```
void client1() {  
    Message msg(DO_THIS);  
    send(srv, msg);  
}
```

```
void client2() {  
    Message msg(DO_THAT);  
    send(srv, msg);  
}
```

Mutual exclusion guaranteed by processing loop in the server. Synchronous *send* blocks a client as long as the server is still busy.

- just like a monitor

```
class Server : public ActiveObject {  
    Msg msg; // Message buffer  
public:  
    ...  
    // Object with control flow!  
    void action() {  
        while (true) {  
            receive(ANY, msg); // receive msg.  
            switch (msg.type()) {  
                case DO_THIS: doThis(); break;  
                case DO_THAT: doThat(); break;  
                default:      handleError();  
            }  
            reply(msg);  
        }  
    }  
};
```

Duality – Active Objects

- Reader—writer synchronization with message exchange

```
void reader() {  
    Msg start_read(START_READ);  
    send(srv, start_read);  
    Msg read_msg(DO_READ);  
    send(srv, read_msg);  
    Msg end_read(END_READ);  
    send(srv, end_read);  
    // use data in 'read_msg'  
}
```

```
void writer() {  
    Msg start_write(START_WRITE);  
    send(srv, start_write);  
    // fill message here  
    Msg write_msg(DO_WRITE);  
    send(srv, write_msg);  
    Msg end_write(END_WRITE);  
    send(srv, end_write);  
}
```

```
class RWServer : public ActiveObject {  
    Msg msg; // Message buffer  
public:  
    ...  
    // Control flow  
    void action() {  
        while (true) {  
            receive(ANY, msg); // receive msg.  
            switch (msg.type()) {  
                case START_READ:  startRead();  break;  
                case DO_READ:     doRead();     break;  
                case END_READ:    endRead();    break;  
                case START_WRITE: startWrite(); break;  
                case DO_WRITE:    doWrite();    break;  
                case END_WRITE:   endWrite();   break;  
                default: msg.type(ERROR); reply(msg);  
            }  
        }  
    }  
};
```

Duality – Active Objects

- Reader—writer synchronization with message exchange
 - Actual read/write operations happen concurrently in a child process

The 'request' message must be copied because it could be overwritten while the child process is being executed.

```
void RWServer::doRead() {  
    Msg copy=msg;  
    if (fork()==0) {  
        // actual read op.  
        copy.set(...) // reply  
        reply(copy);  
    }  
    else {  
    } // Parent proc.: nothing  
}
```

```
void RWServer::doWrite() {  
    Msg copy=msg;  
    if (fork()==0) {  
        // actual write op.  
        // (uses 'copy')  
        reply(copy);  
    }  
    else {  
    } // Parent process: nothing  
}
```

The server process can immediately wait for more requests.

Duality – Active Objects

- Reader—writer synchronization with message exchange

```
void RWServer::startRead() {
    ar++;
    if (aw>0)
        read.copy_enqueue(msg);
    else {
        rr++; reply(msg);
    }
}

void RWServer::endRead() {
    ar--; rr--;
    if (rr==0 && aw>0) {
        Msg wmsg=write.dequeue();
        ww++; reply(wmsg);
    }
    reply(msg);
}
```

```
void RWServer::startWrite() {
    aw++;
    if (ww>0 || rr>0)
        write.copy_enqueue(msg);
    else {
        ww++; reply(msg);
    }
}

void RWServer::endWrite() {
    aw--; ww--;
    if (aw>0) {
        Msg wmsg=write.dequeue();
        ww++; reply(wmsg);
    }
    else while (rr < ar) {
        Msg rmsg=read.dequeue();
        rr++; reply(rmsg);
    }
    reply(msg);
}
```

Duality – Discussion

- Is there a **fundamental difference** between IPC via shared memory and IPC via messages?
 - or more provocatively: Which is better – **microkernels or monoliths**?
- Example: Reader—writer monitor vs. server:
 - Monitor: 2 potential waiting points
 - Client is delayed for mutual exclusion
 - Client is potentially further delayed due to a condition variable
 - Server: 2 potential waiting points
 - Reply is delayed because the server serves other requests
 - Reply is potentially further delayed if the request must be enqueued in a waiting queue
- Conclusion: Synchronization and concurrency identical!

Agenda

- Communication and Synchronization
- IPC via Shared Memory
 - Semaphore, Monitor, Path Expressions
- IPC via Messages
 - Send/Receive
- Basic Abstractions in Operating Systems
- Duality of Concepts
- **Summary**

Summary

- Two central IPC-mechanism classes:
 - IPC via shared memory
 - Message-based IPC
- Mechanisms of both classes exist in real-world OSs
 - However, language mechanisms like monitors and path expressions usually cannot be used in OS development
- Neither class is generally better regarding synchronization behavior and degree of concurrency
 - Advantages and disadvantages lie in other properties (see slides 8 and 23)

Bibliography

- [1] C. A. R. Hoare, *Monitor – An Operating System Structuring Concept*, Communications of the ACM 17, 10, S. 549-557, 1974
- [2] R. H. Campbell and A. N. Habermann, *The Specification of Process Synchronization by Path Expressions*, Lecture Note in Computer Science 16, Springer, 1974
- [3] K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, PhD Thesis, Yale University, 1986