Betriebssysteme (BS)

alias ***Betriebssystembau (BSB)***

Interprozesskommunikation

Abstraktionen und Systemstruktur

Überblick

- Kommunikation und Synchronisation
- IPC über gemeinsamen Speicher
  - Semaphor, Monitor, Pfadausdrücke
- IPC über Nachrichten
  - Send/Receive
- Basisabstraktionen in Betriebssystemen
- Dualität der Konzepte
- Trennung der Belange mittels 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)

IPC über gemeinsamen Speicher

- Anwendungsfälle/Voraussetzungen
  - ungeschütztes System (alle Prozesse im selben Adressraum)
  - System mit sprachbasiertem Speicherschutz
  - Kommunikation zwischen Fäden in 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();
}
```

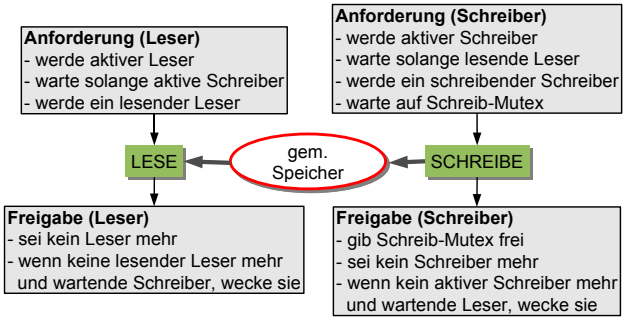
- betriebsmittelorientierte Synchronisation

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

sonst wie beim gegenseitigen Ausschluss

Semaphore – komplexere 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();
```

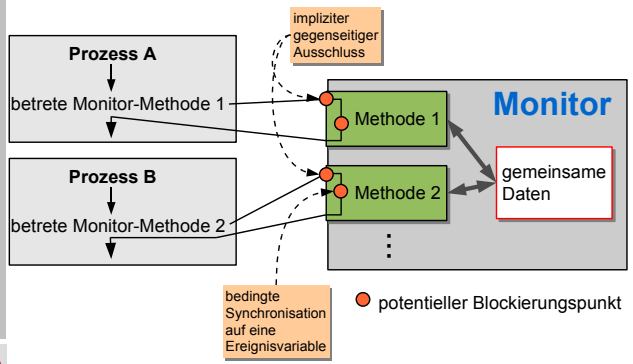
```
// Freigabe (Schreiber)
w_mutex.v();
mutex.p();
aw--; ww--;
while (aw==0 && rr<ar) {
  rr++;
  read.v();
}
mutex.v();
```


Semaphore – Diskussion

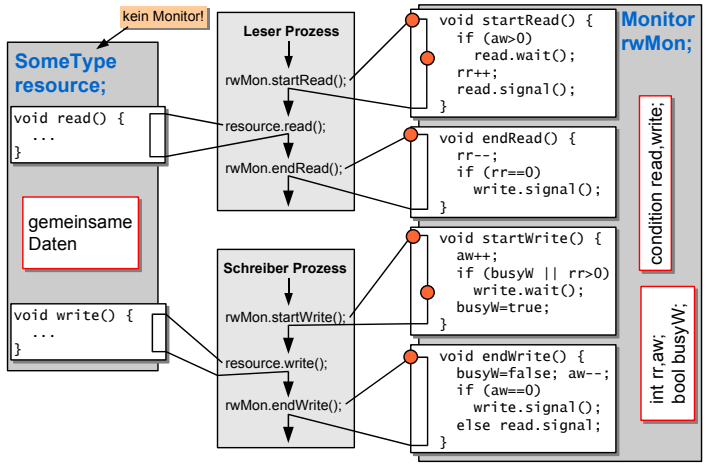
- Erweiterungen
  - nicht-blockierendes p()
  - *Timeout*
  - Felder von Zählern
- Fehlerquellen
  - Semaphorbenutzung wird nicht erzwungen
  - Abhängigkeit kooperierender Prozesse
    - jeder muss die Protokolle exakt einhalten
  - Aufwand bei der Implementierung
- → Unterstützung durch die Programmiersprache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

Monitore – synchronisierte ADTs

- Idee: Abstrakte Datentypen werden mit Synchronisationseigenschaften gekoppelt



Monitore – Leser/Schreiber-Problem



Monitore – Implementierung

- ... auf Basis von Semaphoren

einfache Implementierung, die nur *eine* Bedingungsvariable unterstützt.

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

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)
    c_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 nicht immer natürlich
  - siehe Leser/Schreiber Beispiel
  - gleiches Problem: wie beim Semaphor müssen Programmierer ein Protokoll einhalten

➔ die Synchronisation sollte von der Organisation der Daten und Methoden besser getrennt werden

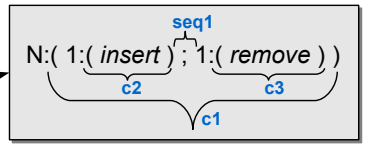
Pfadausdrücke

- 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** 2:(*Pfadausdruck*) **end**
  - max. 2 Kontrollflüssen dürfen gleichzeitig im *Pfadausdruck* sein
- **path** N:(1:(insert); 1:(remove)) **end**
  - z.B. Synchronisation eines N-elementigen Puffers
    - gegenseiter 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;
```

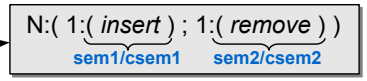
```
bool mayInsert () {
  return c1<N && c2<1;
}
void startInsert () {
  c1++; c2++;
}
void endInsert () {
  c2--; seq1++;
}
```

```
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.




```
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;
  }
  else if (csem2>0 &&
    mayRemove()) {
    csem2--;
    sem2.v();
    return true;
  }
  return false;
}
```


Pfadausdrücke – Diskussion

\+ komplexere Interaktionsmuster als mit Monitoren möglich

\+ Einhaltung der Interaktionsprotokolle wird erzwungen
- weniger Fehler!

\- 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

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

Nachrichtenbasierte Kommunikation

- Wiederholung: 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*

Basisabstraktionen

- Welche IPC Basisabstraktionen bieten Betriebssysteme?
  - UNIX-Systeme: *Sockets; System V Semaphore, Messages* u. *Shared Memory*
  - Windows NT/2000/XP: Shared Memory, Events, Semaphore, Mutant, Sockets, asynchrone Ein-/Ausgabe, ...
  - 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 u. verteilte Betriebssysteme: Nachrichten
    - Basis für Nachrichtenimplementierungen sind aber Synchronisationsprimitiven
  - Monolithische Systeme: Semaphore und gemeinsamen Speicher

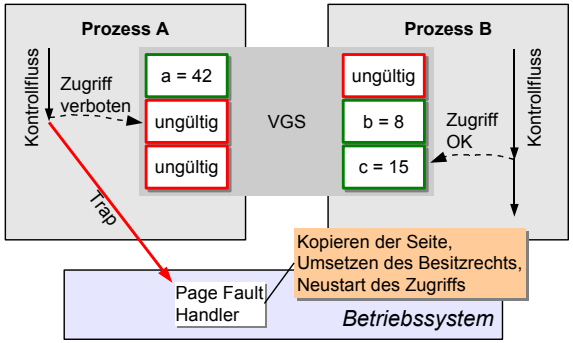
Dualität – Nachrichten im gem. Speicher

- auf Basis von Semaphoren u. gem. 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;
  }
};
```

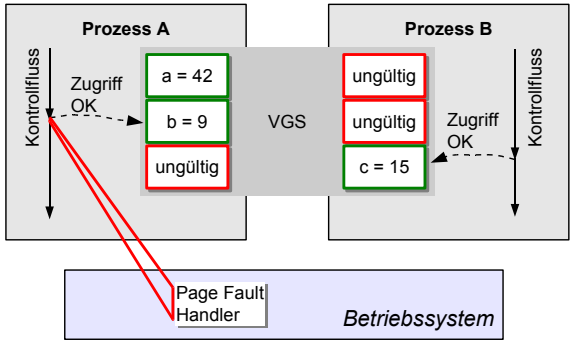
Dualität – Gem. Speicher mit Nachrichten

- „Virtueller gemeinsamer Speicher“ (VGS)



Dualität – Gem. Speicher mit Nachrichten

- „Virtueller gemeinsamer Speicher“ (VGS)



Dualität – VGS Diskussion

- Verteilter 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 werde in Kauf genommen
  - Änderungen asynchron per *Broad-/Multicast* verbreiten

Dualität – Aktive Objekte

- Leser/Schreiber Problem mit Nachrichtenaustausch

```
void reader() {
  Msg buf;
  send_request(srv,
    Msg(START_READ));
  wait_reply(srv);
  send_request(srv, Msg(DO_READ));
  wait_reply(srv, &buf);
  send_request(srv, Msg(END_READ));
  wait_reply(srv);
  // ... benutze Daten in 'buf'
}
```

```
void writer() {
  Msg buf(DO_WRITE);
  // Nachricht füllen
  // ... Protokoll
  send_request(srv, buf);
  wait_reply(srv);
  // ... Protokoll
}
```

```
class RWServer : public ActiveObject {
  Msg msg; // Nachrichtenpuffer
public:
  ...
  // Objekt mit Kontrollfluss!
  void action() {
    while (true) {
      wait_request(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: send_reply (msg, Msg(ERROR));
} } } };
```

gegenseitiger Ausschluss durch die Verarbeitungsschleife

Dualität – Aktive Objekte

- Leser/Schreiber Problem mit Nachrichtenaustausch
  - die eigenlichen Lese- und Schreiboperation erfolgen nebenläufig durch einen Kindprozess

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

```
void RWServer::doRead() {
  Msg copy=msg;
  if (fork()==0) {
    // Kindprozess
    Msg res=... // fülle 'res'
    send_reply(copy, res);
  }
  else {
  } // Elternprozess: nichts
}
```

```
void RWServer::doWrite() {
  Msg copy=msg;
  if (fork()==0) {
    // Kindprozess
    // benutze 'copy'
    send_reply(copy);
  }
  else {
  } // Elternprozess: nichts
}
```

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++; send_reply(msg);
  }
}

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

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

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


Dualität – Diskussion

- Gibt es einen fundamentalen Unterschied zwischen IPC über gem. Speicher und IPC über Nachrichten?
  - oder zugespitzt: sind Mikrokerne oder Monolithen besser?
- Beispiel: Leser/Schreiber Monitor vs. *Server*:
  - Monitor: 2 potentielle Wartepunkte
    - Klient wird verzögert für gegenseitigen Ausschluss
    - Klient 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
  - Unterschied: bei der *Server*-Lösung kann der Klient zwischen dem send_request und wait_reply arbeiten
    - Lösung: fork/join für Monitoraufruf bzw. nur synchrones send
- Fazit: Synchronisation und Nebenläufigkeit gleich!

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() : 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

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 (siehe Folie 4 und 17)
- Ausblick: mit AOP Techniken könnte man von den konkreten Kommunikations- und Synchronisations*mechanismen* abstrahieren