

Betriebssysteme (BS)

VL 13.1 – Interprozesskommunikation

–

Speicher-/Nachrichten-Kommunikation

Volkmar Sieh / Daniel Lohmann

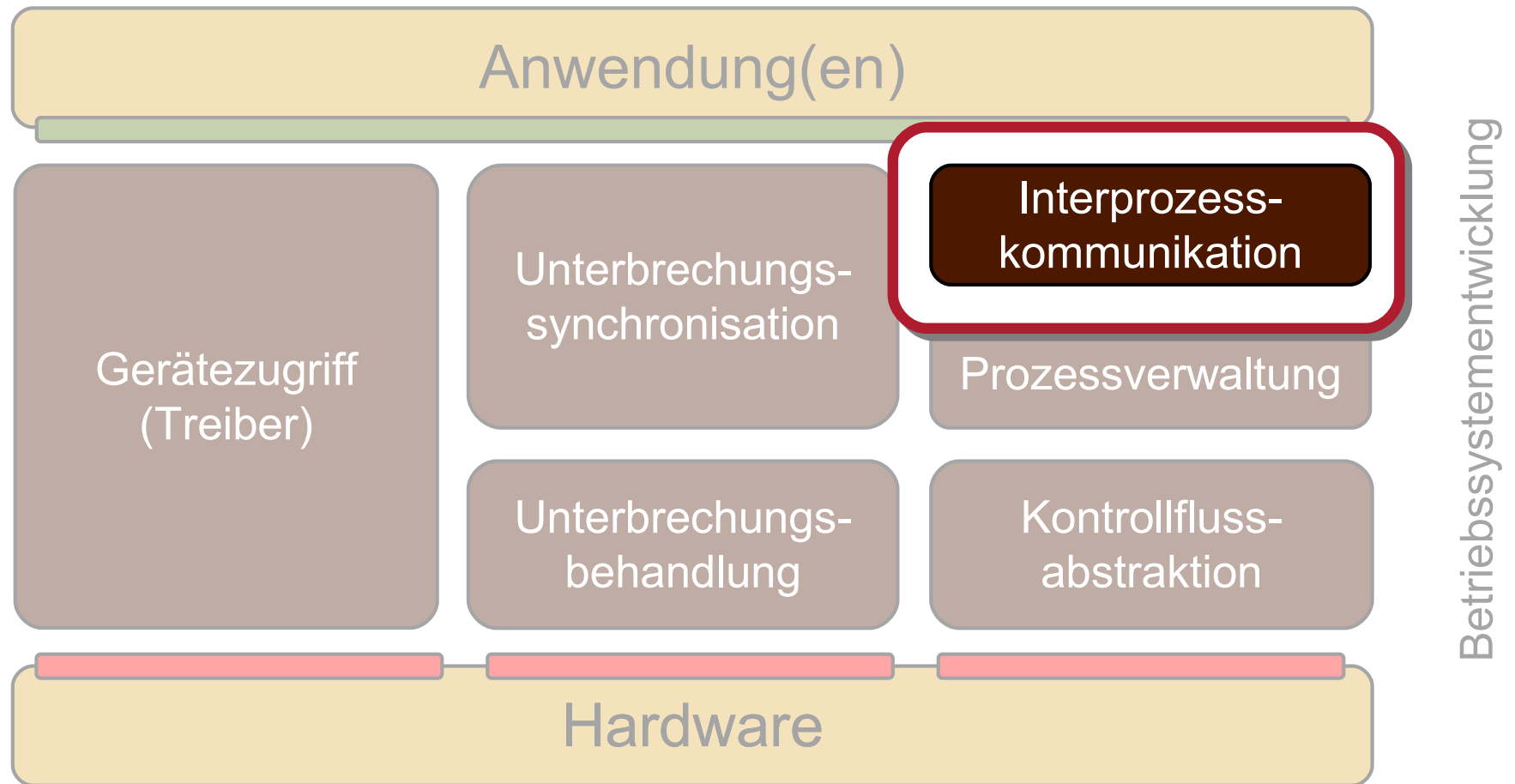
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 21 – 31. Januar 2022



Überblick: Einordnung dieser VL



Agenda

Einordnung
IPC über Speicher
IPC über Nachrichten
Basisabstraktionen
Trennung der Belange mit AOP
Zusammenfassung



Einordnung

Kommunikation und Synchronisation

IPC über Speicher

IPC über Nachrichten

Basisabstraktionen

Trennung der Belange mit AOP

Zusammenfassung



Kommunikation und Synchronisation

- ... sind durch das Kausalprinzip immer verbunden:

Wenn **A** eine Information von **B** benötigt, um weiterzuarbeiten, muss **A** solange *warten*, bis **B** die Information bereitstellt.

- nachrichtenbasierte Kommunikation impliziert Synchronisation (z.B. bei `send()` und `receive()`)
- Synchronisationsprimitiven eignen sich als Basis für die Implementierung von Kommunikationsprimitiven (z.B. Semaphore)



Agenda

Einordnung

IPC über Speicher

Monitore

Pfadausdrücke

IPC über Nachrichten

Basisabstraktionen

Trennung der Belange mit AOP

Zusammenfassung



IPC über gemeinsamen Speicher

- Anwendungsfälle/Voraussetzungen
 - ungeschütztes System (alle Prozesse im selben Adressraum)
 - System mit sprachbasiertem Speicherschutz
 - Kommunikation zwischen Fäden im selben Adressraum
 - gemeinsamer Speicher mit Hilfe des BS und einer MMU (z.B. UNIX System V shared memory)
 - gemeinsamer Kern-Adressraum von isolierten Prozessen
- positive Eigenschaften:
 - atomare Speicherzugriffe erfordern keine zusätzliche Synchronisation
 - schnell: kein Kopieren
 - einfache IPC Anwendungen leicht zu realisieren
 - unsynchronisierte Kommunikationsbeziehungen möglich
 - M:N Kommunikation leicht möglich



Semaphore – einfache Interaktionen

■ gegenseitiger Ausschluss

```
// gem. Speicher  
Semaphore mutex(1);  
SomeType  shared;
```

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

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

■ einseitige Synchronisation

```
// gem. Speicher  
Semaphore elem(0);  
SomeQueue shared;
```

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

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

■ betriebsmittelorientierte Synchronisation

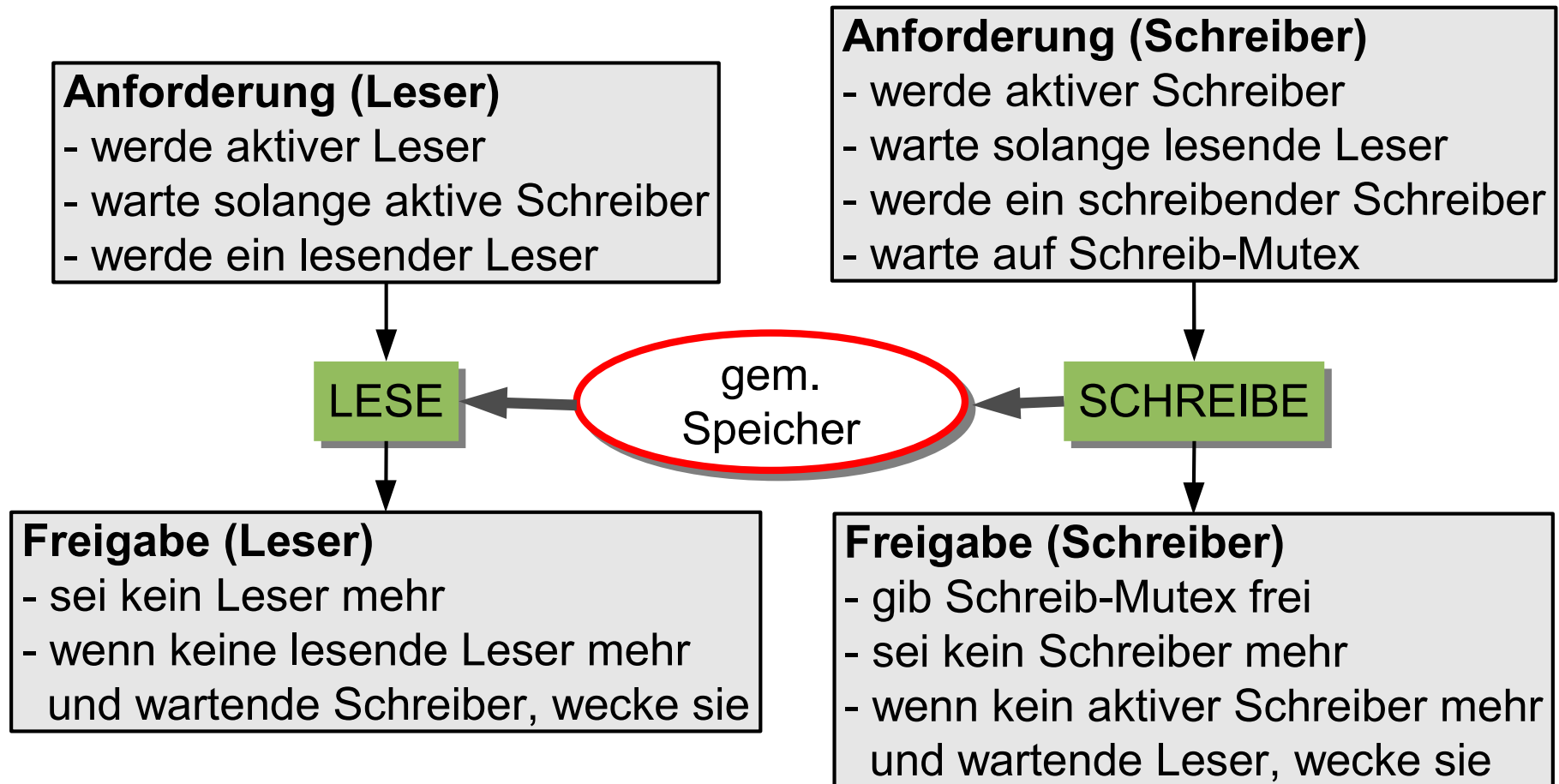
```
// gem. Speicher  
Semaphore    resource(N); // N>1  
SomeResource shared;
```

sonst wie beim
gegenseitigen Ausschluss



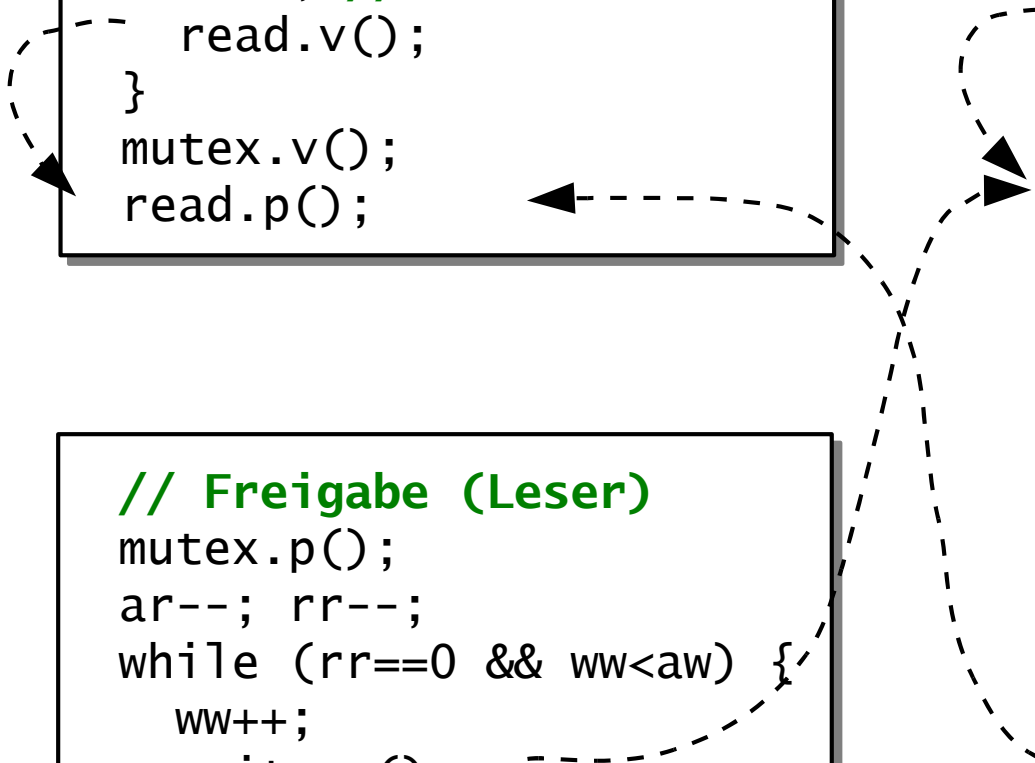
Semaphore – komplexe_{re} Interaktionen

- Leser/Schreiber-Problem
 - Schreiber benötigen den Speicher exklusiv
 - mehrere Leser können gleichzeitig arbeiten



Semaphore – Leser/Schreiber-Problem

```
// Anforderung (Leser)
mutex.p();
ar++; // aktive Leser
if (aw==0) {
    rr++; // lesende Leser
    read.v();
}
mutex.v();
read.p();
```



```
// Anforderung (Schreiber)
mutex.p();
aw++; // aktive Schreiber
if (rr==0) {
    ww++; // schreibende S.
    write.v();
}
mutex.v();
write.p();
w_mutex.p();
```

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

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



Semaphore – Diskussion

■ Erweiterungen

- nicht-blockierendes $p()$
- *Timeout*
- Felder von Zählern

■ Fehlerquellen

- Semaphorebenutzung wird nicht erzwungen
- Abhängigkeit kooperierender Prozesse
 - jeder muss die Protokolle exakt einhalten
- Aufwand bei der Implementierung

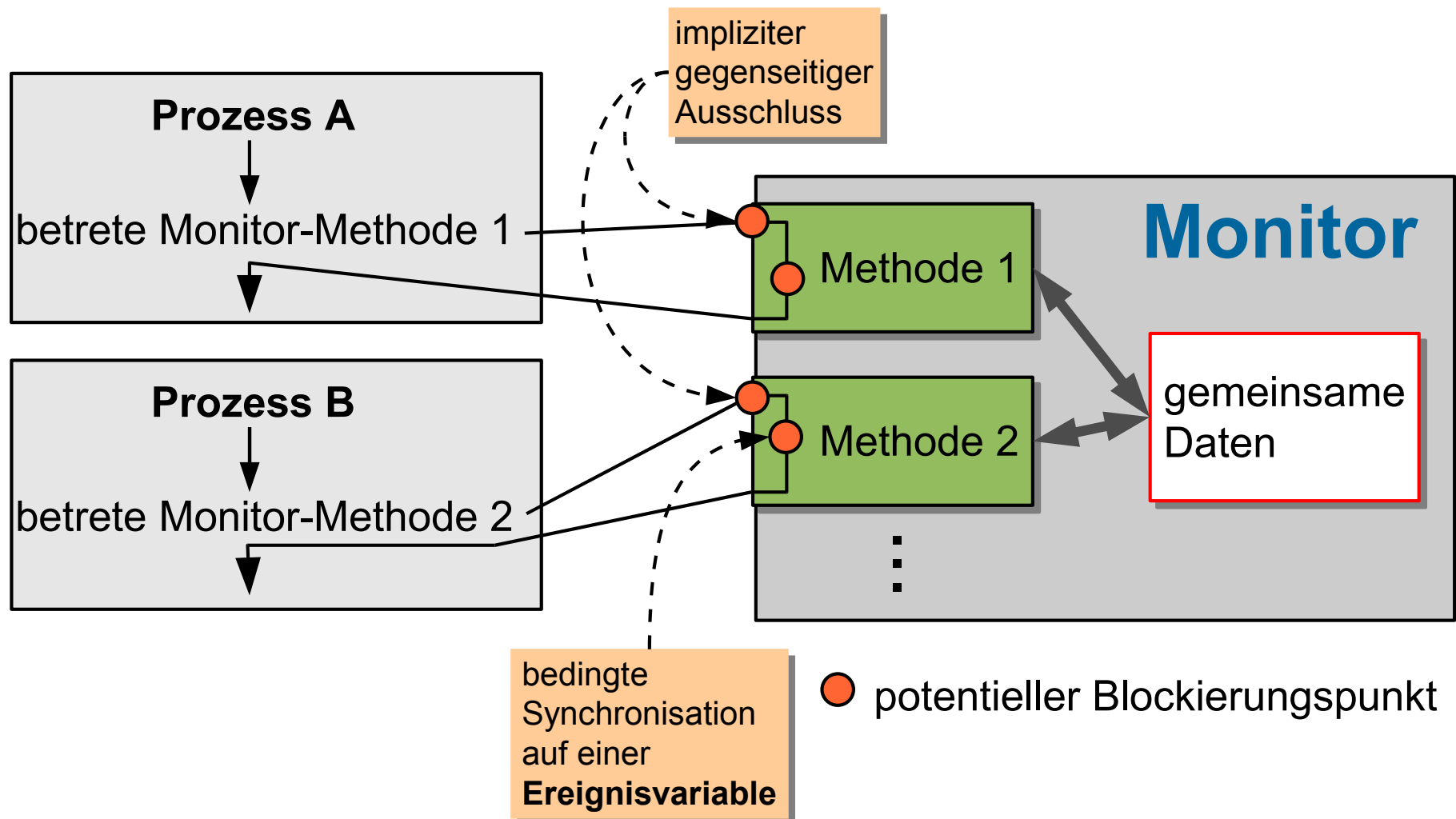
➔ Unterstützung durch die Programmiersprache

- Korrekte Synchronisation wird erzwungen

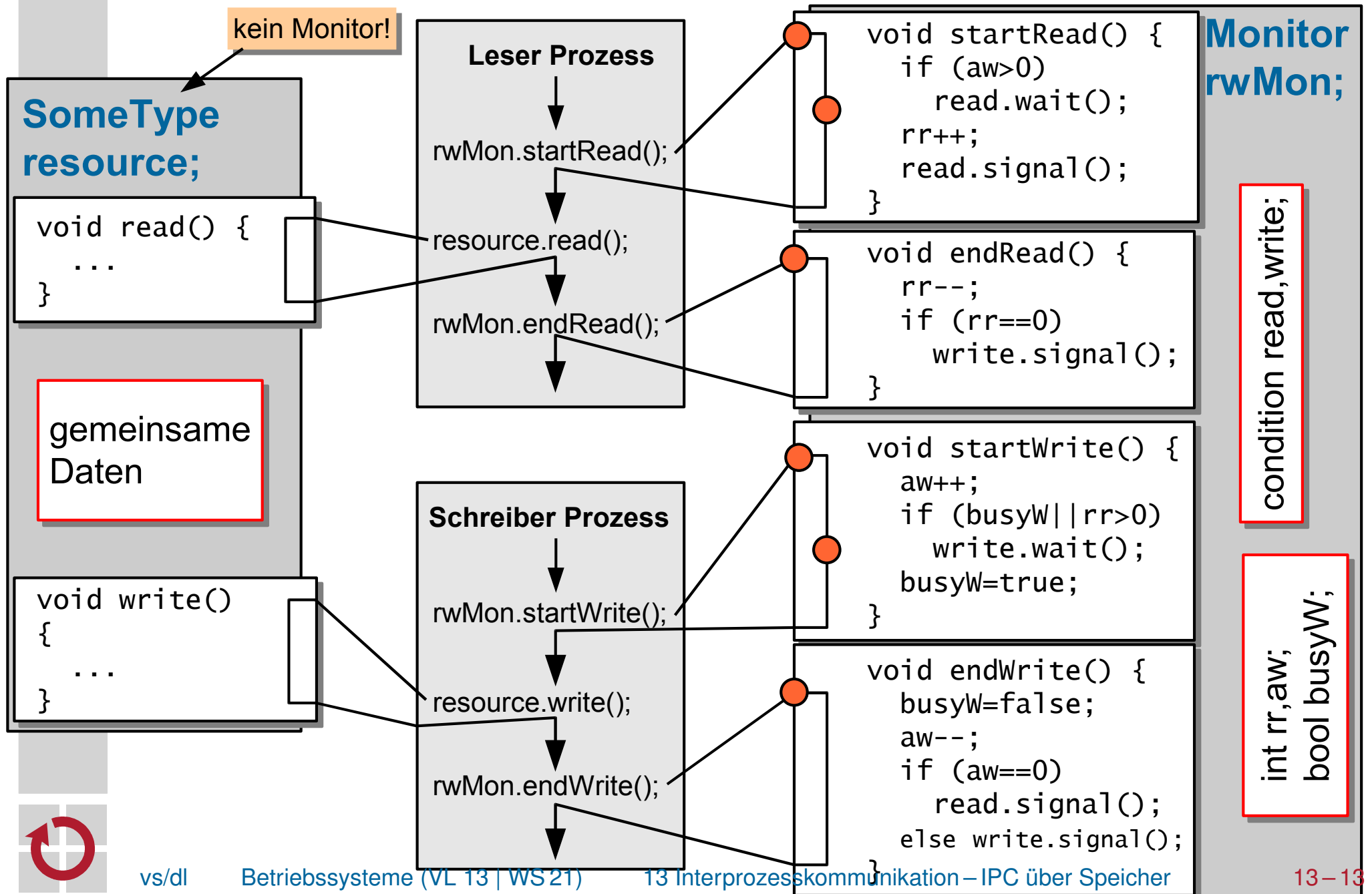


Monitore – synchronisierte ADTs [1]

- Ansatz: Abstrakte Datentypen werden mit Synchronisationseigenschaften gekoppelt



Monitore – Leser/Schreiber-Problem



Monitore – Implementierung

■ ... auf Basis von Semaphoren

einfache Implementierung, die nur *eine* Bedingungsvariable unterstützt.

```
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();  
    ...  
    // ende  
    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--;  
    }  
}
```



Monitore – Diskussion

- Einschränkung der Nebenläufigkeit auf vollständigen gegenseitigen Ausschluss.
 - in Java daher 'synchronized' auch für einzelne Methoden
- Kopplung von logischer Struktur und Synchronisation ist jedoch nicht immer natürlich.
 - siehe Leser/Schreiber Beispiel
 - gleiches Problem wie beim Semaphor:
Programmierer müssen ein Protokoll einhalten
- ➔ Die Synchronisation sollte von der Organisation der Daten und Methoden besser getrennt werden.



Pfadausdrücke [2]

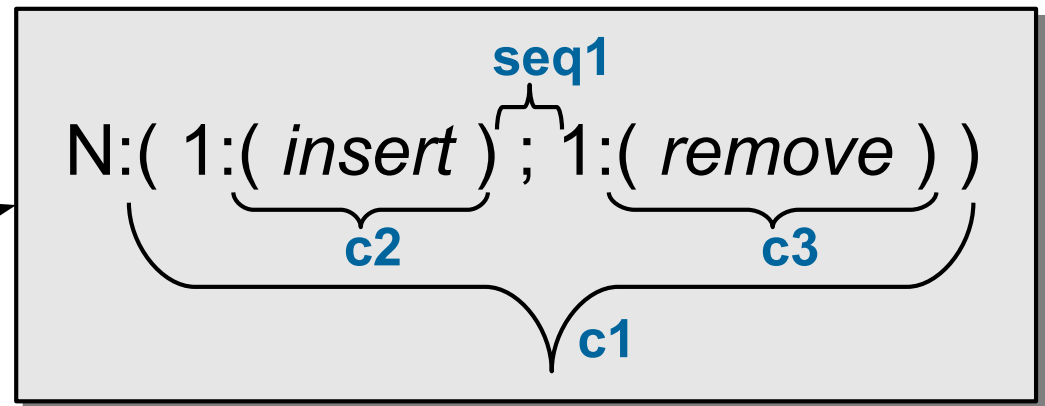
- Idee: flexible Ausdrücke beschreiben erlaubte Reihenfolgen und den Grad der Nebenläufigkeit.
- **path *name1*, *name2*, *name3* end**
 - bel. Reihenfolge und bel. nebenläufige Ausführung von *name1-3*
- **path *name1*; *name2* end**
 - vor jeder Ausführung von *name2* mindestens einmal *name1*
- **path *name1* + *name2* end**
 - alternative Ausführung: entweder *name1* oder *name2*
- **path 2:(*Pfadausdruck*) end**
 - max. 2 Kontrollflüsse dürfen gleichzeitig im *Pfadausdruck* sein
- **path N:(1:(*insert*); 1:(*remove*)) end**
 - z.B. Synchronisation eines N-elementigen Puffers
 - gegenseitiger Ausschluss während *insert* und *remove*
 - vor jedem *remove* muss mindestens ein *insert* erfolgt sein
 - nie mehr als N abgeschlossene *insert*-Operationen



Pfadausdrücke – Implementierung (1)

- Transformation in Zustandsautomaten
- Zustandsänderung bei Ein-/Austritt in die/aus der Operation
- Beispiel:

für jedes 'X:(..)' und
';' wird ein Zähler
eingeführt.



```
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--;
}
```



Pfadausdrücke – Implementierung (2)

■ Transformation der Operationen

für jede Operation
wird ein Semaphor
und ein Zähler
eingeführt.

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

```
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();  
    // original 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;  
}
```



■ Vorteile

- komplexere Interaktionsmuster als mit Monitoren möglich
 - read + 1: write
- Einhaltung der Interaktionsprotokolle wird erzwungen
 - weniger Fehler!

■ Nachteile

- Synchronisationsverhalten kann nicht von Zustandsvariablen oder Parametern abhängen
 - Erweiterung: Pfadausdrücke mit Prädikaten
- Synchronisation des Zustandsautomaten kann Flaschenhals werden
- keine Unterstützung für Pfadausdrücke in gebräuchlichen Programmiersprachen



Agenda

Einordnung

IPC über Speicher

IPC über Nachrichten

Basisabstraktionen

Trennung der Belange mit AOP

Zusammenfassung



- Anwendungsfälle/Voraussetzungen
 - IPC über Rechnergrenzen
 - Interaktion isolierter Prozesse

- positive Eigenschaften:
 - einheitliches Paradigma für IPC mit lokalen und entfernten Prozessen
 - ggf. Pufferung und Synchronisation
 - Indirektion erlaubt transparente Protokollerweiterungen
 - Verschlüsselung, Fehlerkorrektur, ...
 - Hochsprachenmechanismen wie OO-Nachrichten oder Prozeduraufrufe lassen sich gut auf IPC über Nachrichten abbilden (RPC, RMI)



- Bekannt (aus SOS):
Variationen von `send()` und `receive()`
 - synchron/asynchron (blockierend/nicht blockierend)
 - gepuffert/ungepuffert
 - direkt/indirekt
 - feste Nachrichtengröße/variable Größe
 - symmetrische/asymmetrische Kommunikation
 - mit/ohne *Timeout*
 - *Broadcast/Multicast*

