

Betriebssysteme (BS)

VL 12 – Interprozesskommunikation

Daniel Lohmann / Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 16 – 26. Januar 2017

https://www4.cs.fau.de/Lehre/WS16/V_BS

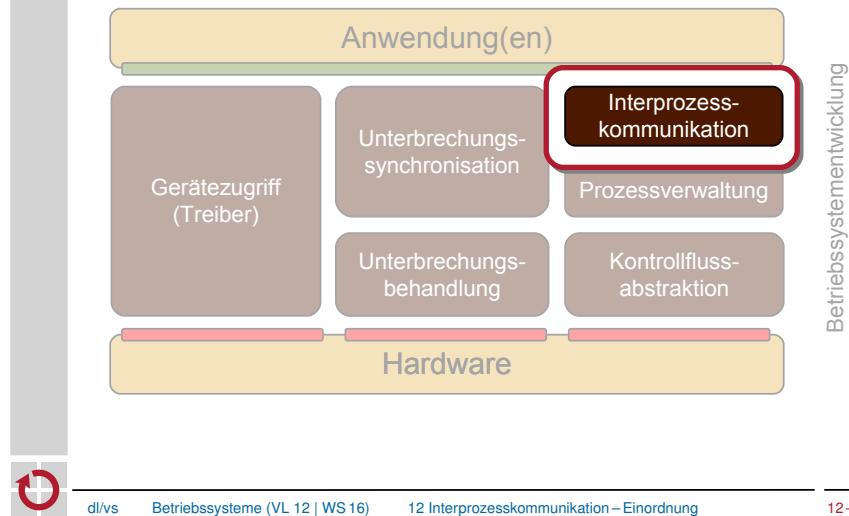


Agenda

- Einordnung
 - Kommunikation und Synchronisation
- IPC über Speicher
 - Semaphor, Monitor, Pfadausdrücke
- IPC über Nachrichten
 - Send/Receive
- Basisabstraktionen
 - Windows/UNIX/...
 - Dualität der Konzepte
- Trennung der Belange mit AOP
- Zusammenfassung



Überblick: Einordnung dieser VL



Betriebssystementwicklung

Agenda

- Einordnung
 - Kommunikation und Synchronisation
- IPC über Speicher
 - Semaphor, Monitor, Pfadausdrücke
- IPC über Nachrichten
 - Send/Receive
- Basisabstraktionen
 - Windows/UNIX/...
 - Dualität der Konzepte
- 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()`)
- Synchronisationsprimitive eignen sich als Basis für die Implementierung von Kommunikationsprimitive (z.B. Semaphore)



Agenda

Einordnung

Kommunikation und Synchronisation

IPC über Speicher

Semaphor, Monitor, Pfadausdrücke

IPC über Nachrichten

Send/Receive

Basisabstraktionen

Windows/UNIX/...

Dualität der Konzepte

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

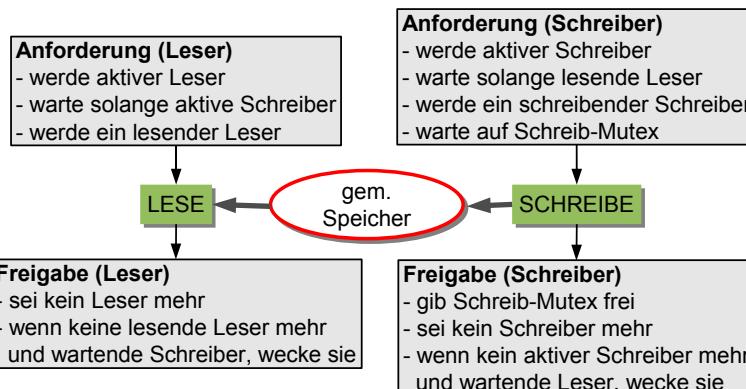
```
// gem. Speicher  
Semaphore resource(N); // N>1  
SomeResource shared;
```

sonst wie beim
gegenseitigen Ausschluss



Semaphore – komplexe_{re} Interaktionen

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

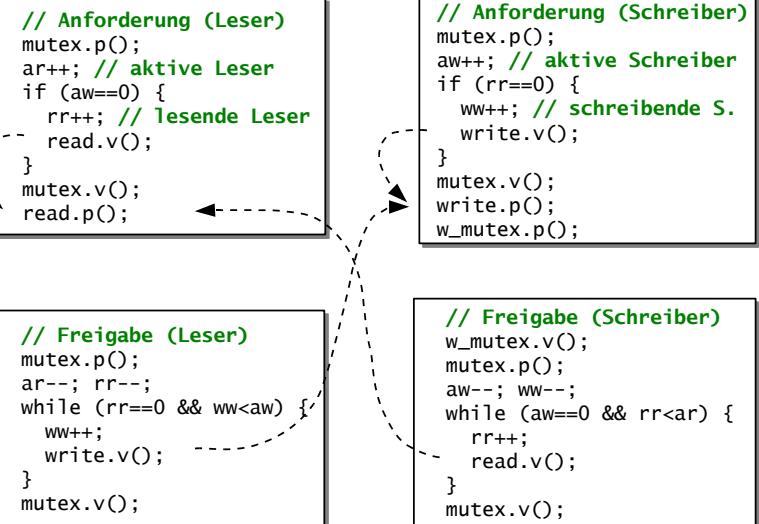


dl/vs Betriebssysteme (VL 12 | WS 16)

12 Interprozesskommunikation – IPC über Speicher

12-9

Semaphore – Leser/Schreiber-Problem



dl/vs Betriebssysteme (VL 12 | WS 16)

12 Interprozesskommunikation – IPC über Speicher

12-10

Semaphore – Diskussion

- Erweiterungen
 - nicht-blockierendes p()
 - *Timeout*
 - Felder von Zählern
- Fehlerquellen
 - Semaphorbenutzung wird nicht erzwungen
 - Abhängigkeit kooperierender Prozesse
 - jeder muss die Protokolle exakt einhalten
 - Aufwand bei der Implementierung
- ➔ Unterstützung durch die Programmiersprache
 - Korrekte Synchronisation wird erzwungen

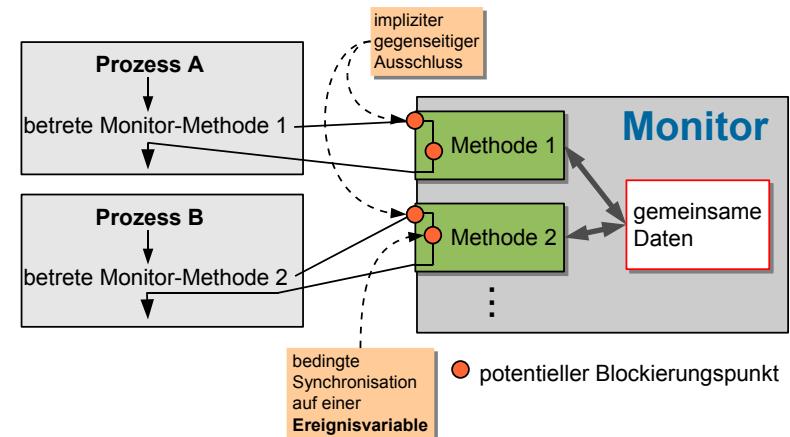
dl/vs Betriebssysteme (VL 12 | WS 16)

12 Interprozesskommunikation – IPC über Speicher

12-11

Monitore – synchronisierte ADTs [1]

- Ansatz: Abstrakte Datentypen werden mit Synchronisationseigenschaften gekoppelt

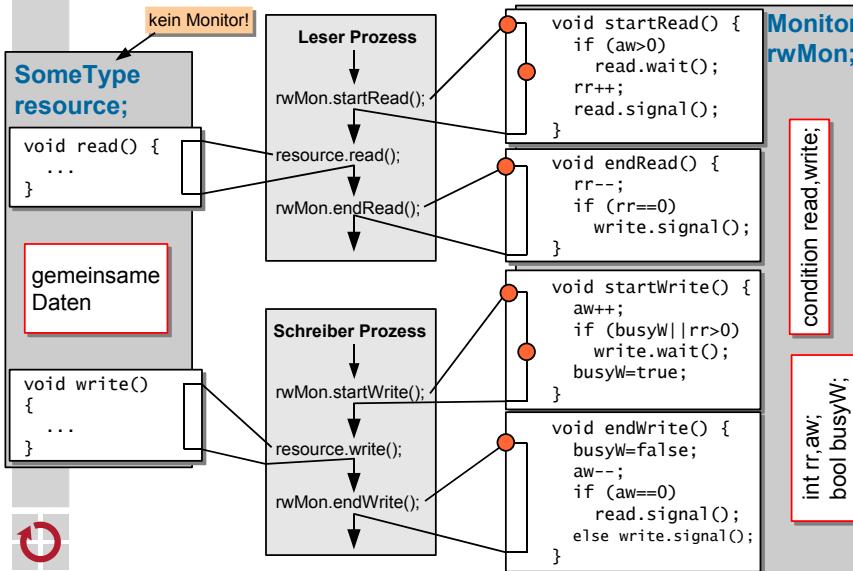


dl/vs Betriebssysteme (VL 12 | WS 16)

12 Interprozesskommunikation – IPC über Speicher

12-12

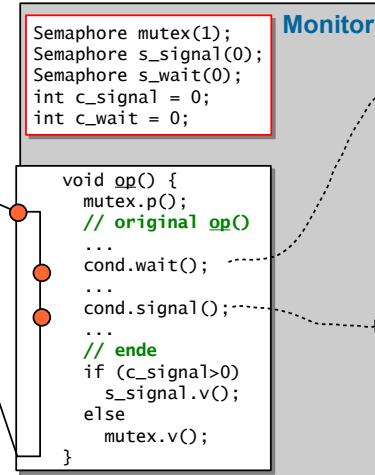
Monitore – Leser/Schreiber-Problem



Monitore – Implementierung

- ... auf Basis von Semaphoren

einfache Implementierung, die nur eine Bedingungsvariable unterstützt.



```

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_wait++;
        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.

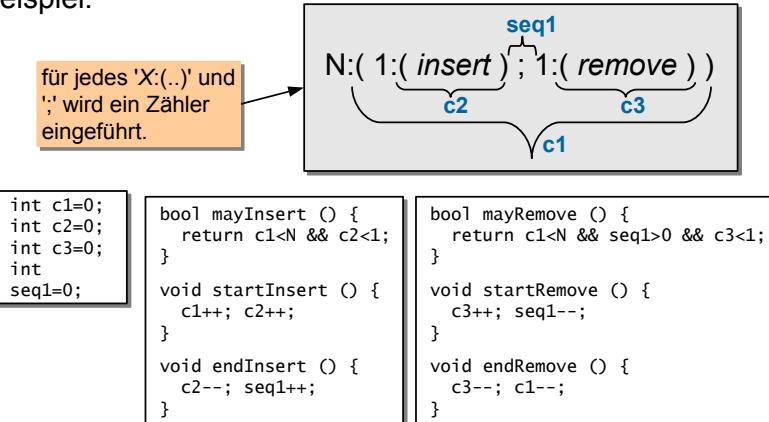
Pfadausdrücke [2]

- 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:



Pfadausdrücke – Implementierung (2)

- Transformation der Operationen

für jede Operation wird ein Semaphor und ein Zähler eingeführt.

N:(1:(insert) ; 1:(remove))
sem1/csem1 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;
}
```



Pfadausdrücke – Diskussion

- 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
Kommunikation und Synchronisation
IPC über Speicher
Semaphore, Monitor, Pfadausdrücke
IPC über Nachrichten
Send/Receive
Basisabstraktionen
Windows/UNIX/...
Dualität der Konzepte
Trennung der Belange mit AOP
Zusammenfassung



IPC über Nachrichten

- 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 Prozeduraufnäufe lassen sich gut auf IPC über Nachrichten abbilden (RPC, RMI)



dl/vs

Betriebssysteme (VL 12 | WS 16)

12 Interprozesskommunikation – IPC über Nachrichten

12-21

Agenda

Einordnung

Kommunikation und Synchronisation

IPC über Speicher

Semaphor, Monitor, Pfadausdrücke

IPC über Nachrichten

Send/Receive

Basisabstraktionen

Windows/UNIX/...

Dualität der Konzepte

Trennung der Belange mit AOP

Zusammenfassung

Nachrichtenbasierte Kommunikation

- 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*



dl/vs

Betriebssysteme (VL 12 | WS 16)

12 Interprozesskommunikation – IPC über Nachrichten

12-22

Basisabstraktionen

- 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



dl/vs

Betriebssysteme (VL 12 | WS 16)

12 Interprozesskommunikation – Basisabstraktionen

12-23



dl/vs

Betriebssysteme (VL 12 | WS 16)

12 Interprozesskommunikation – Basisabstraktionen

12-24

Dualität – Nachrichten in gemeinsamem Speicher

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

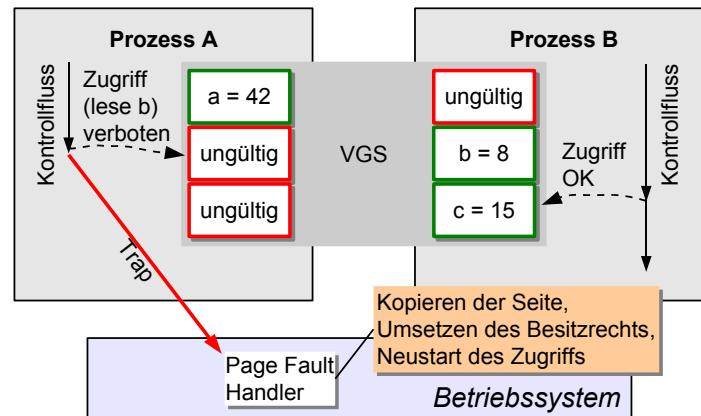
- Nachrichten werden nicht kopiert
 - Sender sorgt für Speicher
- receive* blockiert ggf.
- Mailbox-Abstraktion erlaubt M:N IPC

```
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;
    }
};
```



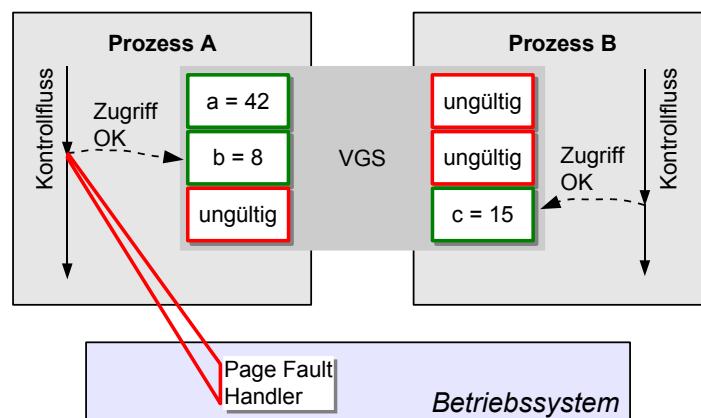
Dualität – Gemeinsamer Speicher mit Nachrichten

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



Dualität – Gemeinsamer Speicher mit Nachrichten

- „Virtueller gemeinsamer Speicher“ (VGS)



Dualität – VGS Diskussion

- 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.

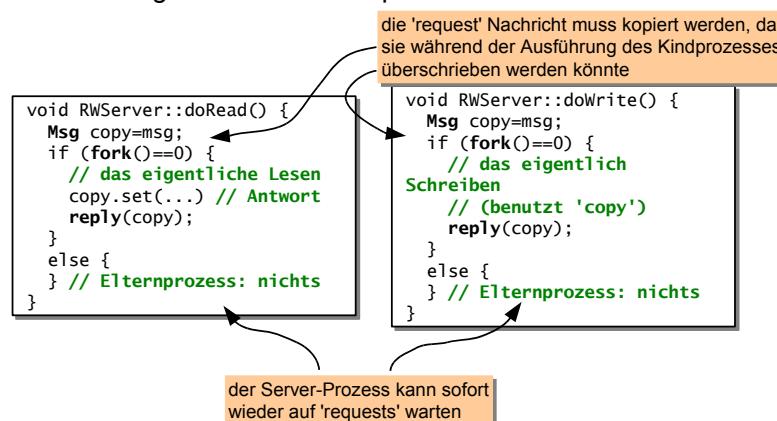
→ genau wie ein Monitor

```
class Server : public ActiveObject {
    Msg msg; // Nachrichtenpuffer
public:
    ...
    // Objekt mit Kontrollfluss!
    void action() {
        while (true) {
            receive(ANY, msg); // empfange 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
 - die eigentliche Lese- und Schreiboperation erfolgt nebenläufig durch einen Kindprozess



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); // empfange 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

```
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 (Mikrokerne) 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
Kommunikation und Synchronisation
- IPC über Speicher
Semaphore, Monitor, Pfadausdrücke
- IPC über Nachrichten
Send/Receive
- Basisabstraktionen
Windows/UNIX/...
- Dualität der Konzepte
- 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
    }
};
```

Agenda

- Einordnung
Kommunikation und Synchronisation
- IPC über Speicher
Semaphore, Monitor, Pfadausdrücke
- IPC über Nachrichten
Send/Receive
- Basisabstraktionen
Windows/UNIX/...
- Dualität der Konzepte
- 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 Synchronisationsmechanismen abstrahieren



Literatur

- [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

