3. Prozesse

3.1. Prozeßsteuerung in Multitaskingsystemen

3.1.1. Prozeßsteuerung unter UNIX

fork-Systemruf

int fork();

- erstellt Kopie, des ihn aufrufenden Prozesses
- kreierender Prozeß ist *parent process*
- erzeugter Prozeß ist *child process*
- wichtige Eigenschaft: *child process* kann auf die Dateien des *parent process* zugreifen
- nach fork() schließt Vater die Dateien, die er für Sohn geöffnet hat, damit nicht beide auf Dateien zugreifen
- *child* erhält Kopie der Prozeßvariablen des *parent processes:*
  - reale User-ID
  - reale Gruppen-ID
  - effektive User-ID
  - effektive Gruppen-ID
  - Prozeßgruppen-ID
  - Terminalgruppen-ID
  - Stammverzeichnis
  - aktuelles Arbeitsverzeichnis
  - Einstellungen des Signalhandlers
  - Maske für Dateistatus

- Unterschiede zwischen *parent* und *child process*
  - *child* besitzt neue eigene Prozeß-ID
  - *child* besitzt eigene Kopie der Filedeskriptoren des *parent processes*
  - Zeit, die zum Setzen eines Alarmsignals vergeht, wird im *child process* auf 0 gesetzt

Wann wird fork() benötigt:

1. Prozeß benötigt Kopie von sich selbst, um in zwei unterschiedlichen Tasks, die gleiche Aufgabe zu erledigen (z.B. bei Netzwerkservern)
2. Wenn ein Prozeß ein anderes Programm ausführen will.
   Zuerst erzeugt Prozeß mit fork() Kopie von sich selbst und startet dann mit exec das neue Programm
3. Wenn Applikation in mehrere Teilprozesse unterteilt werden soll.

exit-Systemruf

- Prozeß beendet sich selbst mit exit()
- es erfolgt kein Rücksprung zum aufrufenden Programm

→ es wird Exitstatus an Kernel übergeben
- Falls Vater mit wait() auf Ende des Sohnes wartet, kann er diesen Exitstatus abfragen.
- Normalerweise ist Exitstatus = 0, bei Fehler ≠ 0

exec-Systemruf

- Programm kann von einem bestehenden Prozeß mit "exec" aufgerufen werden
- gerade ausgeführter Prozeß wird durch das neu auszuführende Programm ersetzt

⇒ Prozeß-ID wird nicht verändert

wait-Systemruf

int wait(int * status)

- zuückgelieferter Wert ist Prozeß-ID des beendeten *child processes*
- wenn der aufrufende Prozeß kein *child* besitzt, dann wird -1 zurückgeliefert
- besitzt der aufrufende Prozeß ein oder mehrere *child*-Prozesse, dann wird der aufrufende Prozeß unterbrochen (geht in den schlafenden Zustand)
- wenn ein *child process* mit exit() beendet wird, dann wird *parent process* fortgesetzt
- in *status* steht der von exit() gelieferte Exitstatus, wenn dieser nicht Null war
  (zur Erinnerung wenn exit 0 liefert, dann ist die Beendigung erfolgreich geschehen)

3.1.2. Beziehungen der Prozesse

- UNIX-Systemkern */unix* in den Hauptspeicher

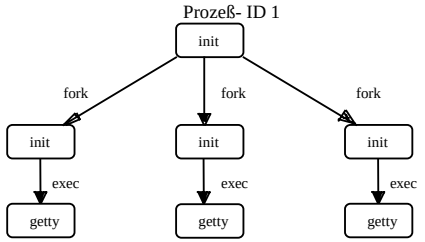
→ kreierter Prozeß erhält Prozeßnummer. 0
= "Kernel-Prozeß"

- startet */etc/init*

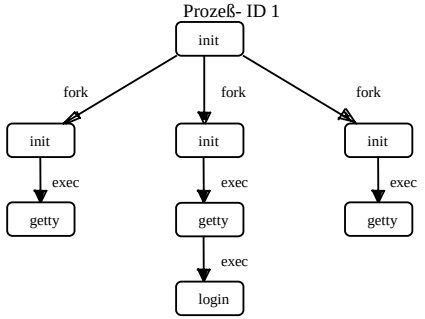
→ kreierter Prozeß erhält Prozeßnummer. 1
= "Urprozeß" (Wurzel der Prozeßhierachie)
- */etc/init* ließt */etc/inittab*
  enthält alle angeschlossenen Terminals und sonstige Leitungen
- für jeden zu aktivierenden Anschluß startet */etc/init* einen Sohnprozeß
  - jeder dieser Prozesse eröffnet zunächst die drei Dateien:
    * stdin
    * stdout
    * stderr

- danach startet */etc/init* → */etc/getty*
  ⇒ Aufforderung zum Login
  → erwartet Eingabe der Login-Kennung
  - */etc/getty* ruft Kommando */bin/login* auf
  ⇒ fragt Password ab
  - wenn Password o.k., dann */bin/sh* (oder andere Shell)
    Beachte: */bin/sh* ist kein neuer Prozeß, gleiche Prozeßnummer. wie */etc/getty*

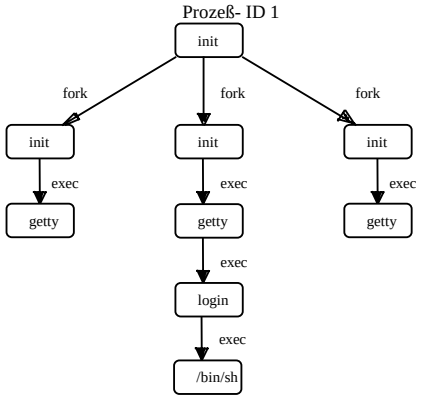
⇒ Nach Beenden der Arbeit mit Ctrl-D stirbt Sohn und */etc/getty* kreiert neuen Sohn für Terminal



Die getty- Prozesse im Wartezustand auf sich einloggende Prozess



Der getty- Prozeß nachdem ein Anwender sich eingelogged hat



Der getty- Prozeß nach dem Starten einer Login- Shell

3.2. Interprozeß-Koordination

beinhaltet **Interprozeß-Kommunikation** und **Interprozeß-Synchronisation**

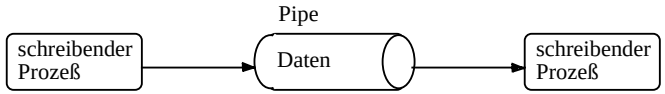
Interprozeß-Kommunikation

⇒ Kommunikation dient vorrangig dem Austausch von Nachrichten und beinhaltet gleichzeitig eine Form der Synchronisation.

1. Datenaustausch zwischen Prozessen
2. Synchronisation von Prozessen
3. Gegenseitiger Ausschluß

3.2.0. Kurze Einführung in die Objekttypen zur IPC

3.2.0.1. Kommunikation über Pipes (Röhre)



Röhre:
Am oberen Ende der Röhre füllt ein Prozeß Daten ein, die der Kommunikationspartner am unteren Ende wieder entnimmt

- lesender Prozeß will aus leerer Pipe lesen

⇒ P. wird suspendiert, bis schreibender Prozeß Daten in Röhre einleitet

- wenn Röhre voll und schreibender Prozeß will weitere Daten einfüllen

⇒ P. wird suspendiert , bis lesender Prozeß Daten aus Röhre abholt

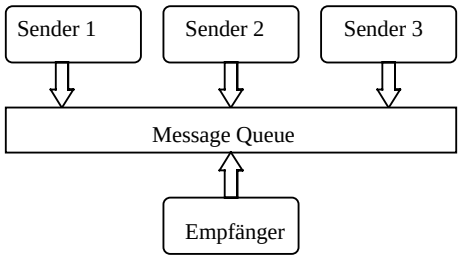
- schreibender Prozeß will Daten in Röhre einleiten, ohne daß Kommunikationspartner vorhanden

⇒ P. wird suspendiert

- Pipe einfach handhabbar, wird wie eine normale Datei behandelt

3.2.0.2. Message-Passing

! BS synchronisiert die kommunizierenden Prozesse



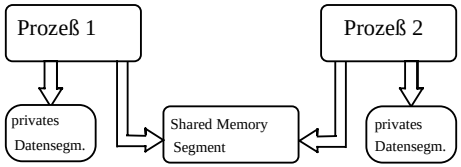
- wenn Empfänger nicht bereit
- o. Nachrichtenwarteschlange voll
- o. z. Zt. der Empfangsanforderung liegen keine Daten vor
  ⇒ P. wird suspendiert, bis Bedingung erfüllt

- Unterschied zur Pipe: Prozeß kann selbst entscheiden, ob er suspendiert werden möchte oder Kontrolle wieder erhält mit entsprechender Fehlermeldung
- Kein direktes Verschicken von z.B. Prozeß A an Prozeß B

→ Vorteil: Größtmögliche Entkopplung aller Prozesse

⇒ Nachricht wird aus dem Adreßteil des Senders entnommen und zum Zeitpunkt des Empfangs in den Adreßraum des Empfängers übertragen.

3.2.0.3. Shared Memory



! Dem Programmierer obliegt Organisation der Übergabe der Daten und die Synchronisation

- Bereich (Shared Memory Segment) wird vom BS angefordert und unter einem systemweiten ID (o. Eintrag im Objektverzeichnis) verfügbar gemacht.
- Prozesse die ID kennen, können auf diese Speicherbereiche zugreifen

3.2.1. Pipe

- Wie können Daten zwischen Prozessen ausgetauscht werden. Denkbar ist Austausch über eine Datei.
- Alternative dazu stellt Pipe dar.

a) Wenn ein Prozeß A Daten zu einem Prozeß B senden will, schreibt er diese Daten in eine Pipe, äquivalent zu einer Datei

b) Pipe ist ein im RAM gewarteter Datenpuffer, der zur Kommunikation zwischen Prozessen eingesetzt wird.

c) Pipe erlaubt Ausgabe eines Programms als Eingabe eines anderen (ohne, daß eine temporäre Datei eingeschaltet wird)

```
- UNIX
$ who > temp    /* mittels temporärer Datei */
$ sort < temp

$ who | sort       /* mittels Pipe */

‘ Ausgabe eines Programms als Eingabe eines anderen

$ cat file1 | sort
‘  Bsp. Pipe

$ cat datei1
Peter sagt Ja
Ulli sagt Nein
Heinz sagt Hallo
$ grep Hallo datei1 (search a file for pattern)
Heinz sagt Hallo
$ grep Hallo datei1 | wc
1
$
```

- Unterschied Datei, Pipe
  - Wenn Prozeß von Plattendatei zu lesen oder zu schreiben versucht, dann geht er sicher, daß die Operationen innerhalb einiger hundert ms beendet sein wird. Im ungünstigsten Fall sind zwei oder drei Plattenzugriffe notwendig.
  - Wenn Prozeß von Pipe lesen will:
    andere Situation
    * Wenn Pipe leer ist, muß Prozeß solange warten bis anderer Prozeß Daten in die Pipe eingibt
    Schreiben ähnliche Situation
  - Versucht ein Prozeß aus Pipe zu lesen oder in Pipe zu schreiben, kann das Dateisystem den Zustand der Pipe unmittelbar prüfen, ob die Operation vollendet werden kann:
    ⇒ ja es wird vollendet
    ⇒ nein Prozeß wird suspendiert
  - Pipe hat sehr begrenzte Kapazität

Schreiben in volle Pipe
Lesen aus leerer Pipe
} Prozeß wird suspendiert

· Beispiele

Datei (Assembler)

```
;NAME datei.asm
;*********************************************************
;*                                                       *
;* DOS-Funktionen zum Eröffnen, Lesen, Schließen einer Datei    *
;*                                                       *
;*********************************************************

        .data

FNAME       db      'nc.exe'
BUFFER      db      512 dup (?)
HANDLE      dw      (?)

        .code

        mov     dx,offset FNAME
        mov     ax,3d00h
        int     21h             ;File eröffnen zum Lesen
        mov     handle,ax
        .
        .
        .
        mov     bx,handle
        mov     cx,200h
        mov     dx,offest BUFFER
        mov     ah,3fh
        int     21h             ;512 Bytes aus File lesen
        .
        .
        .
;       ****************************************
;       *      Daten auswerten                 *
;       *                                      *
;       ****************************************

        mov     bx,handle
        mov     ah,3eh
        int     21h             ;File schließen
        .
        .
        .
```


Datei (C)

```
/* ophandle.c */

# include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
#include <sys\stat.h>

int main (void)
{
        int handle;
        char msg[] = "Hello world";

if ((handle = open ("Test.tst",
                        O_RDWR | O_CREAT | O_TEXT,
                        S_IREAD | S_IWRITE))== -1)
        {
                perror("Fehler\n");
                return 1;
        }
        write (handle,msg,strlen(msg));
        printf ("handle=%d\n",handle);
        close (handle);
        return 0;
}
```

Pipe (C)

```
/* pipe.c */

int main ()

{
int fildes[2],i;
char buf[20];
char msg[]="hello world";

        if (pipe(fildes) == -1)
        {
                perror ("Fehler\n");
                return 1;
        }
        write(fildes[1],msg,strlen(msg));
        /* schreibt max. 10240 Datenbytes in die Pipe.
           Bei Ueberschreitung wird der schreibende Prozess
           blockiert */
        read(fildes[0],buf,strlen(msg));
        printf("%s\n",buf);
        return 0;
}
```

- Pipes werden von Prozeß angelegt
  int fildes[2]
  int pipe(fildes)
  → 2 Dateideskriptoren, einer für Lesen und einer für Schreiben
- Eine gewöhnliche Pipe kann nur über Deskriptoren, nicht über Namen angesprochen werden.

- Vor-u. Nachteile
  - Pipe einfach wie Datei handhabbar
  - Großer Vorteil: Dateinummernweitergabe an Kindprozeß
  - Über Pipes kommunizierende Prozesse benötigen daher entweder einen gemeinsamen Vaterprozeß oder müssen selbst in einem Vater-Sohn-Verhältnis stehen
  - Unterschiede zwischen Pipes, Dateien und Treiber werden voreinander verborgen
  - Kommunikation in zwei Richtungen benötigt zwei Pipes
    Pipe ist Einbahnstraße

- ab UNIX System III named Pipes (FIFO Dateien)
- Prozeß der Pipe anlegt verzweigt im Normalfall (fork()) und gibt beide Dateideskriptoren an seine Sohnprozesse weiter.

3.2.2. Shared Memory

- BS stellt dem Anwender Funktionen Systemfunktionen zur Verfügung um Shared Memory zu realisieren
- Benutzung eines Shared Memory ist schnellste Möglichkeit zum Datenaustausch

- Speicher anfordern unter MS/DOS

Speicher anfordern	
Aufruf	Ergebnis
AH = 48	CF = kein Fehler
BX = Speicherbereich in Paragraphen	AX = Segmentadresse
	CF = 1 Fehler
	AX = Fehlernummer
	BX = max. verfügbare Größe

- Speicher anfordern unter RMX → Segment kreieren

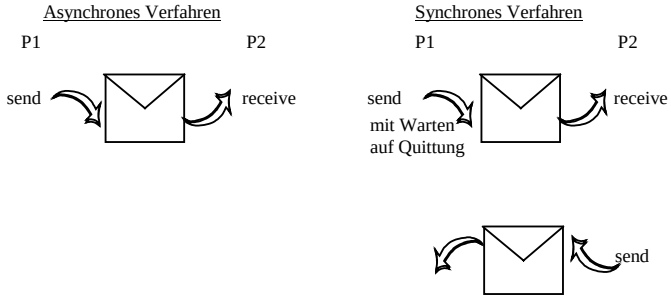
  create$segment
  <AX>:= TOKEN = Basisadresse des Segments
  catalog$object
  Name: Share
  Objekt: TOKEN

- Shared Memory unter UNIX

shmget()	Öffnen des Segments, damit hat Prozeß Zugriff auf dieses Segment, wie auf lokalen Speicher. Wichtig: shmflg muß Lesen und Schreiben erlauben, somit ist Shared Memory Segment wertlos
shmat	attach-Anbinden, hier wird Speicher angelegt
shmdt	detach-Lösen
shmctl	control-Kontrolle, Speichersegment bleibt in der Systemtabelle solange erhalten, bis es explizit mit shmctl gelöscht wird

3.2.3 Messages

- Austausch von Nachrichten über Botschaftenwarteschlangen (message queues), Briefkästen (Mailboxes) oder neuerdings durch Mailslots (Windows-NT)
- Das Absenden einer Botschaft geschieht durch das Abstellen der Botschaft in Message Queues
  (UNIX gesamte Nachricht, Message passing)
  (RMX Kenndaten (ID, TOKEN))
- Herauslesen der Nachricht aus Queue bzw. Mbx. bewirkt ein Löschen der Nachricht in Queue bzw. Mbx.
  → Es ist unmöglich, daß zwei Prozesse dieselbe Nachricht empfangen (gleiche Nachricht 2x senden)



- Betriebssystem interne Warteschlangen:
  - Prozeß wartet an Mbx auf Message
    ⇒ Taskwarteschlange (mehrere Prozesse warten an dieser Mbx)
  - Objekt wartet auf Abholung (mehrere Objekte warten)
    ⇒ Objektwarteschlange

- Message Queue unter UNIX

msgget()	Öffnen einer Messagequeue
msgsnd()	Senden
msgrcv()	Empfangen
msgctl()	Messagequeue-Steuerung

3.2.4. Semaphor

Def.:
Semaphor ist eine Variable, auf die nur über definierte Funktionen zugegriffen werden kann.
Diese Funktionen sind atomar, d.h. sie dürfen nicht unterbrochen werden.
Sie müssen vom Betriebssystem zur Verfügung gestellt werden., da Benutzerfunktionen immer unterbrechbar sind.
Bei der Initialisierung des Semaphors wird festgelegt, wieviel Prozesse sich gleichzeitig im kritischen Abschnitt befinden dürfen.

z.B. 3 Drucker Semaphor = 3
bei Belegung (P-Operation) dekrementieren
bei Freigabe (V-Operation) inkrementieren

binäres Semaphor (0;1)
allgemeines Semaphor (Initialisierung > 1)

! Achtung: Bei der Verwendung mehrerer Semaphore ist höchste Vorsicht geboten, da die Operationen nicht kommutativ sind.
Wirkung des Semaphors wird schnell undurchsichtig.

Definition des Semaphors (Dijkstra 1968)

Semaphor = abstrakter Datentyp, für den die Operationen P und V erlaubt sind.

P(sem) :	V(sem) :

if (sem>0)
then
sem:= sem -1
else
aufrufenden Prozeß
suspendieren
	if (irgendein Prozeß suspendiert aufgrund von P(sem)
	then
	Aktivieren des
	suspendierten Prozesses
	else
	sem:= sem+1

Semaphor
- kann entweder "frei" oder "belegt" sein
- Hilfsmittel zur Synchronisation von Prozessen
  ‘Bsp.:
  Tankstelle
  Portemonnaie
  Drucker

Die speisenden Philosophen (The dining philosophers problem)

- 5 Philosophen sitzen um Tisch
- vor ihnen steht ein Teller Spaghetti
- zum Essen benötigt Philosoph 2 Gabeln

- Anzahl der vorhandenen Gabeln
- 3 Zustände denkend, hungrig, essend

- Problem der speisenden Philosophen beinhaltet wichtigsten Problemstellungen der Synchronisation:
  - Anforderung mehrerer Betriebsmittel
  - die Vermeidung der Verhungerung von beteiligten Prozessen
  - optimale Ausnutzung von Betriebsmitteln

```
/* Scheinlösung nach Tanenbaum */

# define N      5

philosopher(i)
int i
{
        while (TRUE) {
                think();                /* Philosoph denkt */
                take_fork(i);           /* nimmt linke Gabel auf */
                take_fork((i+1) % N);/* nimmt rechte Gabel auf */
                eat();                  /* happen pappen Spaghetti */
                put_fork(i);            /* legt linke Gabel hin */
                put_fork((i+1) % N);  /* legt rechte Gabel hin */
        }
}


/* take_fork wartet solange, bis die bezeichnete Gabel verfügbar ist
   und nimmt sie auf */
```

Leben eines Philosophen besteht aus den Prozessen Denken und Essen.

1. *take_fork* wartet solange bis bezeichnete Gabel verfügbar und nimmt sie auf.
leider falsch:

Wenn alle gleichzeitig linke Gabel aufgreifen kann keiner rechte Gabel ergreifen:

D E A D L O C K (Verklemmung)

2. Modifizierung:

- Nach Aufnahme der linken Gabel, wird geprüft ob rechte verfügbar.
  · falls re. verfügbar, diese aufnehmen
  · falls re. nicht verfügbar, li. hinlegen

immer noch falsch

Wenn alle gleichzeitig linke Gabel aufnehmen
Erkennen, daß re. Gabel nicht verfügbar
Linke Gabel hinlegen, warten
weiter von vorn

S T A R V A T I O N (Aushungern)

3. Zufallsverteilung

Wahrscheinlichkeit gering, daß Gleichschritt auftritt.
Aber, man möchte Lösung, die immer arbeitet und nicht vom Zufall abhängt (z.B. Kernkraftwerk)

4. Mit binärem Semaphor Anweisungen hinter *think()* schützen.

Vor take_fork(): Nach dem Hinlegen der Gabeln:	down auf MUTEX up auf MUTEX

Leistung mangelhaft:
Nur 1 Philosoph kann essen.
Es könnten aber 2 Philosophen essen

5. Lösung nach Tanenbaum

- Vektor *state* speichert ob Philosoph denkt, hungrig ist oder ißt
- Philosoph kann nur dann in den Zustand essen übergehen, wenn die Nachbarn nicht essen
- li. und re. Nachbarn durch Makros LEFT, RIGHT
  (wenn i=2 → LEFT=1, RIGHT=3)
- pro Philosoph ein Semaphor

- Globales Semaphor MUTEX: Für die Zeit, in der der Philosoph Gabeln ergreifen möchte, dürfen andere Philosophen keine Gabeln ergreifen.
- Ein Philosoph muß genau dann zum Essen übergehen, wenn er hungrig und keiner seiner Nachbarn beim Essen ist.

- Durch zwei mögliche Ereignisse:
  a) ein Philosoph wird hungrig
  b) anderer Ph. geben Gabeln frei, nachdem sie gegessen haben

  zu a) Denker muß testen, ob beide Gabeln zur Verfügung stehen
  test(i)
  zu b) Wenn Ph. gegessen hat, muß er für die Nachbarn prüfen, ob diese jetzt essen sollten
  test(LEFT)
  test(RIGHT)

! Damit das Testen zweier Gabeln (a) und des Zustandes des Philosophen nicht durch fremde Zustandsübergänge verwirrt wird, hat dieses unter dem Schutz des gegenseitigen Ausschlusses stattzufinden.


```
/* Lösung nach Tanenbaum */

# define N                  5 /* Anzahl der Philosophen */
# define LEFT               (i-1)%N /* Nummer seines linken Nachbarn */
# define RIGHT              (i+1)%N /* Nummer seines rechten Nachbarn */
# define THINKING           0 /* der Philosoph denkt */
# define HUNGRY             1 /* der Philosoph versucht Gabeln zu erhalten */
# define EATING             2 /* der Philosoph isst */
typedef int semaphore       /* Semaphore sind besondere "int" */
int state[N]                /* Vektor fuer Zustaende */
semaphore mutex=1           /* zum gegenseitigen Ausschluá */
semaphore s[N]

philosopher(i)
int i;                      /* Philosoph Nummer, 0 bis N-1 */
{
        while (TRUE) {
                think();            /* Philosoph denkt */
                take_forks(i);      /* zwei Gabeln erwerben oder blockieren */
                eat();              /* happen pappen Spaghetti */
                put_forks(i);       /* beide Gabeln zurück legen */
        }
}

take_forks(i)
int i;                              /* Philosoph Nummer, 0 bis N-1 */
{
        down(mutex);                /* Eintritt in kritische Region */
        state[i] = HUNGRY;          /* Philosoph "i" ist hungrig */
        test(i);                    /* versucht zwei Gabeln zu erwerben */
        up(mutex);                  /* verlaesst kritischen Abschnitt */
        down(s[i]);                 /* blockiert, wenn Gabeln nicht erhalten */
        }

put_forks(i)
int i;                              /* Philosoph Nummer, 0 bis N-1 */
{
        down(mutex);                /* Eintritt in kritische Region */
        state[i] = THINKING;        /* Philosoph "i" beendet das Essen */
        test(LEFT);                 /* pruefen: kann linker Nachbar nun essen? */
        test(RIGHT);                /* pruefen: kann rechter Nachbar nun essen? */
        up(mutex);                  /* verlaesst kritische Region */
        }

test(i)
int i;                              /* Philosoph Nummer, 0 bis N-1 */
{
 if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGTH] != EATING) {
        state[i] = EATING;
        up(s[i]);
        }
}
```

3.2.5. Zusammenfassung:

- Echter Datenaustausch kann durch Botschaften (messages) oder über einen von allen beteiligten Prozessen gemeinsam benutzten Speicher (shared memory) vollzogen werden. Die reine Synchronisation wird von einer Reihe umfangreicher Semaphor-Operationen abgedeckt.

⇒ Alle drei Mechanismen faßt man unter dem Begriff IPC zusammen

Shared-Memory-Segmente

- werden in den seltensten Fällen verwendet, ohne eine Synchronisationshilfe wie z.B. Semaphore hinzuzuziehen.
- Für jeden der drei Bereiche verwaltet das System eine systeminterne Tabelle

  Eintrag in Message-Tabelle repräsentiert Messagequeue
  Eine Zeile in Semaphor-Tabelle repräsentiert bestimmte Anzahl von Semaphoren (Semaphorevektor)
  Eintrag in Shared-Memory-Tabelle repräsentiert ein Shared Memory-Segment

Botschaften

- kompliziert, aber vielseitig

msgget()	Öffnen einer Messagequeue
msgsnd()	
msgrcv()	
msgctl()	Messagequeue-Steuerung

- Botschaftensystem
  Client-Server-System