UNIVERSITÄT BERN

2405 Betriebssysteme
V. Kommunikation und Synchronisation zwischen Prozessen

Thomas Staub, Markus Anwander
Universität Bern

Inhalt

1. Prozessinteraktion

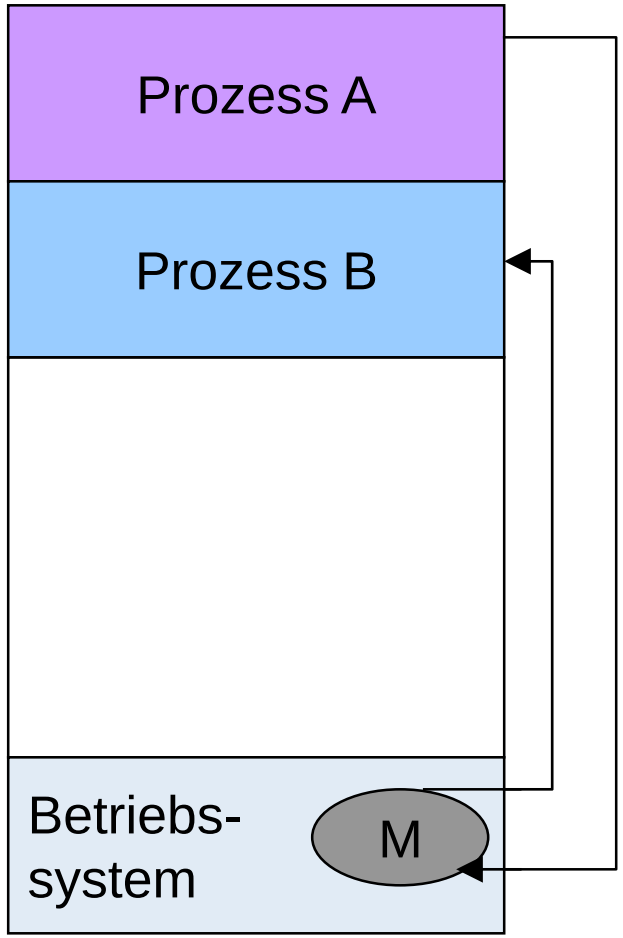
- gegenseitige Beeinflussung kooperierender Prozesse
- Vorteile
  - Teilen von Informationen
  - beschleunigte Verarbeitung in Multiprozessorsystemen
  - Modularität
  - Unterstützung nebenläufiger Aktivitäten eines Benutzers

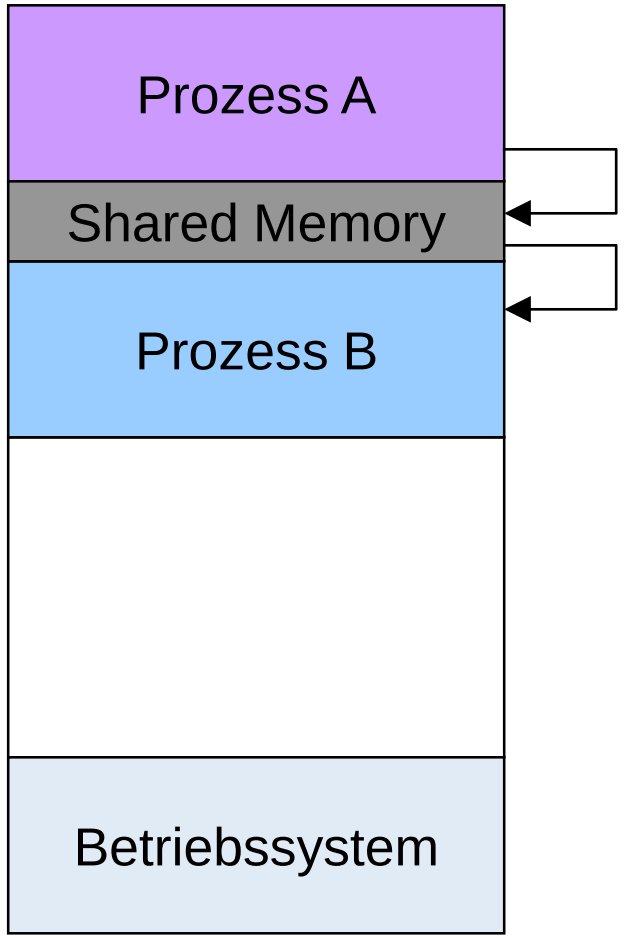
1. Prozessinteraktion

Mechanismen zur Kommunikation zwischen Prozessen und zur Synchronisation ihrer Aktionen

- Speicherbasiert: Zugriff auf Variablen in gemeinsamem Speicher (*Shared Memory*)
  - erfordert Synchronisation des Zugriffs
- Nachrichtenbasiert: Nachrichtenaustausch über send / receive Operationen (*Message Passing*)
  - *direkte / indirekte Kommunikation (Links / gemeinsame Mailbox)*
  - *Puffern von Nachrichten (synchron / asynchron)*
  - symmetrische / asymmetrische Kommunikation
  - send by copy / reference
  - feste / variable Nachrichtenlängen

1.1 Message Passing und Shared Memory



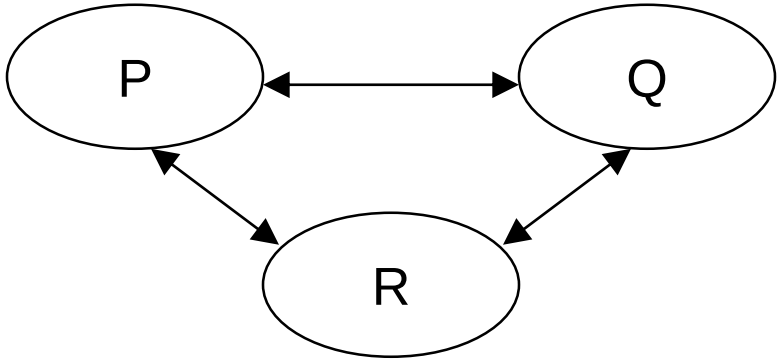


M: Mailbox: Objekt, in das Nachrichten gestellt bzw. aus dem Nachrichten entnommen werden können

1.1.1 Direkte und indirekte Kommunikation

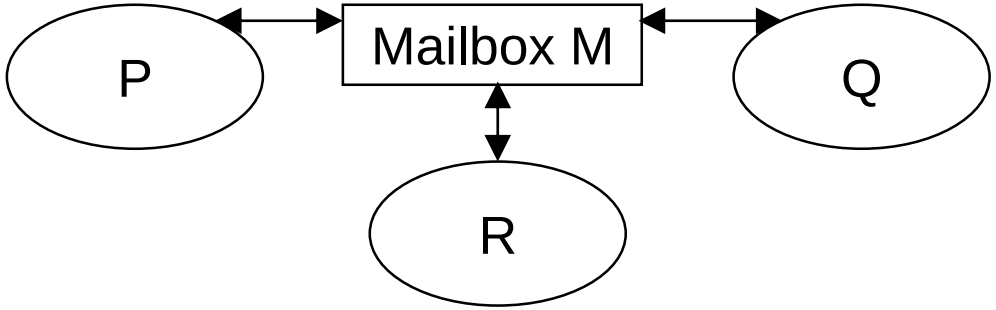
Direkte Kommunikation

- automatische Etablierung von (üblicherweise bidirektionalen) Links zwischen Prozesspaaren
- explizite Adressierung des Partnerprozesses
- Operationen
  - send (P, message)
  - receive (Q, message) oder receive (id, message)

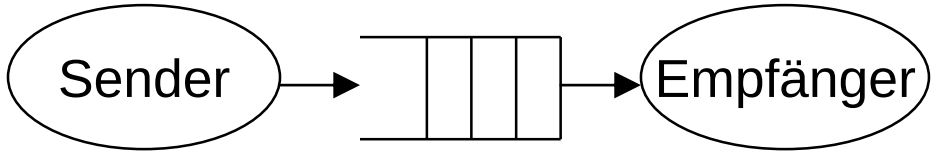


Indirekte Kommunikation

- Kommunizierende Prozesse teilen Mailbox M (gehört zu Prozess oder System).
- Jede Mailbox hat eindeutige Kennung.
- Operationen
  - create (M)
  - send (M, message)
  - receive (M, message)
  - destroy (M)



1.1.2 Puffern von Nachrichten

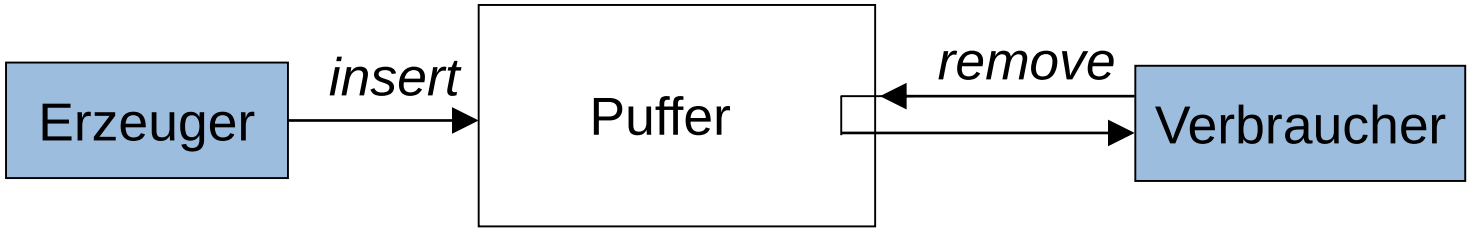


Kommunikationskanal (Link) zwischen Prozessen
mit bestimmter Kapazität zum Speichern von Nachrichten (Warteschlange)

- synchrone Kommunikation
  - *keine* Kapazität zum Speichern von Nachrichten
  - Synchronisation von Sender und Empfänger (Rendezvous)
- asynchrone Kommunikation
  - Sender platziert Nachricht in Warteschlange mit *endlicher* Kapazität und fährt fort.
  - Sender muss bei voller Warteschlange warten, Empfänger muss bei leerer Warteschlange warten.
  - *unendliche* Kapazität: Sender wird nie verzögert.

1.2 Speicherbasierte Prozessinteraktion

- Parallele Prozesse mit Zugriff auf gemeinsame Daten, wodurch Inkonsistenzen verursacht werden können.
- Mechanismen zur Synchronisation von parallelen Prozessen erforderlich
- Beispiel: Erzeuger/Verbraucher-Problem
  - Puffer kann N Informationselemente speichern.
  - → Erzeuger darf nicht in vollen Puffer speichern.
  - → Verbraucher darf nicht aus leerem Puffer entfernen.



1.3 Beispiele für Prozessinteraktion

- POSIX shared memory
  - **`shm_fd = shm_open(name, oflag, mode)`**
  - **`ftruncate(shm_fd, length)`**
  - **`mmap(address, length, flags, shm_fd, offset)`**
- Unix Pipes
  - **`pipe(int fd[])`**: Erzeugen von Pipes, Behandeln von Pipes wie Dateien
  - **`write(), read()`**: Zugriff auf Pipes
- Mach Interprozess-Kommunikation
  - **`port_allocate()`**: Erzeugen einer Mailbox
  - **`msg_send() / msg_receive()`**: Senden / Empfangen
- Sockets
  - vgl. Computernetze
- Remote Procedure Calls
  - vgl. späteres Kapitel
- Remote Method Invocation
  - Entfernter Java Methodenaufruf

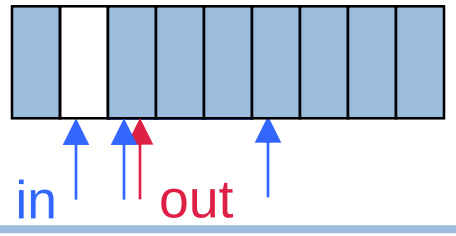
2.1.1 Erzeuger/Verbraucher-Implementierung

```
int in=0,out=0;
item_type buffer[N];
```

```
item_type next_prod;
while (true){
        produce(&next_prod);
        while((in+1)%N==out) /*voller Puffer*/
                no_op;
        buffer[in]=next_prod;
        in=(in+1)%N;
}
```

```
item_type next_cons;
while (true){
        while(in==out) /*leerer Puffer*/
                no_op;
        next_cons=buffer[out];
        out=(out+1)%N;
        consume(&next_cons);
}
```

Problem: höchstes N-1 Elemente im Puffer



Puffer leer
Puffer gefüllt
Puffer voll

in zeigt auf nächsten platz an den wir speichern können

2.1.2 Erzeuger/Verbraucher-Implementierung


```
int in=0,out=0,counter=0;
item_type buffer[N];
```

```
item_type next_prod;
while (true){
        produce(&next_prod);
        while(counter==N) /*voller Puffer*/
                no_op();
        buffer[in]=next_prod;
        in=(in+1)%N;
        counter++;
}
```

```
item_type next_cons;
while (true){
        while(counter==0) /*leerer Puffer*/
                no_op();
        next_cons=buffer[out];
        out=(out+1)%N;
        counter--;
        consume(&next_cons);
}
```

Problem: `counter++` und `counter--` müssen atomar sein.

2.2 Race Conditions

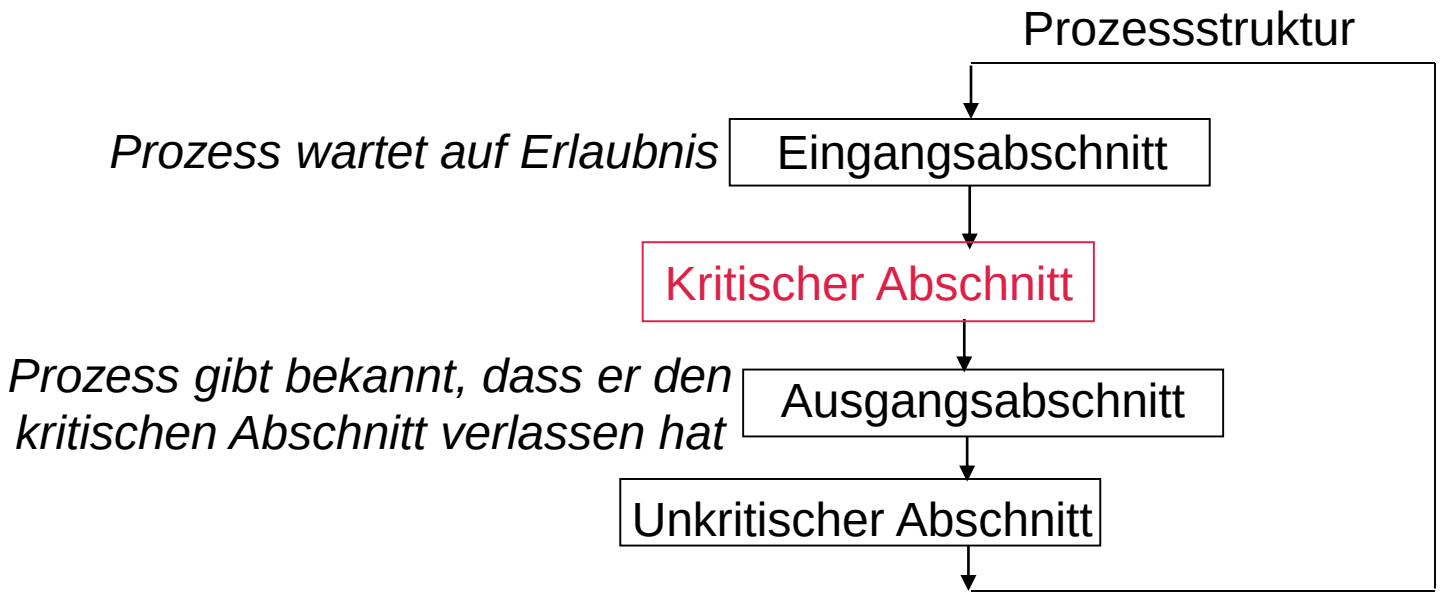

```
counter++:                    counter--:
R1=counter;                   R2=counter;
R1=R1+1;                      R2=R2-1;
counter=R1;                   counter=R2;

                    counter: 5

R1=counter;                   R2=counter;
R2=counter;                   R1=counter;
R1=R1+1;                      R2=R2-1;
R2=R2-1;                      R1=R1+1;
counter=R1;                   counter=R2;
counter=R2; /* =4 */          counter=R1; /* =6 */
```

3. Kritischer Abschnitt

- Kritischer Abschnitt: Folge von Anweisungen oder Code-Segment mit Zugriff auf gemeinsame Daten
- Höchstens ein Prozess darf sich zu einem Zeitpunkt in einem kritischen Abschnitt befinden.
- Beispiele:
  - Zugriff auf ein exklusives Betriebsmittel
  - Manipulation der Counter-Variable in Erzeuger/Verbraucher-Implementierung



3.1 Anforderungen an Lösungsansätze

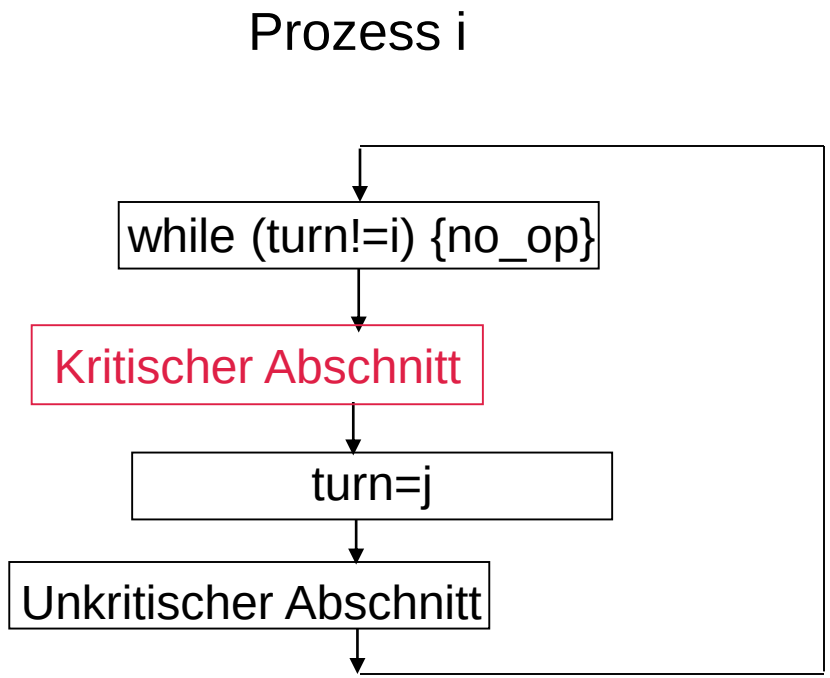
1. Wechselseitiger Ausschluss (Sicherheit)
   - Wenn sich ein Prozess im kritischen Abschnitt befindet, dann darf sich kein anderer darin befinden.
2. Fortschritt
   - Wenn sich kein Prozess im kritischen Abschnitt befindet, dürfen andere Prozesse, die in den kritischen Abschnitt eintreten wollen, nicht blockiert werden.
3. Begrenztes Warten
   - Die Zeit zwischen der Anforderung und der Gewährung des Eintretens in einen kritischen Abschnitt muss begrenzt sein.

3.2.1 Algorithmus 1

- gemeinsame Variable

  ```
  int turn = i;
  turn == i
  ```

  → Prozess i kann in kritischen Abschnitt eintreten.
- Sicherheit
- kein Fortschritt (strikte Alternierung)



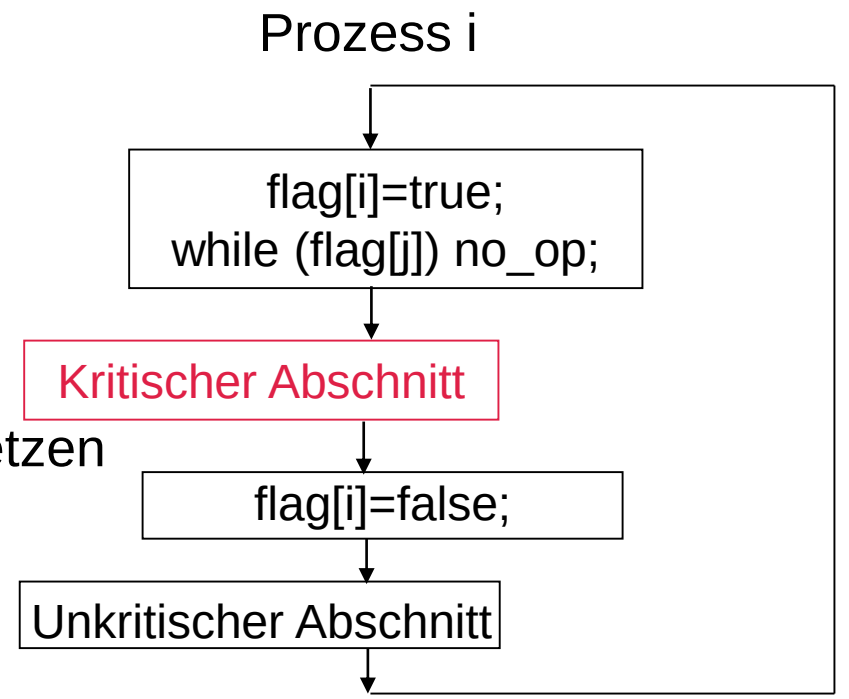
3.2.2 Algorithmus 2

- gemeinsame Variablen

```
enum bool (false, true);
bool flag[2];
```

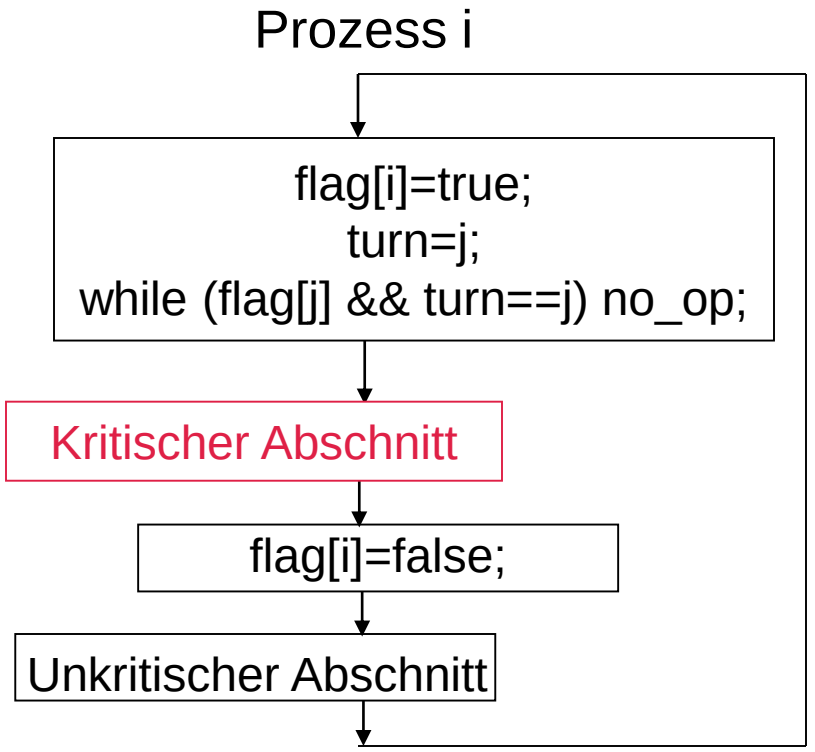
```
flag[j] == false
```

  → Prozess i kann in kritischen Abschnitt eintreten.
- Sicherheit
- kein Fortschritt, wenn beide Prozesse nacheinander ihr Flag setzen



3.2.3 Algorithmus 3: Peterson‘s Lösung

- Kombination von Algorithmus 1 und 2
- erfüllt alle drei Kriterien
- nur geeignet für zwei Prozesse

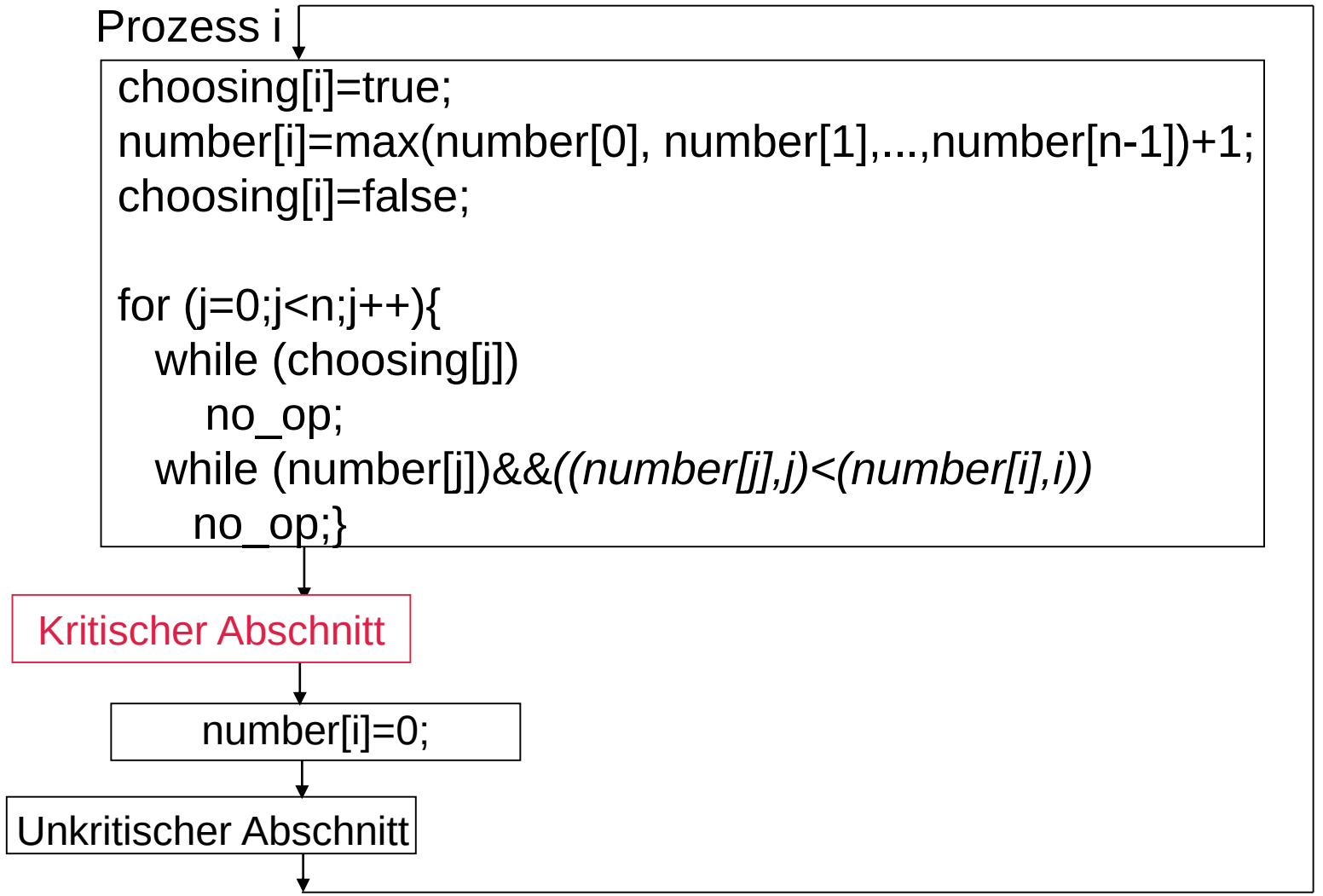


3.2.4 Bakery Algorithmus

- Lösung für N Prozesse
- Vor dem Eintreten in einen kritischen Abschnitt erhält ein Prozess eine Nummer.
- aufsteigende Nummerierung, z.B.: 1, 2, 2, 3, 3, 4, 5, 6, 6, ...
- Prozess mit kleinster Nummer tritt in den kritischen Abschnitt ein.
- Bei Nummerngleichheit gibt Prozess ID den Ausschlag.

3.2.4.1 Implementierung des Bakery Algorithmus

- Notation: *(a,b) < (c,d)*
  (a < c) || ((a==c)&&(b<d))
- gemeinsame Variablen:
  `bool choosing[N];`
  `int number[N];`
  (Initialisierung mit false und 0)



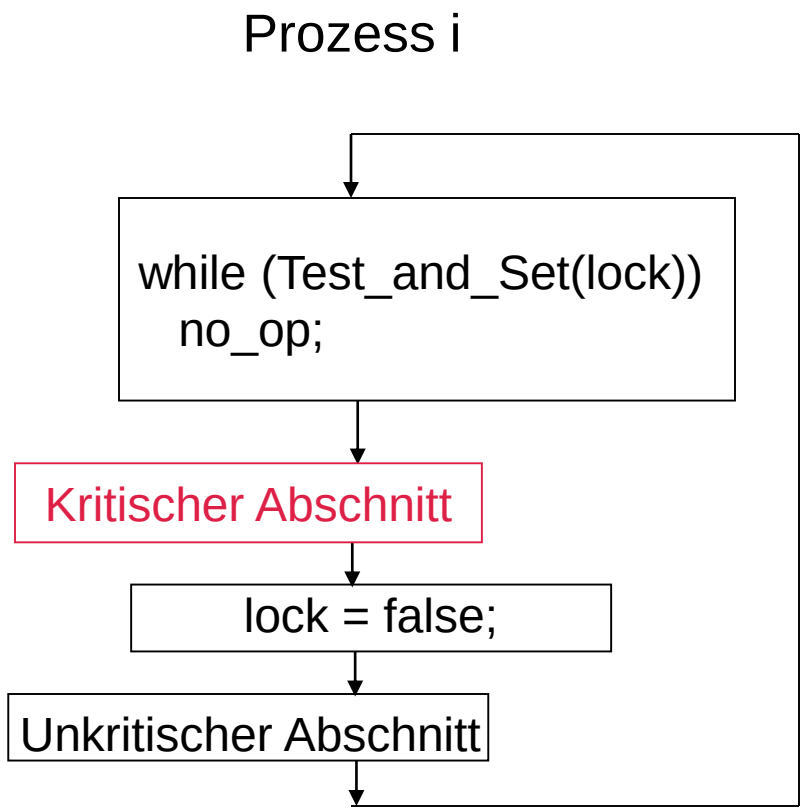
3.2.5.1 Synchronisations-Hardware: Test and Set

- gemeinsame Variable:

```
bool lock=false;
```

- atomare Funktion *Test_and_Set*

```
bool Test_and_Set(bool *target)
{
  bool rv = *target;
  *target = true;
  return rv;
}
```



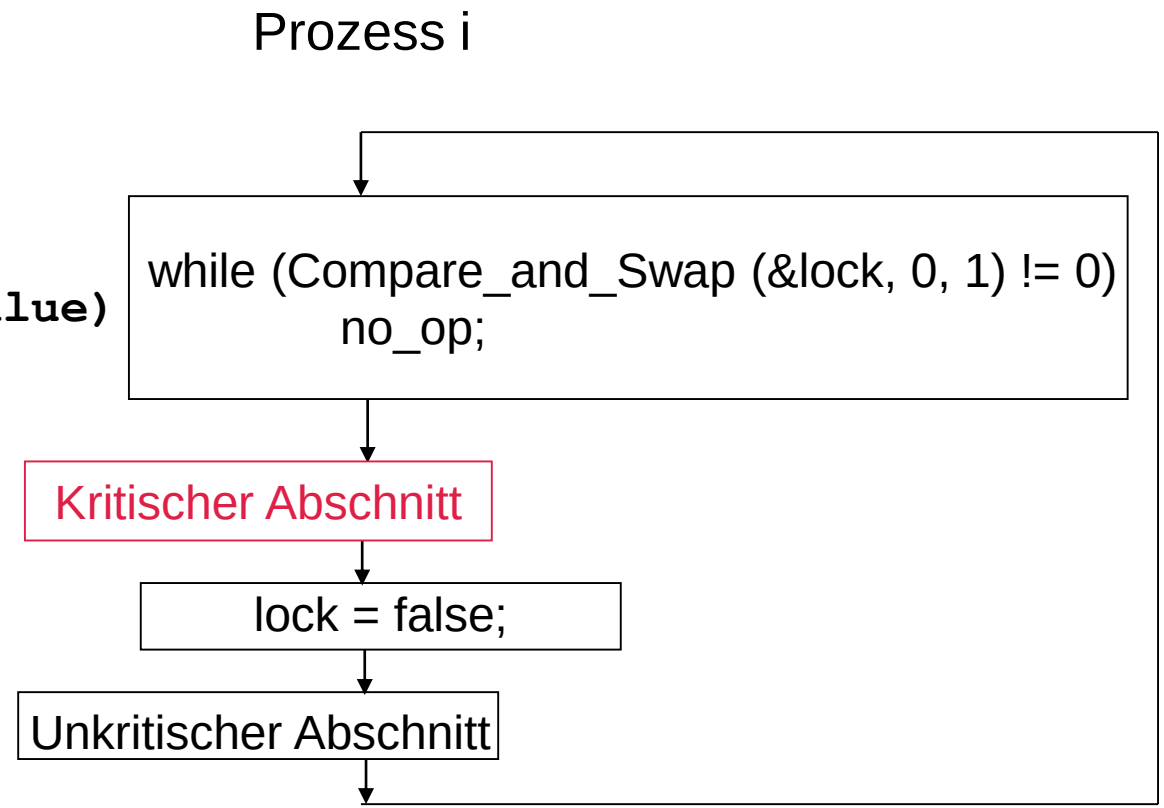
3.2.5.2 Synchronisations-Hardware: Compare and Swap

- gemeinsame Variable:

```
bool lock=false;
```

- atomare Funktion *Compare_and_Swap*

```
void Compare_and_Swap
   (int *value, int expected, int new_value)
{
   int temp = *value;
   if (*value == expected)
       *value =  new_value;
   return temp;
}
```

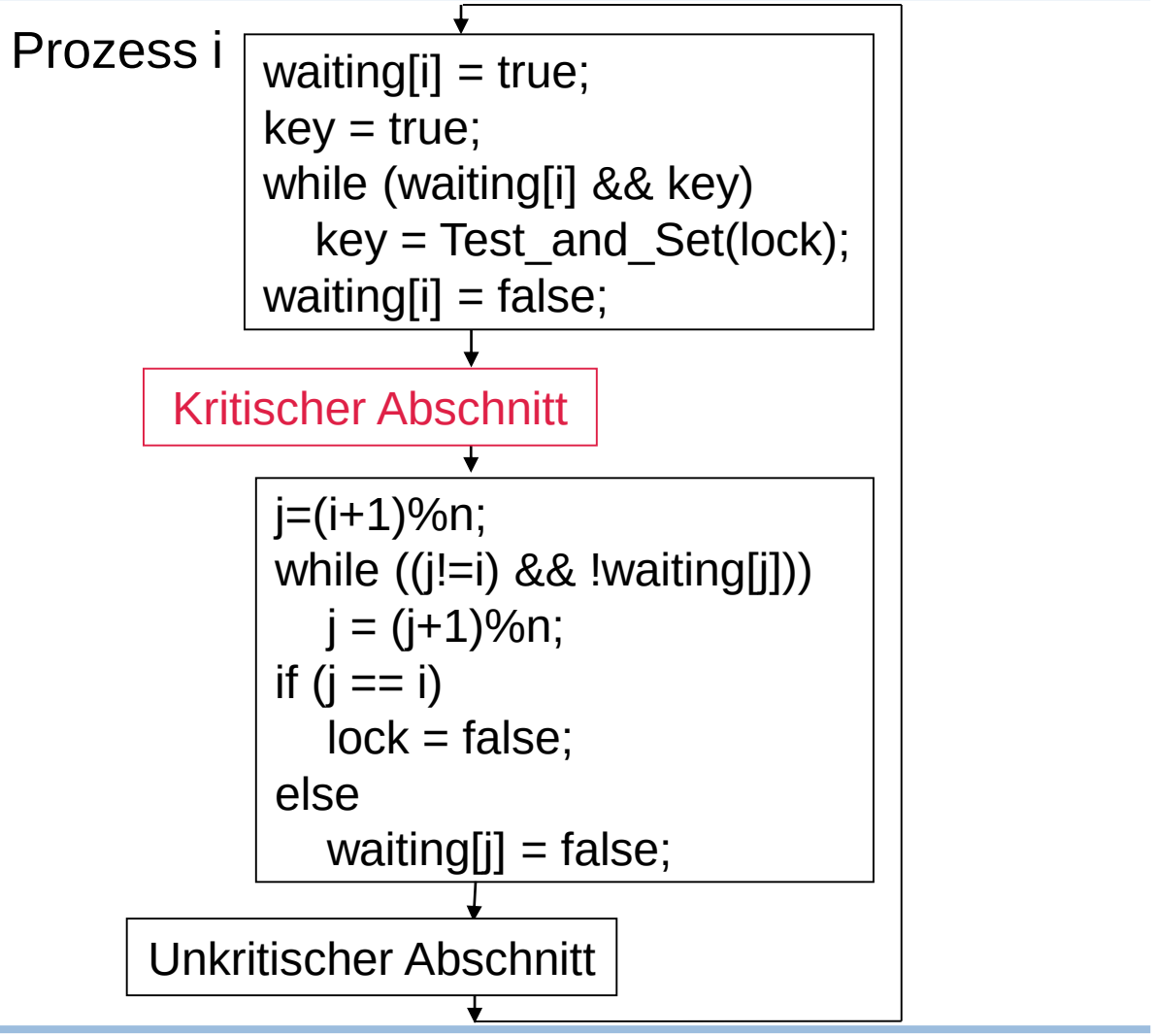


Problem: Einfache Synchronisations-Hardware-Lösungen erfüllen Kriterium 3 (begrenztes Warten) nicht.

3.2.5.3 Lösung mit Synchronisations-Hardware

- Erfüllen von Kriterium 3
- gemeinsame Variable:

```
bool lock=false;
bool waiting[i]=false;
```



4. Semaphore

- geschützte Variable s, auf der nur atomare Operationen **`wait(s)`** und **`signal(s)`** ausgeführt werden können.
- Werte
  - > 0: frei
  - ≤ 0: belegt
- Binäre Semaphor
  - kann nur Wert 0 oder 1 annehmen.
  - Initialisierung auf 1
- Zählende Semaphor
  - Einsatz bei mehreren Instanzen einer verfügbaren Ressource
  - typisch: Initialisierung auf Anzahl der verfügbaren Ressourcen

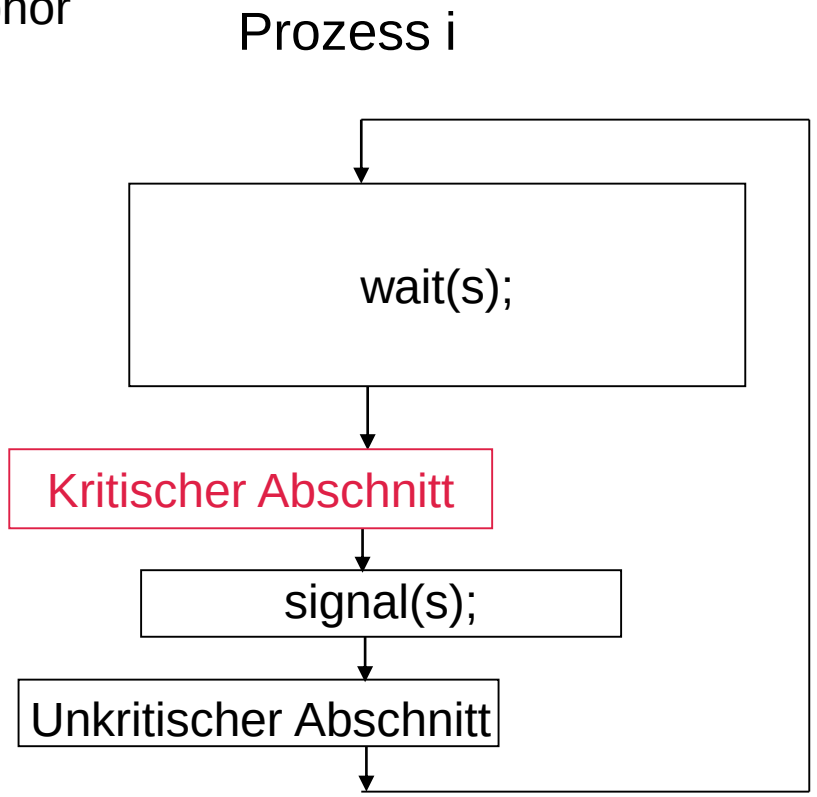
4.1 Semaphoroperationen

```
wait(s):
while (s ≤ 0) no_op; s--;
```

— Belegen einer Semaphor bzw. Warten bei bereits belegter Semaphor

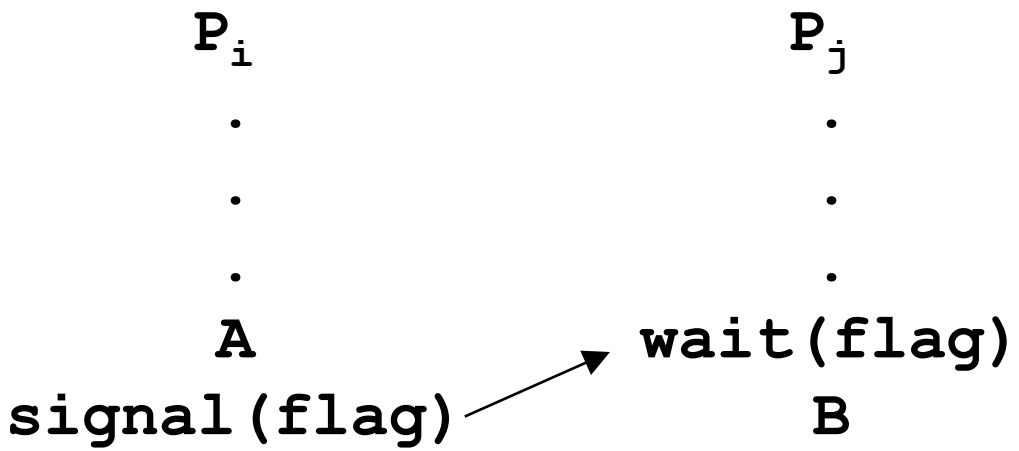
```
signal(s): s++;
```

— Freigeben einer Semaphor



4.2 Synchronisation mit Semaphoren

- Aufgabe:
  - Code-Segment B von Prozess j soll nach Code-Segment A von Prozess i ausgeführt werden.
- Lösung mit Semaphor **`flag`**, Initialisierung mit 0



4.3.1 Implementierung von Semaphoren

- Problem: „Busy Waiting“ bei belegter Semaphor
- Lösung:
  - füge Prozess bei belegter Semaphore einer Liste wartender Prozesse (L) zu.

```
typedef struct s {int value; process_list L} semaphore;
```

  - Zählende Semaphor
    - negativer Wert von **`value`**: Anzeige der Anzahl wartender Prozesse
    - positiver Wert von **`value`**: Anzahl der Prozesse, die noch eintreten dürfen

4.3.2 Implementierung von Semaphoren


```
wait(S):
S.value--;
if (S.value < 0) {
    add_this_process_to(S.L);
    block;} /* Blockieren des aufrufenden Prozess */

signal(S):
S.value++;
if (S.value ≤ 0){
    P=remove_a_process_from(S.L);
    wakeup(P);} /* Deblockieren von P */
```

4.4 Klassische Synchronisationsprobleme

- Erzeuger/Verbraucher mit beschränktem Puffer
- Leser/Schreiber-Problem
  - Entweder ein Schreiber oder N Leser dürfen auf ein gemeinsames Datenobjekt zugreifen.
- Dining Philosophers

4.4.1 Erzeuger/Verbraucher mit Semaphoren

```
item_type    buffer[N], next_cons, next_prod;
semaphore    full=0, empty=N, mutex=1
             /* empty zählt leere Speicherplätze */
             /* full zählt volle Speicherplätze */
             /* mutual exclusion */
```

Erzeuger

```
for (;;){
        produce(&next_prod);
        wait(empty);
        wait(mutex);
        buffer[in]=next_prod;
        in=(in+1)%N;
        signal(mutex);
        signal(full);
}
```

Verbraucher

```
for (;;){
        wait(full);
        wait(mutex);
        item_cons=buffer[out];
        out=(out+1)%N;
        signal(mutex);
        signal(empty);
        consume(&next_cons);
}
```

4.4.2 Leser/Schreiber-Problem


```
semaphore     mutex=1, wrt=1;
int           readcount=0;
```

Leser

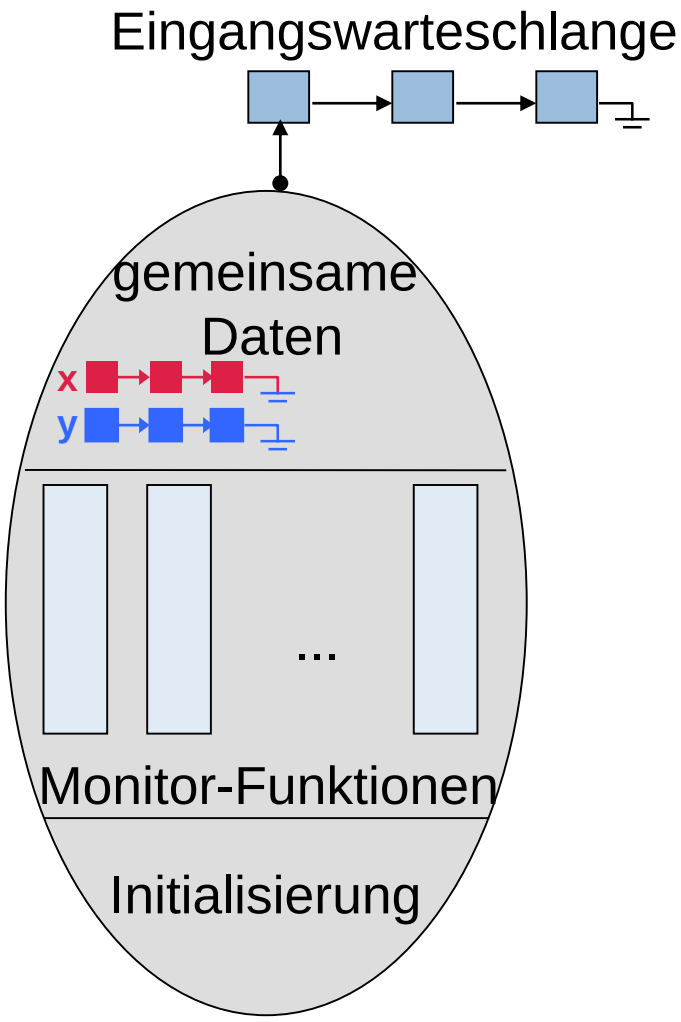
```
wait(mutex);
readcount++;
if(readcount == 1)
    wait(wrt);
signal(mutex);
reading();
wait(mutex);
readcount--;
if(readcount == 0)
    signal(wrt);
signal(mutex);
```

Schreiber

```
wait(wrt);
writing();
signal(wrt);
```

5. Monitore

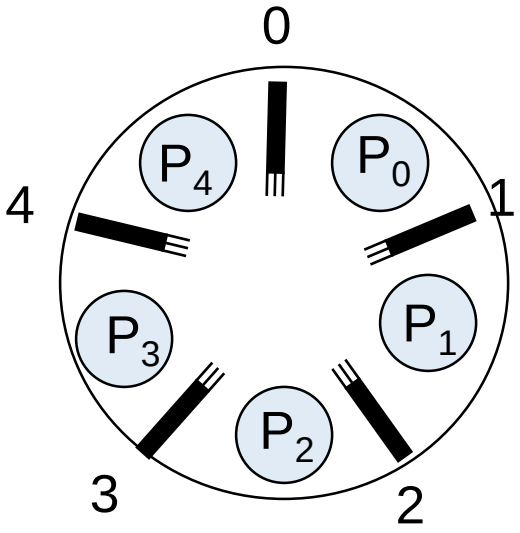
- Monitor = Sammlung von Prozeduren, Variablen und Datenstrukturen in einem Modul (information hiding)
- In einem Monitor existiert zu einem Zeitpunkt nur 1 aktiver Prozess.
- Monitor-Initialisierung: **`Monitor(Parameter)`**
- Aufruf der Monitor-Funktionen: **`Monitor.Funktion(Parameter)`**
- Condition-Variable x zur Formulierung kritischer Abschnitte im Monitor
  - Zugriff auf Condition-Variable über
    - **`x.wait`**
      - Aufrufender Prozess wird verdrängt bis ein anderer die Funktion **`x.signal`** aufruft.
    - **`x.signal`**
      - Aktivieren eines wartenden Prozesses



4.4.3 Dining Philosophers

Philosophen (P_i) denken und essen mit 2 Gabeln.

1 Semaphore pro Gabel




```
for (;;){
        wait(gabel[i]);
        wait(gabel[(i+1)%5]);
        essen();
        signal(gabel[i]);
        signal(gabel[(i+1)%5]);
        denken();
}
```

Problem: Alle Philosophen nehmen gleichzeitig die rechte Gabel.

➔ Verklemmung

5.1 Erzeuger/Verbraucher mit Monitor


```
Monitor ProducerConsumer
condition full, empty;
int count=0;
insert(){
        if (count==N)
                full.wait();
        enter_item(); count++;
        if (count==1)
                empty.signal();}

remove(){
        if (count==0)
                empty.wait();
        remove_item(); count--;
        if(count==N-1)
                full.signal();}
```

```
Producer
producer(){
 for(;;){
  produce_item();
  ProducerConsumer.insert();
}}
```

```
Consumer
consumer(){
 for(;;){
  ProducerConsumer.remove();
  consume_item();
}}
```

5.2 Implementierung von Monitoren mit Semaphoren

- Semaphor **`mutex = 1`** für erstmaligen Monitorzutritt
- Semaphor **`next = 0`** für Monitorzutritt nach vorübergehendem Verlassen des Monitors über **`wait`** oder **`signal`**
- Variable **`next_count = 0`** zählt an **`next`** wartende Prozesse.

- Funktion F

```
wait(mutex)
...
F;
...
if (next_count > 0)
    signal(next);
else
    signal(mutex);
```

5.3 Signal-Varianten

- Für jede Condition-Variable **`x`** wird eine Semaphor **`x`** und ein zugehöriger Zähler **`x_count`** eingeführt.
- Signal-Varianten
  1. Signalisierender Prozess bleibt im Monitorbesitz und ein wartender Prozess wird aus Warteschlange der Condition-Variable entfernt. Wartender Prozess muss sich aber erneut um Monitorzutritt (über **`next`**) bewerben.
  2. Unterschied zu 1.): Alle an Condition-Variable wartende Prozesse werden deblockiert. (für Situationen, in denen mehrere Prozesse Bedingungen für Fortführung erfüllen)
  3. Besitzwechsel: Signalisierender Prozess muss sich erneut um Monitorzutritt (über **`next`**) bewerben.

5.3.1 Signal-Variante 1

x.wait()

```
x_count++;
if (next_count > 0)
      signal(next);
else
      signal(mutex);
wait(x);
wait(next);
next_count--;
```

x.signal()

```
if (x_count > 0){
      signal(x);
      x_count--;
      next_count++;
}
```

5.3.2 Signal-Variante 2

x.wait()

```
x_count++;
if (next_count > 0)
      signal(next);
else
      signal(mutex);
wait(x);
wait(next);
next_count--;
```

x.signal()

```
while (x_count > 0){
      signal(x);
      x_count--;
      next_count++;
}
```

5.3.3 Signal-Variante 3

x.wait()

```
x_count++;
if (next_count > 0)
      signal(next);
else
      signal(mutex);

/* Prozess verlässt */
/* Monitor.         */
wait(x);

/* Prozess betritt */
/* Monitor erneut. */
x_count--;
```

x.signal()

```
if (x_count > 0){
      next_count++;
      signal(x);
      wait(next);
      next_count--;
}
```

6. Transactional Memory

- zur Unterstützung nebenläufiger Prozesse auf Multicore-Rechnern
- Memory Transaction = atomare Sequenz von Lese- und Schreiboperationen
- Falls alle Operationen erfolgreich verlaufen sind, kann die Transaktion bestätigt werden (**`commit`**). Sonst muss sie abgebrochen (**`abort`**) und der Ursprungszustand wiederhergestellt werden (**`rollback`**).
- Software Transactional Memory
  - Programmiersprachenunterstützung zur Kennzeichnung atomarer Sequenzen (z.B. **`atomic{}`**)
  - Compiler fügt Code hinzu.
- Hardware Transactional Memory
  - Cache-Hierarchien und Cache-Kohärenz-Protokolle