Betriebssysteme (BS)

VL 13.2 – Interprozesskommunikation – Dualität

Volkmar Sieh / Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 20 – 8. Februar 2021

https://www4.cs.fau.de/Lehre/WS20/V_BS

Agenda

- Einordnung
- IPC über Speicher
- IPC über Nachrichten
- Basisabstraktionen
  - Windows/Unix/...
  - Dualität der Konzepte
- Trennung der Belange mit AOP
- Zusammenfassung

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

Dualität – Nachrichten in gemeinsamem Speicher

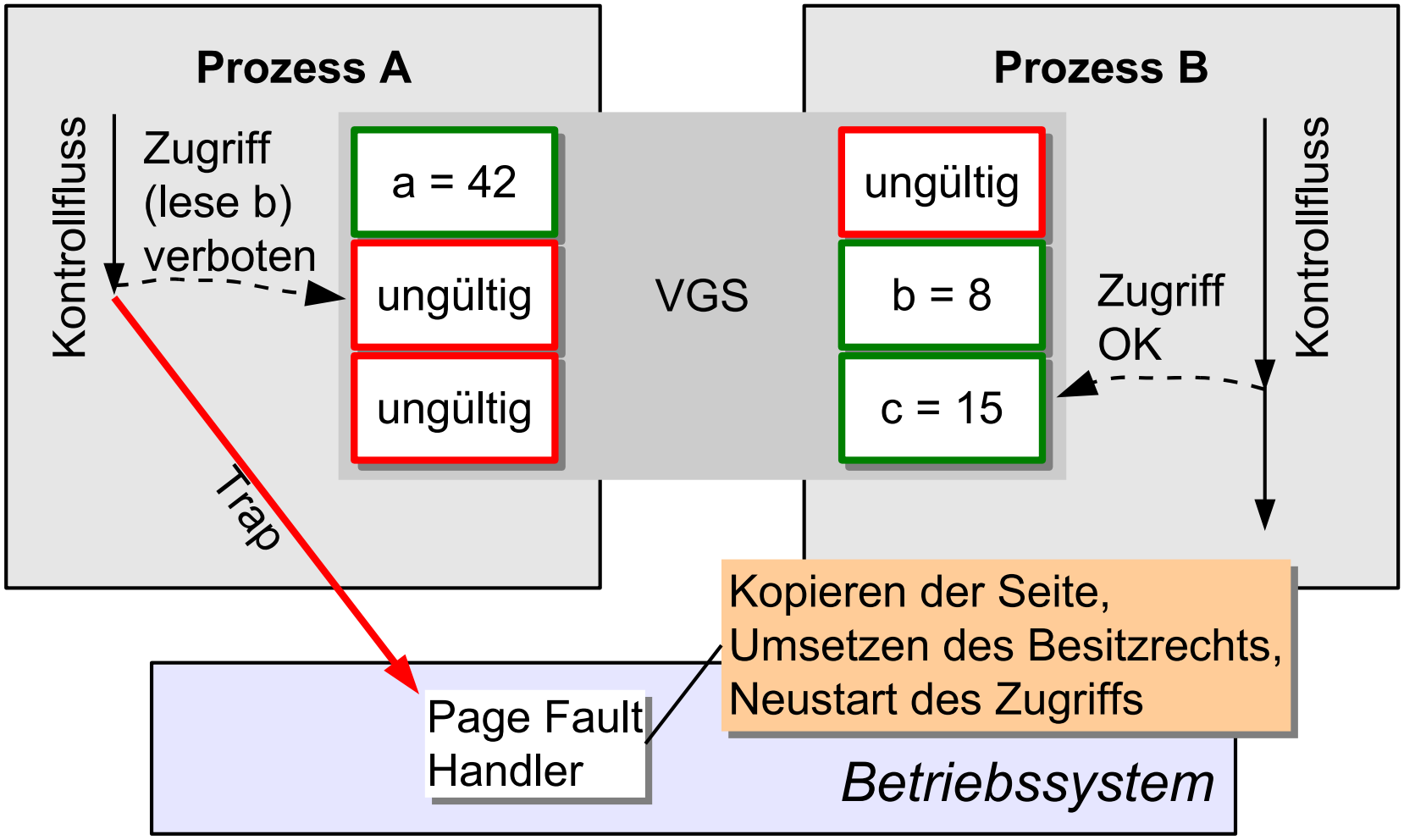
- auf Basis von Semaphoren und gemeinsamem Speicher lässt sich leicht eine *Mailbox*-Abstraktion realisieren:

- 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;
  }
};
```

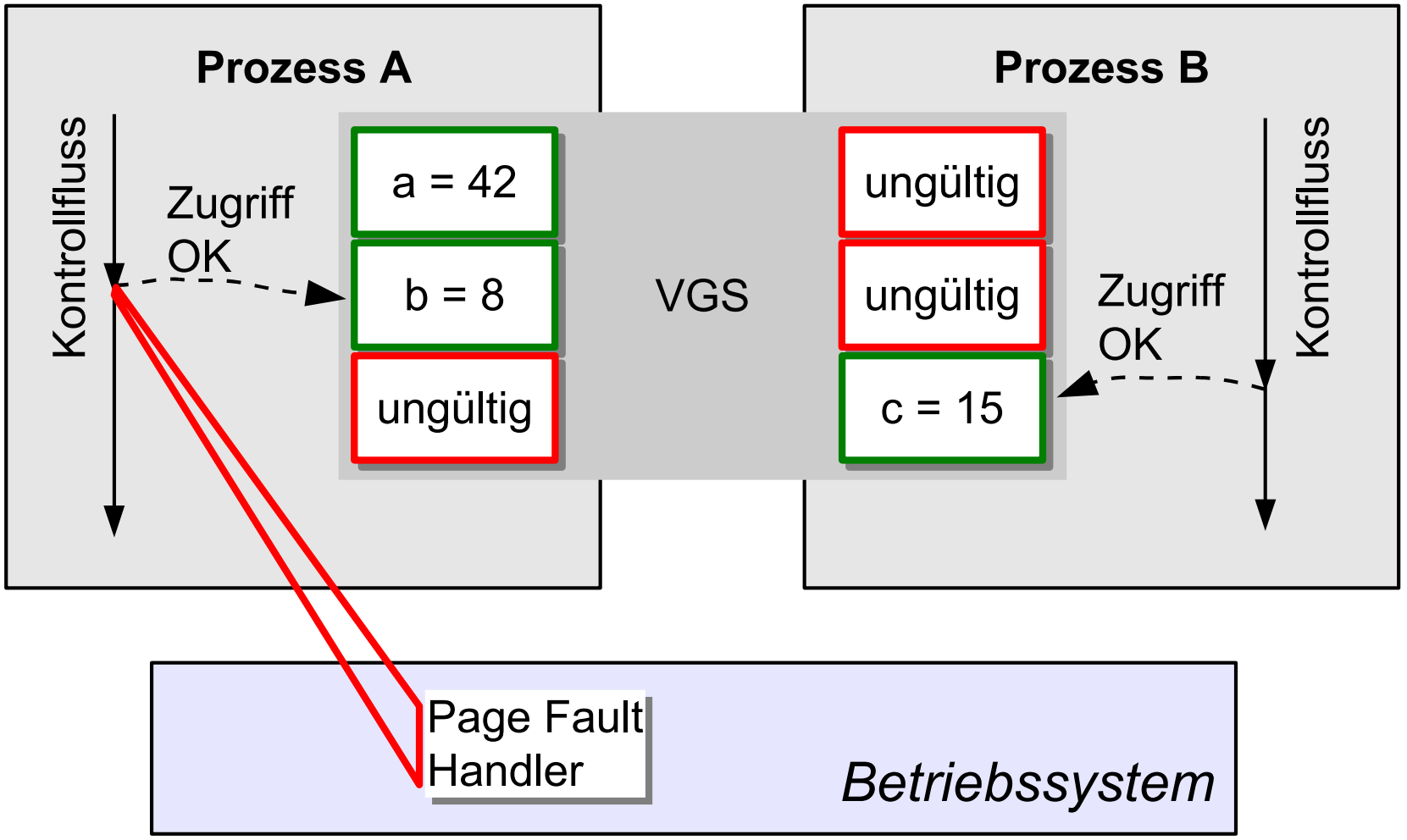
Dualität – Gemeinsamer Speicher mit Nachrichten

- „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

```
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
  - die eigentliche Lese- und Schreiboperation erfolgt 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) {
    // das eigentliche Lesen
    copy.set(...) // Antwort
    reply(copy);
  }
  else {
  } // Elternprozess: nichts
}
```

```
void RWServer::doWrite() {
  Msg copy=msg;
  if (fork()==0) {
    // das eigentlich
Schreiben
    // (benutzt 'copy')
    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++; 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

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