Script zur Vorlesung
-Parallele-
-Datenverarbeitung-

Basierend auf dem Mitschrieb der Vorlesung Parallele Datenverarbeitung von Prof. B. Rösch

Kritik / Verbesserungsvorschläge / Korrekturen an **kirstenm@fh-bingen.de**

Inhaltsverzeichnis

1.: Parallele Architekturen:

Ziel eines parallelen / verteilten Algorithmus ist die Einbindung mehrerer Programme, Prozesse oder Prozessoren in eine Lösung einer Datenverarbeitungsaufgabe.

Das bedeutet, Parallelität läßt sich unter 2 Gesichtspunkten betrachten:

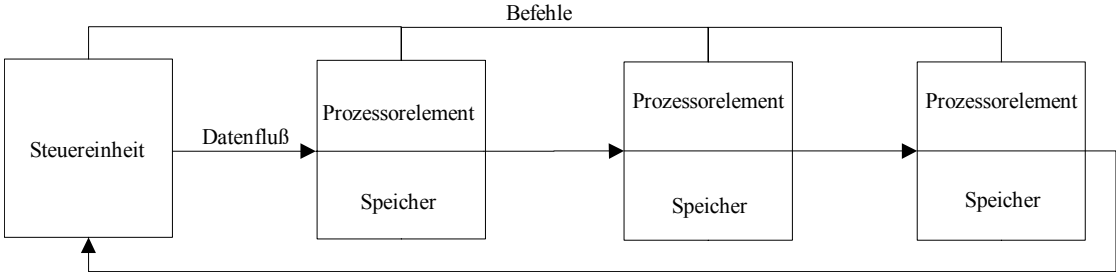
1) Klassifikation nach Rechnerarchitektur

Flym charakterisierte die Architekturen folgendermaßen:

SISD single instruction single data Von Neumann Rechner	***SIMD*** single instruction multiple data Vektor / Array Rechner
MISD multiple instruction single data Pipeline Rechner	***MIMD*** multiple instruction multiple data Mehrprozessor Rechner, verteiltes System

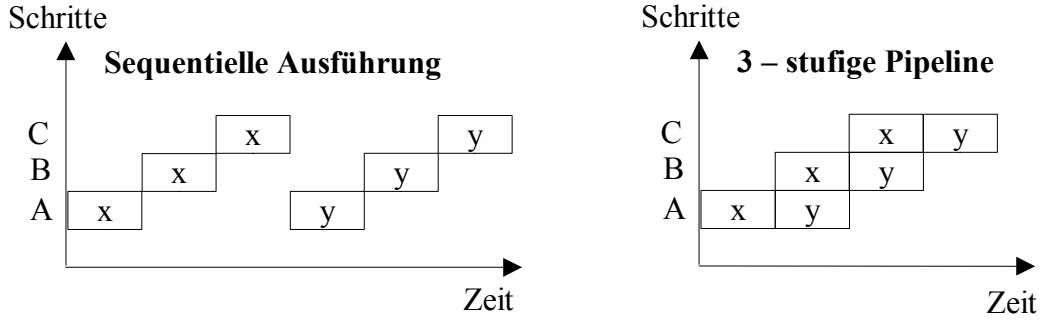
zu MISD (Pipeline Rechner):

Grundsätzlicher Aufbau:

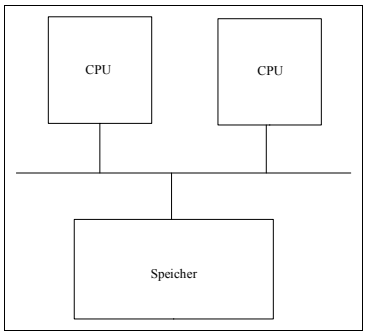


Beispiel: Folgendes soll mehrmals ausgeführt werden:

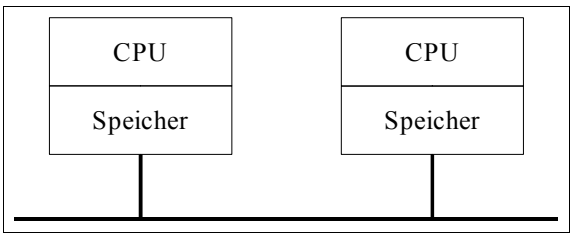
A: Lade Werte aus x und y
B: multipliziere die Werte
C: addiere Ergebnis zu S



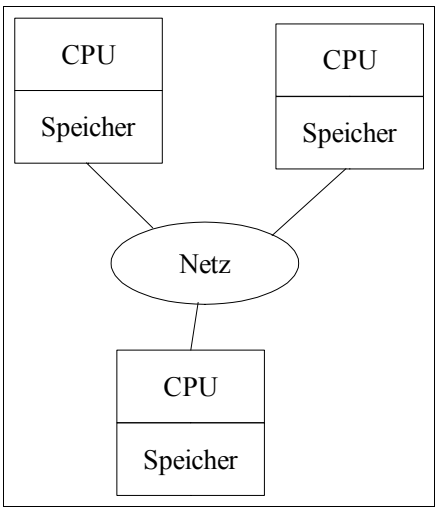
Zu MIMP (Mehrprozessor - Rechner, verteilte Systeme):



MIMD mit gemeinsamen Speicher
„eng gekoppeltes System“

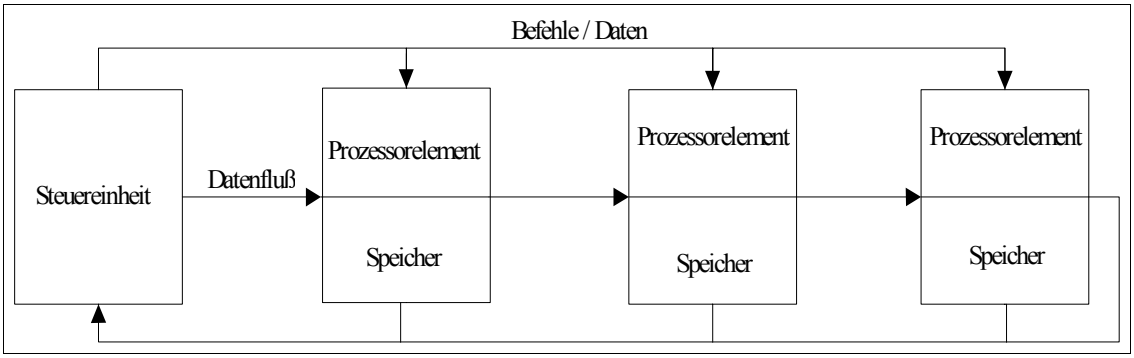


MIMD mit getrennten Speicher
„lose gekoppeltes System“
Kommunikations- & Synchronisationsaufwand



verteiltes System
Aufwand für Kommunikation und Synchronisation
Laufzeit für Nachrichten nicht vernachlässigbar

Zu SIMD (Vektor- / Array -Rechner):



Beispiel:

Addition von 2 Vektoren: $\begin{pmatrix}1\\2\\3\end{pmatrix}+\begin{pmatrix}4\\5\\6\end{pmatrix}$

Diese Aufgabe bewirkt, daß das 1. Prozessorelement die Daten „1“ und „4“ zur Addition erhält.
2. " „2“ und „5“ ....

2) Klassifikation nach parallelen Abläufen:

Ebene	***Verarbeitungseineit***	***Beispielsystem***
Programmebene	Prozeß (laufendes Programm) (=Job, =Task)	Multitasking-System
Prozedur Ebene	Prozeß oder Thread (gleichgewichtiger Prozeß)	Multitasking-System +MIMD
Ausdrucksebene	Instruktion (z.b. Funktion vektoradd steht schon zur Verfügung)	SIMD
Bit – Ebene	Innerhalb der Instruktion (paralleles Addieren in der Alu)	Von Neumann Rechner

Der Inhalt der Vorlesung ist grau hinterlegt

Vorteile von verteilten / parallelen Algorithmen:

1) Zeitgewinn
2) Nebenläufigkeit

   Es gibt Problemstellungen (Aufgaben), die nicht unbedingt sequentiell zu lösen sind

3) Wiederverwendung von Programmen

   Durch geeignete Zerlegung einer Aufgabe in viele Kleinere, die dann durch einzelne Programme realisiert werden, ist es möglich, verschiedenste Aufgabenstellungen zu lösen. (<=> Reduzierung des Programmieraufwandes)

   Beispiel: ls -l | more

Methoden der Problemzerlegung:

1) Zerlegung der Eingabedaten
   - Kartoffeln schälen und in Wasser werfen => lose gekoppelt

- Mauer bauen => eng gekoppelt => Synchronisation

2) Zerlegung des Algorithmus
   - Pipelinemodelle
   - 1 Verteiler – n Arbeiter – Modell
   - Team – Modell (lauter identische Prozesse, wer gerade frei ist übernimmt die Arbeit)

2.: Netzmodelle:

Netzmodelle werden zur Beschreibung der dynamischen Eigenschaften eines Systems verwendet. Es sollen also die Objekte und deren gegenseitige Beeinflussung dargestellt werden. Dazu gibt es 2 Techniken:

– Zustandsgraph
– Petri Netz

2.1.: Zustandsgraph:

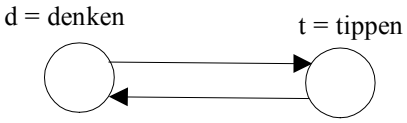
Das System
   – durchläuft eine endliche Anzahl von Zuständen
   – befindet sich zu einem Zeitpunkt in genau einem Zustand
   – wechselt unter gewissen Voraussetzungen von einem Zustand in den nächsten

Ein Zustandsgraph wird wie folgt dargestellt:

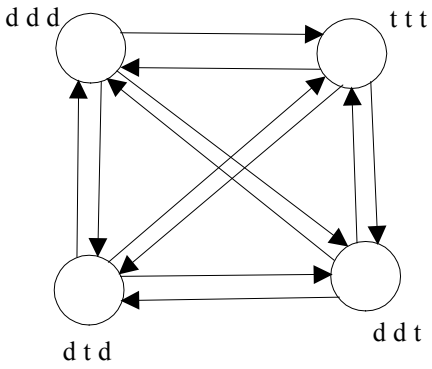
Beispiel: n Programmierer teilen sich m Terminals mit n >= m

Fall 1: n = m = 1

System hat 2 Zustände:
   - Programmierer denkt
   - Programmierer tippt



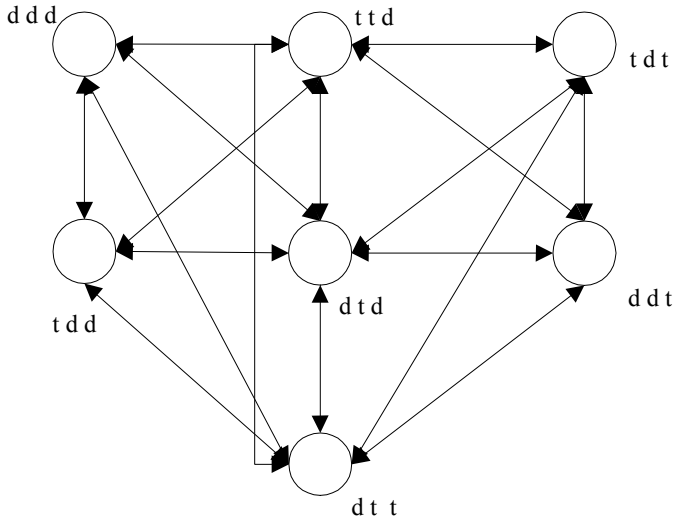
Fall 2: n = 3, m = 1



Starke Kopplung

=> Komplexität steigt nicht stark an

Fall 3: n = 3, m = 2



lose gekoppelt

=> Komplexität steigt stark an

Lose gekoppelte Systeme lassen sich so schlecht darstellen. Einfache Teilsysteme werden im Gesamtsystem unübersichtlich. Nebenläufigkeiten oder Verklemmungen sind unter Umständen nicht leicht erkennbar.

Hauptursache: Betrachtung aller möglichen Zustandsübergänge

2.2.: Petri Netze:

Hier handelt es sich um einen sogenannten Ereignisorientierten Ansatz.

2 Knotentypen:

- Kreis (genannt: Stelle) stellt Teilzustand dar
- Rechteck (genannt: Transition) stellt Ereignis dar, das zu einer Zustandsänderung führt

Kanten:

Pfeile, die zwischen Knoten unterschiedlichen Typs gezogen werden können (Pfeil kann nur von Stelle zu Transition oder umgekehrt; nie von Stelle zu Stelle oder Transition zu Transition)

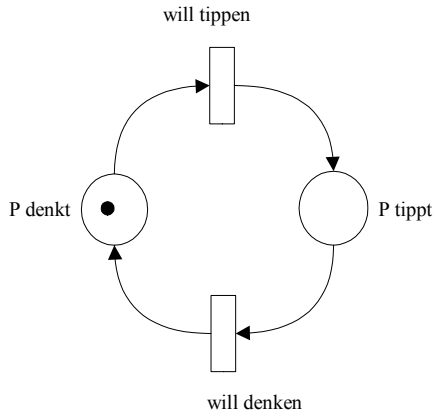
Punkte (genannt: Marken):

Das Vorliegen eines Punktes in einer Stelle markiert, daß dieser Teilzustand gültig ist. Der Gesamtzustand des Systems wird durch die aktuelle Markierung im Netz ausgedrückt.

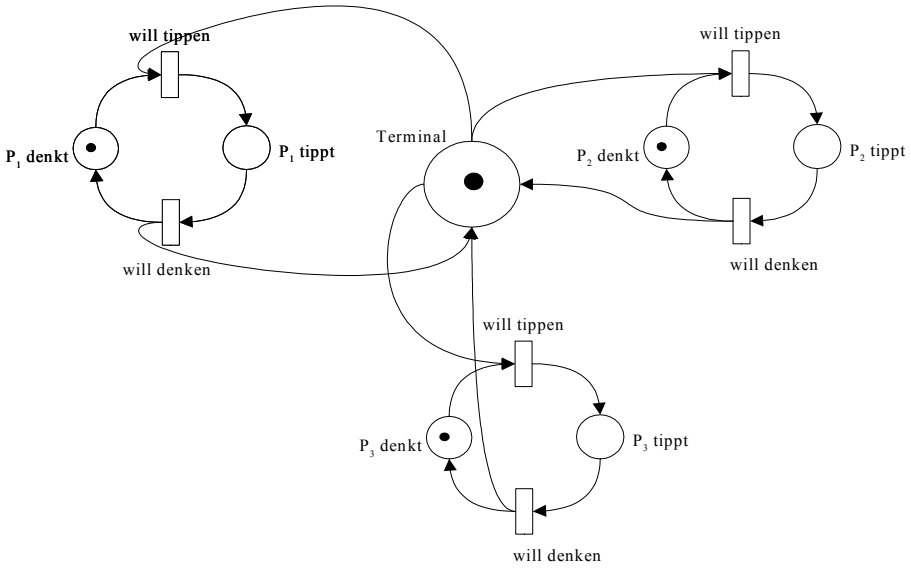
Beispiel:

n Programmierer, m Terminals

Fall 1: n = m = 1

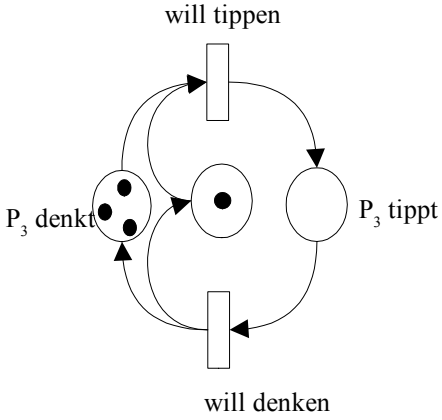


Fall 2: n = 3, m = 1



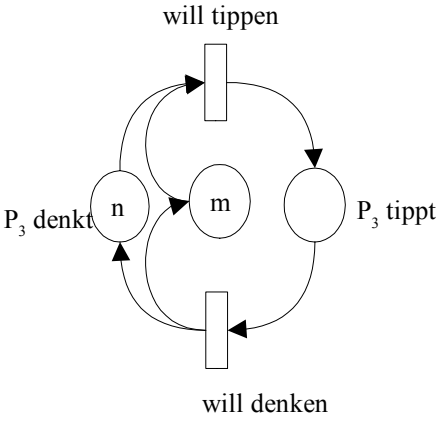
Marke in Terminal bedeutet, Terminal ist frei. Wer tippen will muß die Marke aus Terminal herausholen und diese anschließend wieder abgeben.

Einfache Lösung:



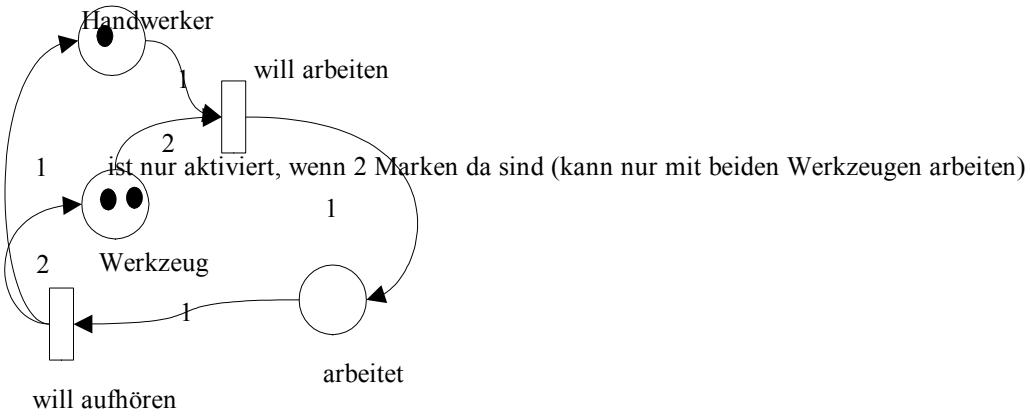
Falls n und m beliebig:

Schalten im Petri Netz:



Eine Transition ist schaltfähig (aktiviert), wenn:

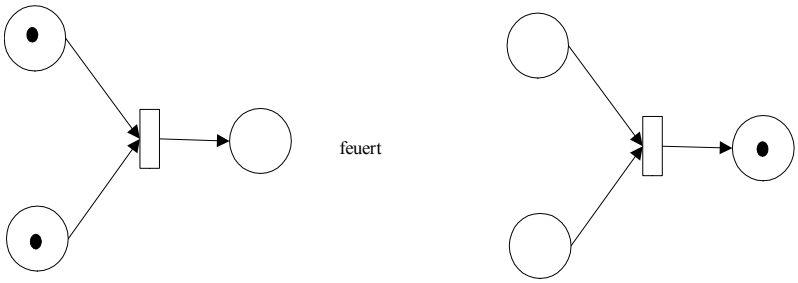
- alle Eingangsstellen markiert sind (Vorbereich)
- alle Ausgangsstellen eine genügend große Aufnahmekapazität zur Verfügung stellen (Nachbereich)



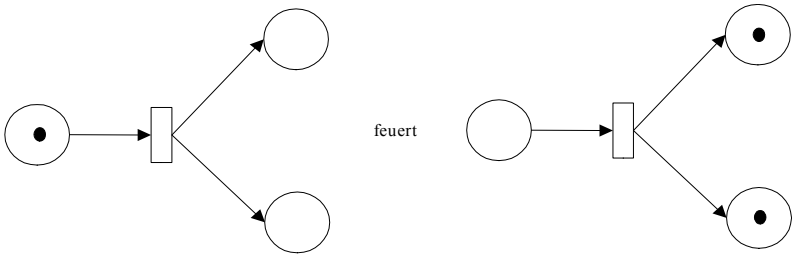
Der Schaltvorgang bewirkt:

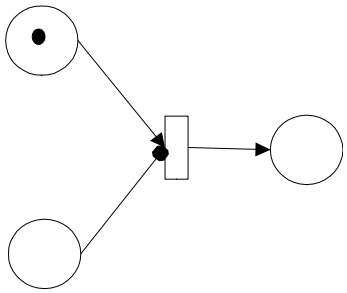
- löschen der Marken in sämtlichen Eingangsstellen (gemäß der Pfeilangabe in Richtung Transition)
- Ablegen von Marken (gemäß der Pfeilangabe von der Transition weg) in sämtlichen Ausgangsstellen)

Beispiele:



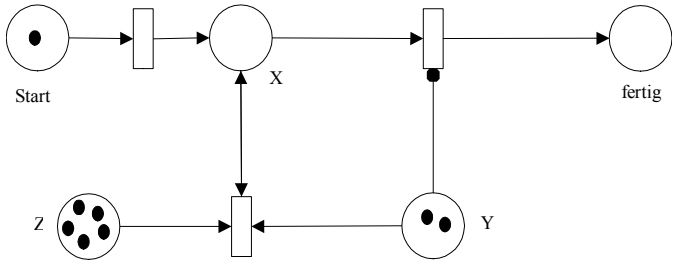
Inhibitorkante:





Wird bei einer Transition eine Inhibitorkante (Negationskante) verwendet, so ist die Transition aktiviert, wenn in der zugeordneten Stelle keine Marke liegt.

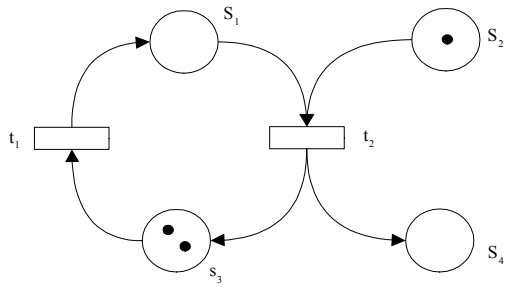
Beispiel:

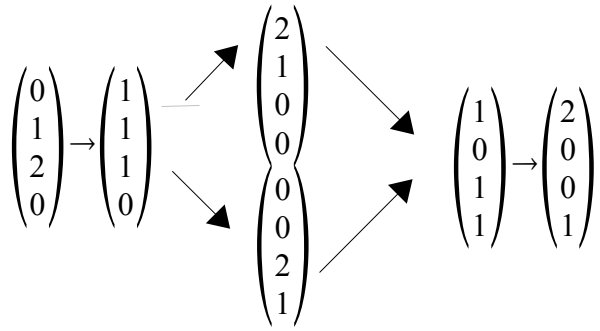


Wie lautet die Folge der Markierungen?

$$\begin{pmatrix} Start \\ fertig \\ X \\ Y \\ Z \end{pmatrix} \rightarrow \begin{pmatrix} 1 \\ 0 \\ 0 \\ 2 \\ 5 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 0 \\ 1 \\ 2 \\ 5 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 0 \\ 1 \\ 2 \\ 4 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 3 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 3 \end{pmatrix}$$

Petri Netz mit Erreichbarkeitsbaum:





Petri Netz ist tot!

2.3.: Typische Problemstellungen:

Erzeuger – Verbraucher – Problem:

Erzeuger – Prozeß:

- erzeugt Daten
- legt Daten in einem gemeinsamen Speicherbereich ab

Verbraucher – Prozeß:

- lese Daten aus gemeinsamen Speicherbereich
- verbrauche Daten

Annahme: Es paßt nur ein Datensatz in den Datenbereich:

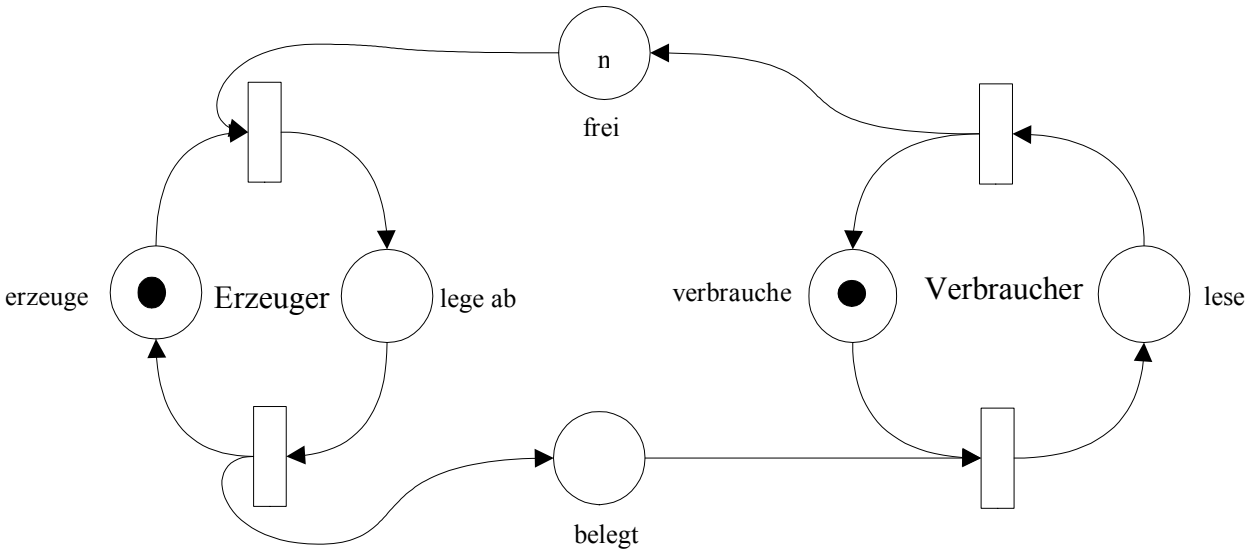
Es können folgende Probleme auftreten:

1) Wenn Datenbereich belegt, dann darf Erzeuger den Datensatz nicht ablegen. Er muß warten, bis Verbraucher den Datensatz gelesen hat. (Erzeuger sehr schnell, Verbraucher langsam)
2) Es könnte passieren, daß der Verbraucher lesen will, obwohl nichts neues abgelegt wurde (unter Umständen liest er den Datensatz zweimal)

Somit müssen wir ein Petri Netz entwickeln, mit dem die Synchronisationsprobleme gelöst werden können.

Kooperation:

Falls mehr als ein Datensatz abgelegt werden kann (n Datensätze), sieht das Petri Netz wie folgt aus:



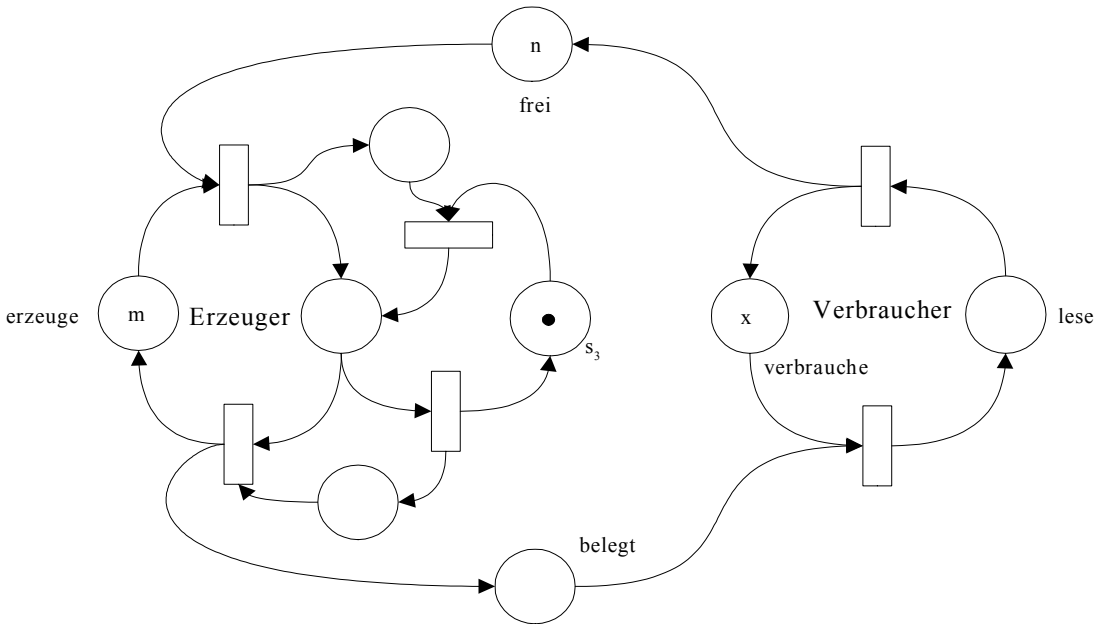
Diese Aufgabenstellung wird auch „Bounded Buffer“ – Problem genannt.

Wichtig ist folgende „Beobachtung“:

Da nur ein Erzeugerprozeß existiert, kann der Index, um auf die entsprechende Stelle im Datenbereich zuzugreifen, lokal (d.h. innerhalb des Prozesses) gehalten werden. Entsprechendes gilt auch beim Verbraucherprozeß.

Allgemeinster Fall:

- m Erzeugerprozesse
- x Verbraucherprozesse
- n Datensätze



Verbraucher entsprechend ergänzen!

Zu unterscheiden:

1. soll exklusiver Zugriff auf Datenbereich für alle eingerichtet sein? => nur 1 S_3
2. soll nur Index von Erzeuger & Verbraucher geschützt werden? => 2 S_3 – Stellen (einfach Kopie der Erweiterung beim Erzeuger)

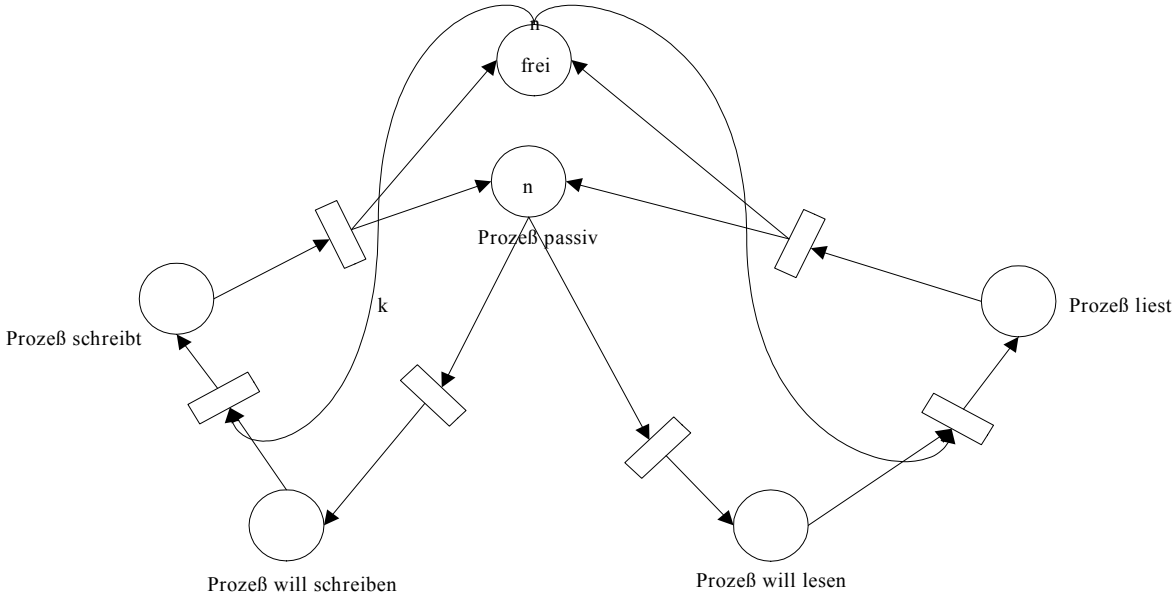
Lese – Schreib – Problem:

Gegeben:

n Prozesse, die alle auf einen Speicherbereich lesend oder schreibend zugreifen können.

Wenn kein Prozeß schreibt, können bis zu k <= n Prozesse gleichzeitig lesen.

Wenn ein Prozeß schreibt, darf kein anderer schreiben oder lesen.



3.: Nichtsequentielle Programmierung:

3.1: Begriff „Nebenläufigkeit“:

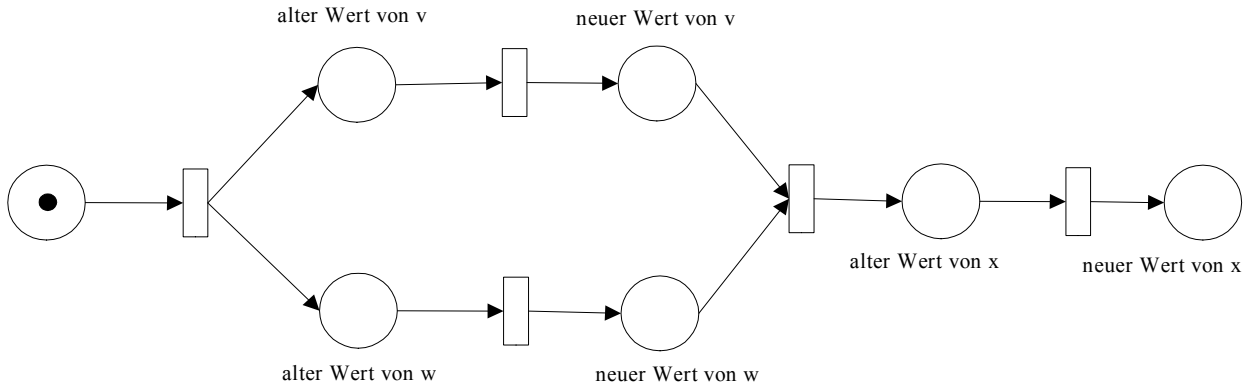
Beispiel:

v = x;

w = x;

x = y;

Wie das folgende Petri Netz zeigt, liegt eine willkürliche Festlegung der Reihenfolge vor:



Verzichtet man auf die sequentielle Reihenfolge, kann dies zweierlei bedeuten:

1) Parallele Ausführung
2) Anweisungen werden zeitlich verzahnt von nur einem Prozessor ausgeführt

Diese beiden Ausführungsmöglichkeiten sind gemeint, wenn man von Nebenläufigkeit spricht. Um diese in einem Programmtext ausdrücken zu können, ist ein Sprachkonstrukt notwendig.

cobegin

$s_1 \parallel s_2$

coend

s_1 und s_2 sind zwei Anweisungsfolgen, die nebenläufig abgearbeitet werden können.

Für das Beispiel:

cobegin

v = x || w = x

coend

3.2.: Anforderung der Determiniertheit:

Ein Algorithmus heißt:

determiniert: wenn er bei gleichen Eingabewerten gleiche Ausgabewerte liefert

deterministisch: wenn der Ablauf reproduzierbar ist.

Es gelten folgende Zusammenhänge:

1. Jeder deterministische Algorithmus ist determiniert
2. Jeder nichtsequentielle Algorithmus ist nicht deterministisch

3. Aus nichtsequentiell folgt nicht notwendigerweise nichtdeterminiert

Wie das Erzeuger – Verbraucher – Problem zeigt, kann man durch geeignete Synchronisation Determiniertheit erreichen.

Paradigma der nichtsequentiellen Programmierung:

Nichtsequentielle Algorithmen in Verbindung mit Synchronisationsmaßnahmen sind determiniert.

3.3: Synchronisationsbedarf:

1) Schreib – Schreib – Konflikt:

   Beispiel: cobegin

   x = x + 1|| x = x + 1

   coend

   Je nach Ausführungsart wird x einmal oder zweimal erhöht

2) Schreib – Lese – Konflikt:

   Probleme können entstehen, wenn Daten nebenläufig gelesen und geschrieben werden.

3) Gewährleistung einer logischen Abfolge

   Beim Erzeuger – Verbraucher – Problem liegt keine Konkurrenz Situation vor, die beteiligten Prozesse müssen in geeigneter Weise kooperieren.

3.4.: Synchronisationsarten:

Man unterscheidet:

- Konkurrenz
  - mehrseitige Synchronisation
  - soll B begonnen werden, während A durchgeführt wird, dann wird B verzögert, bis A fertig ist (und umgekehrt!).
- Kooperation
  - einseitige Synchronisation
  - B wird erst beginnen, wenn A fertig ist

3.5.: Kritischer Abschnitt:

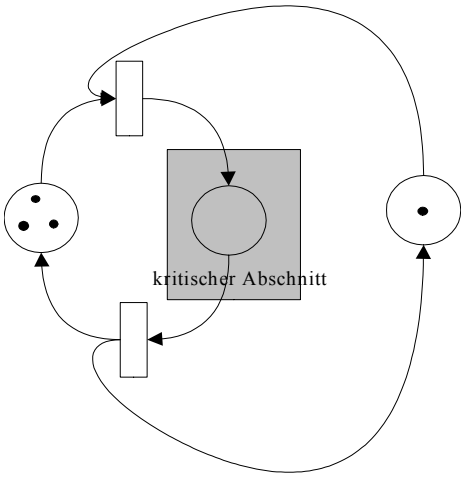
Kennzeichnend für die mehrseitige Synchronisation ist, daß Aktivitäten, die von ihr betroffen sind, im gegenseitigen Ausschluß stehen. Eine Anweisungsfolge heißt atomar (unteilbar), wenn sie nicht durch nebenläufige Anweisungen unterbrochen werden kann. Eine atomare Anweisungsfolge wird auch kritischer Abschnitt genannt.

Kennzeichen:

- Zugriff auf gemeinsame Objekte
- aus der Sicht des anderen Prozesses erscheint die Folge unteilbar

– Ausführung erfordert gegenseitigen Ausschluß

Im Petri Netz drückt sich der kritische Abschnitt wie folgt aus:



Am Petri Netz kann man folgendes erkennen:

Eintrittsprotokoll:

Ein Prozeß gibt geeignet bekannt, daß er den kritischen Abschnitt betreten will. Im Petri Netz bedeutet dies, das wegnehmen der Synchronisationsmarke.

Austrittsprotokoll:

Ein Prozeß gibt geeignet bekannt, daß er den kritischen Abschnitt verlassen hat. Im Petri Netz bedeutet dies das zurücklegen der Synchronisationsmarke.

3.6.: Schloßalgorithmen:

Es sollen Möglichkeiten vorgestellt werden, wie Ein- und Austrittsprotokolle unter Einsatz von gemeinsamen Variablen implementiert werden können. Diese Variablen heißen Schloßvariablen. Die möglichen Algorithmen Schloßalgorithmen.

3.7.: Dekkerscher Algorithmus:

Es soll eine Lösung für den gegenseitigen Ausschluß für zwei Prozesse P_1 und P_2 auf Hochsprachenebene gefunden werden. Dabei soll eine schrittweise Entwicklung der Lösung erfolgen (nach Diykstra).

Die beiden Prozesse führen in einer Endlosschleife aus:

Kritischer Abschnitt	krit_1 ()	krit_2 ()
Unkritischer Abschnitt	unkrit_1 ()	unkrit_2 ()

Die Zeitliche Ausführung von krit_1 () und krit_2 () darf sich zeitlich nicht überlappen.

Die Lösung muß 3 Kriterien erfüllen:

1. **Exklusivität**
2. **Blockadefreiheit**:
   - keine Verklemmung
   - kein „Aushungern“ eines Prozesses
3. **Zeitinvarianz**
   - Korrekter Ablauf bei beliebigen zeitlichen Eintreten der Prozesse in den kritischen Abschnitt

Versuch 1:

Schloßvariable turn:

- von beiden Prozessen schreib & lesbar
- Wert kann 1 oder 2 sein
- Wert gibt an, welcher Prozeß als nächster in den kritischen Abschnitt darf
- Initialwert: 1

P_1	P_2

while (true)
{
while (turn == 2)
; // tue nichts
krit_1 ();
turn = 2;
unkrit_1 ();
}
	while (true)
	{
	while (turn == 1)
	; // tue nichts
	krit_2 ();
	turn = 1;
	unkrit_2 ();
	}

Beobachtung:

1) Exklusivität ist erreicht
2) Blockade?!?

Es liegt eine Behinderung vor, wenn sich P1 z.B. sehr lange in unkrit_1 aufhält und P2 den Zyklus in der Zwischenzeit einmal durchlaufen hat.

Versuch2:

Jeder Prozeß bekommt seine eigene Schloßvariable

Schloßvariable c_1

- für P_1 schreib & lesbar
- für P_2 lesbar
- Wert 0 oder 1
- 0 bedeutet : bin in kritischen Abschnitt
- 1 bedeutet : nicht in kritischen Abschnitt
- Initialwert: 1

Schloßvariable c_2: analog

P_1	P_2

while (true)
{
while (c_2 == 0)
; // tue nichts
c_1 = 0;
krit_1 ();
c_1 = 1;
unkrit_1 ();
}
	while (true)
	{
	while (c_1 == 0)
	; // tue nichts
	c_2 = 0;
	krit_2 ();
	c_2 = 1;
	unkrit_2 ();
	}

	c_1	c_2
Initialwert	1	1
P_1 prüft c_2	1	1
P_2 prüft C_1	1	1
P_1 setzt c_1	0	1
P_2 setzt c_2	0	0
P_1 in k.A.	0	0
P_2 in k.A.	0	0

Versuch 3: (erst setzen, dann prüfen)

```
P1
while (true)
{
 c1 = 0;
 while (c2 == 0)
        ; // tue nichts
 krit_1 ();
 c1 = 1;
 unkrit_1 ();
}
```

```
P2
while (true)
{
 c2 = 0;
 while (c1 == 0)
        ; // tue nichts
 krit_2 ();
 c2 = 1;
 unkrit_2 ();
}
```

Bewertung: => Deadlock wenn beide c's auf 0!!!

Versuch 4: (Temporäre Aufgabe von der Absicht (Rückziehmethode (:))

```
P1
while (true)
{
 c1 = 0;
 while (c2 == 0)
   {
        c1 = 1;
        sleep ();
        c1 = 0;
   }
 krit_1 ();
c1 = 1;
unkrit_1 ();
}
```

```
P2
while (true)
{
 c2 = 0;
 while (c1 == 0)
   {
        c2 = 1;
        sleep ();
        c2 = 0;
   }
 krit_2 ();
c2 = 1;
unkrit_2 ();
}
```

„Nach – Ihnen – Problem“ (selbst bei unterschiedlichen „Schlafwerten“)

Versuch 5: Dekkerscher Algorithmus

```
P1
while (true)
{
 c1 = 0;
 while (c2 == 0)
   {
        if (turn == 2)
        {
          c1 = 1;
          while (turn == 2);
          c1 = 0;
        }
   }
 krit_1 ();
 turn = 2;
 c1 = 1;
 unkrit_1 ();
}
```

```
P2
while (true)
{
 c2 = 0;
 while (c1 == 0)
   {
        if (turn == 1)
        {
          c2 = 1;
          while (turn == 1);
          c2 = 0;
        }
   }
 krit_2 ();
 turn = 1;
 c2 = 1;
 unkrit_2 ();
}
```

Diskussion der Lösung:

1) Initialwerte: $c_1 = 1$, $c_2 = 1$, turn = 1
2) P_1 kann in den kritischen Abschnitt (so oft wie P_1 will), wenn P_2 nicht vorhat in den kritischen Abschnitt zu gehen, unabhängig vom turn – Wert.
3) P_1 im kritischen Abschnitt => P_2 kann nicht in den kritischen Abschnitt
   $c_1 = 0$ — P_2 will in kritischen Abschnitt => $c_2 = 0$ $c_2 = 1$ (while Schleife blockiert erstmal) => P_2 kann nicht in kritischen Abschnitt, wegen $c_1 = 0$
4) Beide wollen in den kritischen Abschnitt , d.h. $c_1 = c_2 = 0$

   => die turn Variable regelt, wer den kritischen Abschnitt betreten darf

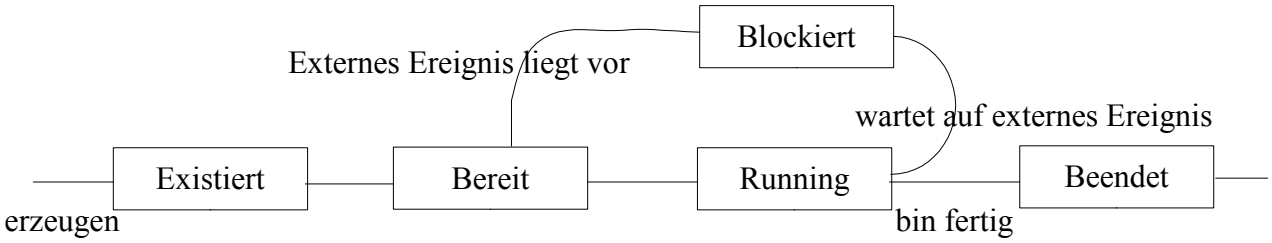
Erläuterung: P_1 läuft durch, geht wieder nach oben und setzt $c_1 = 0$, obwohl P_2 schon die gante Zeit auf auf die Erlaubnis wartet in den kritischen Abschnitt eintreten zu dürfen. Dieser Fall wird allerdings durch turn geregelt, so daß P_2 trotzdem zuerst dran kommt.

Kritik am Dekkerschen Algorithmus:

1) Ist nur für 2 Prozesse definiert
   (Erweiterung ist relativ kompliziert)
2) Aktives Warten
   (Prozeß überprüft ständig, ob sich ein Wert geändert hat)
   mit Betriebssystemunterstützung kann passives Warten realisiert werden (-> Semaphoren)
3) Durch das Benutzen von gemeinsamen Variablen kann das Verfahren nicht in verteilten System angewandt werden.

4.: Semaphoren:

Es handelt sich um ein Synchronisationskonstrukt, welches von Diykstra um 1965 vorgeschlagen wurde. Mit Hilfe der Prozeßverwaltung des Betriebssystems soll eine Lösung realisiert werden.



Zustandsdiagramm für Prozesse

Eine Semaphor ist ein abstrakter Datentyp, der aus:

- einem Zähler
- einer Warteschlange

besteht. Durch einen positiven Wert des Zählers wird ausgedrückt, wieviele Prozesse in den kritischen Abschnitt dürfen. Die Warteschlange spielt dann eine Rolle, wenn der kritische Abschnitt belegt ist. Die Prozesse, die in den kritischen Abschnitt wollen, werden dort vom Betriebssystem verwaltet (-> Zustand blockiert).

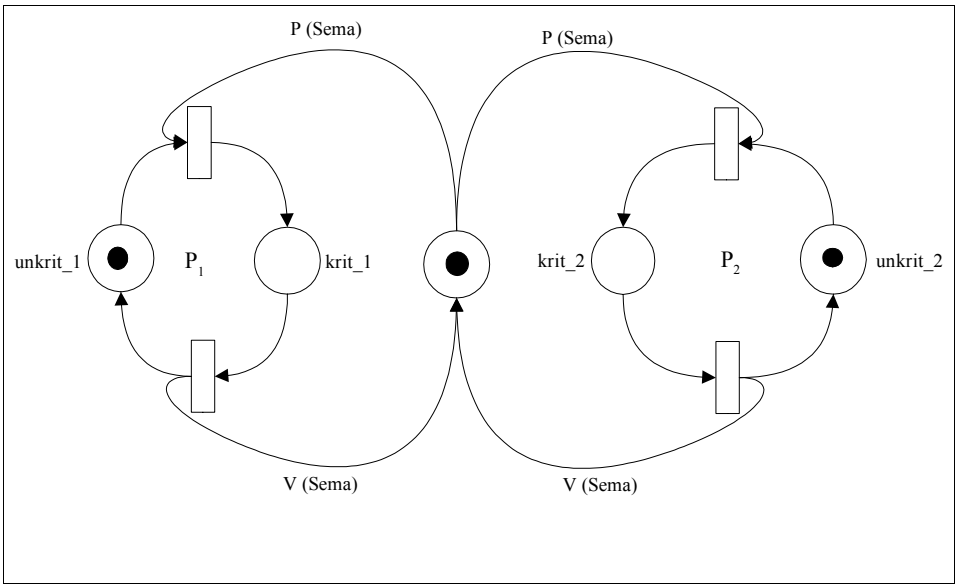
Auf einer Semaphor sind folgende Operationen definiert:

- P-Operation:
  - wird zum Betreten des kritischen Abschnitts benötigt
  - reduziert Zähler um 1
  - falls Zähler < 0, dann kommt aufrufender Prozeß in die Semaphor-Warteschlange
- V-Operation:
  - wird zum Verlassen des kritischen Abschnitts benötigt
  - erhöht Zähler um 1
  - falls Zähler <= 0, dann wird ein Prozeß aus der Warteschlange geweckt (-> kommt in Bereit-Warteschlange)

4.1.: Beispiel:

Init: Zähler = 2

1. P_1 macht P-Operation: Zähler = 1
2. P_2 macht P-Operation: Zähler = 0
3. P_3 macht P-Operation: Zähler = -1
4. P_4 macht P-Operation: Zähler = -2
5. P_1 macht V-Operation: Zähler = -1 => P_3 darf rein



Petri-Netz für Semaphor Operationen

P_1	P2

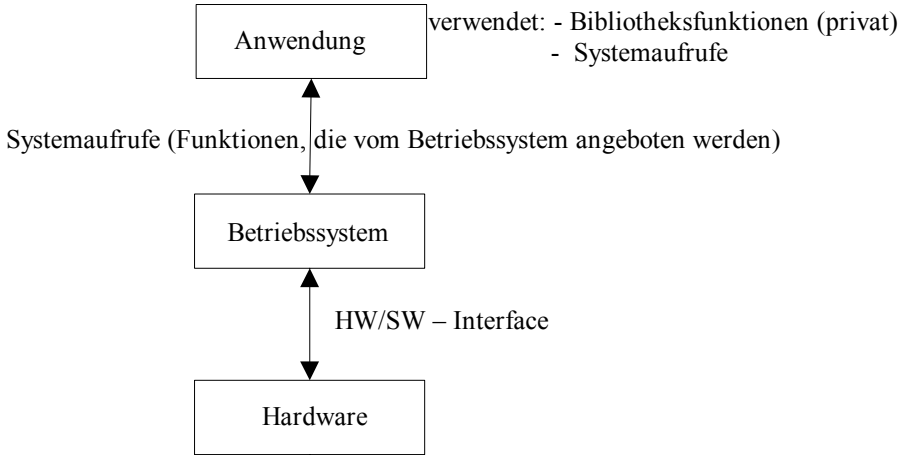
while (true)
{
P (Sema);
krit_1 ();
V (Sema);
unkrit_1 ();
}
	while (true)
	{
	P (Sema);
	krit_2 ();
	V (Sema);
	unkrit_2 ();
	}

zugehöriges Programm

Einseitige Synchronisation (Kooperation):

- Jeder Synchronisationsbedingung wird eine Semaphor zugeordnet
- Die einander zugeordneten Semaphor-Operationen stehen in unterschiedlichen Prozessen
- Mit der P-Operation wartet ein Prozeß darauf, daß ihm ein anderer Prozeß das Eintreten des Ereignisses anzeigt.
- Wenn die Bedingung zu Beginn noch nicht gilt, ist der Initialwert der Semaphor = 0.

5.: Systemnahes Arbeiten:



Wie arbeitet eine Bibliotheksfunktion?

printf (.....)

{ ...

- Formatierung der Argumente
- Danach Ablage des Ausgabestrings in einen Puffer
- verwenden eines Systemaufrufs (hier write)

...}

Wie arbeitet ein Systemaufruf?

- Ein Systemaufruf kann in einer Anwendung benutzt werden
- Der Linker bindet für jeden benutzten Systemaufruf den entsprechenden „Systemaufrufstummel“ hinzu. Dieser enthält im Wesentlichen einen Sprung ins Betriebssystem
- Das Betriebssystem ruft die entsprechende Funktion intern auf.
- Diese Funktion überprüft die Zulässigkeit der Argumente und gibt unter Umständen eine Fehlermeldung zurück.
- Wenn alles o.k. ist, wird die Funktion ausgeführt.
- Die Funktion meldet einen Funktionswert zurück.

Fehlermeldung: in den meisten Fällen wird eine -1 zurückgegeben. Die eigentliche Fehlerursache steht in der Variablen: *int errno.* (<errno.h> muß includiert werden)

Mit der Fehlerfunktion *perror (char*s)* kann der dazugehörige Fehlertext ausgegeben werden.

Grundlegende Dateioperationen unter UNIX:

Beispiele zur Kopie:

open:

fd = open („/usr/home/para1/a.c“, O_RDONLY);

=> Filedescriptor muß bei read / write verwendet werden

read:

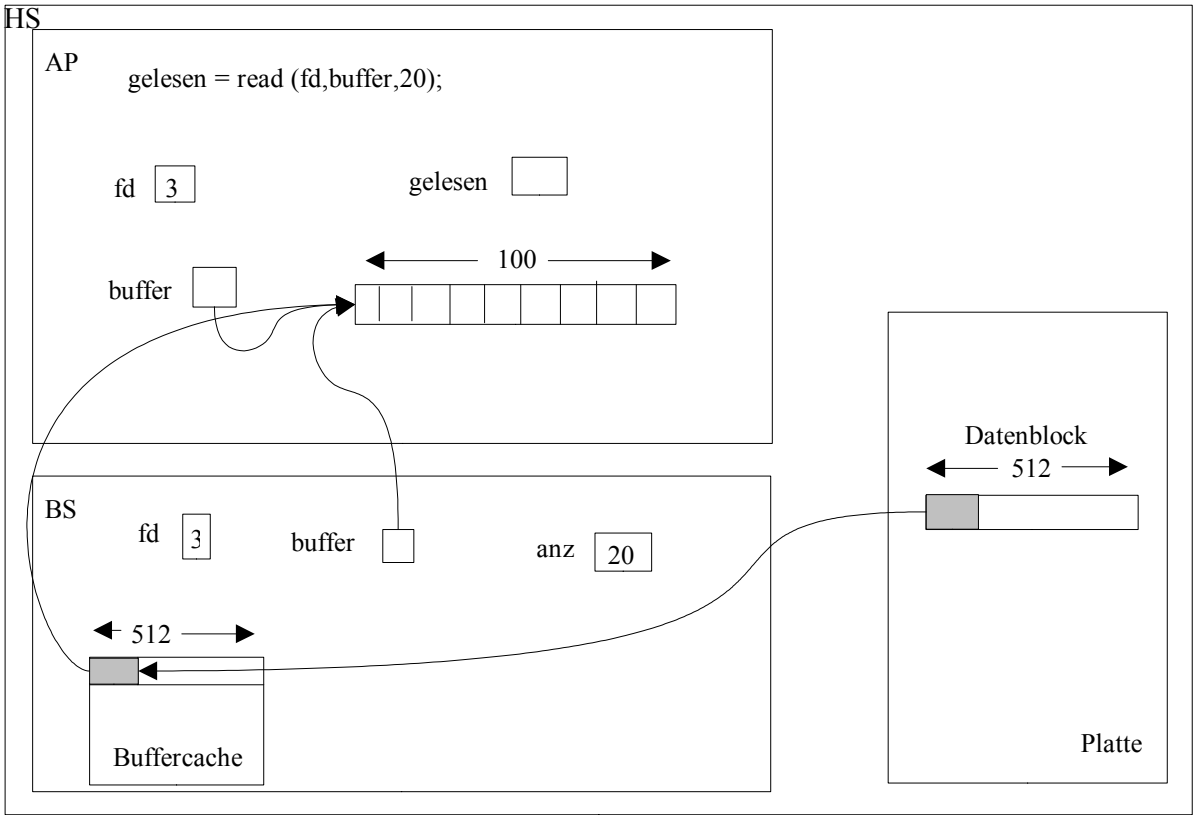
main ()

{ char buffer [100]; returnwert = -1 bei Fehler

fehlervar = read (fd, buffer ,100); } = 0 bei Datei leer / Dateiende

Der read Befehl wird vom Betriebssystem wie folgt abgearbeitet:

1. lese Datenblock von Platte



2. lege Block in Buffercache ab
3. kopiere die ersten 20 Byte ins AP
4. Gebe Anzahl der ins AP kopierten Bytes zurück

Beispiel:

// lesen von der Standardeingabe und schreiben auf die Standardausgabe

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
```

```
main ()
{       // Standardeingabe (Tastatur) => FD = 0; Standardausgabe (Monitor) => FD = 1; Standart-
        // fehlerausgabe => FD = 2
...
 char buf [1024];
 int len;
 while ((len = read (0,buf,1024)) > 0)
 {
  write (1,buf,len);
 }
 if  (len < 0)
  { perror ("read");
    exit ();
  }
}
```

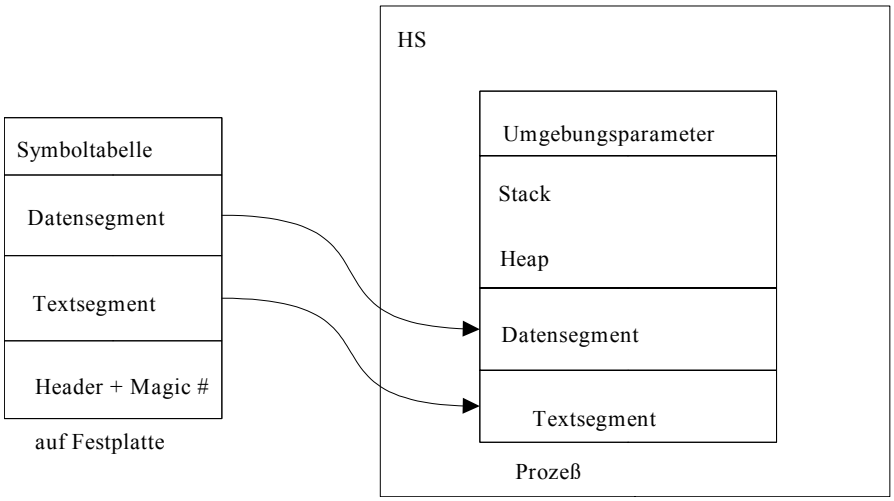
6.: Parallele Prozesse unter UNIX:

1) Was ist ein Programm?

   Folge von Anweisungen, die über einen Compiler in eine ausführbare Datei übersetzt wurden. Diese Datei befindet sich auf der Festplatte und besteht mindestens aus: Symboltabelle, Datensegment, Textsegment und Header + „Magic Number“ (Bild siehe unten)

2) Was ist ein Prozeß?

   Ein Prozeß ist die Ausführung eines Programmes in einer gegebenen Umgebung.



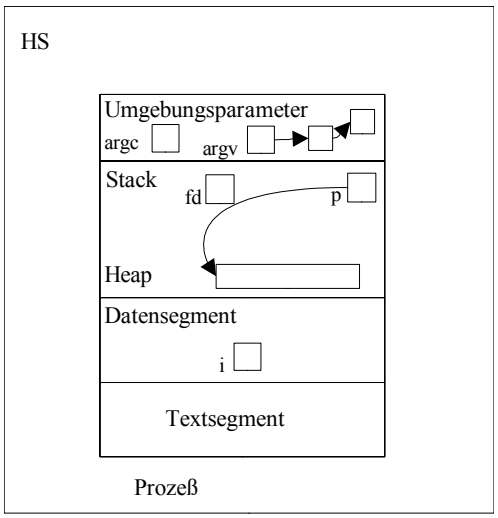


Abbildung folgender Variablen im Speicher:

```
int i;
main (int argc; char ** argv)
{
        int fd;
        char *p;
        P = malloc (100);
}
```

Jedem Prozeß wird vom Betriebssystem eine Prozeßidentifikationsnummer (pid) zugeordnet.

Ermitteln der pid:

1) Programmebene:

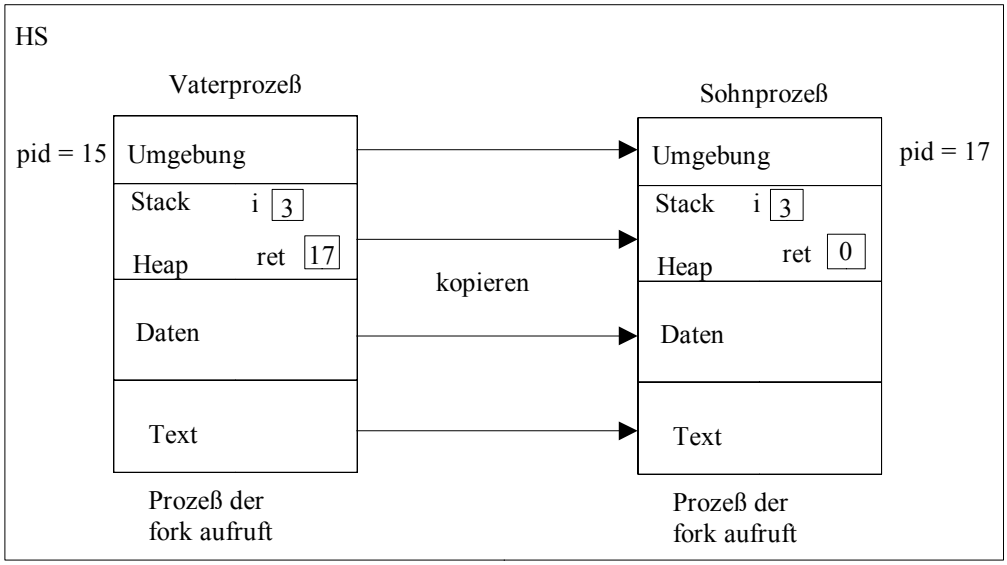
   int getpid () // zurückgegeben wird die pid des aufrufenden Prozesses

2) ps Kommando

Erzeugen eines weiteren Prozesses aus einem laufenden Prozesses heraus mit

int fork ();

Durch diesen Aufruf wird folgendes bewirkt:



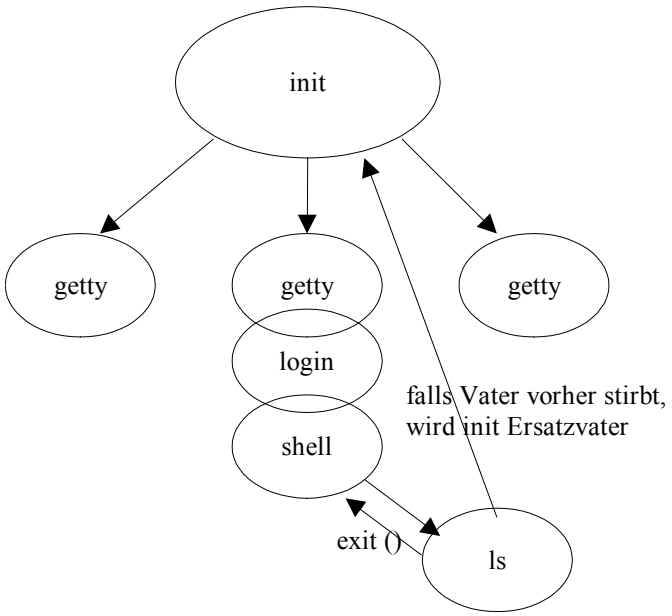
Der Sohnprozeß ist die Kopie des Vaterprozesses. Beide Prozesse arbeiten an der Stelle weiter, die dem fork – Aufruf folgt. Über den Returnwert des fork – Aufrufes teilt das Betriebssystem mit, welcher Prozeß aktiv ist. Der Vaterprozeß erhält die pid des Sohnes (im Bsp.: 17). Sohnprozeß erhält 0.

Programmfragment:

```
main ()
```

```
{...
int ret;
 ret fork ();
 if (ret >0)       // Vater
   {.....}         // wird nur vom Vater ausgeführt!
 if (ret == 0)     // Sohn
   {....}          // wird nur vom Sohn ausgeführt!
 ausgabe ();       // wird von beiden ausgeführt!
}
```

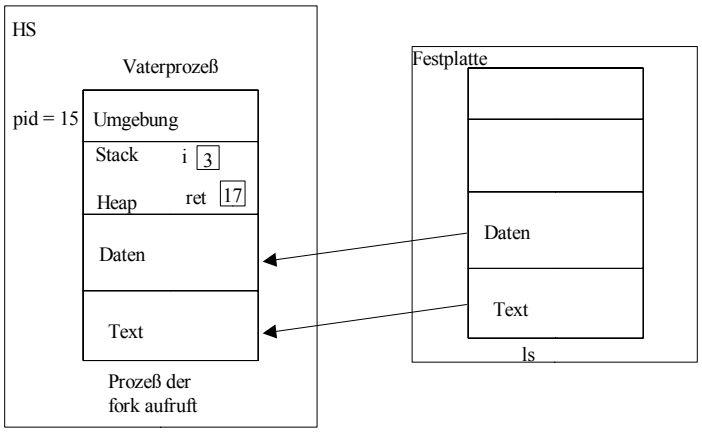
Prozeß – Hierarchie in UNIX:



Ein Prozeß, der sich beendet hat, heißt Zombieprozeß, wenn sein exit – Wert vom Vater noch nicht empfangen wurde.

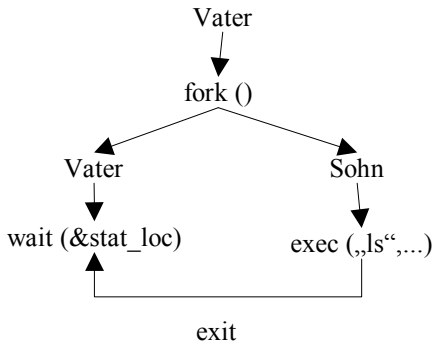
Exec Systemaufruf

Ziel: Ein Prozeß soll sich mit einem neuen Programm überlagern.



über exec wird Daten & Textsegment eingelagert.

Dies wird wie folgt angewendet:



Man kann den folgenden Suchmechanismus verwenden:

PATH=/bin:/usr/bin:. (. = Verzeichnis, in dem man sich befindet)

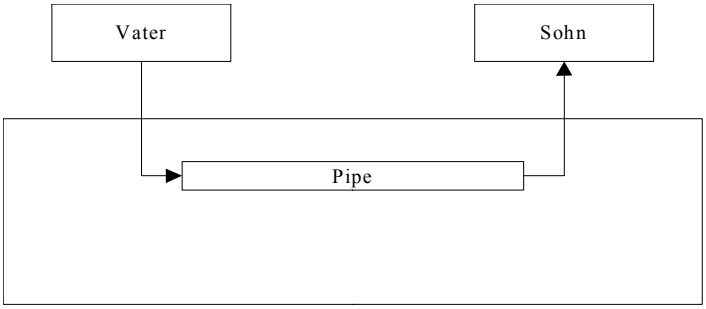
Durch den Path – Mechanismus ist es ausreichend, den Kommando – Namen anzugeben. Ansonsten muß der absolute Pfadname angegeben werden.

execl ("/bin/ls",...)

execlp ("ls",...)

Der Pipe – Mechanismus:

Möglichkeit zur Kommunikation zwischen 2 verwandten Prozessen.



- Die Pipe ist ein Einweg – Kanal, der durch das Betriebssystem zur Verfügung gestellt wird.
- In die Pipe können mit write Nachrichten abgelegt werden. Diese können mit read wieder ausgelesen werden. Es gilt das FIFO – Prinzip
- Systemaufruf: int pipe (int *pipefd);

pipefd ist ein Zeiger auf ein Array von 2 int Werten.

pipefd[0] = lesen

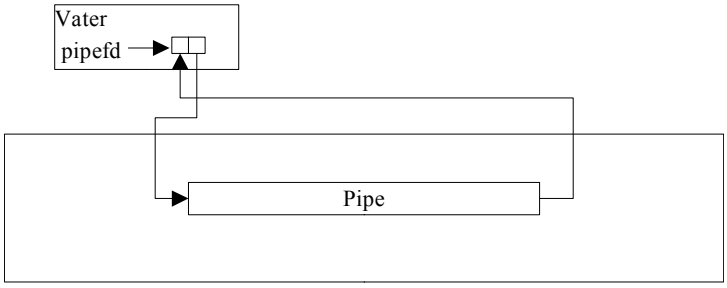
pipefd[1] = schreiben

Rückgabewert: -1 : Fehler

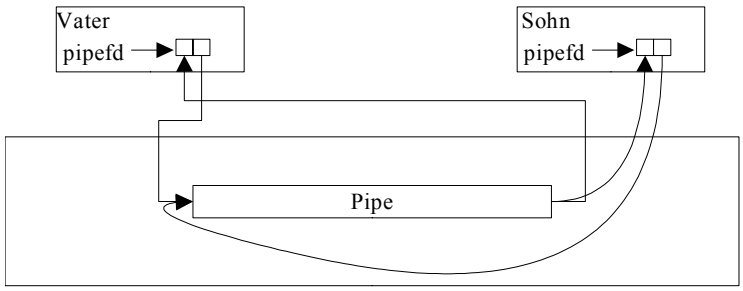
0 : falls ok

Korrektes Einrichten einer Pipe – Kommunikation:

1. Vater erzeugt Pipe:



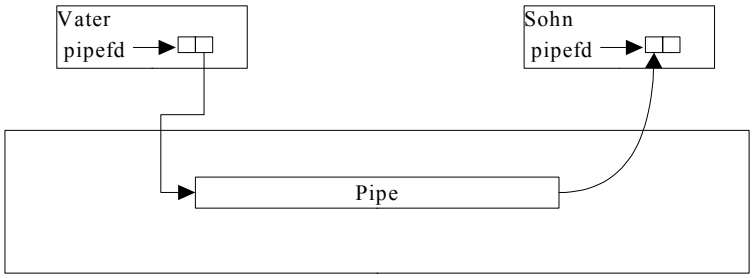
2. Vater erzeugt Sohn:



3. Einrichten der Kommunikationsrichtung (Vater -> Sohn)

Vater: close (pipefd[0]);

Sohn close (pipefd[1]);

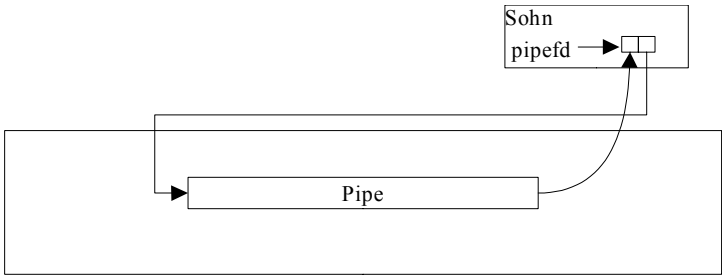


Die Kommunikation muß sauber eingerichtet werden, da sonst Probleme auftreten können.

Deadlock:

- Sohn setzt ein read ab
- In der Pipe stehen keine Daten => Sohn blockiert
- Vater beendet sich aufgrund eines Fehlers

=> Sohn wartet vergebens auf Daten



Zum Einrichten ist folgendes Vorgehen notwendig:

1) Vater führt pipe – Aufruf durch
2) Vater erzeugt Sohn
3) Einrichten der Kommunikationsrichtung (z.B. Vater -> Sohn)

   Vater: close (pipefd[0]);

   Sohn: close (pipefd[1]);

Die Kommunikationsrichtung muß „sauber“ eingestellt werden, damit folgende Unterstützung des Betriebssystems möglich ist:

1) Lesen aus der Pipe, wenn Schreibseite geschlossen: lesender Prozeß erhält 0 als Returnwert vom read
2) Schreiben in eine Pipe, wenn Lese – Seite geschlossen ist: schreibender Prozeß erhält durch das Betriebssystem ein Signal SigPipe. Dieses muß durch den Prozeß geeignet bearbeitet werden.

Wichtige Anwendung des Pipe – Mechanismus:

Die Shell erlaubt folgende Kommandoeingabe: ls|sort

Die Shell arbeitet den Aufruf wie folgt ab:

1) Pipe einrichten
2) Zwei Prozesse starten, die später ls bzw. sort ausführen sollen
3) Schließen überflüssiger FD's

   shell: 2x close

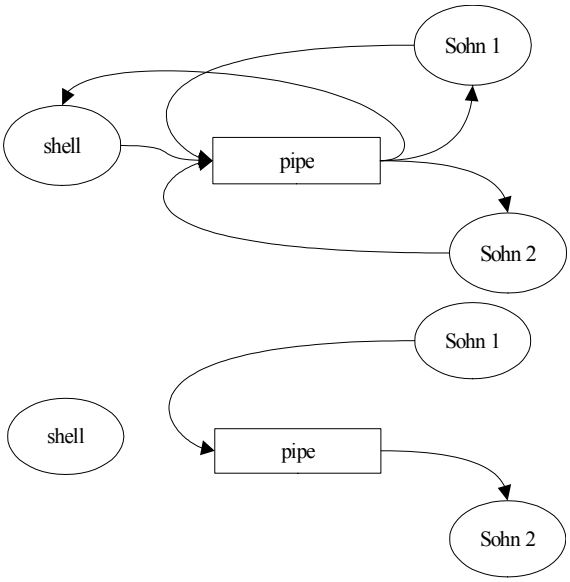
   Sohn 1: close auf lese FD

   Sohn 2: close auf schreibe FD

4) Um zu erzwingen, daß nach der Überlagerung von Sohn 1 mit ls wirklich in die Pipe geschrieben wird, muß Sohn 1 folgendes machen:

   close (1);

   newfd = dup (pipefd[1]);



Entsprechend Sohn 2:

   close (0);

   newfd = dup (pipefd[0]);

5) Überlagerung mit den jewailigen Programmen:

   Sohn 1: execlp („ls“, „ls“, Null);

   Sohn 2: execlp („sort“, „sort“, Null);

Die Shell arbeitet wie folgt:

1) Schreibe Propmt auf Bildschirm
2) Lese Benutzer – Eingabe ein (read (0, buffer, 512);)
3) Analyse der Benutzereingaben, u.A.: Wenn | - Zeichen: rufe Funktion „pipe -Handler“ auf. Dieser erledigt die Schritte 1) – 5)

```
pid1 = fork ();
if (pid1 > 0)
 { pid2 = fork ();
    if (pid2>0)
         { close (pipefd[0];
           close (pipefd [1]);
           wait (&status);
         }
  }
if (pid2 == 0)
 { close (pipefd[1];
```

```
    close (0);
    dup (pipefd[0]);
    execlp („sort“, „sort“, Null);
  }
if (pid1 == 0)
{....}
```

4) Falls keinebesonderen Leistungen erforderlich: => Söhne erzeugen => Söhne überlagern

Shared Memory (Gemeinschaftsspeicher):

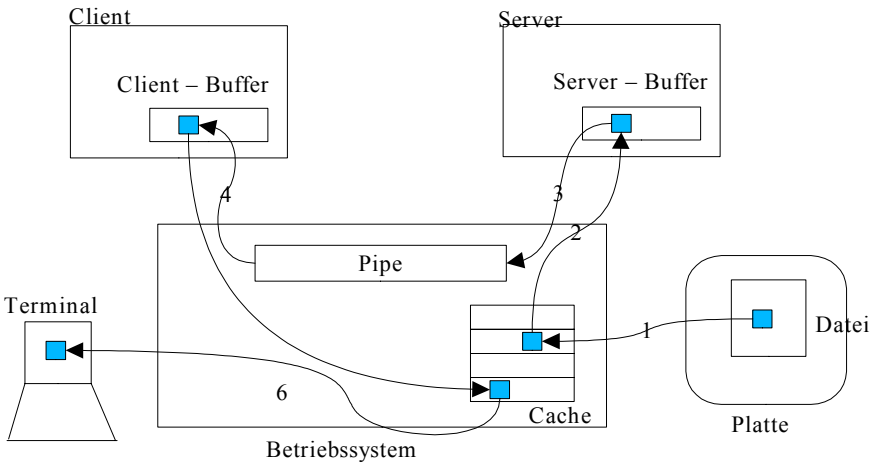
Bsp.: Client – Server – Kopierprogramm

Client:

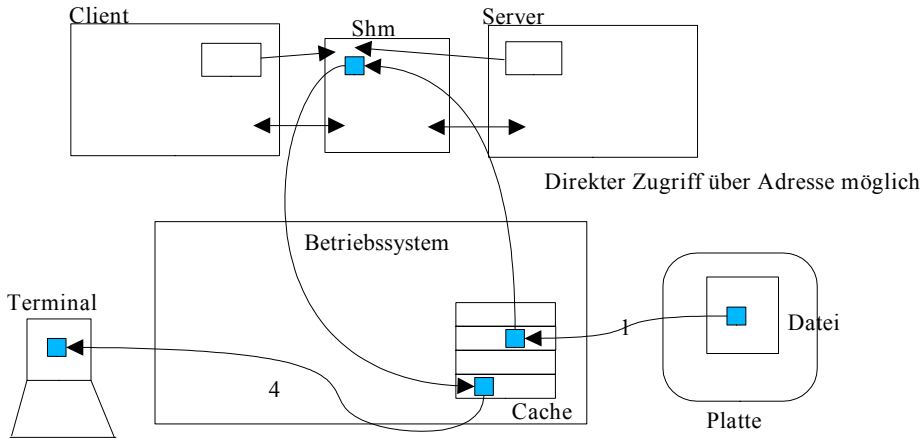
1) Erfragt Dateinamen vom Benutzer
2) Schreibt Dateinamen zum Server
2) liest Antwort (Datenblock der Datei) vom Server
3) gibt die Antwort auf den Bildschirm aus

Server:

1) liest Dateinamen vom Client
2) öffnet Datei
3) liest Datenblock von Datei
4) schreibt gelesenen Datenblock an Client



Datenblock wird 3 mal im Betriebssystem und zusätzlich je einmal im Client bzw. Server zwischengespeichert.



Zugriff geht schneller, aber es muß synchronisiert werden (Semaphor!)

Systemaufrufe für das Arbeiten mit dem Shared Memory:

Aktion	***Systemaufruf***
Einrichten des Shm bzw Zugriff öffnen	shmget (...)
Bezug herstellen	shmat (...)
Bezug lösen	shmdt (...)
kontrollieren	shmctl (...)

Folgende Header – Files sind notwendig:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

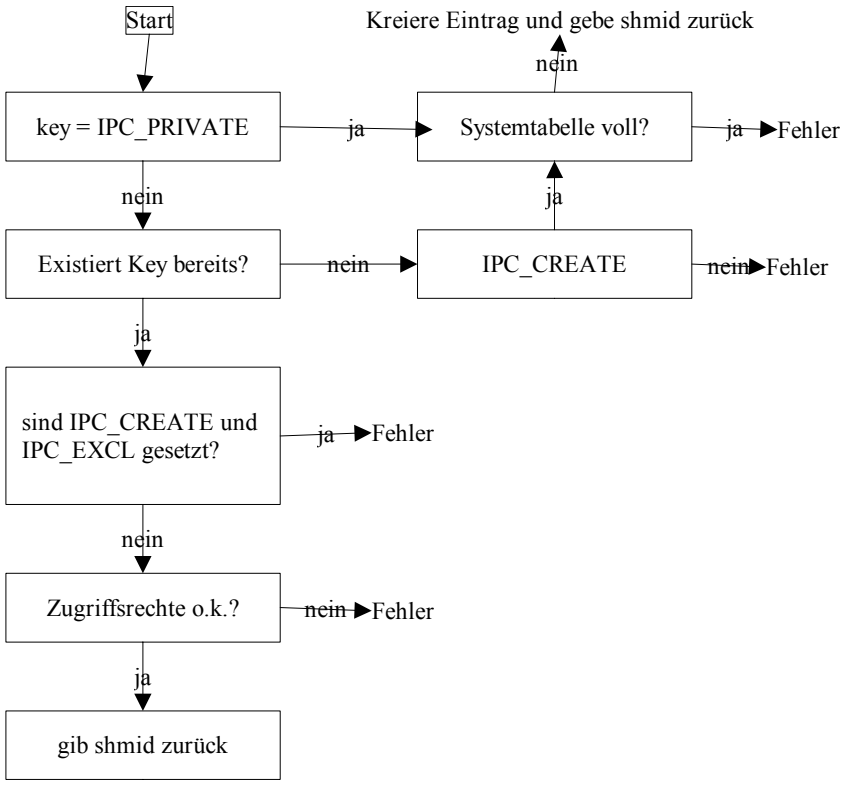
Einrichten des Shm:

int shmget (int key, int n, int mode);

Rückgabewert:	**key**:	**n**:	**mode**:
shmid: >= 0 falls ok -1 falls Fehler	numerischer Name des Shm; alle beteiligten Prozesse müssen diesen kennen	Größe des Shm	Zugriffsrechte & Flags - IPC_CREAT - IPCCREAT|IPC_EXCL

Zugriffsrechte:

0	6	4	0
	Eigen-tümer	Gruppe	Welt
r	4	4	4
w	2	2	2
x	1	1	1



Prozeß1:

```
...
main ()
{ int shmid;
  shmid = shmget (17, 25, IPC_CREAT|IPC_EXCL|0644);
          ....}
```

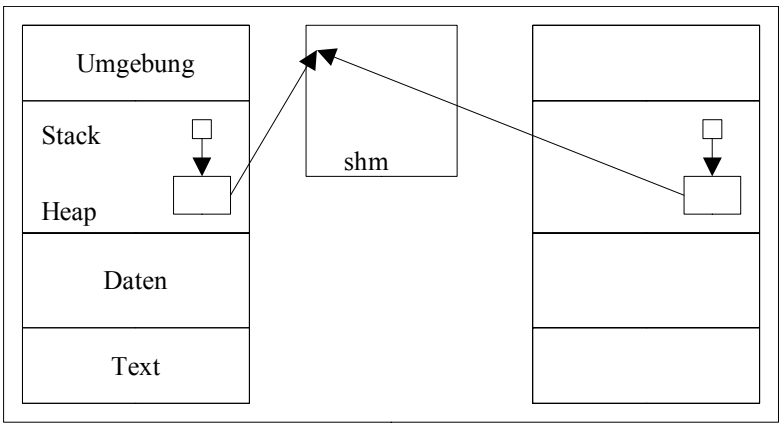
Prozeß2

```
...
main ()
{ int shmid;
  shmid = shmget (17, 25, IPC_CREAT);
...
```

Bezug herstellen:

```
char * shmat (int shmid, char * addr, int flags)
```

Adresse Vom 1. Byte des shm Null bei Fehler	Returnwert von shmget	NULL: Betriebssystem sucht nach geeigneter Stelle	read/write 0 = r&w, 1 = r, 2 = w



Durch shmat erhält man eine Adresse, die innerhalb des Prozeßlayouts liegt (im Bereich von Stack/Heap). Beim Zugriff auf shm wird diese Adresse durch das Betriebssystem auf die richtige umgesetzt (Achtung: Listen mit Pointern funktionieren nicht).

shm kontrollieren:

int shmctl (int shmid, int cmd, struct shmid_ds *buffer);

Semaphoren unter UNIX:

1. Notwendige Headerfiles:
   <sys/types.h>
   <sys/ipc.h>
   <sys/sem.h>

2. Aufrufe:

semget	Einrichten der Semaphor
semctl	Initialisieren der Semaphor
sempo	Beeinflussen der Semaphor

3. Aufruf semget

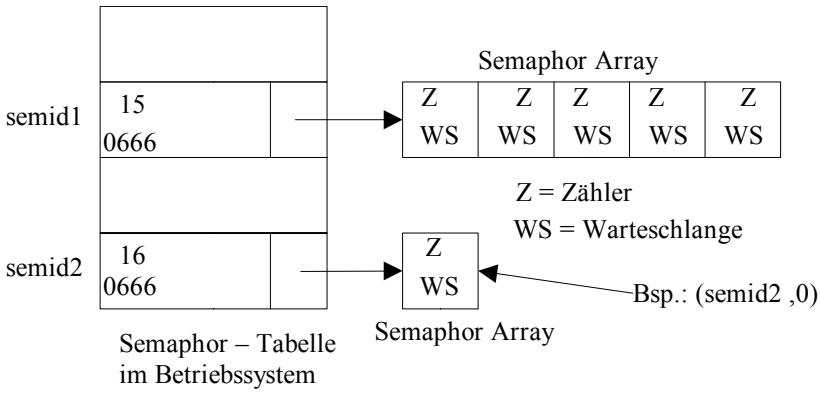
int semget (int key, int nsems, int flag);

falls o.k. > 0 (semid) sonst -1	Numerischer Name, muß allen Prozessen bekannt sein	Anzahl der Semaphoren, die unter key erreichbar sind	IPC_CREAT IPC_EXCL Zugriffsrechte siehe shmget

Beispiel:

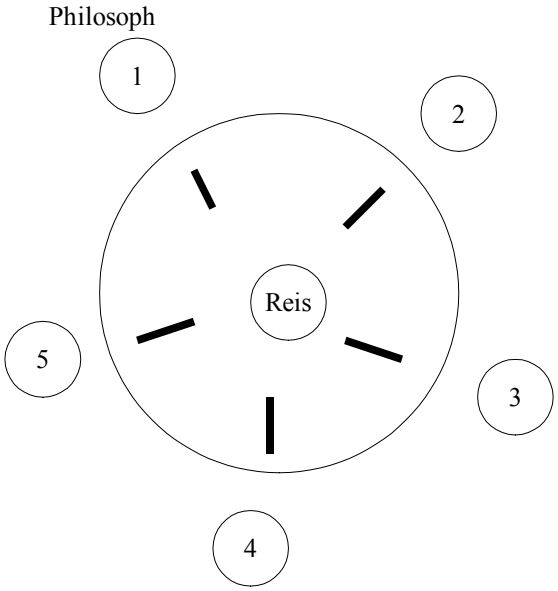
semid1 = semget (15, 5, IPC_CREAT|0666);

semid2 = semget (16, 1, IPC_CREAT|0666);



Eine Semaphor ist beschrieben durch: (semid, Nummer im Array (Index) (siehe Bsp).

Durch das Semaphor – Array ist es möglich, mehrere Semaphoren gleichzeitig zu beeinflussen.
Beispiel: 5 Philosophen – Problem:



Philosophen pendeln zwischen essen und denken.

Um eine Deadlock – Sitation zu vermeiden, kann mit Hilfe des Semaphor – Arrays auf 2 Stäbchen gleichzeitig zugegriffen werden.

4) Zähler beeinflussen

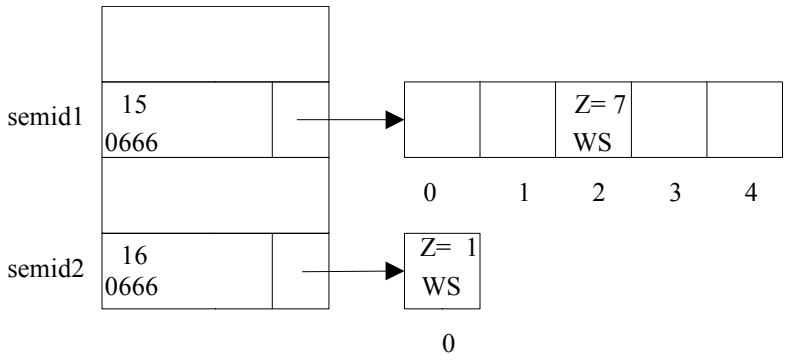
int semctl (int semid, int semnum, int cmd, int arg);

-1: Fehler 0: sonst nicht negativ bei GETVAL	Returnwert von semget	Index im Semaphor - Array	IPC_RMID GETVAL SETVAL	Wertangabe für SETVAL NULL bei GETVAL

Beispiel:

semid1 = semget (15, 5, IPC_CREAT|0666);

```
semctl (semid1, 2, SETVAL, 7);
semid2 = semget (16, 1, IPC_CREAT|0666);
semctl (semid2, 0, SETVAL, 1);
```



Ist folgendes korrekt?

```
semid1 = semget (15, 5, ...);
semctl (semid1, 2, SETVAL, 7);
Zaehler = semctl (semid1, 2, GETVAL);
if (Zaehler == 7)
{ //betrete k.A.
  semctl (semid1, 2, SETVAL. 6);
...
```

Es ist nicht korrekt, da der Wert des Semaphorzählers in der Zwischenzeit von anderen Prozessen beeinflusst werden kann. Man muß mit der Semaphoroperation (P & V) arbeiten.

5) Semaphoroperationen

```
int semop  (int semid,     struct sembuf ** sops,        int nsops);
```

0 : o.k. -1: sonst	Returnwert von semget	sops: Zeiger auf struct sembuf	Anzahl der gewünschten Semaphor - Operationen

neue Linux Version:

```
int semop (int semid, struct sembuf sops[], int nsops);

struct sembuf
{ short sem_num;      // Index
  short sem_op;       // Semaphoroperation „P“: -5, „V“: +1
  short sem_flg;      // IPC_NOWAIT, IPC_WAIT
}
```

klassische P – Operation:

```
struct sembuf P ={0, -1, 0};
```

erste Semphor in Semaphorarray	erniedrige Semaphorzähler um 1	warte wenn k.A. besetzt

```
semop (semid, &P, 1);
```

Klassische V – Operation:

```
struct sembuf V = {0, 1, 0};
```

semop (semid, &V, 1};

Beispiel: Erzeuger – Verbraucher – Problem

Gelöst über Vater – Sohn – Beziehung, shm und Semaphoren.

```
// Header files
main ()
{
  int shmid, semid, pid, i;
  int *shmmem;
  struct sembuf P = {0, -1, 0};
  struct sembuf V = (0, 1, 0};

// shm anlegen
  shmid = shmget (IPC_PRIVATE, sizeof (int), IPC_CREAT|0666);
  if (shmid == -1)
        { perror („Anlegen shm“);
          exit (1);
        }
// shm an Prozeß hängen
  shmmem = (int*) shmat (shmid, NULL, 0);  // NULL Zeiger bei Fehler, mit shmctl shm löschen
// Semaphor anlegen
  semid = semget (IPC_PRIVATE, 2, IPC_CREAT|0666);
// Zähler der Semaphoren setzen
  semctl (semid, 0, SETVAL, 1);
  semctl (semid, 1, SETVAL, 0);
// Vater – Sohn
  switch (pid = fork () )
        {
         case -1: {} //Fehlerbehandlung
         case 0:  // Sohn
                {
                  P.sem_num = 1;
                  while (1)
                        { semop (semid, &P, 1);       // P(sem1)
                          printf („gelesen %d\n“, *shmmem);
                          semop (semid, &V, 1);       // V(sem0)
                        }
        default:          // Vater
                { V.sem_num  = 1;
                  for (i = 1, i < 5, i++)
                        { semop (semid, &P, 1);       // P(sem0)
                          *shmmem = i;
                          semop (semid, &V, 1);       // V(sem1)
                        }
                kill (pid, SIGKILL);              // Sohnprozeß wird gekillt
                break;
                } // switch
// Semaphoren + shm freigeben
```

```
  shmctl (shmid, IPC_RMID, NULL);
  semctl (semid, 0, IPC_RMID, 0)
  semctl (semid, 1, IPC_RMID, 0);
}
```

Signale unter UNIX:

Bisher war die Kommunikation zwischen den Prozessen immer synchron. Allerdings gibt es Situationen, bei denen einem Prozeß eine Information asynchron mitgeteilt werden muß. Beispielsweise teilt das Betriebssystem einem Prozeß asynchron mit, daß er einen Zeigerfehler gemacht hat. Auch andere Prozesse können sich asynchrone Meldungen zusenden (z.B. einen Kill – Befehl). Ebenso ist es dem Anwender möglich über die Tastatur einen Prozeß asynchron zu beeinflussen.