

Betriebssysteme (BS)

VL 13 – Interprozesskommunikation

Volkmar Sieh / Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

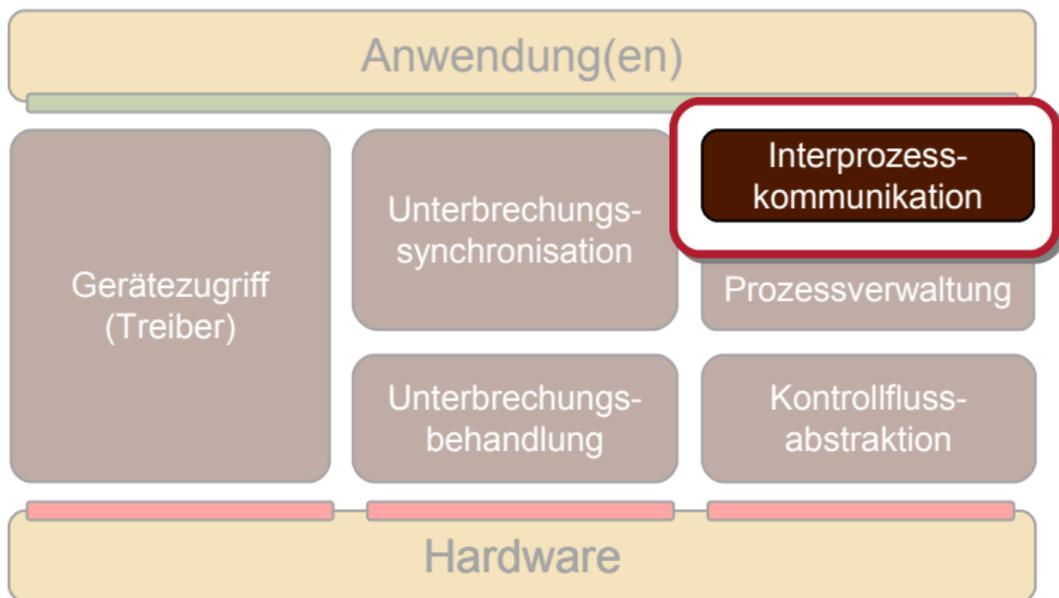
Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 23 – 31. Januar 2024

<https://sys.cs.fau.de/lehre/ws23/bs>



Überblick: Einordnung dieser VL



Betriebssystementwicklung



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)



Einordnung

IPC über Speicher

Monitore

Pfadausdrücke

Thread-Safety-Analyse

IPC über Nachrichten

Basisabstraktionen

Trennung der Belange mit AOP

Zusammenfassung



- 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

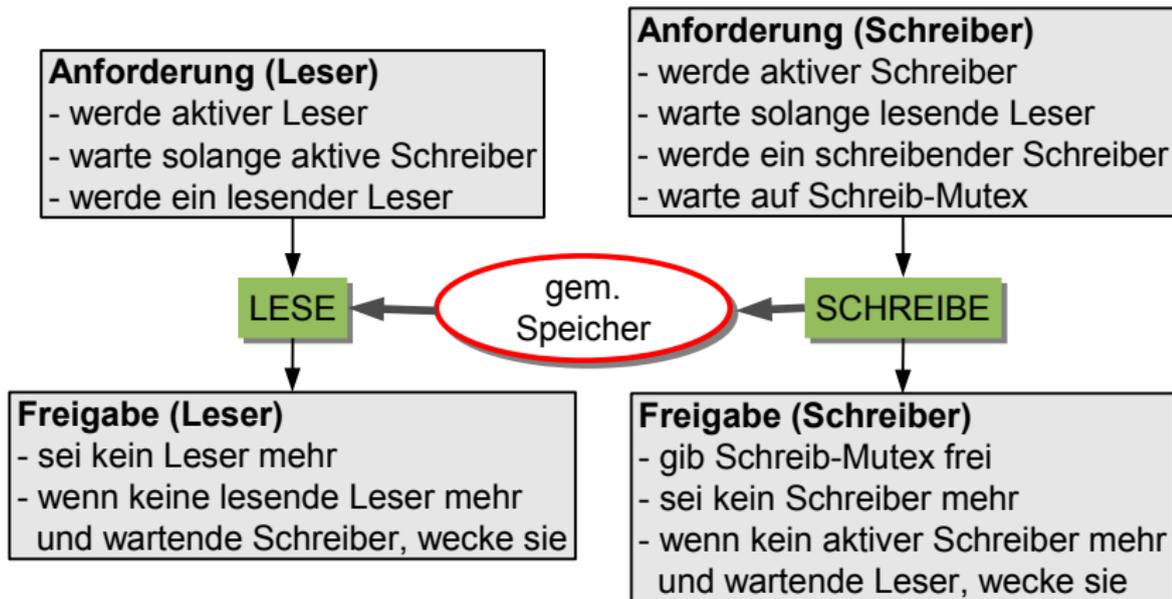
```
// gem. Speicher  
Semaphore resource(N); // N>1  
SomeResource shared;
```

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();
```

■ 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

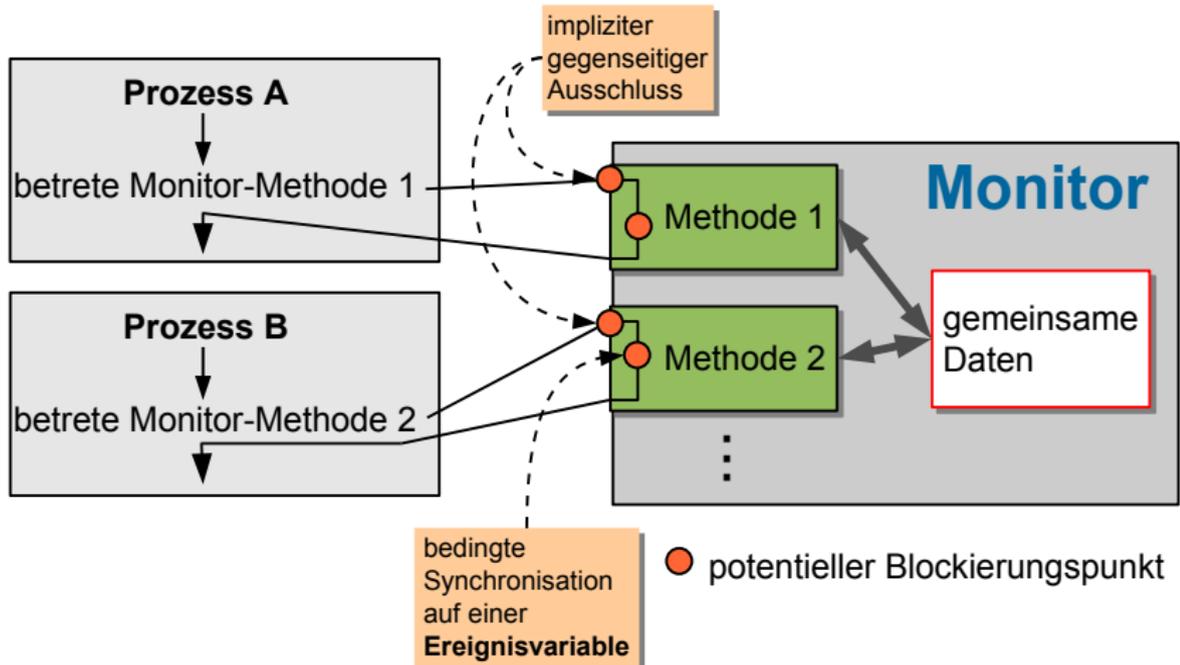
→ Unterstützung durch die Programmiersprache

- Korrekte Synchronisation wird erzwungen



Monitore – synchronisierte ADTs [1]

- Ansatz: Abstrakte Datentypen werden mit Synchronisationseigenschaften gekoppelt



Monitore – Leser/Schreiber-Problem

kein Monitor!

SomeType
resource;

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

gemeinsame
Daten

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

Leser Prozess

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

Schreiber Prozess

```
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;



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]

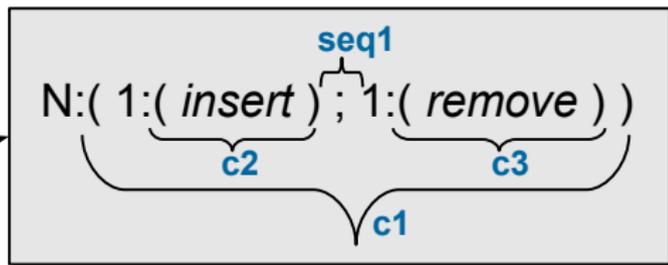
- Idee: flexible Ausdrücke beschreiben erlaubte Reihenfolgen und den Grad der Nebenläufigkeit.
- **path *name1*, *name2*, *name3* end**
 - bel. Reihenfolge und bel. nebenläufige Ausführung von *name1-3*
- **path *name1*; *name2* end**
 - vor jeder Ausführung von *name2* mindestens einmal *name1*
- **path *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



Thread-Safety-Analyse (clang) – Motivation

```
typedef struct { ... } lock_t;
extern void lock(lock_t *l);
extern void unlock(lock_t *l);

lock_t l;
int sum;

extern void alert(void);

int update(int c) {
    lock(&l);
    sum += c;
    if (0 <= sum) {
        unlock(&l)
        return 1;
    }
    alert();

    unlock(&l);
    return 0;
}
```

<= kann man vergessen

<= kann man leicht vergessen

<= kann man auch ohne lock aufrufen

<= kann man vergessen

... und der Compiler warnt nicht!



Annotieren des Programms mit Attributen:

```
#define CAPABILITY(x)    attribute((capability(x)))  
#define GUARDED_BY(x)   attribute((guarded_by(x)))  
#define ACQUIRE(x)     attribute((acquire_capability(x)))  
#define RELEASE(x)      attribute((release_capability(x)))  
#define REQUIRES(x)     attribute((requires_capability(x)))
```

- das Halten eines Locks ist eine „Capability“ (Befähigung)
- Zugriffe auf Variablen können geschützt sein („guarded_by“)
- Capabilities kann man erlangen („acquire“)
- Capabilities kann man abgeben („release“)
- Capabilities können notwendig sein („require“)



Thread-Safety-Analyse – Umsetzung

```
typedef struct { ... } lock_t CAPABILITY;
extern void lock(lock_t *l) ACQUIRE(l);
extern void unlock(lock_t *l) RELEASE(l);

lock_t l;
int sum GUARDED_BY(l);

extern void alert(void) REQUIRES(l);

int update(int c) {
    lock(&l);
    sum += c;
    if (0 <= sum) {
        unlock(&l);
        return 1;
    }
    alert();

    unlock(&l);
    return 0;
}
```

... und der Compiler warnt ggf.!



```
typedef struct { ... } irq_disabled_t CAPABILITY;
extern irq_disabled_t irq_disabled;
extern void irq_disable(void) ACQUIRE(irq_disabled);
extern void irq_enable(void) RELEASE(irq_disabled);

char buf GUARDED_BY(irq_disabled);

char kbd_get(void)
{
    irq_disable();
    char c = buf;
    irq_enable();
    return c;
}

int kbd_prologue(void) REQUIRES(irq_disabled)
{
    buf = ...;
    return 1;
}
```

Aber: kbd_get könnte mit ausgeschalteten Interrupts aufgerufen werden!



Thread-Safety-Analyse – virtuelle Capabilities

```
typedef struct { ... } irq_enabled_t CAPABILITY;
typedef struct { ... } irq_disabled_t CAPABILITY;
extern irq_enabled_t irq_enabled;
extern irq_disabled_t irq_disabled;
extern void irq_disable(void) RELEASE(irq_enabled) ACQUIRE(irq_disabled);
extern void irq_enable(void) RELEASE(irq_disabled) ACQUIRE(irq_enabled);

char buf GUARDED_BY(irq_disabled);

char kbd_get(void) REQUIRES(irq_enabled)
{
    irq_disable();
    char c = buf;
    irq_enable();
    return c;
}

int kbd_prologue(void) REQUIRES(irq_disabled)
{
    buf = ...;
    return 1;
}
```



Vier Bedingungen für einen Deadlock (mit Locks):

- 1 Locks können nicht entzogen werden
- 2 Fäden halten Locks und wollen weitere
- 3 Locks können nicht geshared werden
- 4 es können Zyklen auftreten

Deadlock-Verhinderung:

- Punkte 1-3 können i.A. nicht geändert werden
- Punkt 4: Zyklen verbieten

=> Lock-Ebenen einführen



Thread-Safety-Analyse – Deadlock-Verhinderung

```
extern void lock_0(lock_0_t *l) RELEASE(level_0) ACQUIRE(l);
extern void unlock_0(lock_0_t *l) RELEASE(l) ACQUIRE(level_0);
extern void lock_1(lock_1_t *l) RELEASE(level_1) ACQUIRE(l);
extern void unlock_1(lock_1_t *l) RELEASE(l) ACQUIRE(level_1);

lock_0_t l_0;
int var_0 GUARDED_BY(l_0);
lock_1_t l_1;
int var_1 GUARDED_BY(l_1);

void bar(void) REQUIRES(level_0)
{
    lock_0(&l_0);
    var_0 = ...;
    unlock_0(&l_0);
}

void foo(void) REQUIRES(level_0) REQUIRES(level_1)
{
    lock_1(&l_1);
    var_1 = ...;
    bar();
    unlock_1(&l_1);
}
```



Thread-Safety-Analyse (clang)

Zusammenfassung/Erfahrungen clang-Thread-Safety-Analyse:

- elementare Funktionen, die Capabilities erzeugen/abgeben, müssen mit „NO_THREAD_SAFETY_ANALYSIS“ markiert werden

```
void irq_disable(void)
    RELEASE(irq_enabled) ACQUIRE(irq_disabled)
    NO_THREAD_SAFETY_ANALYSIS
{
    asm volatile ("cli");
}
```

- Funktionszeiger können nicht attribuiert werden
- ist eine Art boolesche „und“-Verknüpfung
- „nicht“-Operator über inverse Capabilities
- Tipp-Arbeit
- dokumentiert Code

JITTY Erfahrungen: **sehr** wertvoll!



Einordnung

IPC über Speicher

IPC über Nachrichten

Basisabstraktionen

Trennung der Belange mit AOP

Zusammenfassung



IPC über Nachrichten

- Anwendungsfälle/Voraussetzungen
 - IPC über Rechengrenzen
 - 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*



Einordnung

IPC über Speicher

IPC über Nachrichten

Basisabstraktionen

Windows/Unix/...

Dualität der Konzepte

Trennung der Belange mit AOP

Zusammenfassung



- Welche IPC Basisabstraktionen bieten Betriebssysteme?
 - UNIX-Systeme: Sockets, System V Semaphore, Messages, Shared Memory
 - Windows NT/2000/XP: Shared Memory, Events, Semaphore, Mutant (Mutex), Sockets, Pipes, Named Pipes, Mailslots, ...
 - Mach: Nachrichten an Ports und Shared Memory (mit *Copy on Write*)
- Welche Abstraktionen nutzen die Systeme i.d.R. intern?
 - Semaphore erlauben gegenseitigen Ausschluss und einseitige Synchronisation, also sehr häufige Anwendungsfälle
 - werden praktisch immer benutzt
 - Mikrokerne und verteilte Betriebssysteme: Nachrichten
 - Monolithische Systeme: Semaphore und gemeinsamen Speicher



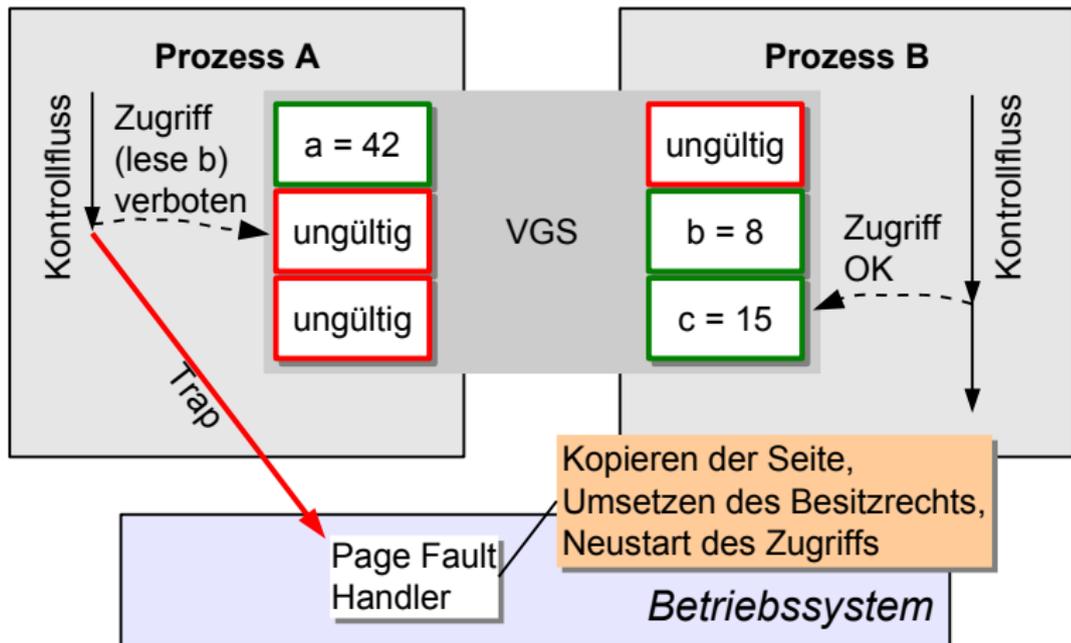
- auf Basis von Semaphoren und gemeinsamem Speicher lässt sich leicht eine *Mailbox*-Abstraktion realisieren:

- Nachrichten werden nicht kopiert
 - Sender sorgt für Speicher
- receive blockiert ggf.
- Mailbox-Abstraktion erlaubt M:N IPC

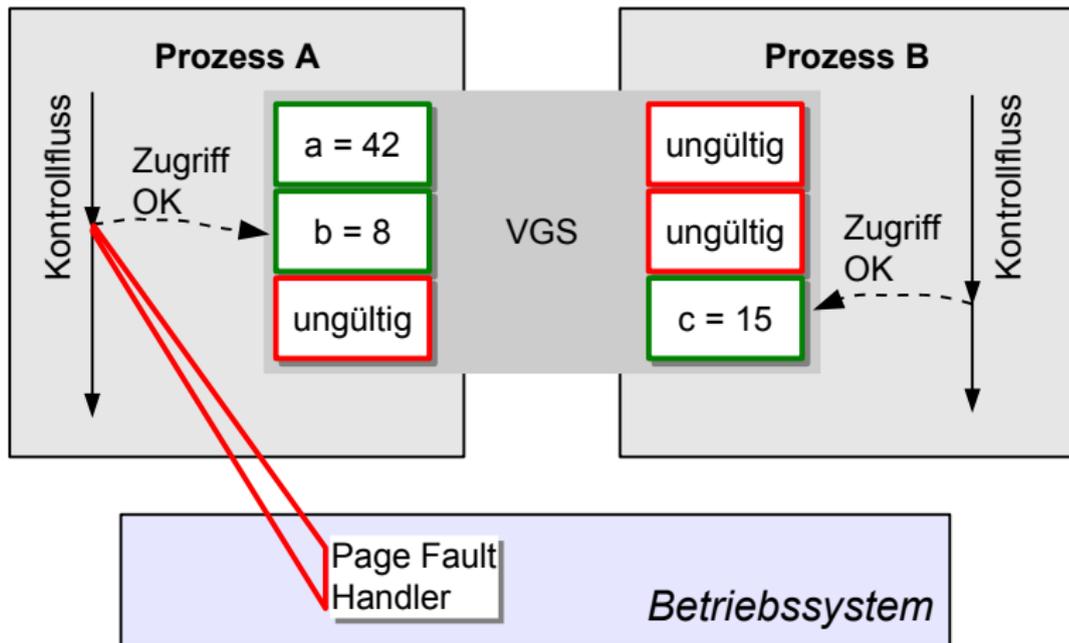
```
class Mailbox : public List {
    Semaphore mutex; // (1)
    Semaphore has_elem; // (0)
public:
    Mailbox() : mutex(1), has_elem(0) {}
    void send(Message *msg) {
        mutex.p();
        enqueue(msg); // aus List
        mutex.v();
        has_elem.v();
    }
    Message *receive() {
   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



■ „Virtueller gemeinsamer Speicher“ (VGS [3])



■ „Virtueller gemeinsamer Speicher“ (VGS)



- Verteilter virtueller gemeinsamer Speicher ermöglicht...
 - das Programmiermodell von Multiprozessoren auf Mehrrechnersystemen zu nutzen
 - IPC über (virtuellen) gemeinsamen Speicher trotz getrennter Adressräume
- Probleme:
 - Latenzen der Kommunikation und Trap-Behandlung
 - „*false sharing*“ - Seitengröße entspricht nicht Objektgröße
- Lösungsansätze:
 - schwache Konsistenzmodelle, z.B.:
 - nicht jeder Zugriff führt zu einem Trap, veraltete Werte werden in Kauf genommen
 - Änderungen asynchron per *Broad-/Multicast* verbreiten



Dualität – Aktive Objekte

- Objekte mit Kontrollfluss
- gut geeignet zur Zugriffssynchronisation in Systemen mit nachrichtenbasierter IPC

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

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

Gegenseitiger Ausschluss durch die Verarbeitungsschleife wird garantiert. Durch das synchrone `send()` blockiert ein Client solange der Server noch beschäftigt ist.

→ genau wie ein **Monitor**

```
class Server : public ActiveObject {  
    Msg msg; // Nachrichtenpuffer  
public:  
    ...  
    // Objekt mit Kontrollfluss!  
    void action() {  
        while (true) {  
            receive(ANY, msg); // empfangen Nachr.  
            switch (msg.type()) {  
                case DO_THIS: doThis(); break;  
                case DO_THAT: doThat(); break;  
                default:      handleError();  
            }  
            reply(msg);  
        }  
    }  
};
```



Dualität – Aktive Objekte

■ Leser/Schreiber Problem mit Nachrichtenaustausch

```
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);  
    // benutze Daten in 'read_msg'  
}
```

```
void writer() {  
    Msg start_write(START_WRITE);  
    send(srv, start_write);  
    // hier Nachricht füllen  
    Msg write_msg(DO_WRITE);  
    send(srv, write_msg);  
    Msg end_write(END_WRITE);  
    send(srv, end_write);  
}
```

```
class RWServer : public ActiveObject {  
    Msg msg; // Nachrichtenpuffer  
public:  
    ...  
    // Kontrollfluss  
    void action() {  
        while (true) {  
            receive(ANY, msg); // empfangen N.  
            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);  
            }  
        }  
    }  
};
```



Dualität – Aktive Objekte

- Leser/Schreiber Problem mit Nachrichtenaustausch
 - die eigentliche Lese- und Schreiboperation erfolgt nebenläufig durch einen Kindprozess

```
void RWServer::doRead() {  
    Msg copy=msg;  
    if (fork()==0) {  
        // das eigentliche Lesen  
        copy.set(...) // Antwort  
        reply(copy);  
    }  
    else {  
    } // Elternprozess: nichts  
}
```

```
void RWServer::doWrite() {  
    Msg copy=msg;  
    if (fork()==0) {  
        // das eigentliche Schreiben  
        // (benutzt 'copy')  
        reply(copy);  
    }  
    else {  
    } // Elternprozess: nichts  
}
```

die 'request' Nachricht muss kopiert werden, da sie während der Ausführung des Kindprozesses überschrieben werden könnte

der Server-Prozess kann sofort wieder auf 'requests' warten



Dualität – Aktive Objekte

■ Leser/Schreiber Problem mit Nachrichtenaustausch

```
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);
}
```

Ergebnis: Die Semantik / Parallelität entspricht der Monitor-basierten Implementierung

Dualität – Diskussion

- Gibt es einen fundamentalen Unterschied zwischen IPC über gem. Speicher und IPC über Nachrichten?
 - zugespitzt: sind oder prozedurorientierte BS (Monolithen) oder prozessorientierte BS (Mikrokern) besser?
- Beispiel: Leser/Schreiber Monitor vs. *Server*:
 - Monitor: 2 potentielle Wartepunkte
 - Client wird verzögert für gegenseitigen Ausschluss.
 - Client wird ggf. wegen einer Ereignisvariablen weiter verzögert.
 - *Server*: 2 potentielle Wartepunkte
 - *Reply* wird verzögert, da der *Server* noch andere *Requests* bearbeitet.
 - *Reply* wird ggf. weiter verzögert, wenn der *Request* in eine Warteschlange gehängt werden muss.
- Fazit: Dualität in Synchronisation und Nebenläufigkeit [4]



Agenda

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



Trennung der Belange mittels AOP

- „Aspektorientierte Programmierung“ erlaubt die *modulare* Implementierung „querschneidender“ Belange
- Beispiel in AspectC++:

```
// Festlegung der Monitore des Systems
pointcut monitors() = "FileTable"||"BufferCache";

// Synchronisation per Aspekt
aspect MonitorSynch {
  advice monitors() : slice struct {
    Semaphore _mutex;
  };
  advice construction(monitors()) : before() {
    tjp->that()->_mutex.init(1);
  }
  advice execution(monitors()) : around() {
    tjp->that()->_mutex.p(); // Monitor sperren
    tjp->proceed();         // Fkt. ausführen
    tjp->that()->_mutex.v(); // Monitor freigeben
  }
};
```

"Einfügung" eines Semaphors in die Monitor-Klassen

"Code-Advice" für Ereignisse im Programmablauf



Agenda

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



Zusammenfassung und Ausblick

- Es gibt zwei Hauptklassen von IPC Mechanismen:
 - IPC über gemeinsamen Speicher
 - nachrichtenbasierte IPC
- Mechanismen beider Klassen sind in realen Betriebssystemen anzutreffen
 - Sprachmechanismen wie Monitore und Pfadausdrücke können bei der BS-Entwicklung allerdings i.d.R. nicht verwendet werden
- Bzgl. des Synchronisationsverhaltens und dem Grad der Nebenläufigkeit zeichnet sich keine Klasse besonders aus
 - Vor- und Nachteile liegen woanders
 - Ausblick: mit AOP Techniken könnte man von den konkreten Kommunikations- und Synchronisations*mechanismen* abstrahieren



- [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
- [4] Lauer, H. C. and Needham, R. M. 1979. On the duality of operating system structures. SIGOPS Oper. Syst. Rev. 13, 2 (Apr. 1979), 3-19

