3 Prozesse und Nebenläufigkeit

3.1 Das Prozeßkonzept

- Programm: Folge von Anweisungen (statisch), wird im Speicher hinterlegt
- Prozeß: In Ausführung befindliches Programm mit den zugehörigen Daten (dynamisch)
  hier Rechenprozeß, nicht technischer Prozeß

  Ein Programm kann sich mehrfach in Ausführung befinden!

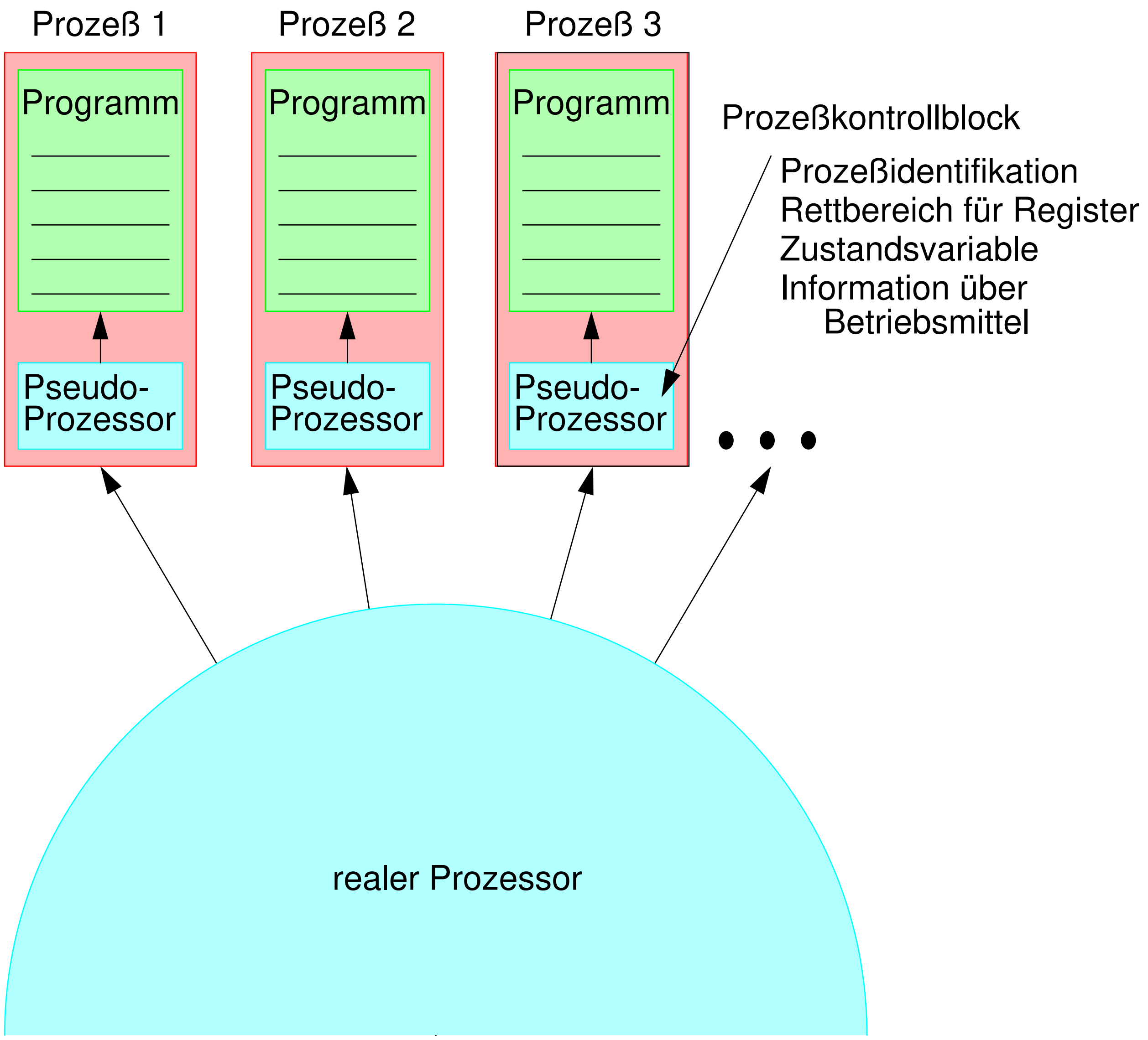
- Prozeßzustände
  - Bereit (*ready*)

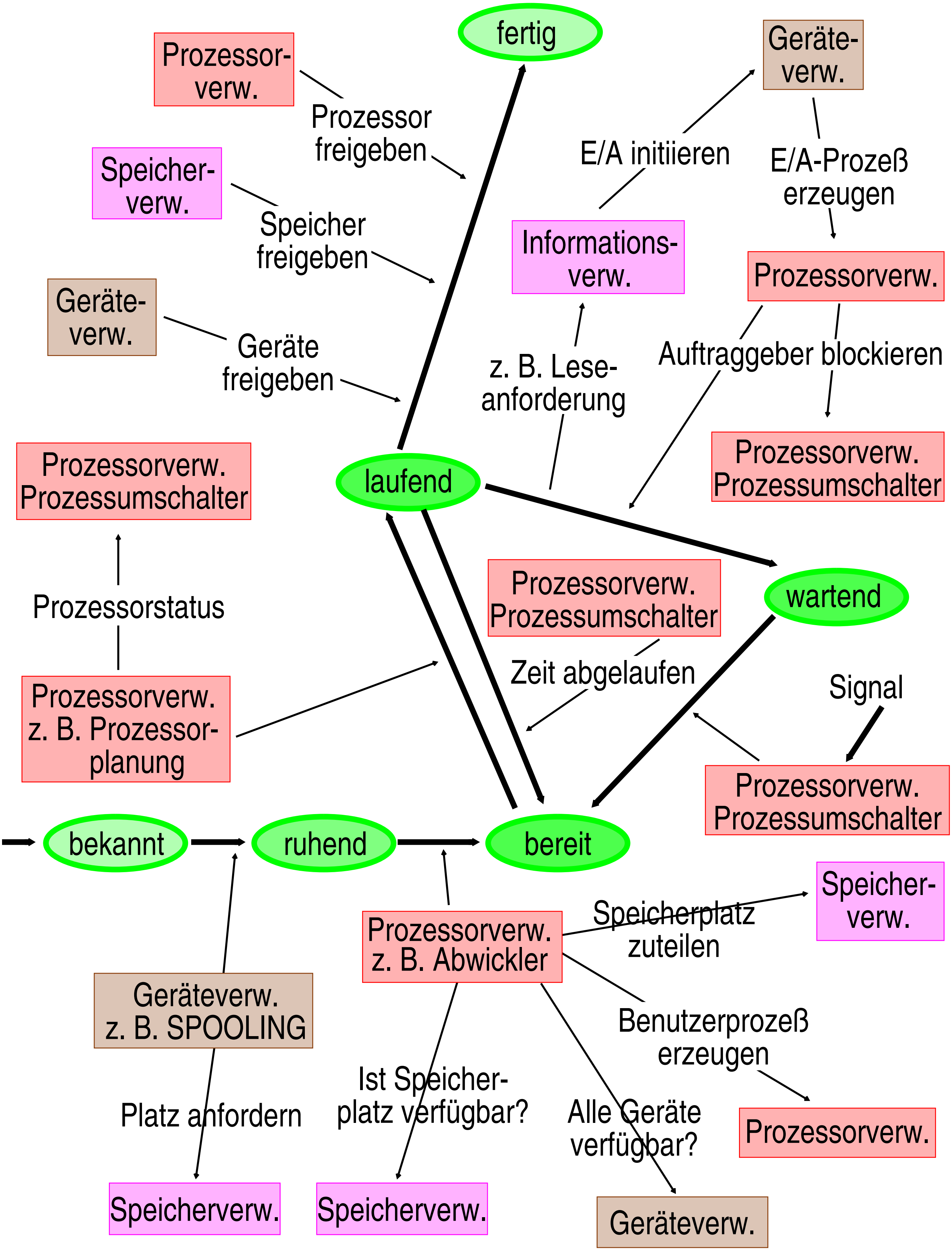
    Der Prozeß bewirbt sich um die Zuteilung des realen Prozessors und hat bereits alle anderen Betriebsmittel, die für seine Weiterarbeit erforderlich sind, zugeteilt bekommen.
  - Laufend (*running*)

    Der Prozeß wird vom realen Prozessor ausgeführt.
  - Blockiert (*blocked*)

    Der Prozeß bewirbt sich nicht um die Zuteilung des Prozessors, da er auf die Erfüllung einer Bedingung (z. B. dem Abschluß eines E/A-Transfers) wartet.
  - Beendet (*terminated*)

    Der Prozeß hat all seine Berechnungen beendet und die ihm zugeteilten Betriebsmittel freigegeben.


Prozeß 1
Prozeß 2
Prozeß 3
Programm
Programm
Programm
Pseudo-Prozessor
Pseudo-Prozessor
Pseudo-Prozessor
Prozeßkontrollblock
Prozeßidentifikation
Rettbereich für Register
Zustandsvariable
Information über Betriebsmittel
• • •
realer Prozessor

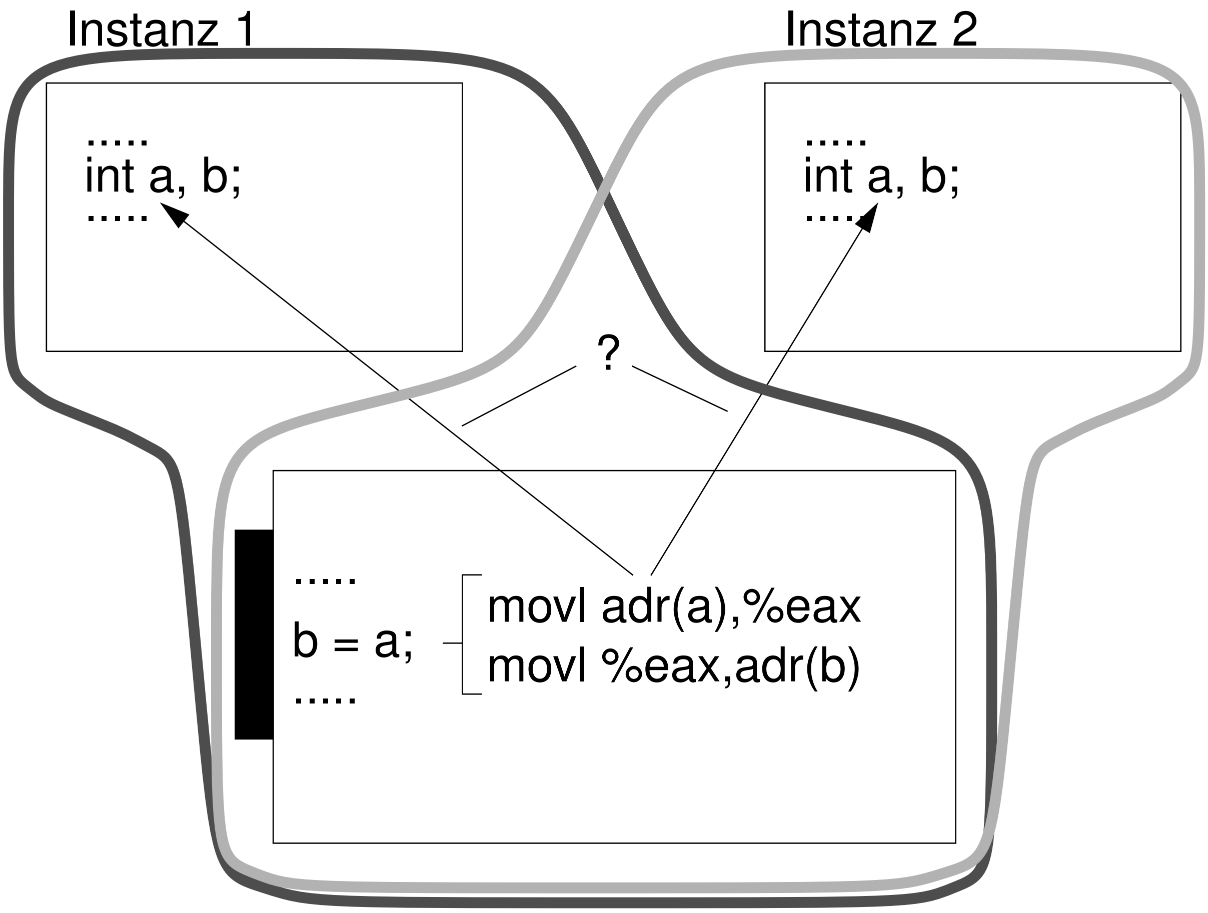

fertig
Prozessor-verw.
Prozessor freigeben
Speicher-verw.
Speicher freigeben
Geräte-verw.
Geräte freigeben
E/A initiieren
Geräte-verw.
E/A-Prozeß erzeugen
Informations-verw.
Prozessorverw.
z. B. Lese-anforderung
Auftraggeber blockieren
Prozessorverw. Prozessumschalter
laufend
Prozessorverw. Prozessumschalter
Prozessorstatus
Prozessorverw. Prozessumschalter
wartend
Zeit abgelaufen
Signal
Prozessorverw. z. B. Prozessor-planung
Prozessorverw. Prozessumschalter
bekannt
ruhend
bereit
Speicher-verw.
Speicherplatz zuteilen
Prozessorverw. z. B. Abwickler
Geräteverw. z. B. SPOOLING
Benutzerprozeß erzeugen
Ist Speicher-platz verfügbar?
Platz anfordern
Alle Geräte verfügbar?
Prozessorverw.
Speicherverw.
Speicherverw.
Geräteverw.

- Bestandteile
  - Befehlssegment (pro Klasse)
  - Datensegment (pro Instanz)
  - Stapelsegment (pro Aktivitätsträger)

- Ablauf
  - Register: Befehlszähler, Index-/Basisregister
  - Methodenaufruf und -beendigung

- Ausnahmebehandlung
  - Programmstatuswort
  - Unterbrechungen, Signale
    - Traps
    - Interrupts

Bindung Befehlsadressen - Datensegment



Mögliche (wenn auch unbefriedigende) Lösung:

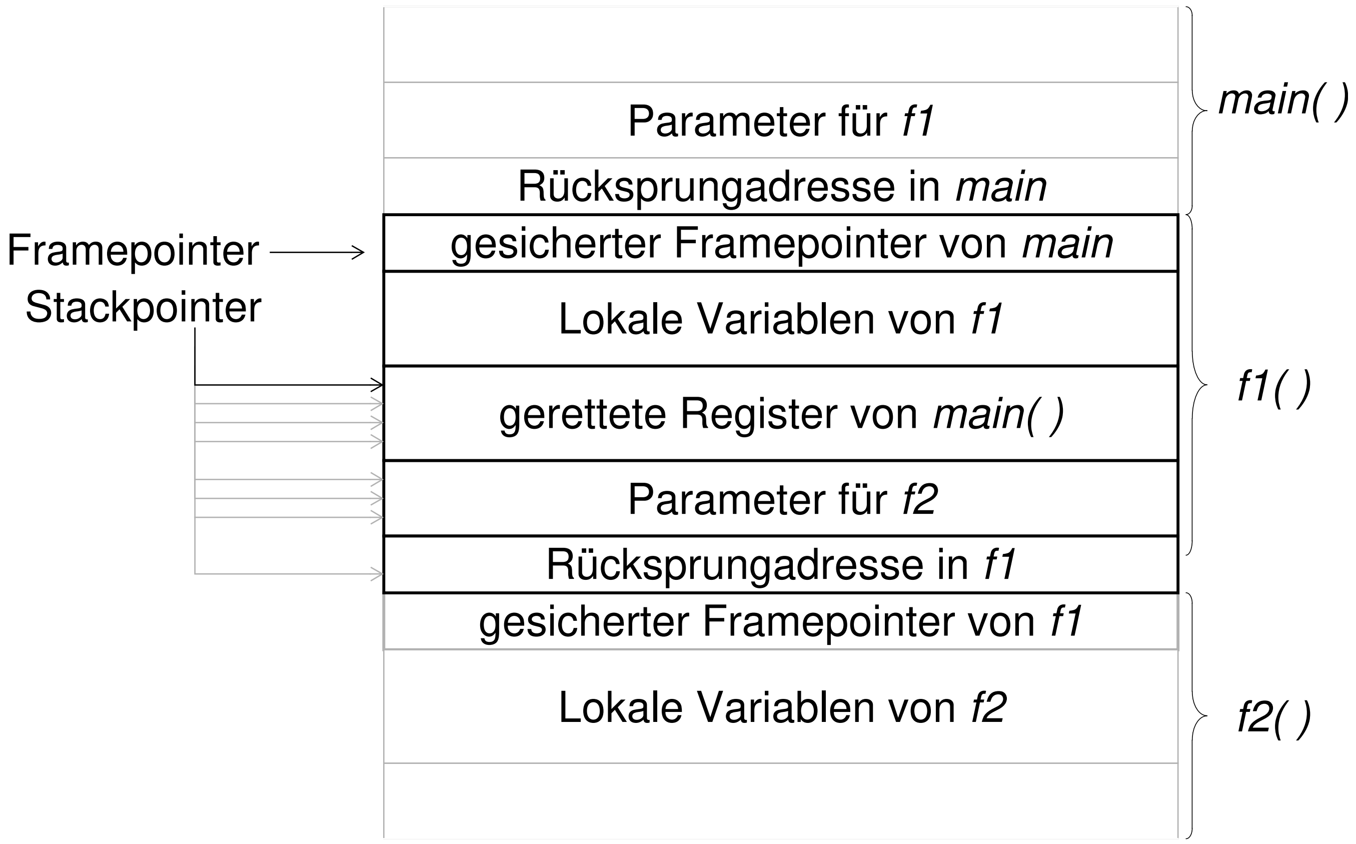
Benutzung indizierter Befehle

```
                            basis:
  0000 00000000             a:         .long    0
  0004 00000000             b:         .long    0

                            start:
  0008 8D 2D 00000000   leal basis,%ebp
  000e 8B 85 00000000   movl a-basis(%ebp),%eax
  0014 89 85 04000000   movl %eax,b-basis(%ebp)
```

Stapelaufbau

- Bei jedem Methodenaufruf wird ein ***Stack-Frame*** angelegt, in dem
  - lokale Variablen der Methode
  - Aufrufparameter an weitere Methoden
  - Registerbelegung der Methode während des Aufrufs weiterer Methoden

  gespeichert werden.
- Stapelorganisation ist Prozessor-, Compiler- und Betriebssystem-abhängig (Beispiele aus einem UNIX auf Motorola 68k-Prozessor)
- Aufbau eines *Stack-Frame* (Funktionen *main( ), f1( ), f2( )*)

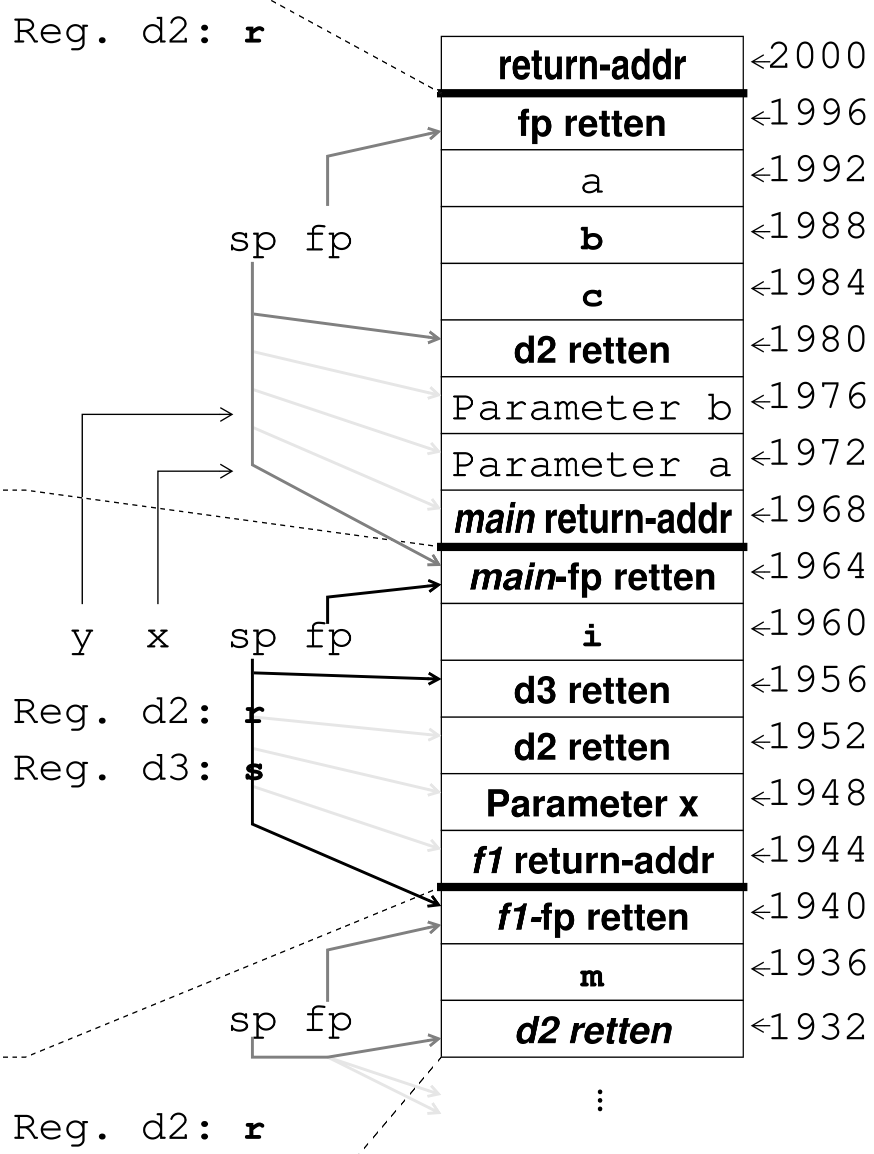


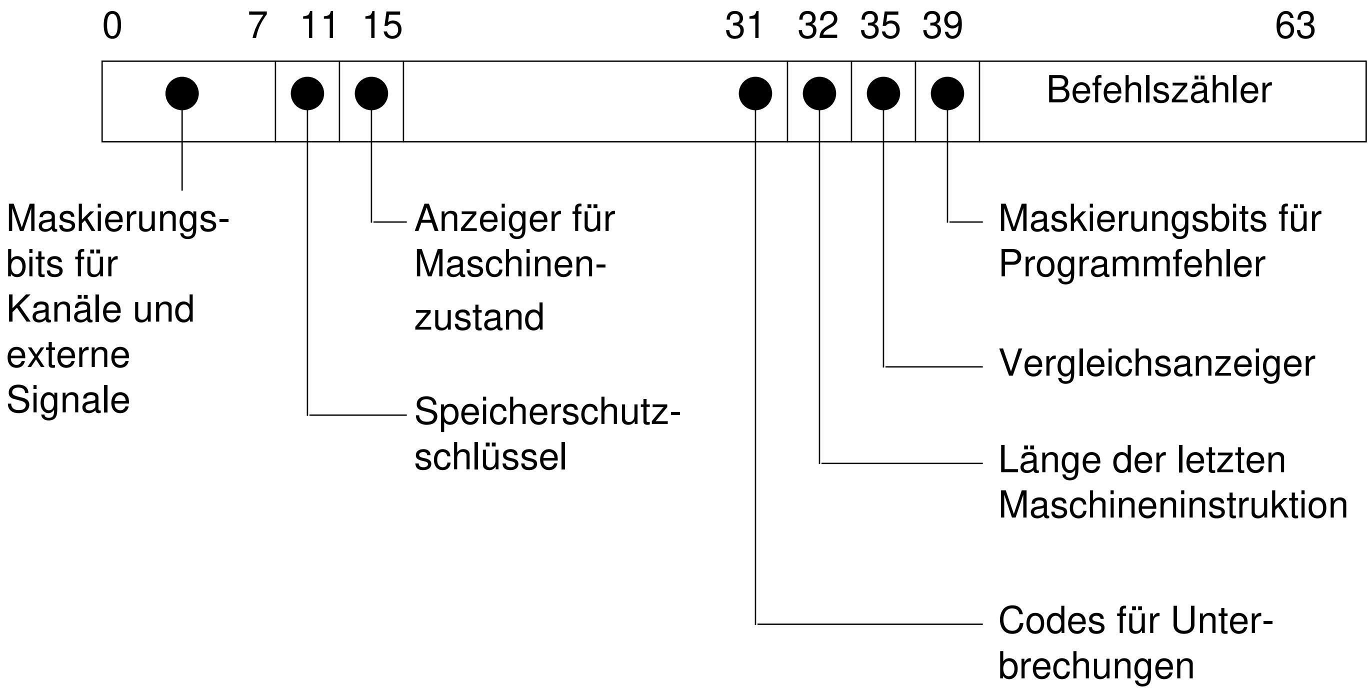
Stapel bei geschachtelten Methodenaufrufen

```
main()
{
        int a, b, c;
        register r;

        a = 10;
        b = 20;
        r = 40;

        f1(a, b);

        return(a);
}
```

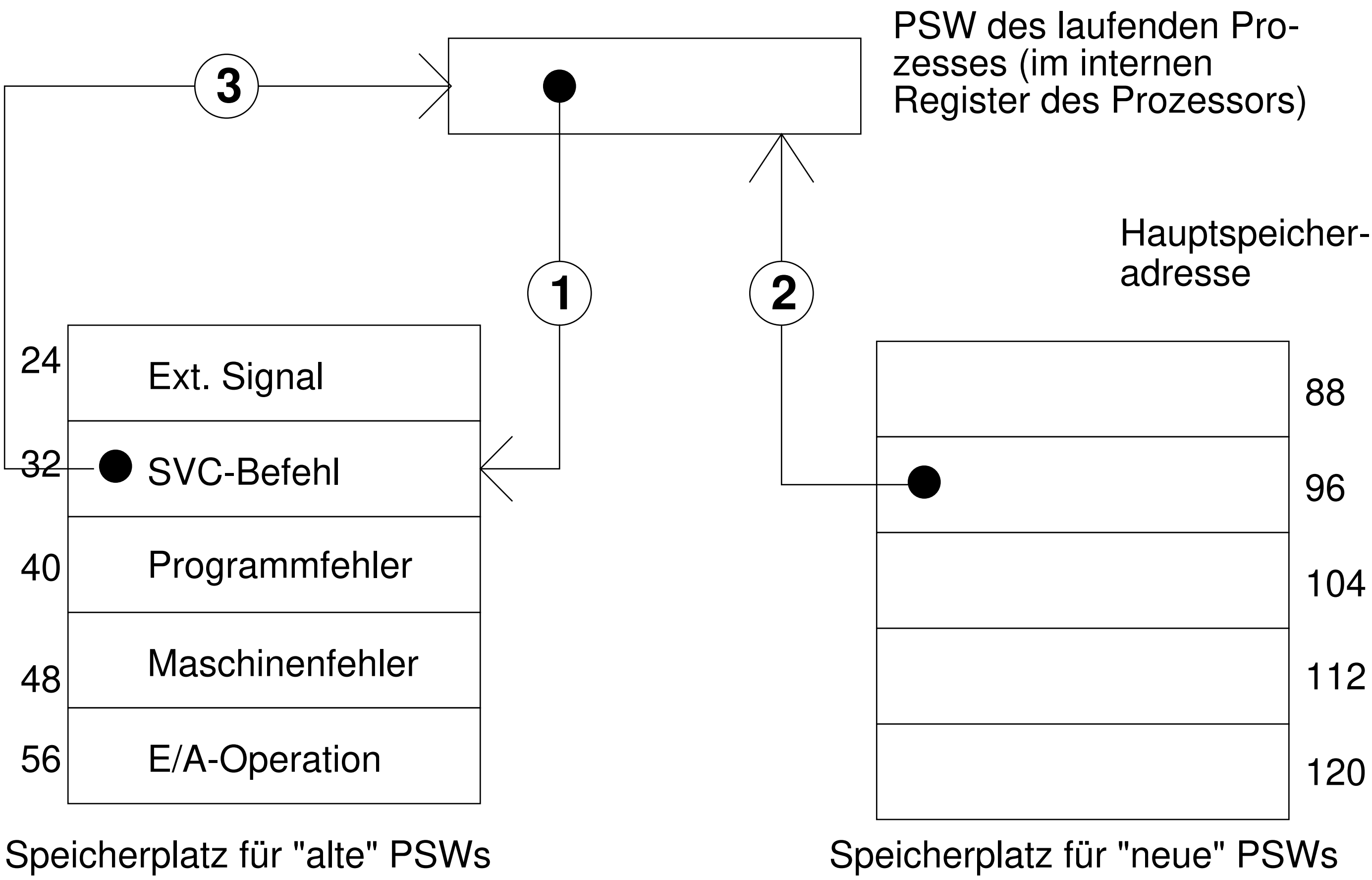
```
f1(x,y)
{
        int i;
        register r;
        register s;

        x++;

        i = f2(x);

        return(i);
}
```

```
f2(z)
{
        int m;
        register r;

        r = z;
        m = 100;

        return(z+1);
}
```





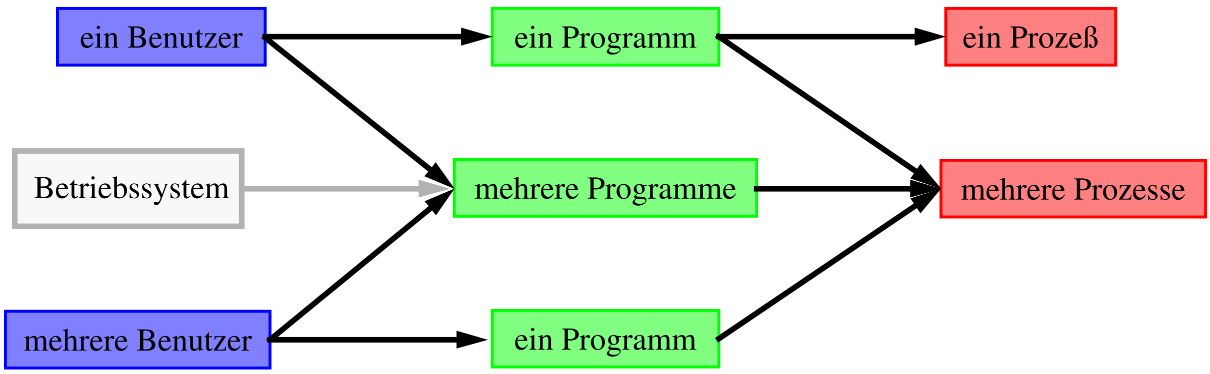
Programmstatuswort des Systems IBM 360



Austausch der PSWs im System IBM 360

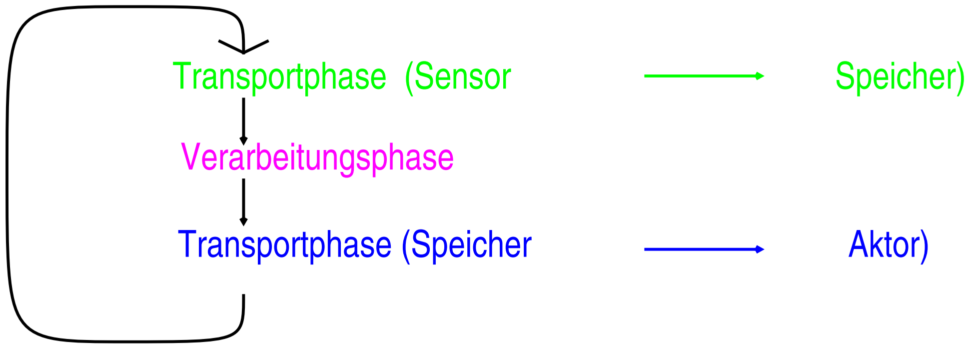
3.2 Prozeßzuordnung

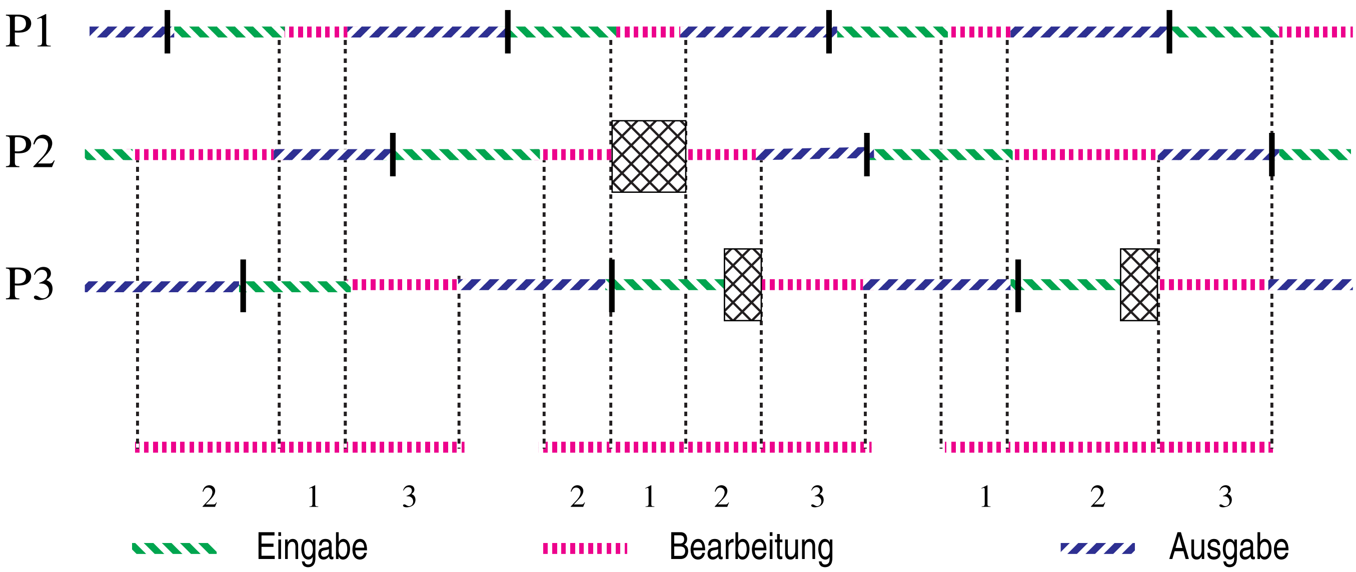
- Ein- und Mehrprozeßbetrieb:



- Ein Vorteil: Bessere Gesamtnutzung des Systems:

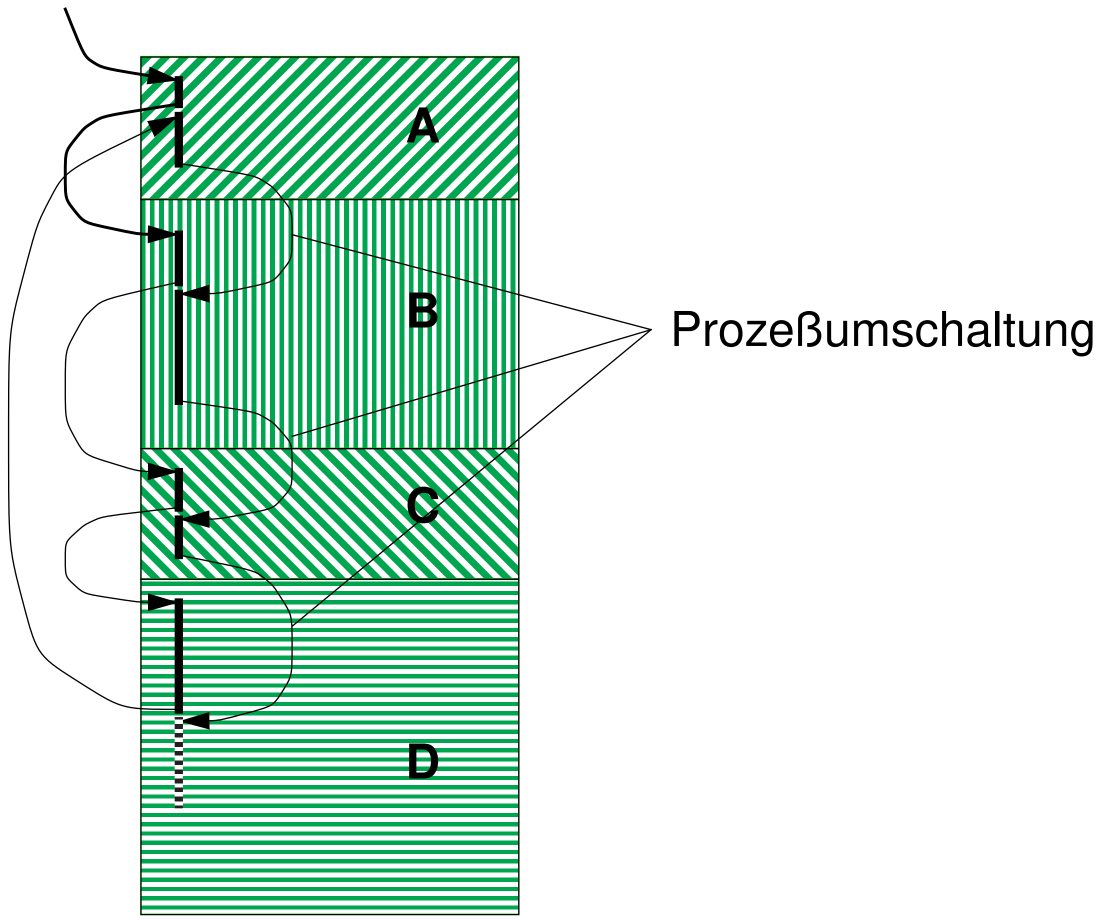
Typischer Ablauf eines Benutzerauftrags



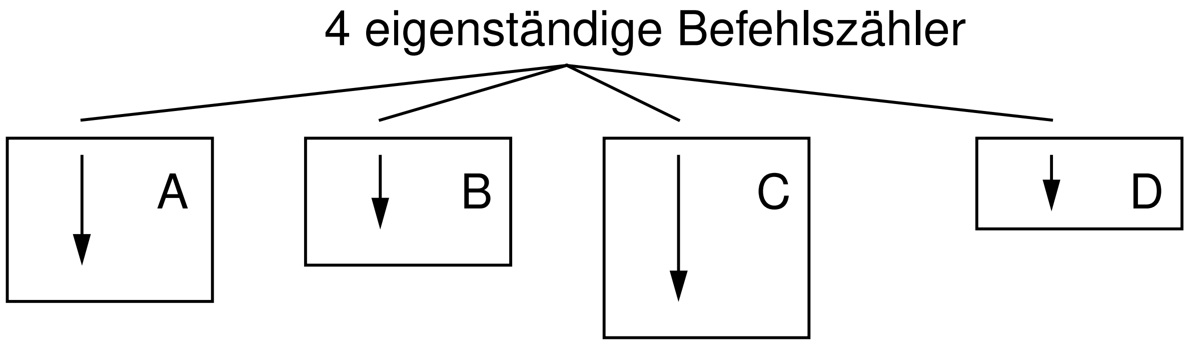


- Pseudoparallelität
  - Prozeßkontrollblöcke
  - Prozeßumschalter
- Prozeßerzeugung
  - statisch - dynamisch
  - Prozeßhierarchie

Mehrprogrammbetrieb bei vier Prozessen (Aktivitätsträgern)



Konzeptionelles Modell



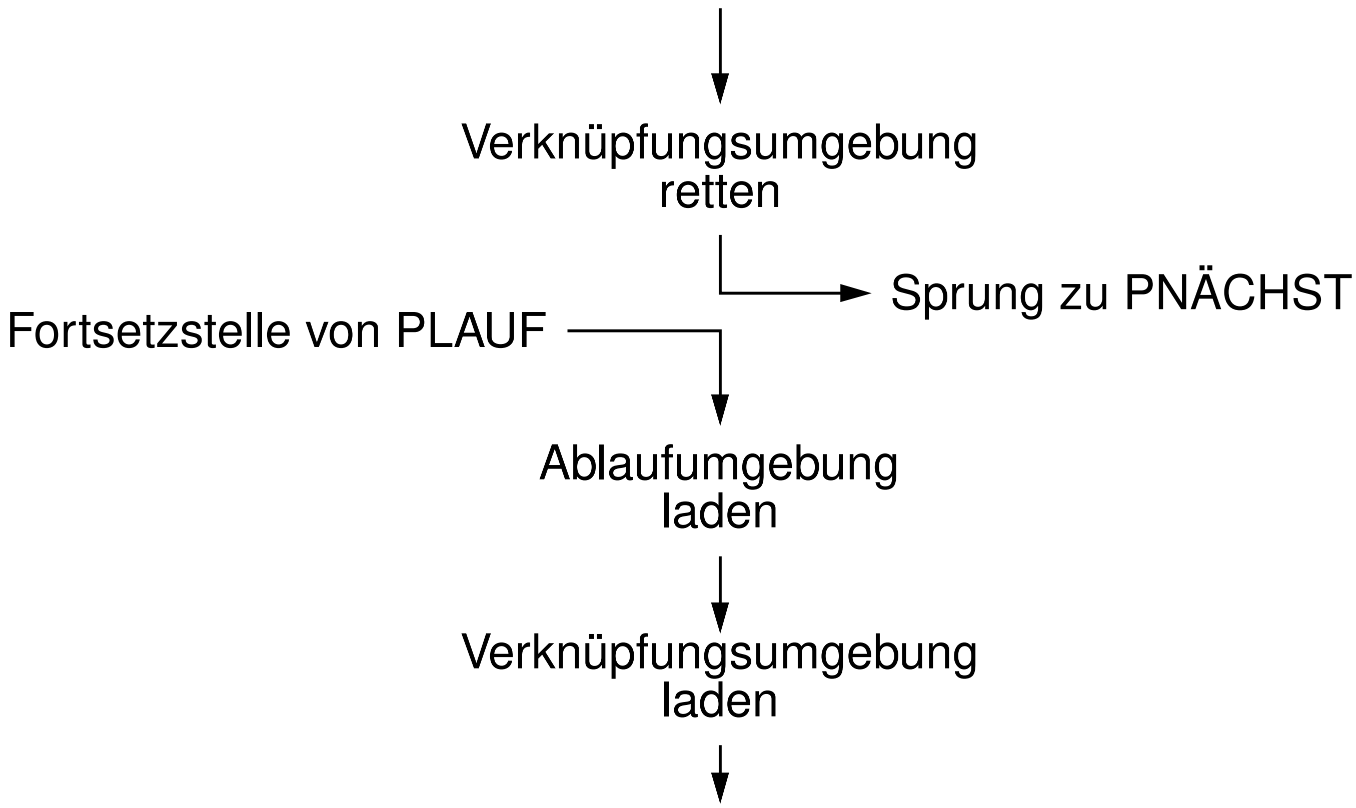
- Verknüpfungsumgebung

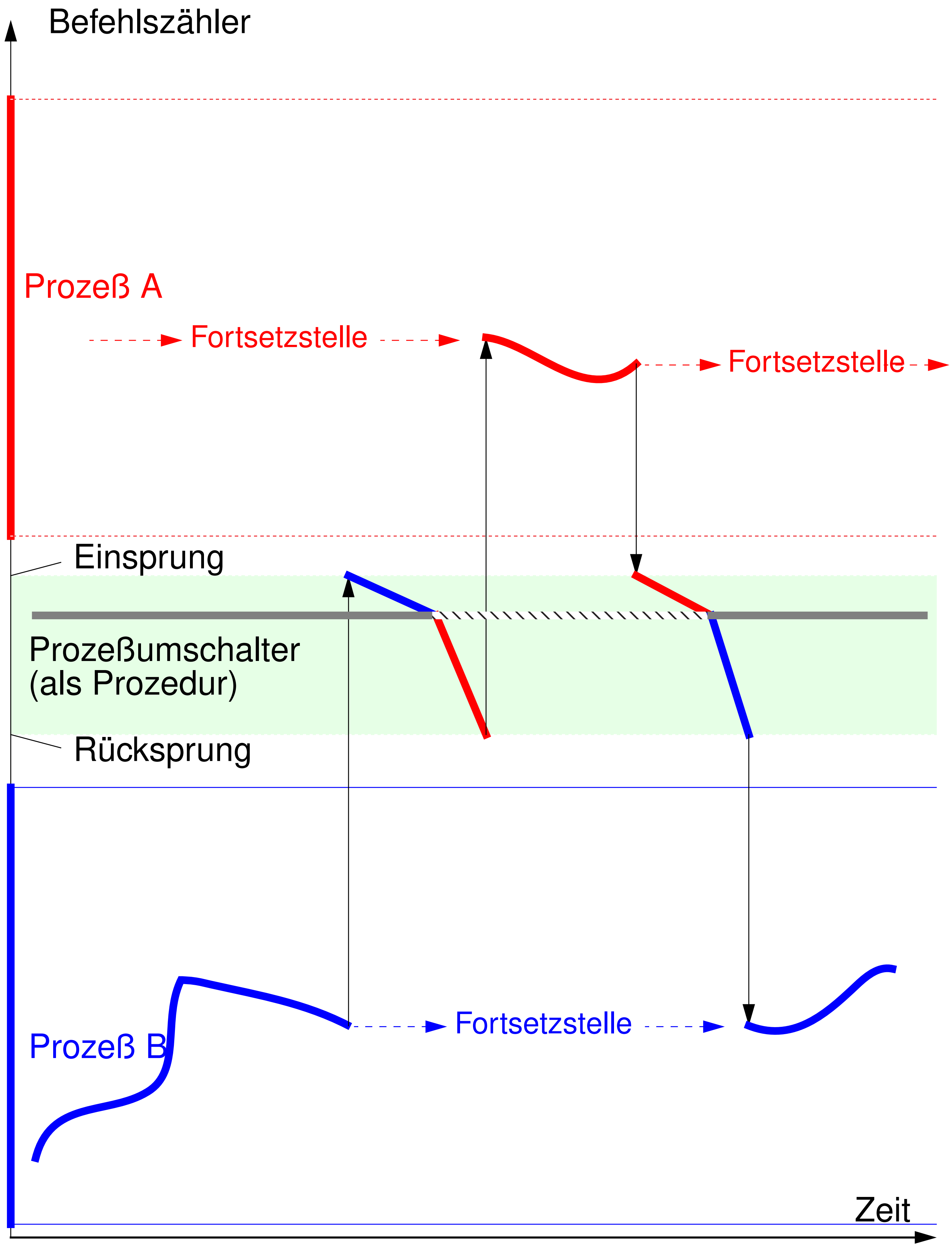
  Größen, die aus der eigentlichen Informationsverknüpfung resultieren, wie etwa Inhalte von Rechenregistern, Indexregistern, Ereignismerkern u. ä.

- Ablaufumgebung

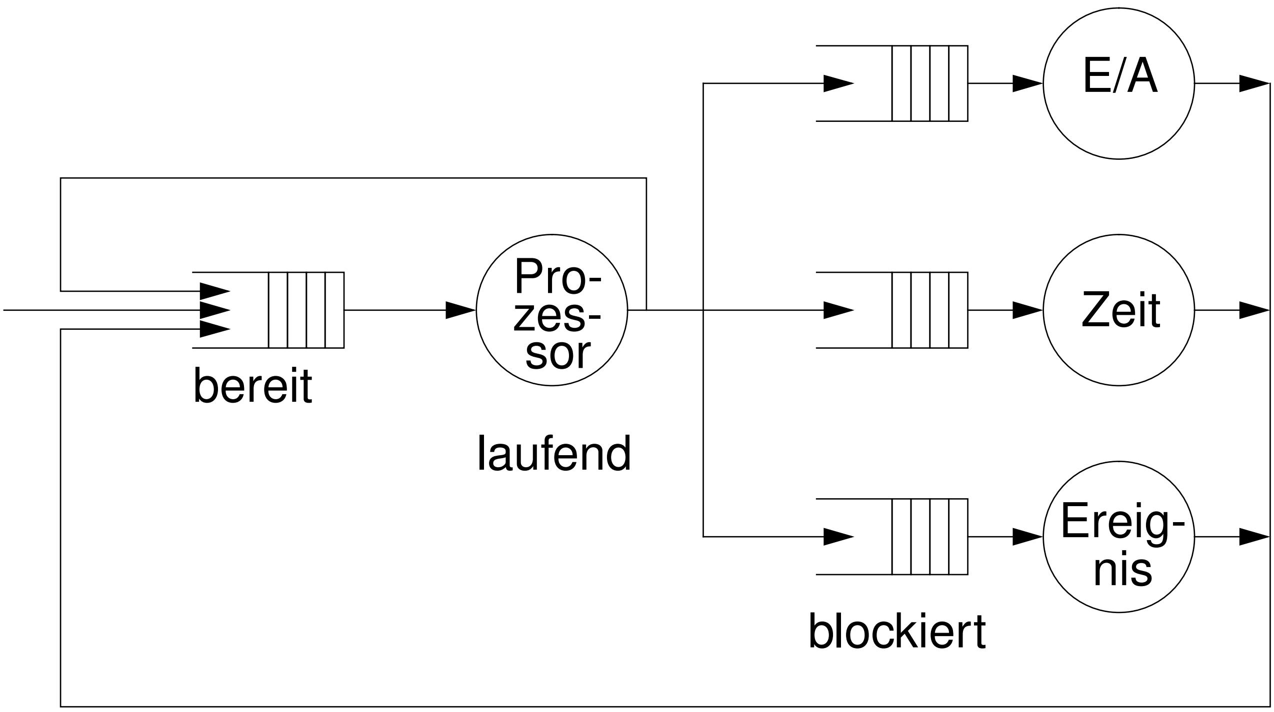
  Alle Größen, die zum eigentlichen Ablauf notwendig sind, etwa Adreßregister, Seiten- oder Segmenttabellen, Unterbrechungsmasken, Schutz- und Zugriffskontrollangaben u. ä.

- Prozeßumschaltung:




Befehlszähler
Prozeß A
Fortsetzstelle
Fortsetzstelle
Einsprung
Prozeßumschalter
(als Prozedur)
Rücksprung
Fortsetzstelle
Prozeß B
Zeit

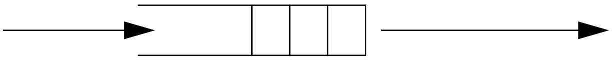
3.3 Auswahlstrategien (Einplanungsstrategien, Zuordnungsstrategien, *scheduling*)



- ❑ Klassifikation
  - ◆ nicht verdrängend verdrängend
  - ◆ statisch dynamisch
- ❑ Gütekriterien für Auswahlstrategien:
  - ◆ CPU-Auslastung
  - ◆ Durchsatz
  - ◆ Mittlere Verweilzeit
  - ◆ Mittlere Wartezeit
  - ◆ Mittlere Antwortzeit

- Auswahl nach Ankunftsreihenfolge
  (*First Come First Served , FCFS*)

  ..., i+2, i+1, i, ...

  ..., i+2, i+1, i, ...

- Auswahl (verdrängend) nach kürzester Restrechenzeit
  (*Preemptive Shortest Job First, PSJF*)

  minimiert die mittlere Verweilzeit

- Auswahl nach Prioritäten

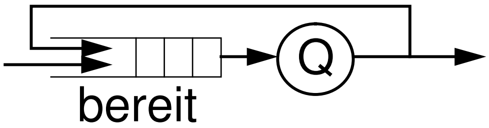
  statisch - dynamisch; verdrängend - nicht verdrängend

  Wird vorwiegend in Echtzeitsystemen benutzt, wegen der besseren Vorhersagbarkeit von Reaktionszeiten.

- Auswahl (verdrängend) nach steigenden Zielzeiten
  (*Earliest Deadline First, EDF*)

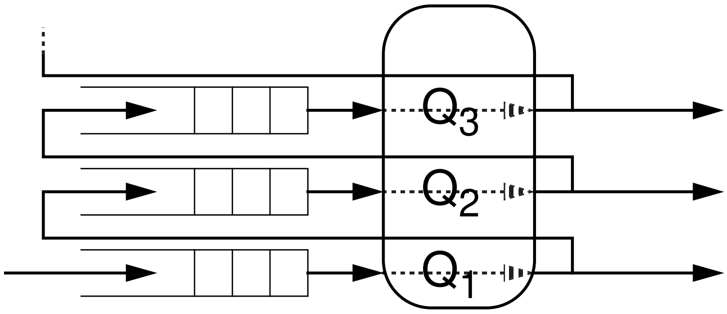
  minimiert die maximale Verspätung

- Auswahl reihum mit festem Zeitquant Q
  (*Round Robin, RR*)



  Jeder Prozeß sieht im Mittel eine auf ein n-tel ihrer Leistung verminderte Bedienung, wenn im Mittel n Prozesse zu bearbeiten sind.

- Auswahl nach geringster verbrauchter Rechenzeit
  (*Multilevel Feedback, MLFB*)



❑ Timesharing Scheduling unter SunOS 5.5 (Solaris 2.5)

ts_quantum	ts_tqexp	ts_slpret	ts_maxwait	ts_lwait	Level
200	0	50	0	50	0
200	0	50	0	50	1
200	0	50	0	50	2
200	0	50	0	50	3
200	0	50	0	50	4
200	0	50	0	50	5
200	0	50	0	50	6
200	0	50	0	50	7
200	0	50	0	50	8
200	0	50	0	50	9
160	0	51	0	51	10
160	1	51	0	51	11
160	2	51	0	51	12
160	3	51	0	51	13
160	4	51	0	51	14
160	5	51	0	51	15
		•••			
80	24	53	0	53	34
80	25	54	0	54	35
80	26	54	0	54	36
80	27	54	0	54	37
80	28	54	0	54	38
80	29	54	0	54	39
40	30	55	0	55	40
40	31	55	0	55	41
40	32	55	0	55	42
40	33	55	0	55	43
40	34	55	0	55	44
40	35	56	0	56	45
40	36	57	0	57	46
40	37	58	0	58	47
40	38	58	0	58	48
40	39	58	0	59	49
40	40	58	0	59	50
40	41	58	0	59	51
40	42	58	0	59	52
40	43	58	0	59	53
40	44	58	0	59	54
40	45	58	0	59	55
40	46	58	0	59	56
40	47	58	0	59	57
40	48	58	0	59	58
20	49	59	32000	59	59

3.4 Prozeßmanipulation

- Erzeugung von Prozessen
  - Erzeugender Prozeß, Vaterprozeß (*parent process*) ruft Systemaufruf auf:
    - Unix: fork()
    - NT: spawn
  - Erzeugter Prozeß, Kindprozess (*child process*)
- Tilgen von Prozessen
  - Selbstterminierung
  - Fremdterminierung
- Erzeugen und Tilgen von Fäden
- Suspendierung, Fortsetzung
- Beeinflussung von Attributen (z. B. für Scheduling)
- Festlegung von Zugriffsrechten zu den Betriebsmitteln (z. B. Dateien)

3.5 Interagierende Prozesse/Aktivitätsträger

- Gründe für die Benutzung kooperierender Aktivitätsträger
  - Gleichzeitige Nutzung von Informationen durch mehrere Anwendungen
  - Verkürzung der Bearbeitungszeit bei Verwendung von Parallelrechnern

- Interaktion
  - Drei Grundformen
    - Kommunikation

      Mechanismen, die Aktivitätsträgern ermöglichen, daß Daten aus einem ihm zugänglichen Speicherbereich, in einen Speicherbereich hineinzuverknüpfen, zu dem andere Aktivitätsträger Zugang haben.

    - Kooperation

      Mechanismen, die mehreren Fäden ermöglichen im gleichen oder in überlappenden Adreßräumen zu arbeiten und diese Tatsache zur gegenseitigen Beeinflussung zu nutzen.

    - Koordination

      Mechanismen, die einem Aktivitätsträger bei Erreichen einer bestimmten Stelle seines Ablaufs erlauben, auf die Fortführung oder Verzögerung anderer Aktivitätsträger Einfluß zu nehmen.

- Kommunikation
  - Daten sind aus einem Speicherbereich in einen anderen zu transportieren.
  - Transporteinheiten werden als **Nachrichten** bezeichnet.
  - Der Transport obliegt einem **Kommunikationsobjekt** (*pipe, queue, port, link, mailbox*), das Operationen zum Senden und Empfangen bereitstellt.

    Bei der Konkretisierung dieser Vorstellungen findet sich eine große Variationsbreite.

    Wesentlichste Klassifikationsmerkmale sind:

    - Art der Koordination, z. B. ob der sendende Faden bis zur Ankunft der Nachricht blockiert wird oder nicht.
    - Art der Adressierung, z. B. ob als Empfänger Adreßräume, Fäden oder Geräte dienen, oder ob Dienste wie “Drucken” als Adressat dienen können.
    - Art der Verbindung, z. B. ob Nachrichten nur an einen oder gleichzeitig an mehrere Empfänger gesendet werden können.
    - Art der Pufferung und Flußkontrolle, z. B. ob bei zu dichter Sendefolge Nachrichten verloren gehen oder der Sender vorübergehend blockiert wird.

- Häufig Kommunikation in Art eines Methodenaufrufs:

  Mit einer ersten Nachricht wird - evtl. über Adreßraum- und Rechnergrenzen hinweg - ein Auftrag erteilt und mit einer Rücknachricht der Abschluß mitgeteilt sowie gegebenenfalls das Ergebnis übermittelt.

  Wegen der Ähnlichkeit mit üblichen Prozeduraufrufen innerhalb eines Adreßraums wird diese Interaktionsform **Fernaufruf** (*remote procedure call*, **RPC**) genannt.

  Fernaufrufe können auch so realisiert werden, daß sie Adreßräume auf unterschiedlichen Rechnern verbinden und sind deshalb im Zusammenhang mit verteilten Systemen von besonderem Interesse.

3.6 Kooperation und Koordination

- Beispiel: Beobachter und Protokollierer

  Mittels Induktionsschleife werden Fahrzeuge gezählt. Protokollierung der Anzahl vorbeigekommener Fahrzeuge im Abstand von 10 Minuten.

```
x = 0;
```

Beobachter

```
on indication
    x++;
```

Protokollierer

```
every 10 minutes
    y = x;
    x = 0;
    cout << ... << y << ... ;
```

Gemeinsame Daten

```
shared_int counter = 0;
```

Beobachter

```
int observer()
{ char c;
  while (TRUE) {
    cin.get(c);   // Simulation eines Impulses
                  // der Induktionsschleife durch
                  // drücken einer Terminal-Taste
    counter++;
  }
}
```

Protokollierer

```
int recorder()
{ int last_count;
  cout << "\n\n";
  while (TRUE) {
    sleep(5);   // 5 Sekunden pausieren
    last_count = counter;
    sleep(1);   // 1 Sekunde pausieren
    counter = 0;
    cout << "Anzahl beobachteter Ereignisse: "
         << last_count << "\n" << flush;
  }
}
```

Beobachtetes Ergebnis:

```
Anzahl beobachteter Ereignisse: 0
Anzahl beobachteter Ereignisse: 0
123456789
Anzahl beobachteter Ereignisse: 10
Anzahl beobachteter Ereignisse: 0
Anzahl beobachteter Ereignisse: 0
123456789
Anzahl beobachteter Ereignisse: 10
123456789
Anzahl beobachteter Ereignisse: 1
Anzahl beobachteter Ereignisse: 0
Anzahl beobachteter Ereignisse: 0
Anzahl beobachteter Ereignisse: 0
123456789
Anzahl beobachteter Ereignisse: 0
Anzahl beobachteter Ereignisse: 0
Anzahl beobachteter Ereignisse: 0
123456789
Anzahl beobachteter Ereignisse: 10
Anzahl beobachteter Ereignisse: 0
Anzahl beobachteter Ereignisse: 0
123456789
Anzahl beobachteter Ereignisse: 10
Anzahl beobachteter Ereignisse: 0
123456789
Anzahl beobachteter Ereignisse: 5
Anzahl beobachteter Ereignisse: 0
Anzahl beobachteter Ereignisse: 0
123456789
Anzahl beobachteter Ereignisse: 4
Anzahl beobachteter Ereignisse: 9
Anzahl beobachteter Ereignisse: 0
Anzahl beobachteter Ereignisse: 0
```

Ursache: Konkurrierende Ausführung der Anweisungen

```
counter++;
```

und

```
counter = 0;
```

T_0: Beobachter	$Register_B$ = counter	($Register_B$ = 8)
T_1: Beobachter	$Register_B$ = $Register_B$ + 1	($Register_B$ = 9)
T_2: Protokollierer	$Register_P$ = 0	($Register_P$ = 0)
T_3: Protokollierer	counter = $Register_P$	(counter = 0)
T_4: Beobachter	counter = $Register_B$	(counter = 9 !!)

d. h. das Rücksetzen des Zählers geht verloren, also werden Impulse mehrfach gezählt.

T_0: Protokollierer	$Register_P$ = counter	($Register_P$ = 6)
T_1: Protokollierer	last_counter = $Register_P$	(last_counter = 6)
T_2: Protokollierer	$Register_P$ = 0	($Register_P$ = 0)
T_3: Beobachter	$Register_B$ = counter	($Register_B$ = 6)
T_4: Beobachter	$Register_B$ = $Register_B$+1	($Register_B$ = 7)
T_5: Beobachter	counter = $Register_B$	(counter = 7)
T_6: Protokollierer	counter = $Register_P$	(counter = 0)

d. h. ein Impuls geht bei der Zählung verloren.

Abhilfe: Manipulation gemeinsamer Daten nur unter gegenseitigem Ausschluß.

- Abschnitte, in denen gemeinsame Daten bearbeitet werden heißen kritische Abschnitte.
- Die Anweisungen kritischer Abschnitte müssen atomar ausgeführt werden, d. h. ihre Abarbeitung erscheint für andere Aktivitätsträger als zeitlich unteilbar.

Problem: Wie kann der gegenseitige Ausschluß erzielt werden?

Nachfolgende Problemanalyse folgt den Arbeiten von Dijkstra.

1. Lösungsversuch

- Initialisierung

```
shared_int turn = 1;
```

- Aktivitätsträger 0

```
int thread_0()
{ shared_int turn;
  while (TRUE) {
    while (turn == 1) ;
    critical_section_0();
    turn = 1;
    remainder_0();
  }
}
```

- Aktivitätsträger 1

```
int thread_1()
{ shared_int turn;
  while (TRUE) {
    while (turn == 0) ;
    critical_section_1();
    turn = 0;
    remainder_1();
  }
}
```

Problem: Läßt nur alternierende Abarbeitung zu
(unvollständige Implementation)

2. Lösungsversuch

- Initialisierung

```
shared_int thread_0_in_crit = FALSE,
           thread_1_in_crit = FALSE;
```

- Aktivitätsträger 0

```
int thread_0()
{ shared_int thread_0_in_crit, thread_1_in_crit;
  while (TRUE) {
    while (thread_1_in_crit) ;    // active waiting
    thread_0_in_crit = TRUE;
    critical_section_0();
    thread_0_in_crit = FALSE;
    remainder_0();
  }
}
```

- Aktivitätsträger 1

```
int thread_1()
{ shared_int thread_0_in_crit, thread_1_in_crit;
  while (TRUE) {
    while (thread_0_in_crit) ;
    thread_1_in_crit = TRUE;
    critical_section_1();
    thread_1_in_crit = FALSE;
    remainder_1();
  }
}
```

Problem: Gegenseitiger Ausschluß wird nicht mit Sicherheit erreicht (Implementation unsicher);
aktives Warten (Implementation nicht prompt)

3. Versuch

- Initialisierung

```
shared_int thread_0_in_crit = FALSE,
           thread_1_in_crit = FALSE;
```

- Aktivitätsträger 0

```
int thread_0()
{ shared_int thread_0_in_crit, thread_1_in_crit;
  while (TRUE) {
    while (!thread_0_in_crit) {
      thread_0_in_crit = TRUE;
      if (thread_1_in_crit)
        { thread_0_in_crit = FALSE; }
    }
    critical_section_0();
    thread_0_in_crit = FALSE;
    remainder_0();
  }
}
```

- Aktivitätsträger 1

```
int thread_1()
{ shared_int thread_0_in_crit, thread_1_in_crit;
  while (TRUE) {
    while (!thread_1_in_crit) {
      thread_1_in_crit = TRUE;
      if (thread_0_in_crit)
        { thread_1_in_crit = FALSE; }
    }
    critical_section_1();
    thread_1_in_crit = FALSE;
    remainder_1();
  }
}
```

Problem: Überhöflich (Bitte nach Ihnen!, starvation)

Algorithmus von Peterson (1981)

- Initialisierung

```
shared_int turn = 0,
           interested[2] = {FALSE, FALSE};
```

- Aktivitätsträger 0

```
thread_0()
{
  while (TRUE) {
    interested[0] = TRUE;
    turn = 0;
    while (turn == 0 && interested[1] == TRUE) ;
    critical_section_0();
    interested[0] = FALSE;
    remainder_0();
  }
}
```

- Aktivitätsträger 1

```
thread_1()
{
  while (TRUE) {
    interested[1] = TRUE;
    turn = 1;
    while (turn == 1 && interested[0] == TRUE) ;
    critical_section_1();
    interested[1] = FALSE;
    remainder_1();
  }
}
```

Problem: Aktives Warten

- Aktivitätsträger 0

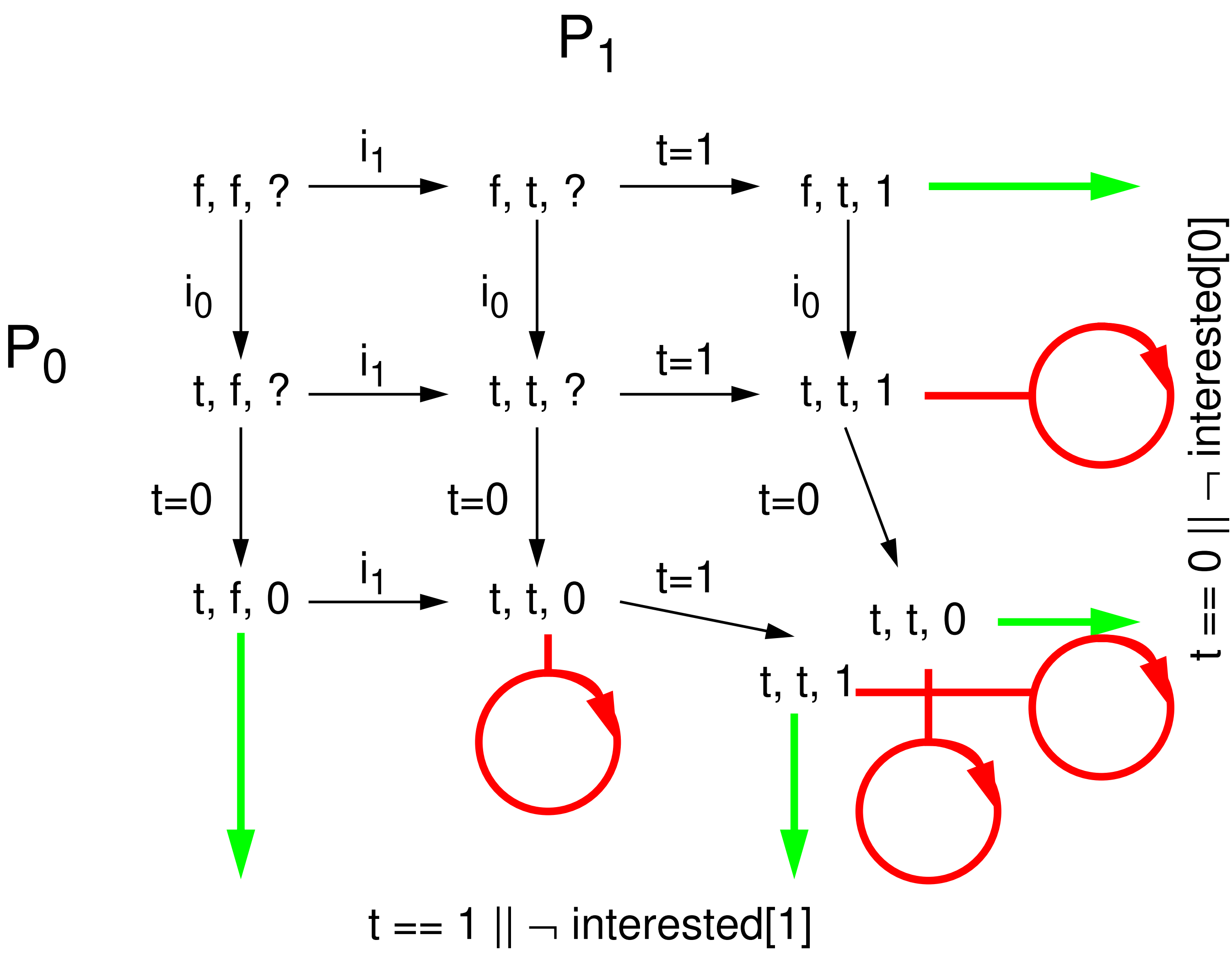
```
thread_0()
{
  while (TRUE) {
    interested[0] = TRUE;
    turn = 0;
    while (turn == 0 && interested[1] == TRUE) ;
    cout << "P 0 ist im krit. Absch.; "
         << "interested = " << interested[0] << ", "
         << interested[1] << "\n" << flush;
    critical_section_0();
    interested[0] = FALSE;
    cout << "P 0 ist ausserhalb krit. Absch.; "
         << "interested = " << interested[0] << ", "
         << interested[1] << "\n" << flush;
    remainder_0();
  }
}
```

- Aktivitätsträger 1

```
thread_1()
{
  while (TRUE) {
    interested[1] = TRUE;
    turn = 1;
    while (turn == 1 && interested[0] == TRUE) ;
    cout << "P 1 ist im krit. Absch.; "
         << "interested = " << interested[0] << ", "
         << interested[1] << "\n" << flush;
    interested[1] = FALSE;
    critical_section_1();
    cout << "P 1 ist ausserhalb krit. Absch.; "
         << "interested = " << interested[0] << ", "
         << interested[1] << "\n" << flush;
    remainder_1();
  }
}
```

```
P 1 ist ausserhalb krit. Absch.; interested = 1, 0
P 0 ist ausserhalb krit. Absch.; interested = 0, 1
P 1 ist im krit. Absch.; interested = 0, 1
P 1 ist ausserhalb krit. Absch.; interested = 1, 0
P 0 ist im krit. Absch.; interested = 1, 1
P 1 ist im krit. Absch.; interested = 0, 1
P 0 ist ausserhalb krit. Absch.; interested = 0, 1
P 1 ist ausserhalb krit. Absch.; interested = 1, 0
P 0 ist im krit. Absch.; interested = 1, 0
P 0 ist ausserhalb krit. Absch.; interested = 0, 1
P 1 ist im krit. Absch.; interested = 1, 1
P 0 ist im krit. Absch.; interested = 1, 0
P 1 ist ausserhalb krit. Absch.; interested = 1, 0
P 0 ist ausserhalb krit. Absch.; interested = 0, 1
P 1 ist im krit. Absch.; interested = 1, 1
P 0 ist im krit. Absch.; interested = 1, 0
P 1 ist ausserhalb krit. Absch.; interested = 1, 0
P 0 ist ausserhalb krit. Absch.; interested = 0, 1
P 1 ist im krit. Absch.; interested = 0, 1
P 1 ist ausserhalb krit. Absch.; interested = 1, 0
P 0 ist im krit. Absch.; interested = 1, 0
P 0 ist ausserhalb krit. Absch.; interested = 0, 1
P 1 ist im krit. Absch.; interested = 1, 1
P 0 ist im krit. Absch.; interested = 1, 0
P 1 ist ausserhalb krit. Absch.; interested = 1, 0
```

Nachweis des gegenseitigen Ausschlusses



Lösung durch Hardwareunterstützung: Swap-Befehl

- Wirkung des swap(exchange)-Befehls

```
void swap(register &r, shared int &m)
{ int temp;
  begin_atomic();
    temp = m; m = r; r = temp;
  end_atomic();
};
```

- Initialisierung

```
shared_int flag = 0;
```

- Aktivitätsträger 0

```
thread_0()
{
  register r;
  while (TRUE) {
    while (r = 1, swap(r, flag), r == 1) ;
    critical_section_0();
    flag = 0;
    remainder_0();
  }
}
```

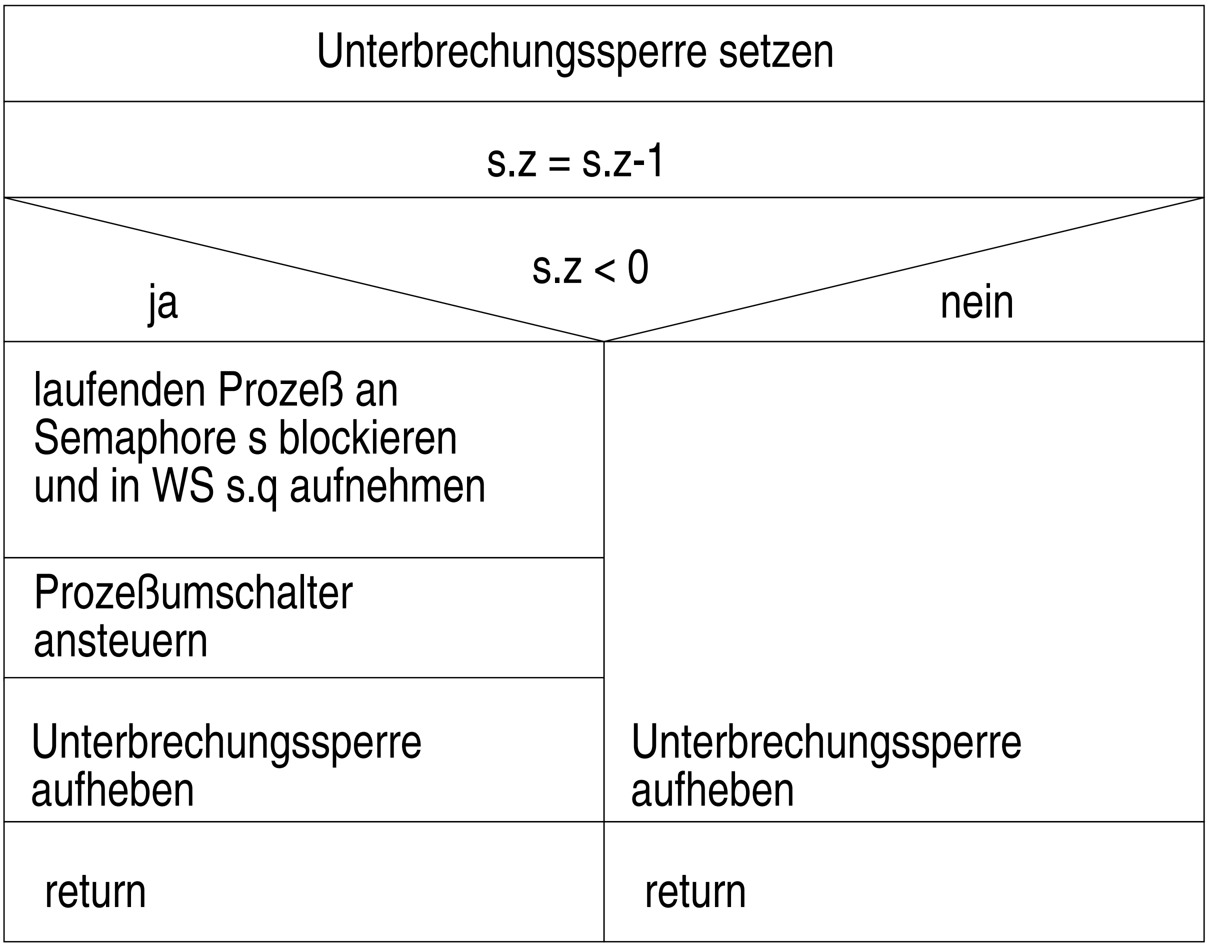
- Aktivitätsträger 1

```
thread_1()
{
  register r;
  while (TRUE) {
    while (r = 1, swap(r, flag), r == 1) ;
    critical_section_1();
    flag = 0;
    remainder_1();
  }
}
```

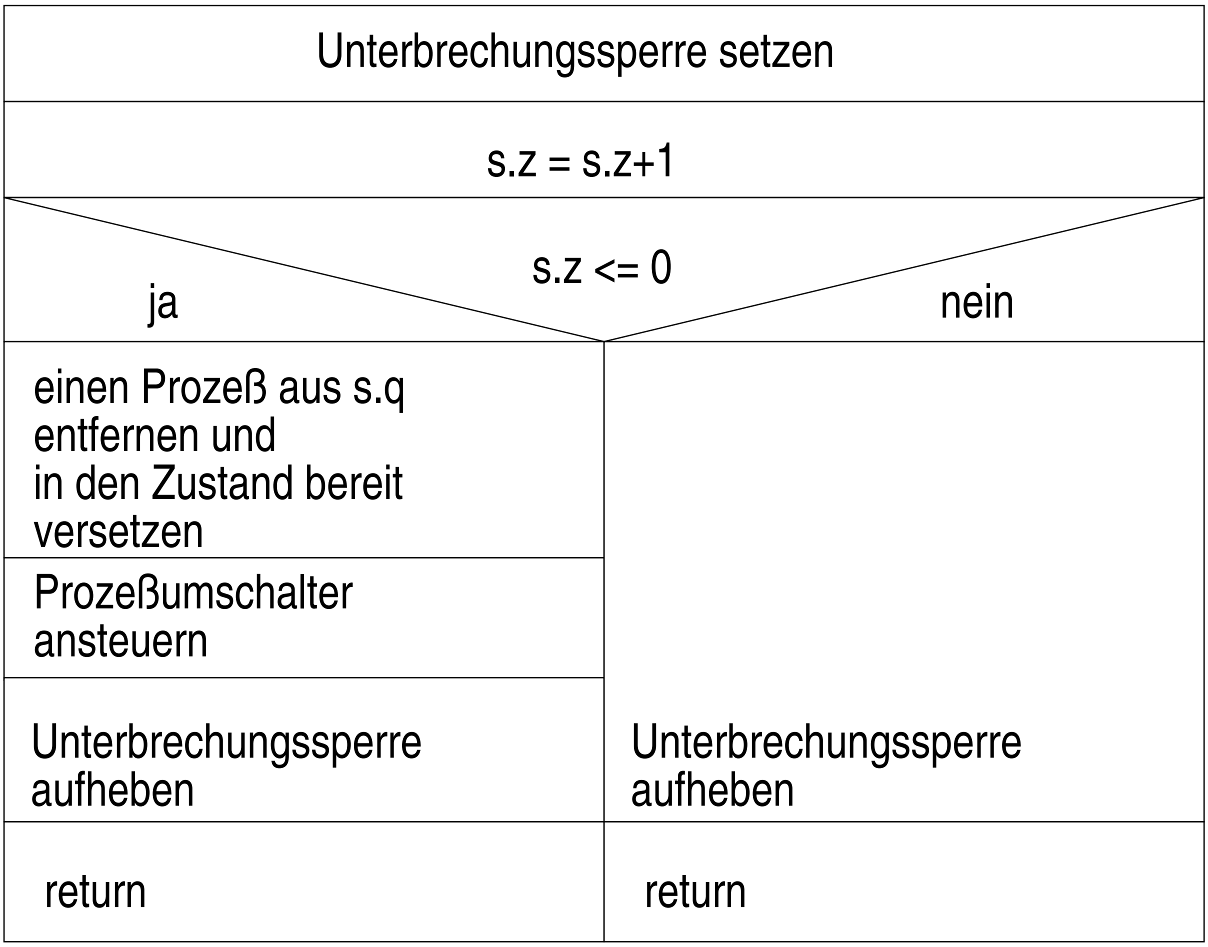
Lösung für Monoprozessoren: Semaphore

```
typedef  ..........  queue;
class semaphore
  { int z;
    queue q;
  public:
     semaphore() { };
     semaphore(int z0) {z = z0; ... };
     void P();
     void V()
  } s;
```

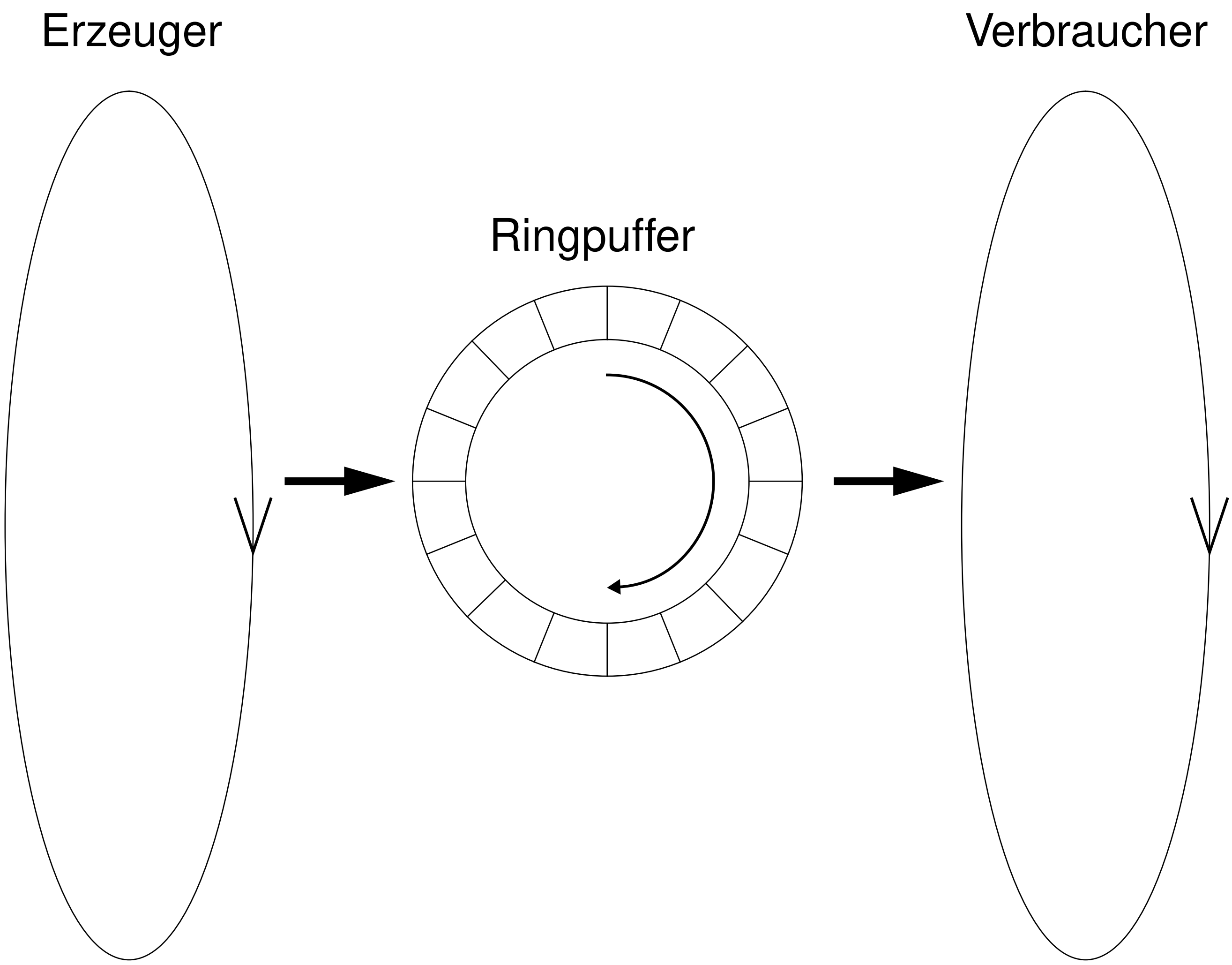
s.P()   (Monoprozessor)



s.V() (Monoprozessor)



Erzeuger-Verbraucher mit endlichem Puffer



- Initialisierung

```
semaphore mutex(1), empty(N), full(0);
shared_int buffer[N];
```

- Aktivitätsträger 0 (Erzeuger, Producer)

```
int producer()
{ int value, slot = 0;
  while (TRUE) {
    cout << "Producer: " << flush; cin >> value;
    empty.P();
    mutex.P();
      buffer[slot] = value;
    mutex.V();
    full.V();
    if (++slot == N) slot = 0;
  }
}
```

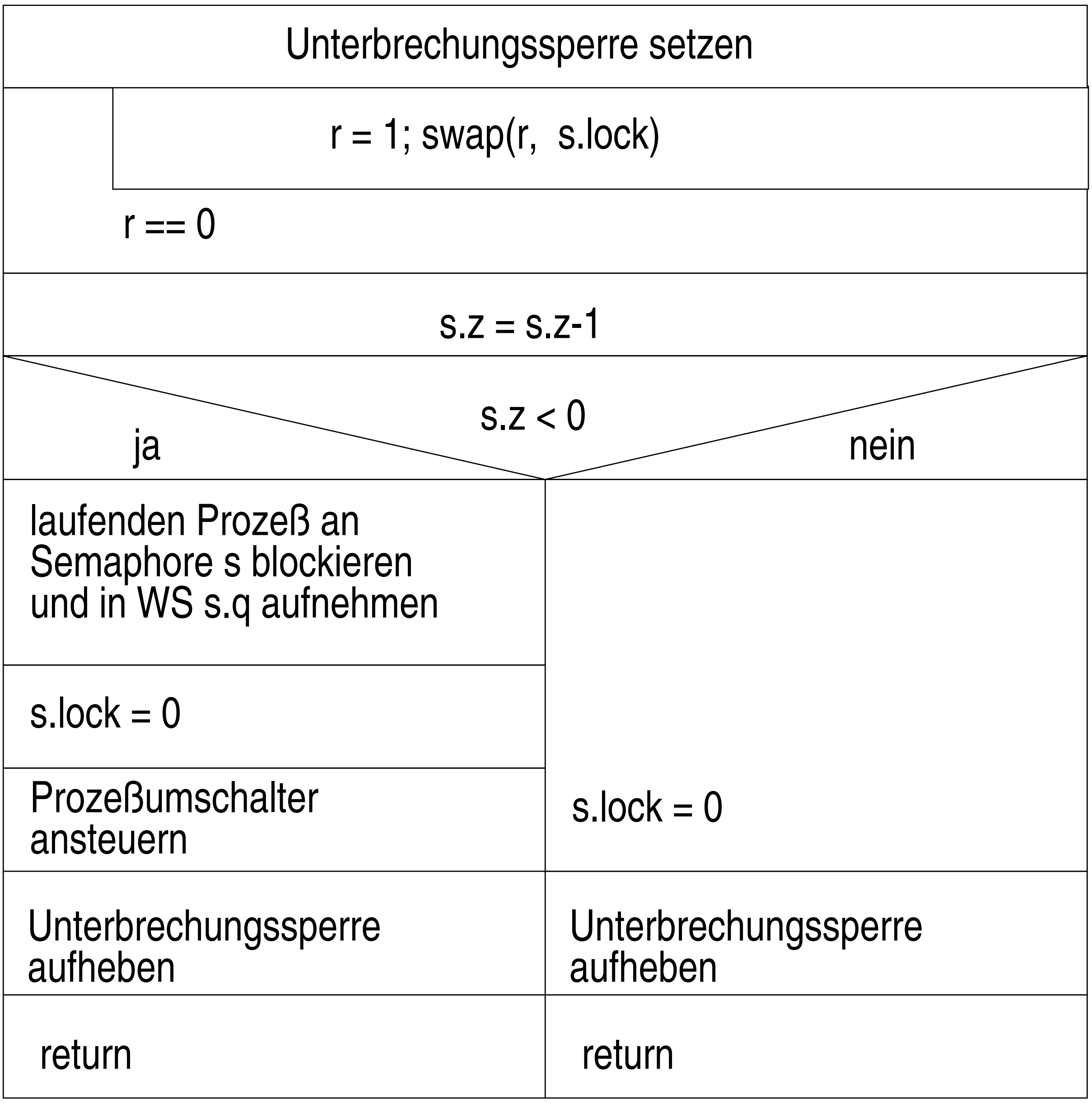
- Aktivitätsträger 1 (Verbraucher, Consumer)

```
int consumer()
{ int value, slot = 0;
  while (TRUE) {
    full.P();
    mutex.P();
      value = buffer[slot];
    mutex.V();
    empty.V();
    if (++slot == N) slot = 0;
    cout << "Consumer: " << value
         << "\n" << flush;
  }
}
```

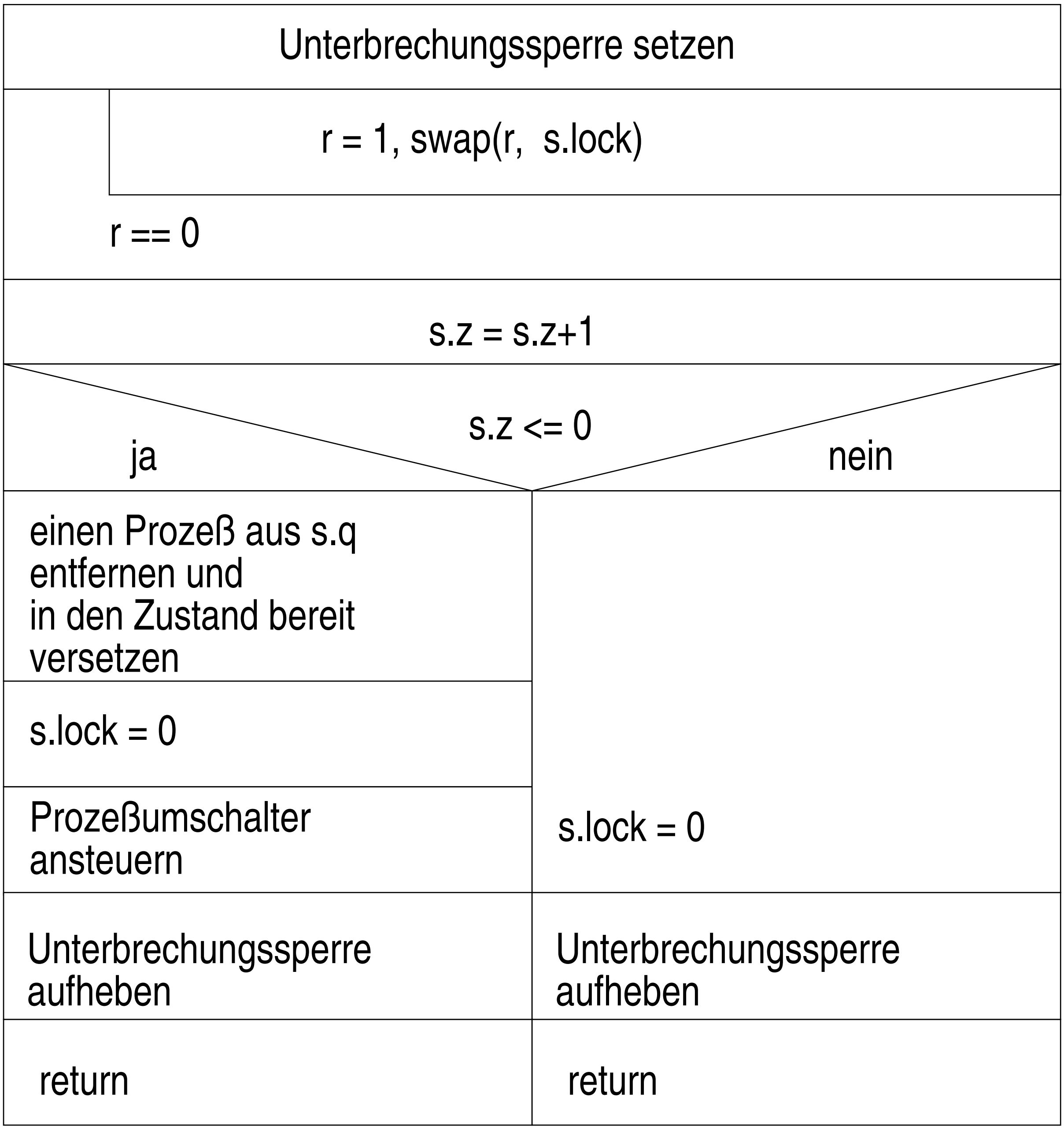
Semaphore für Multiprozessoren

```
typedef  .........  queue;
class semaphore
  { int z;
    queue q;
    int lock;
  public:
     void P();
     void V()
  } s;
```

s.P() (Multiprozessor)



s.V() (Multiprozessor)



3.7 Klassische Koordinierungsprobleme

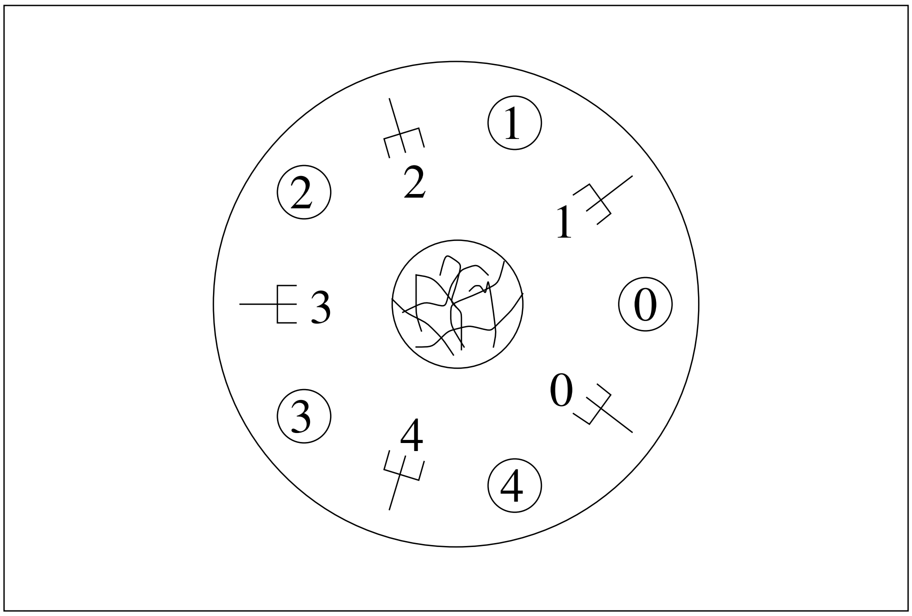
3.7.1 Das Problem der speisenden Philosophen (Dijkstra 1971)

- Die Aufgabenstellung:

  "The live of a philosopher consists of an alternation of thinking and eating:

```
CYCLE   BEGIN   think;
                eat
        END.
```

Five philosophers, numbered from 0 through 4 are living in a house where the table is laid for them, each philosopher having his own place at the table:



Their only problem - besides those of philosophy - is that the dish served is a very difficult kind of spaghetti, that has to be eaten with two forks. There are two forks next to each plate, so that presents no difficulty: as a consequence, however, no two neighbours may be eating simultaneously.

A very naive solution associates with each fork a binary semaphore with the initial value = 1 (indicating that the fork is free and, naming in each philosopher these semaphores in a local terminology, we could think the following solution for the philosopher's life adequate

```
CYCLE   BEGIN   think;
                P(left hand fork); P(right hand fork);
                    eat;
                V(left hand fork); V(right hand fork);
        END.
```

But this solution - although it guarantees that no two neighbours are eating simultaneously - must be rejected because it contains the danger of deadly embrace."

- Initialisierung

```
semaphore fork[5] = {semaphore(1), semaphore(1),
                     semaphore(1), semaphore(1),
                     semaphore(1)};
shared_int number_of_philosophers = 5;
```

- Philosoph

```
int thread_I()
{ int my_number = I;
  semaphore &left_fork = fork[I],
            &right_fork
              = fork[(I + 1)
                    % number_of_philosophers);

  while (TRUE) {
    cout << "Philosopher " << setw(1) << my_number
         << " is thinking     ----\n" << flush;
    sleep(1);
    cout << "Philosopher " << setw(1) << my_number
         << " is hungry       ++++\n" << flush;
    left_fork.P(); right_fork.P();
    cout << "Philosopher " << setw(1) << my_number
         << " is eating       ####\n" << flush;
    sleep(1);
    cout << "Philosopher " << setw(1) << my_number
         << " has enough      ----\n" << flush;
    left_fork.V(); right_fork.V();
  }
}
```

```
Philosopher 0 is thinking     ----
Philosopher 3 is thinking     ----
Philosopher 2 is thinking     ----
Philosopher 4 is thinking     ----
Philosopher 1 is thinking     ----
Philosopher 0 is hungry       ++++
Philosopher 3 is hungry       ++++
Philosopher 2 is hungry       ++++
Philosopher 4 is hungry       ++++
Philosopher 1 is hungry       ++++
```

und es tut sich nichts mehr!

- Konstruktion einer verklemmungsfreien Lösung
  - Koordinierungszustand

    denkend[i], hungrig[i], speisend[i] $0 \le i < 5$

    Änderung nur in kritischen Abschnitten
  - Verbotene Zustände V:

    $\exists i(0 \le i < 5 \wedge \text{speisend}[i] \wedge \mathit{speisend}[(i+1) \bmod 5])$
  - Systeminvariante:

    Eigenschaft des Systems, die immer dann erfüllt ist, wenn sich kein Aktivitätsträger (Prozeß) innerhalb eines kritischen Abschnittes aufhält.
  - Schwächste Systeminvariante: $\neg V$

    $$\neg(\exists i(0 \le i < 5 \wedge \mathit{speisend}[i] \wedge \mathit{speisend}[(i+1) \bmod 5]))$$

    $$=$$

    $$\forall i(0 \le i < 5 \Rightarrow \neg \mathit{speisend}[i] \vee \neg \mathit{speisend}[(i+1) \bmod 5])$$
  - Stabile und instabile Zustände

    Die Menge der erlaubten, d. h. nicht verbotenen, Zustände läßt sich unterteilen in

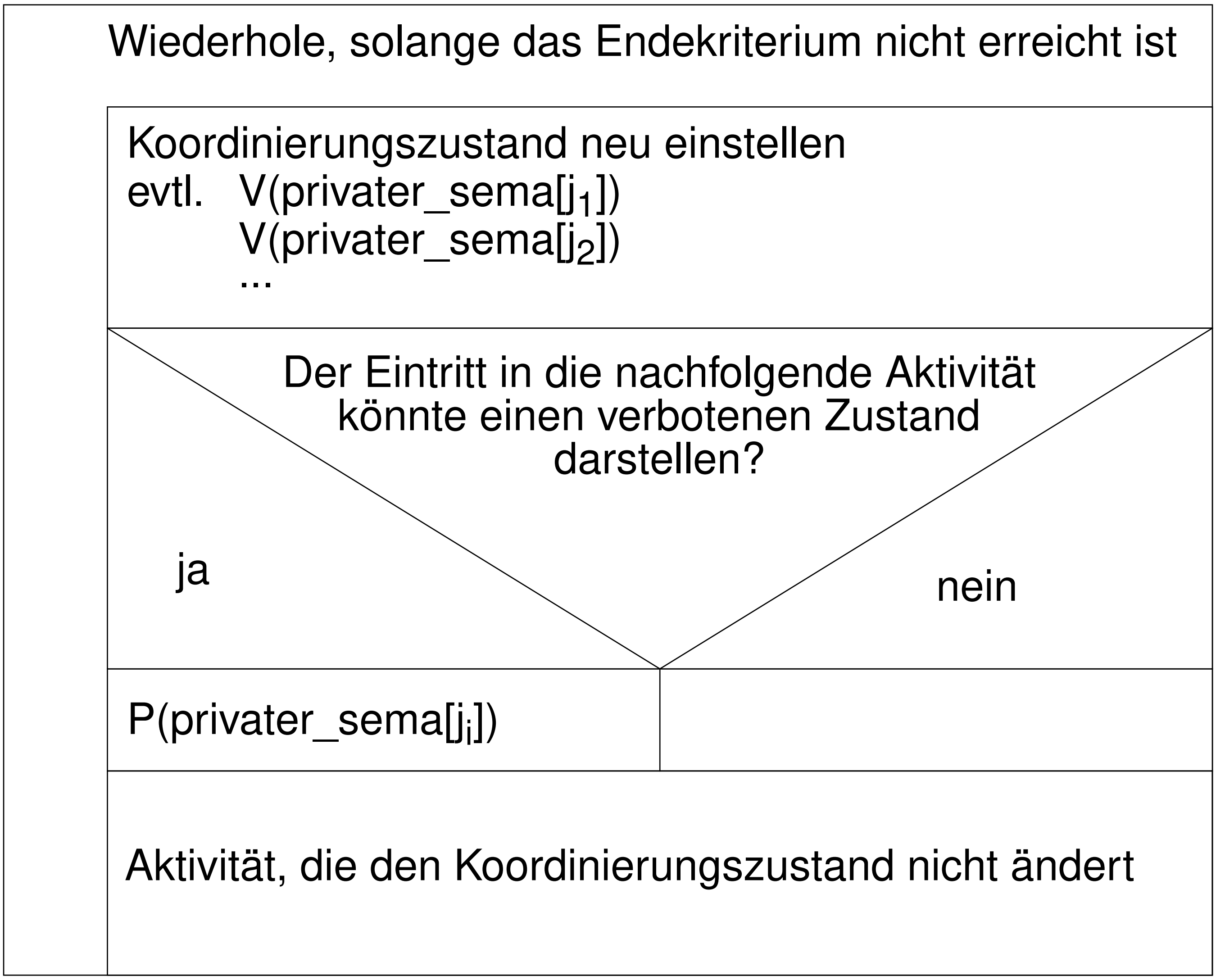
    - die Menge der instabilen Zustände, die zwar nicht als verboten zu betrachten sind, die jedoch möglichst schnell verlassen werden sollten, und
    - die Menge der stabilen Zustände, in denen das System beliebig lange verharren darf.

    $$\begin{aligned} INST = \exists i(0 \le i < 5 \wedge \neg \mathit{speisend}[(i-1) \bmod 5] \\ \wedge\ \mathit{hungrig}[i] \\ \wedge \neg \mathit{speisend}[(i+1) \bmod 5]) \end{aligned}$$

    $$STAB = \neg V \wedge \neg INST$$

- Implementierungsstruktur:

Aktivitätsträger i



- Initialisierung

```
#define THINKING 0
#define HUNGRY   1
#define EATING   2
shared_int number_of_philosophers = 5;
shared_int state[5] = {THINKING, THINKING,
                       THINKING, THINKING,
                       THINKING};
semaphore mutex(1);
semaphore private_sema[5]
            = {semaphore(0), semaphore(0),
               semaphore(0), semaphore(0),
               semaphore(0)};
```

- Philosoph

```
char state_text[3][2] = {"T", "H", "E"};
#define LEFT (i - 1 + 5) % 5
#define RIGHT (i + 1) % 5

void think(int i)
{ sleep(1);
}

void eat(int i)
{ int j;
  mutex.P();
  for (j = 0; j < 5; j++) {
    cout << state_text[state[j].value] << " ";
  }
  cout << "\n" << flush;
  mutex.V();
  sleep(1);
}
```

```
void test(int i)
{ if (state[i] == HUNGRY
        && state[(i + 4) % 5] != EATING
        && state[(i + 1) % 5] != EATING) {
      state[i] = EATING; private_sema[i].V();
    }
}

void take_forks(int i)
{ mutex.P();
      state[i] = HUNGRY;
      test(i);
  mutex.V();
  private_sema[i]->P();
}

void put_forks(int i)
{ mutex.P();
      state[i] = THINKING;
      test(LEFT); test(RIGHT);
  mutex.V();
}
```

```
thread_I()
{ int my_number = I;
  while (TRUE) {
      think(my_number);
      take_forks(my_number);
      eat(my_number);
      put_forks(my_number);
    }
}
```

```
T T T T E
T E T H E
H E H E T
E T H E T
E T E T H
T H E T E
T E T H E
H E T E T
E T H E T
E T E T H
T H E T E
T E T H E
H E T E T
E T H E T
E T E T H
T H E T E
T E T H E
T E T E T
E T H E T
E T E T H
T H E T E
T E T T E
H E T E T
E T H E T
E T E T H
T H E T E
```

3.7.2 Das Leser-Schreiber-Problem (Courtois e. a., 1971)

- Allgemeine Aufgabenstellung

  "We discuss two related problems of practical significance in which we recognize two classes of processes wishing to use the resource. The processes of the first class, named writers, must have exclusive access as in the original problem, but processes of the second class, the readers, may share the resource with an unlimited number of other readers."

- Problem 1

  "We demand of our solution that no reader be kept waiting unless a writer has already obtained permission to use the resource; i. e. no reader should wait simply because a writer ist waiting for other readers to finish."

- Initialisierung

```
shared_int readcount(0);
pv mutex(1), w(1);
```

- Leser

```
PROCESS_L()
{ while (1) {
      mutex.P();
         readcount++;
         if (readcount == 1) w.P();
      mutex.V();
         . . .  r e a d i n g
      mutex.P();
         readcount--;
         if (readcount == 0) w.V();
      mutex.V();
      remainder_L();
   }
}
```

- Schreiber

```
PROCESS_S()
{ while (1) {
      w.P();
         . . .  w r i t i n g
      w.V();
      remainder_W();
   }
}
```

- Problem 2

  "Here we retain the requirement that writers must have exclusive access while readers may share, but we add the requirement that once a writer is ready to write, he performs his write as soon as possible."

  - Initialisierung

```
shared_int readcount(0), writecount(0);
pv mutex1(1), mutex2(1), mutex3(1), w(1), r(1);
```

- Leser

```
PROCESS(0)
  shared_int readcount(0), writecount(4);
  pv mutex1(0), mutex2(1), mutex3(2), w(3), r(4);
  while (1) {
    mutex3.P();
      r.P();
        mutex1.P();
          readcount++; if (readcount == 1) w.P();
        mutex1.V();
      r.V();
    mutex3.V();
      . . .  r e a d i n g
    mutex1.P();
    readcount--; if (readcount == 0) w.V();
    mutex1.V();
  }
PROCESS_END
```

- Schreiber

```
PROCESS(1)
  shared_int readcount(0), writecount(4);
  pv mutex1(0), mutex2(1), mutex3(2), w(3), r(4);
  while (1) {
    mutex2.P();
    writecount++; if (writecount == 1) r.P();
    mutex2.V();
    w.P();
     . . .  w r i t i n g
    w.V();
    mutex2.P();
    writecount--; if (writecount == 0) r.V();
    mutex2.V();
    sleep(2);
  }
PROCESS_END
```

3.8 Kommunikation über Nachrichten

- Klassen
  - Kanäle (pipes)
  - Briefkästen (queues)
  - Kommunikationsendpunkte (sockets)
- Übertragungsrichtung
  - unidirektional
  - bidirektional (voll-duplex, halb-duplex)
- Adressierung
  - implizit: UNIX-Pipes
  - explizit: CSP, OCCAM
  - globale Adressierung: Sockets, Ports
  - Gruppenadressierung: multicast, broadcast
  - funktionale Adressierung: Dienste
- Protokollfamilien
  - stromorientiert
  - nachrichtenorientiert
- Klassifikation
  - blockierend - nichtblockierend
  - gepuffert - ungepuffert
  - zuverlässig - unzuverlässig

UNIX-queues

- Initialisierung

```
#include "shared_file.h"
#include <sys/msg.h>
int main()
{ shared_int msqid(3008);
  key_t key = 1;

  msqid = msgget(key, int(0600|IPC_PRIVATE|IPC_CREAT));

  msgctl(msqid, IPC_RMID, 0);

  sleep(10);

  msqid = msgget(key, int(0600|IPC_PRIVATE|IPC_CREAT));

  return 1;
}
```

- Prozeß 0 (Erzeuger)

```
PROCESS(0)
  shared_int msqid(3008); char item[200]; int length;
  struct msgform
    { long mtype; char mtext[200];} buffer;
  while (1) {
    length = msgrcv(msqid,(msgbuf *)&buffer,200,1,0);
    cout<<"Producer received: "<<buffer.mtext
        <<"  sends: "<<flush;
    cin >> item;
    strcpy(buffer.mtext, item); buffer.mtype = 2;
    msgsnd(msqid,(msgbuf *)&buffer,strlen(item)+1,0);
    }
  }
PROCESS_END
```

- Prozeß 1 (Verbraucher)

```
PROCESS(1)
  shared_int msqid(3008); char item[200]; int i,
length;
  struct msgform
    { long mtype; char mtext[200];} buffer;
  buffer.mtype = 1; strcpy(buffer.mtext, "empty");
  for (i = 0; i < 3; i++)
    msgsnd(msqid,(msgbuf *)&buffer,
           strlen(buffer.mtext)+1,0);

  while (1) {
    length = msgrcv(msqid,(msgbuf *)&buffer,200,2,0);
    cout<<"Consumer received: "<<buffer.mtext
        <<"\n"<<flush;
    strcpy(buffer.mtext, "empty"); buffer.mtype = 1;
    msgsnd(msqid,(msgbuf *)&buffer,
           strlen(buffer.mtext)+1,0);
    sleep(1);
  }
PROCESS_END
```

UNIX-sockets

- Initialisierung

```
#include "shared_file.h"
int main()
{ unlink("consumer_socket");
  unlink("producer_socket");
  return 1;
}
```

- Prozeß 0 (Erzeuger)

```
PROCESS(0)
  char item[200];
  int sd;
  struct sockaddr_un producer_socket, consumer_socket;
  producer_socket.sun_family = AF_UNIX;
  strcpy(producer_socket.sun_path, "producer_socket");
  consumer_socket.sun_family = AF_UNIX;
  strcpy(consumer_socket.sun_path, "consumer_socket");
  unlink("producer_socket");

  sd = socket(AF_UNIX, SOCK_DGRAM, 0);
  bind(sd, (struct sockaddr *) &producer_socket,
            strlen(producer_socket.sun_path)
              + sizeof(producer_socket.sun_family));
  while (connect(sd,
                 (struct sockaddr *)&consumer_socket,
                 strlen(consumer_socket.sun_path)
                   + sizeof(consumer_socket.sun_family))
         < 0) {
    cerr << "\nError in connecting
         << "producer to consumer: "
         << sys_errlist[errno] << "\n"
         << "Trying again\n" << flush;
    sleep(1);
  }

  while (1) {
    recv(sd, item, 199, 0);
    cout << "Producer received: " << item
         << "\n" << flush;
    cout << "Message to be sent by the producer: "
         << flush;
    cin >> item;
    send(sd, item, strlen(item) + 1, 0);
  }
PROCESS_END
```

- Prozeß 1 (Verbraucher)

```
PROCESS(1)
  char item[200];
  int sd, i;
  struct sockaddr_un producer_socket, consumer_socket;
  producer_socket.sun_family = AF_UNIX;
  strcpy(producer_socket.sun_path, "producer_socket");
  consumer_socket.sun_family = AF_UNIX;
  strcpy(consumer_socket.sun_path, "consumer_socket");
  unlink("consumer_socket");

  sd = socket(AF_UNIX, SOCK_DGRAM, 0);
  bind(sd, (struct sockaddr *) &consumer_socket,
           strlen(consumer_socket.sun_path)
             + sizeof(consumer_socket.sun_family));
  while (connect(sd,
                 (struct sockaddr *) &producer_socket,
                 strlen(producer_socket.sun_path)
                   + sizeof(producer_socket.sun_family))
         < 0) {
    cerr << "\nError in connecting
         << "consumer to producer: "
         << sys_errlist[errno] << "\n"
         << "Trying again\n" << flush;
    sleep(1);
  }
  for (i = 0; i < 3; i++) {
    strcpy(item, "Please send next item");
    send(sd, item, strlen(item) + 1, 0);
  }
  while(1) {
    recv(sd, item, 199, 0);
    cout << "Consumer received: "
         << item << "\n" << flush;
    strcpy(item, "Please send next item");
    send(sd, item, strlen(item) + 1, 0);
  }
PROCESS_END
```

3.9 Verallgemeinerungen des Semaphorkonzeptes und Konzepte höherer Abstraktion

- (Φ, Θ)-Koordinierungssysteme

System-name	Ausführbarkeits-bedingung	Wirkung	Operationsbezeichnung
PV	$'S_j > 0$ true	$S_j = 'S_j - 1$ $S_j = 'S_j + 1$	$P(S_j)$ $V(S_j)$
PV two way bounded $k > 0$	$'S_j > 0$ $'S_j < k$	$S_j = 'S_j - 1$ $S_j = 'S_j + 1$	$P_k(S_j)$ $V_k(S_j)$
PV chunk $t > 0$	$'S_j \geq t$ true	$S_j = 'S_j - t$ $S_j = 'S_j + t$	$P(S_j{:}t)$ $V(S_j{:}t)$
PV multiple $J = \{1, ...,k\}$	$\forall j(j \in J \Rightarrow 'S_{i_j} > 0)$ true	$\forall j(j \in J \Rightarrow S_{i_j} = 'S_{i_j} - 1)$ $\forall j(j \in J \Rightarrow S_{i_j} = 'S_{i_j} + 1)$	$P(\{S_{i_j} \mid j \in J\})$ $V(\{S_{i_j} \mid j \in J\})$
PV general $J = \{1, ...,k\}$ $\forall j(j \in J \Rightarrow t_j > 0)$	$\forall j(j \in J \Rightarrow 'S_{i_j} \geq t_j)$ true	$\forall j(j \in J \Rightarrow S_{i_j} = 'S_{i_j} - t_j)$ $\forall j(j \in J \Rightarrow S_{i_j} = 'S_{i_j} + t_j)$	$P(\{S_{i_j}{:}t_j \mid j \in J\})$ $V(\{S_{i_j}{:}t_j \mid j \in J\})$
up/down	$\sum_{r=1}^{k} 'S_{i_r} \geq 0$ $\sum_{r=1}^{k} 'S_{i_r} \geq 0$	$S_j = 'S_j + 1$ $S_j = 'S_j - 1$	$\{S_{i_r} \mid 1 \leq r \leq k\}{:}up(S_j)$ $\{S_{i_r} \mid 1 \leq r \leq k\}{:}down(S_j)$

Systemname	Φ	Θ	Operationsbezeichnung
Vektor-ersetzung $e_1, e_2 \in Z^m$	$'S + e_1 \geq 0$	$S = 'S + e_2$	$<e_1 \vert e_2>$
Vektor-addition $e \in Z^m$	$'S + e \geq 0$	$S = 'S + e$	$<e>$

3.10 Einschlägige Systemaufrufe bei SunOS (Auswahl)

3.10.1 Prozesse

- Erzeugung, Überlagerung

  create a new process
  ```
  pid_t fork()
  ```
  execute a file
  ```
  int execve(char *path, char *argv[], char *envp[])
  ```

- Prozeß beenden (und tilgen)

  terminate a process
  ```
  void _exit(int status)
  ```

- Prozeßidentifikator

  get process identification
  ```
  int getpid()
  ```
  get process identification of parent
  ```
  int getppid()
  ```

- Warten auf Beendigung eines untergeordneten Prozesses

  wait for process to terminate or stop, examine returned status
  ```
  pid_t wait(int *statusp)
  ```

3.10.2 Semaphore

- Erzeugung

    get set of semaphores

```
int semget(key_t key, int nsems, int semflg)
```

- Operationen

    semaphore operations

```
int semop(int semid, struct sembuf *sops, int nsops)
    struct sembuf {
        short sem_num;    /* semaphore # */
        short sem_op;     /* semaphore operation */
        short sem_flg;    /* operation flags */
    };
```

semaphore control operations

```
int semctl(int semid, int semnum, int cmd, semun arg)
    union semun {
        int                 val;
        struct semid_ds     *buf;
        ushort              *array;
    };
    #define  GETNCNT  3        /* get semncnt */
    #define  GETPID   4        /* get sempid */
    #define  GETVAL   5        /* get semval */
    #define  GETALL   6        /* get all semval's */
    #define  GETZCNT  7        /* get semzcnt */
    #define  SETVAL   8        /* set semval */
    #define  SETALL   9        /* set all semval's */
```

3.10.3 Nachrichten

3.10.3.1 Kommunikationskanäle, Pipes

- Erzeugung

create an interprocess communication channel

```
int pipe(int fd[2])
```

- Operationen

read input

```
int read(int fd, char *buf,  int nbyte)
```

read input and scatter

```
int readv(int fd, struct iovec *iov, int iovcnt)
```

write output

```
int write(int fd, char *buf,  int nbyte)
```

write output and gather

```
int writev(int fd, struct iovec *iov, int iovcnt)
```

3.10.3.2 Briefkästen, mailboxes, queues, ...

- Erzeugung

  get message queue

```
int msgget(key_t key, int msgflg)
```

- Operationen

  message operations

```
int msgsnd(int msqid,
           struct msgbuf *msgp, int msgsz,
           int msgflg)
int msgrcv(int msqid,
           struct msgbuf *msgp, int msgsz,
           long msgtyp, int msgflg)
```

3.10.3.3 Kommunikationsendpunkte, Sockets

- Erzeugung

create an endpoint for communication

```
int socket(int domain, int type, int protocol)
```

create a pair of connected sockets

```
int socketpair(int d, int type,
               int protocol, int sv[2])
```

- Operationen

bind a name to a socket

```
int bind(int s, struct sockaddr *name, int namelen)
```

initiate a connection on a socket

```
int connect(int s,
            struct sockaddr *name, int namelen)
```

send a message from a socket

```
int send(int s, char *msg, int len, int flags)
int sendto(int s, char *msg, int len, int flags,
           struct sockaddr *to, int tolen)
int sendmsg(int s, struct msghdr *msg, int flags)
```

receive a message from a socket

```
int recv(int s, char *buf, int len, int flags)
int recvfrom(int s, char *buf, int len, int flags,
             struct sockaddr *from, int *fromlen)
int recvmsg(int s, struct msghdr *msg, int flags)
```

listen for connections on a socket

```
int listen(int s, int backlog)
```

accept a connection on a socket

```
int accept(int s, struct sockaddr *addr,
          int *addrlen)
```