Nebenläufigkeit und Synchronisation

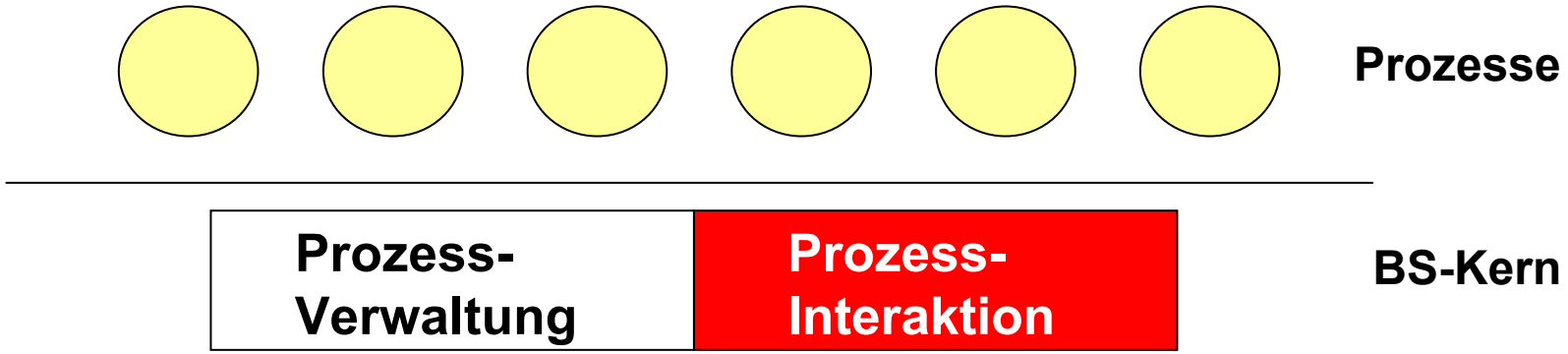
Betriebssysteme I
WS 2007/2008

Jörg Kaiser
IVS – EOS

Otto-von-Guericke-Universität Magdeburg

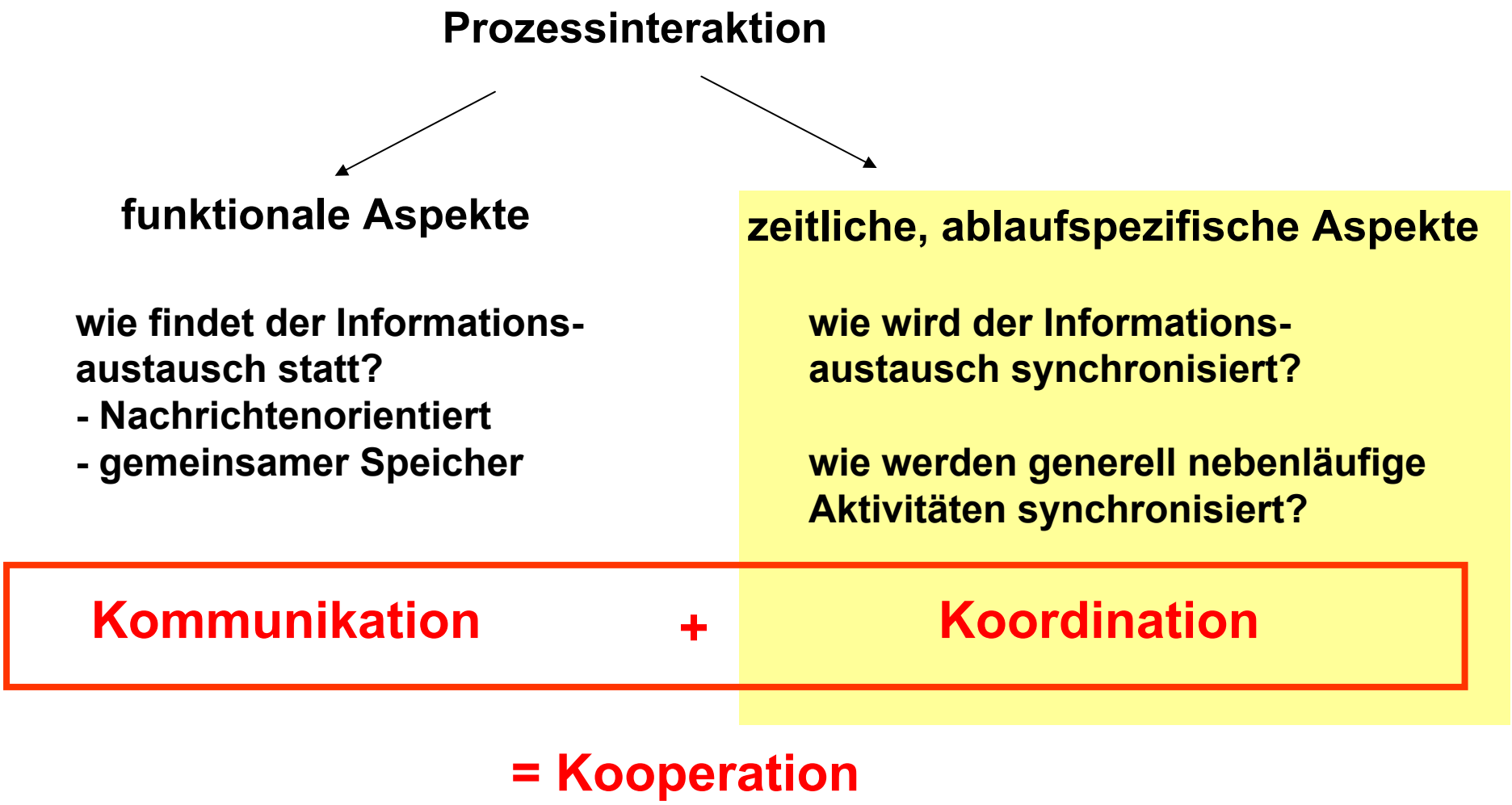
Prozessinteraktion: Kommunikation, Koordination, Kooperation

Prozesse interagieren durch:
- **expliziten Aufruf**
- **Nachrichtenaustausch**
- **Abstimmung**
- **aufeinander Warten**



Prozessinteraktion ist wesentliche Aufgabe des BS-Kerns

Prozessinteraktion: Kommunikation, Koordination, Kooperation



Koordination

- ☆ **Die Koordination nebenläufiger Aktivitäten ist eine elementare (und schwierige) Aufgabe des Betriebssystems.**
- ☆ **Koordination findet auf unterschiedlichen Ebenen statt.**
- ☆ **Koordination ist der Mechanismus der garantiert, dass keine Effekte auftreten, die nur durch die Nebenläufigkeit hervorgerufen werden und bei einer streng sequentiellen Abarbeitung nicht auftreten.**

Problemformulierung

Das Problem: Nebenläufigkeit + Nutzung gemeinsamer Daten, d.h. die Prozesse interagieren miteinander. Asynchronität durch mehrere Prozessoren oder durch systembezogene Unterbrechungen.

Gefordertes Verhalten: Die Ergebnisse eines Prozesses müssen unabhängig von der eigenen Ausführungsgeschwindigkeit und der anderer nebenläufiger Prozesse sein.

Mechanismus: Synchronisation

Möglichkeiten der Interaktion

Konkurrenz

Prozesse agieren ohne Wissen um die Existenz anderer Prozesse.
- Keine Kooperation, d.h. Ergebnisse sind unabhängig.
- Konkurrenz im Hinblick auf Ressourcen.

indirekte Kooperation

Prozesse nutzen gemeinsame Ressourcen.
- Ergebnisse hängen von anderen Prozessen ab.
- Geordneter Zugriff auf gemeinsame Resource notwendig.

direkte Kooperation

Prozesse kommunizieren explizit miteinander.
- Kooperation durch explizite Kommunikation.
- Ergebnisse hängen von anderen Prozessen ab.

Konkurrenz um Ressourcen

Prozesse nutzen gemeinsame Ressourcen ohne das zu wissen.

Es findet keine Kommunikation statt.

Trotzdem besteht eine Wechselwirkung.

Das BS teilt die Ressource einem Prozess zu, der andere muss warten.

Konsistenz der Daten muss gewährleistet sein.

Wechselseitiger Ausschluss (Mutual Exclusion)

Ist ein Read-Modify-Write Zyklus ausreichend?

Bezieht sich nur auf die Ausführung einer einzigen unteilbaren Instruktion, die ein Speicherwort liest und aktualisiert!

Beispiele: Test and Set, Compare and Swap

Kooperation durch gemeinsame Ressourcen (Sharing)

Buchführungsprogramm:
Verhältnis zwischen a und b soll aufrecht erhalten werden.

P1:

```
a = a + 1
b = b + 1
```

P2:

```
b = 2 * b
a = 2 * a
```

Ausführungsreihenfolgen:

```
a = a + 1
b = b + 1
b = 2 * b
a = 2 * a
```

```
b = 2 * b
a = 2 * a
a = a + 1
b = b + 1
```

.....

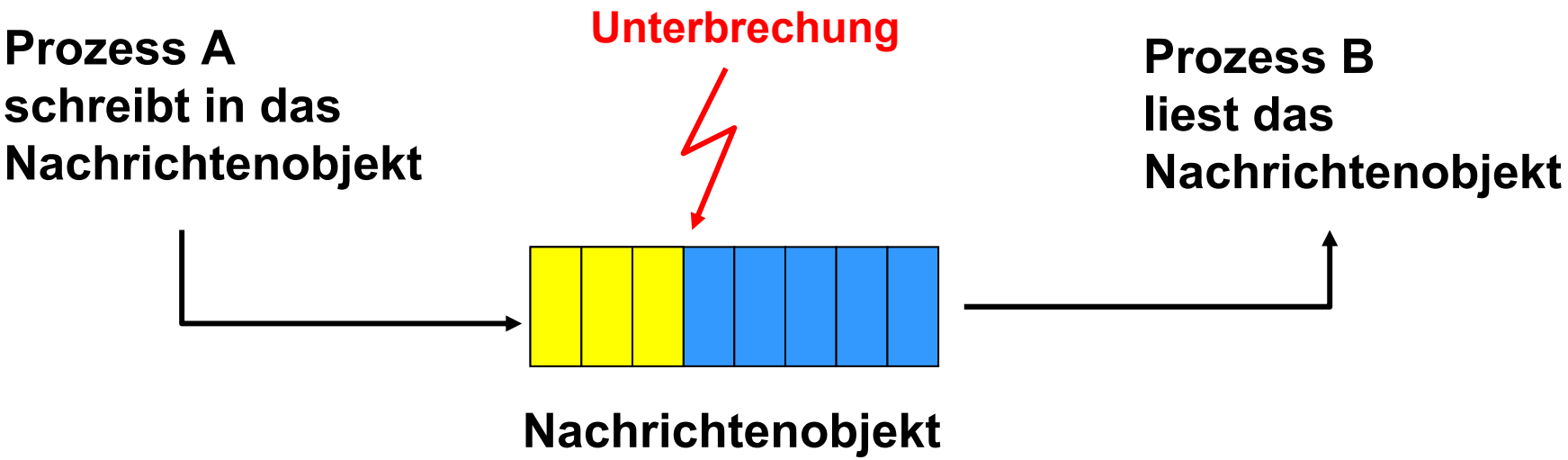
```
a = a + 1
b = 2 * b
b = b + 1
a = 2 * a
```

Für jede Variable wird der wechselseitige Ausschluss befolgt. Trotzdem entspricht das Ergebnis keinem Ergebnis, das ohne Nebenläufigkeit erzielt werden kann!

Konsistenz einer Ausführung

Das Ergebnis einer Ausführung von nebenläufigen, unabhängigen Prozessen ist konsistent, wenn es durch irgendeine strikt sequentielle Ausführung der Prozesse erreicht wird.

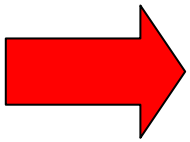
Kooperation durch Kommunikation



Auch bei expliziter Kommunikation ist eine Synchronisation erforderlich !

Wechselseitiger Ausschluss

Forderung: Garantieren des exklusiven Zugriffs für längere Befehlssequenzen.



Konzept des kritischen Abschnitts

Kritischer Abschnitt

```
/* Programm Wechselseitiger Ausschluss */
const int n     /* Anzahl der Prozesse */

void P(int i)
{
      while (true)
      {
            entercritical (i);        /* Prolog zur Steuerung der Zugangskontrolle */
            /* kritischer Abschnitt */;
            exitcritical (i);         /*Epilog zur Steuereung der Zugangskontrolle*/
            / * Restliche Programmzeilen * /;
      }
}
void main ( )
{
      parbegin (P1, P2, ...., Pn);
}
```

Wechselseitiger Ausschluss

Voraussetzungen:

- Es werden keine Annahmen über die relative Ausführungsgeschwindigkeit der Prozesse, ihre Anzahl oder die Anzahl der Prozessoren gemacht.
- Nur EIN Prozess darf in den Kritischen Abschnitt (KA) eintreten.
- Ein Prozess bleibt nur endliche Zeit in einem KA.
- Wenn sich kein anderer Prozess im KA befindet, muss der Zugang ohne Verzögerung gewährt werden.

Software-Ansätze: Wechselseitiger Ausschluss wird ohne Unterstützung durch eine Programmiersprache oder das BS durchgesetzt.

1. Versuch

Annahme: ein einzelner Speicherzugriff ist unteilbar.

gemeinsam genutzte globale Variable: int turn = 0;

```
/* Prozess 0 */
.
.
while  (turn !=0)
      */ do nothing*/;
/* critical section */;
turn = 1;
.
.
```

```
/* Prozess 1 */
.
.
while  (turn !=1)
      */ do nothing*/;
/* critical section */;
turn = 0;
.
.
```

Code zur Steuerung der Variablen zur Zugangskontrolle gehören mit zur "kritischen" Software, die nicht ausfallen darf.

Prinzip: Freiwillige Abgabe der Kontrolle über den kritischen Abschnitt.

Probleme:

- Aktives Warten,
- Streng alternierend; langsamer Prozess bestimmt die Periode.
- Bei Ausfall eines Prozesses (auch außerhalb der kritischen Abschnitts) kann der kritische Abschnitt nicht mehr betreten werden.

2. Versuch

gemeinsam benutzte globale Variable*:

enum boolean {false = 0; true = 1}
boolean flag[2] = {false, false}

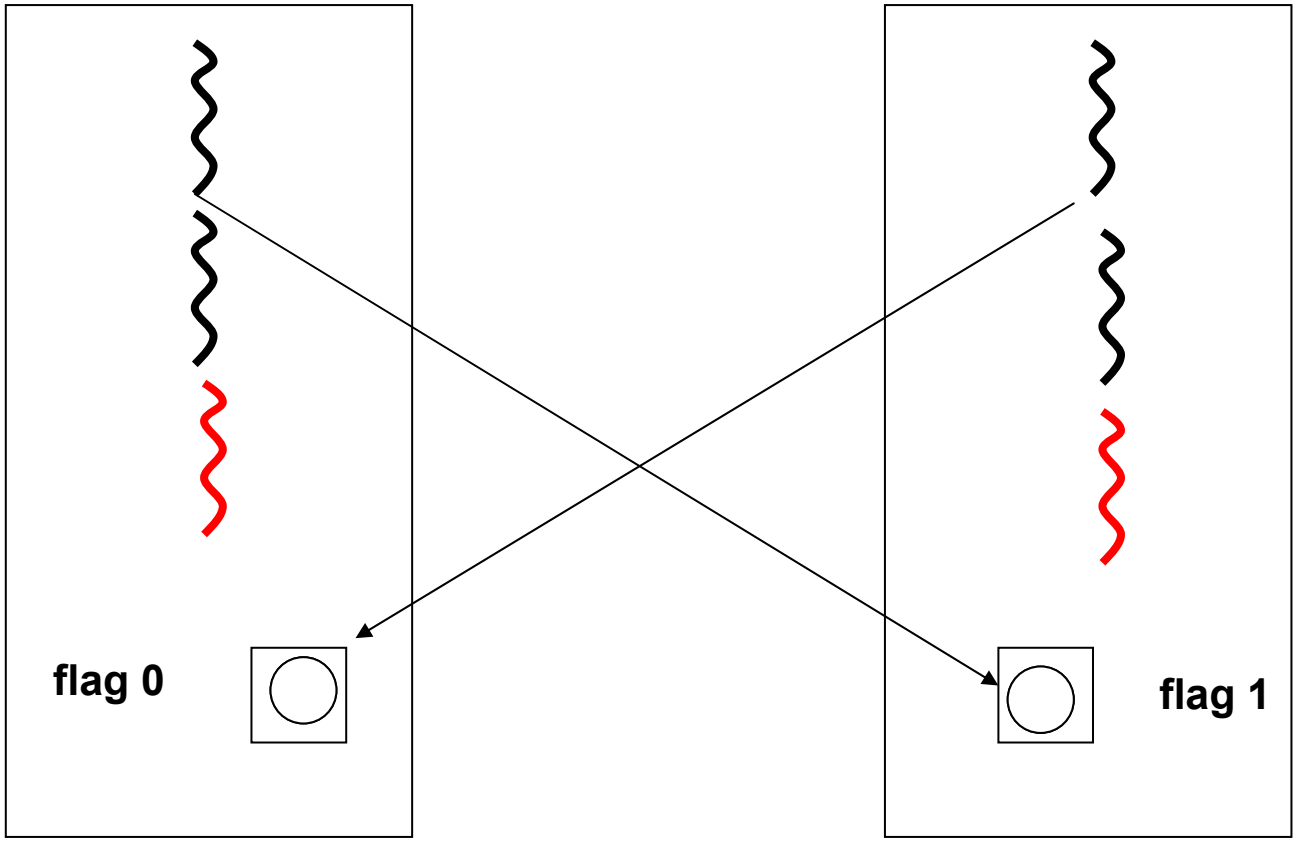
```
/* Prozess 0 */
.
.
while  (flag[1])
      */ do nothing*/;
flag[0] = true;
/* critical section */;
flag[0] = false;
.
.
```

```
/* Prozess 1 */
.
.
while  (flag[0])
      */ do nothing*/;
flag[1] = true;
/* critical section */;
flag[1] = false;
.
.
```

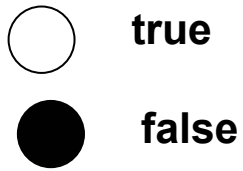
Probleme: ➡ **Kein wechselseitiger Ausschluss gewährleistet!**

*** wird häufig auch als Schlossvariable (lock variable) bezeichnet**

2. Versuch




```
/* Prozess 0/1 */
.
.
while  (flag[1/0])
      */ do nothing*/;
flag[0/1] = true;
/* critical section */;
flag[0/1] = false;
.
.
```



Flag des anderen Prozesses lesen und das eigene ändern geschieht nicht atomar!

3. Versuch

gemeinsam benutzte globale Variable:

enum boolean {false = 0; true = 1}
boolean flag[2] = {false, false}

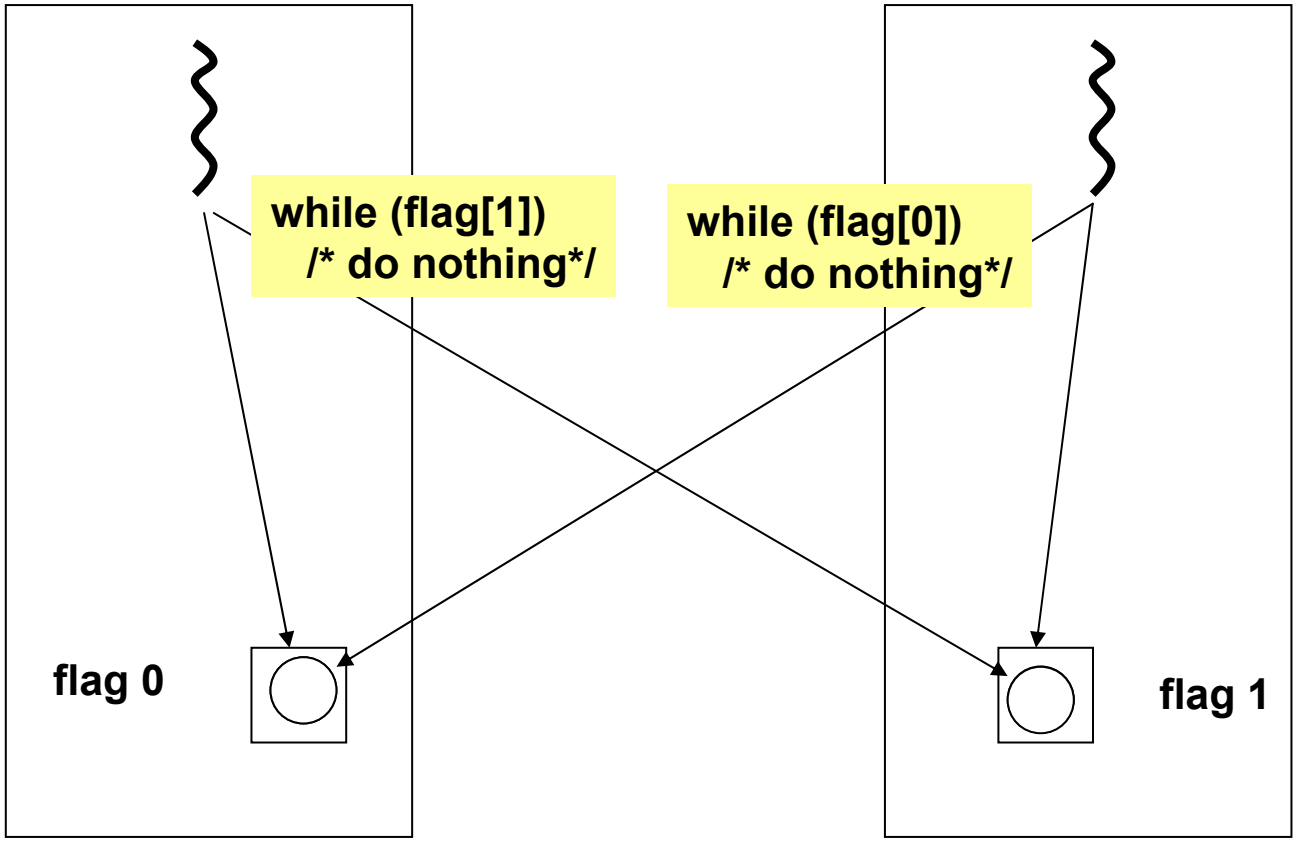
```
/* Prozess 0 */
.
.
flag[0] = true;
while  (flag[1])
      */ do nothing*/;
/* critical section */;
flag[0] = false;
.
.
```

```
/* Prozess 1 */
.
.
flag[1] = true;
while  (flag[0])
      */ do nothing*/;
/* critical section */;
flag[1] = false;
.
.
```

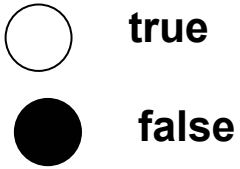
Probleme:

- **Bei Ausfall innerhalb des kritischen Abschnitts: Blockade.**
- **Falls beide Prozesse die Flags auf "true" setzen: Deadlock (Verklemmung).**

3. Versuch




```
/* Prozess 0/1 */
.
.
flag[0/1] = true;
while  (flag[1/0])
      */ do nothing*/;
/* critical section */;
flag[0/1] = false;
.
.
```



4. Versuch

gemeinsam benutzte globale Variable:

enum boolean {false = 0; true = 1}
boolean flag[2] = {false, false}

```
/* Prozess 0 */
.
.
flag[0] = true;
while  (flag[1])
{
      flag[0] = false;
      /* delay */;
      flag[0] = true;
}
/* critical section */;
flag[0] = false;
.
```

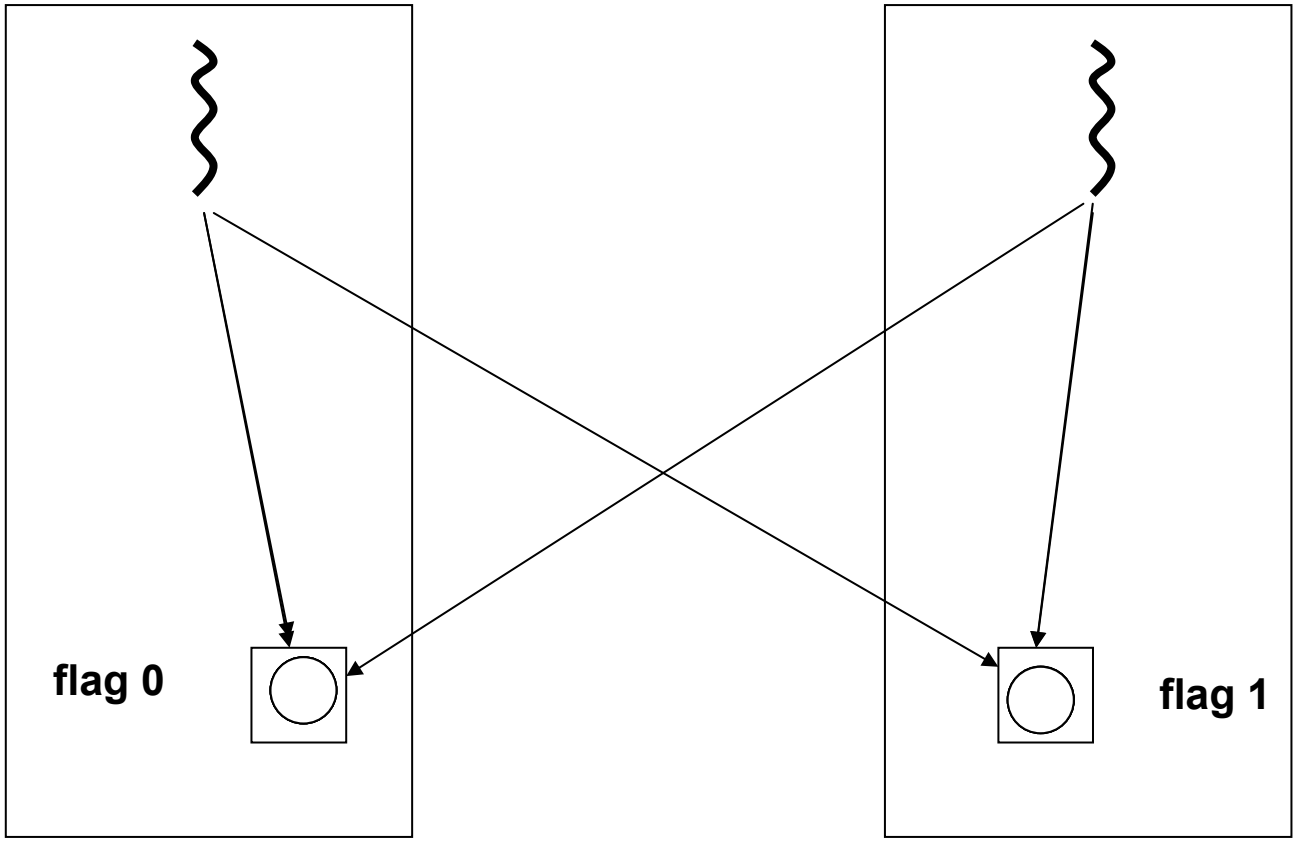
```
/* Prozess 1 */
.
.
flag[1] = true;
while  (flag[0])
{
      flag[1] = false;
      /* delay */;
      flag[1] = true;
}
/* critical section */;
flag[1] = false;
.
```

Versuch der zufälligen Auflösung des Deadlocks

Probleme:

- **Bei Ausfall innerhalb des kritischen Abschnitts: Blockade.**
- **Falls kein Prozesse "nachgibt": Lifelock (Dynamische Verklemmung).**

4. Versuch




```
/* Prozess 0/1 */
.
.
flag[0/1] = true;
while  (flag[1/0])
{
     flag[0/1] = false;
     /* delay */;
     flag[0/1] = true;
}
/* critical section */;
flag[0/1] = false;
.
```

true

false

Aus E.W.Dijkstra's Arbeit (1965):

Quite a collection of trial solutions have been shown to be incorrect and at some moment people that had played with the problem started to doubt whether it could be solved at all. To the Dutch mathematician Th.J.Dekker the credit is due for the first correct solution. It is, in fact, a mixture of our previous efforts: it uses the "safe sluice" of our last constructions, together with the integer "turn" of the first one, but only to resolve the indeterminateness when neither of the two immediately succeeds. The initial value of "turn" could have been 2 as well.

```
"begin  integer c1, c2, turn;
        c1:= 1; c2:= 1; turn:= 1;
        parbegin
        process 1: begin A1: c1:= 0;
                        L1: if  c2 = 0  then
                                begin  if  turn = 1  then  goto  L1;
                                       c1:= 1;
                                   B1: if  turn = 2  then  goto  B1;
                                       goto  A1
                                end;
                            critical section 1;
                            turn:= 2; c1:= 1;
                            remainder of cycle 1; goto A1
                   end;
        process 2: begin A2: c2:= 0;
                        L2: if  c1 = 0  then
                                begin  if  turn = 2  then  goto  L2;
                                       c2:= 1;
                                   B2: if  turn = 1  then  goto  B2;
                                       goto  A2
                                end;


                                critical section 2;
                                turn:= 1; c2:= 1;
                                remainder of cycle 2; goto  A2
                            end
                parend
        end"    .
```

Dekker's Algorithmus

Sync. Code für P0

setze Flag 0 auf true
überprüfe Flag 1
if flag 1 = false
tritt in den Kritischen Abschnitt ein

if Flag 1 = true
überprüfe turn
if turn = 1 --> setze Flag 0 auf false **und**
gehe zum Anfang

sonst (turn = 0) überprüfe Flag 1
if Flag 1=false
tritt in den Kritischen Abschnitt ein

Kritischer Abschnitt; führe code aus

setze turn= 1
setze Flag 0 auf false

Sync. Code für P1

setze Flag 1 auf true
überprüfe Flag 0
if Flag 0 = false
tritt in den Kritischen Abschnitt ein

if Flag 0 = true
überprüfe turn
If turn = 0 --> setze Flag 1 auf false und
gehe zum Anfang

sonst (turn = 1) überprüfe Flag 0
if Flag 0=false
tritt in den Kritischen Abschnitt ein

Kritischer Abschnitt; führe code aus

setze turn= 0
setze Flag 1 auf false

Dekker's Algorithmus

```
boolean flag [2];
int turn;
```

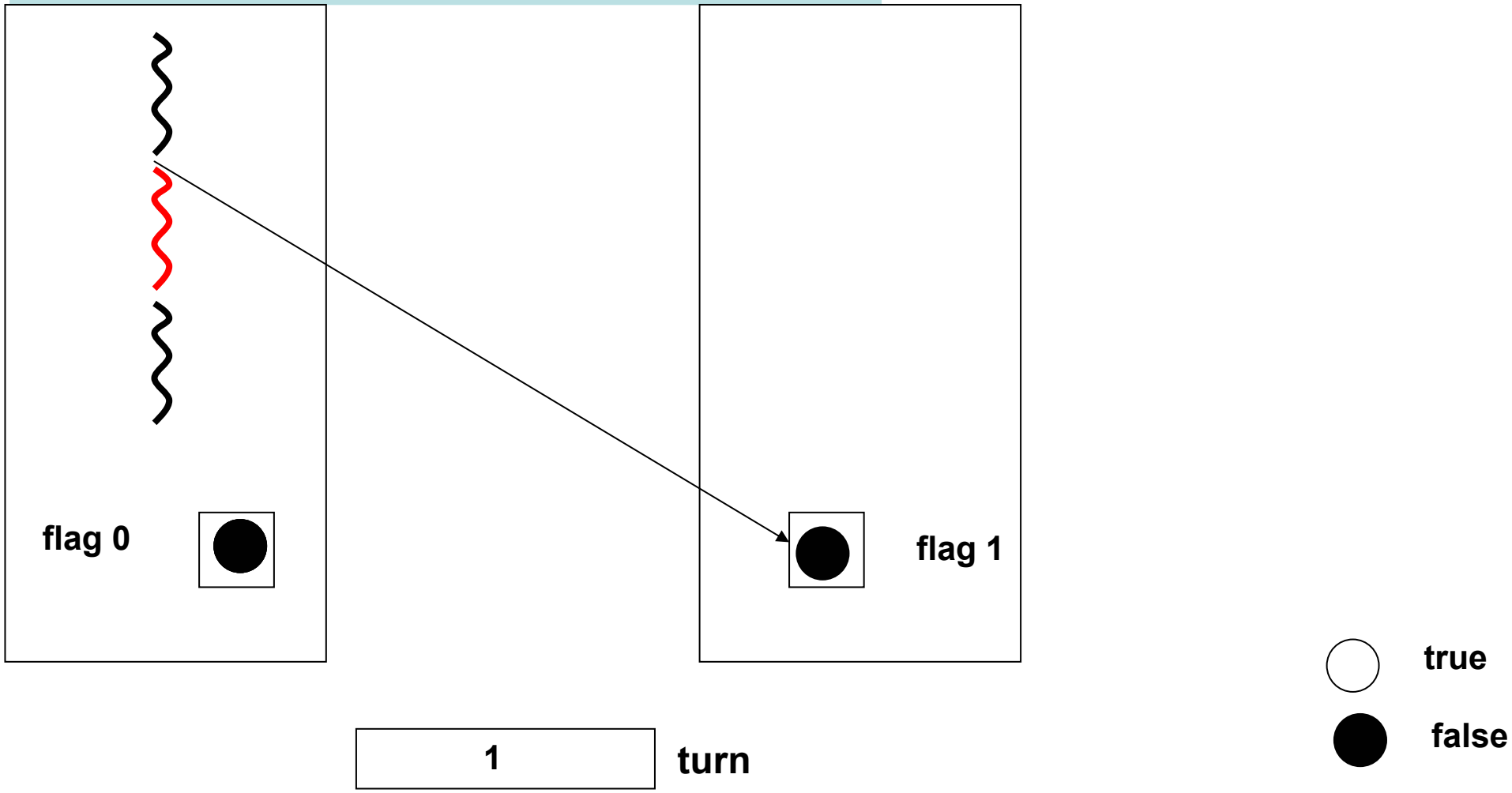
```
void P0()
{
    while (true)
    {
        flag [0] = true;
        while (flag [1])
            if (turn == 1)
            {
                flag [0] = false;
                while (turn == 1)
                    / * do nothing */
                flag [0] = true;
            }
        /* kritischer Abschnitt
        turn = 1;
        flag [0] = false;
        / * Restliche Programmzeilen * /;
    }
}
```

wird ausgeführt wenn flag [1] == false

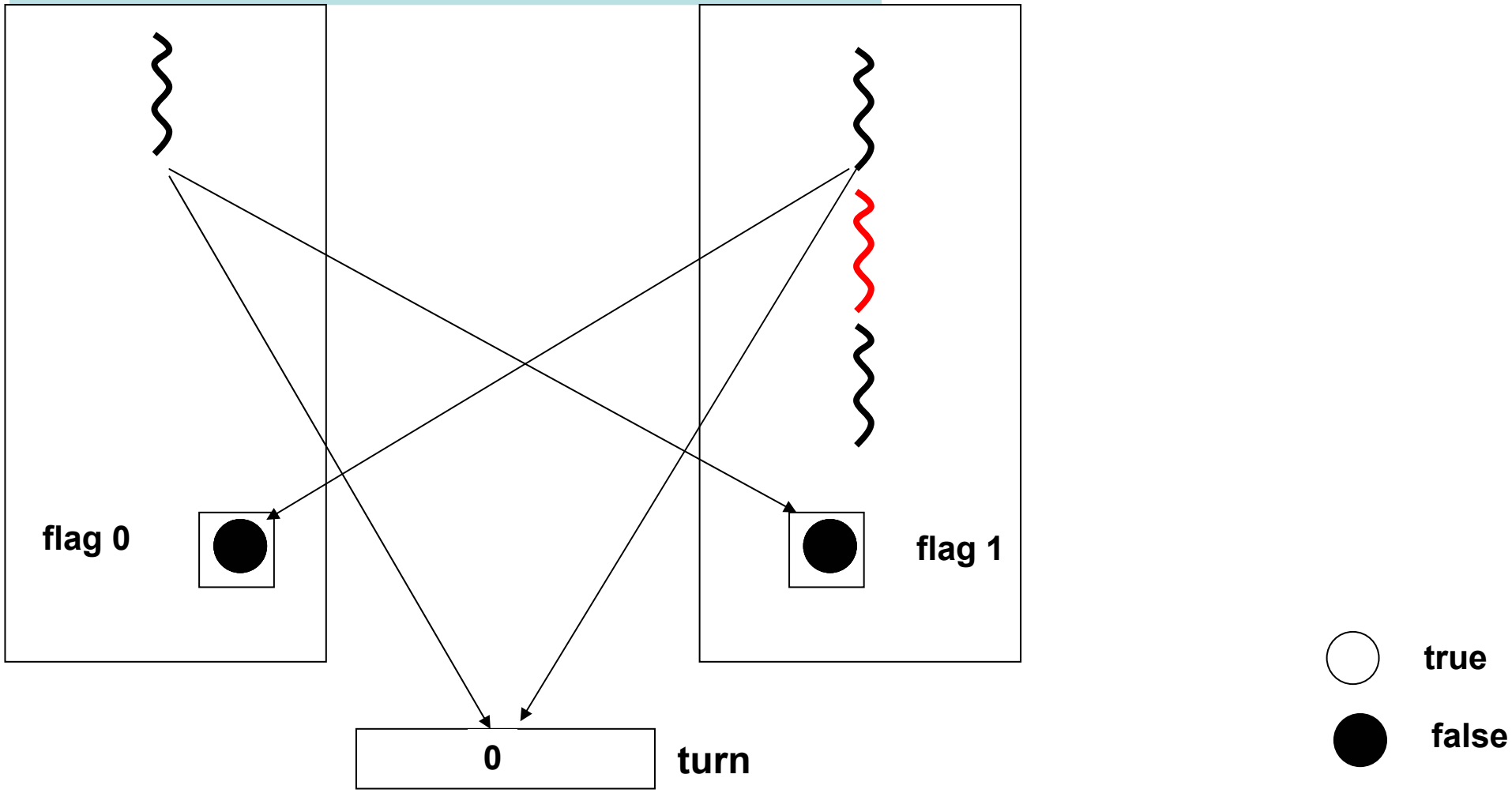
```
void P1()
{
    while (true)
    {
        flag [1] = true;
        while (flag [0])
            if (turn == 0)
            {
                flag [1] = false;
                while (turn == 0)
                    / * do nothing */
                flag [1] = true;
            }
        / * kritischer Abschnitt
        turn = 0;
        flag [1] = false;
        / * Restliche Programmzeilen
    }
}
```

```
void main 0
{
flag [0] = false;
flag [1] = false;
turn = 1;
parbegin (P0, Pl);
}
```

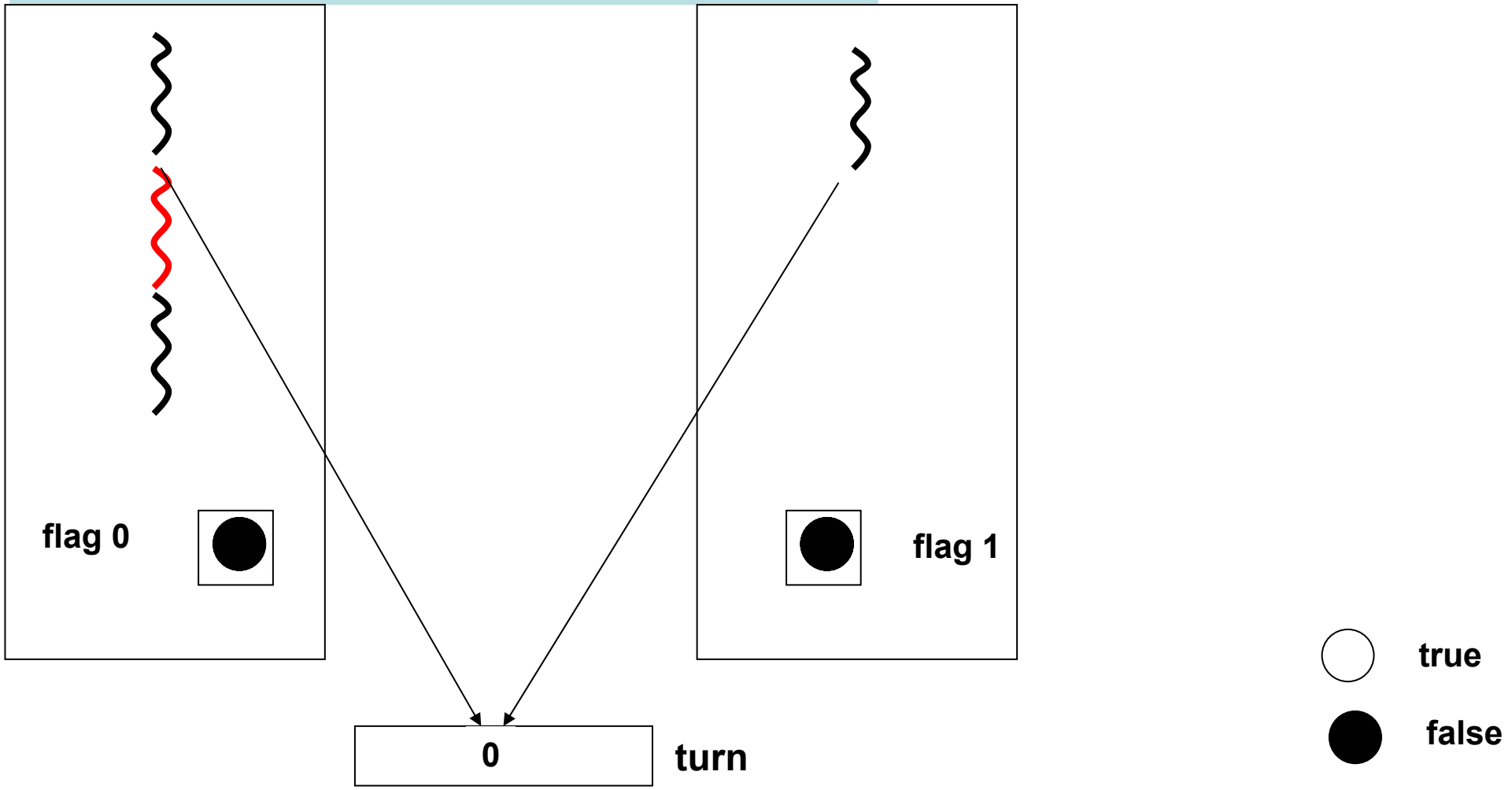
Dekker's Algorithmus



Dekker's Algorithmus



Dekker's Algorithmus



Peterson's Algorithmus

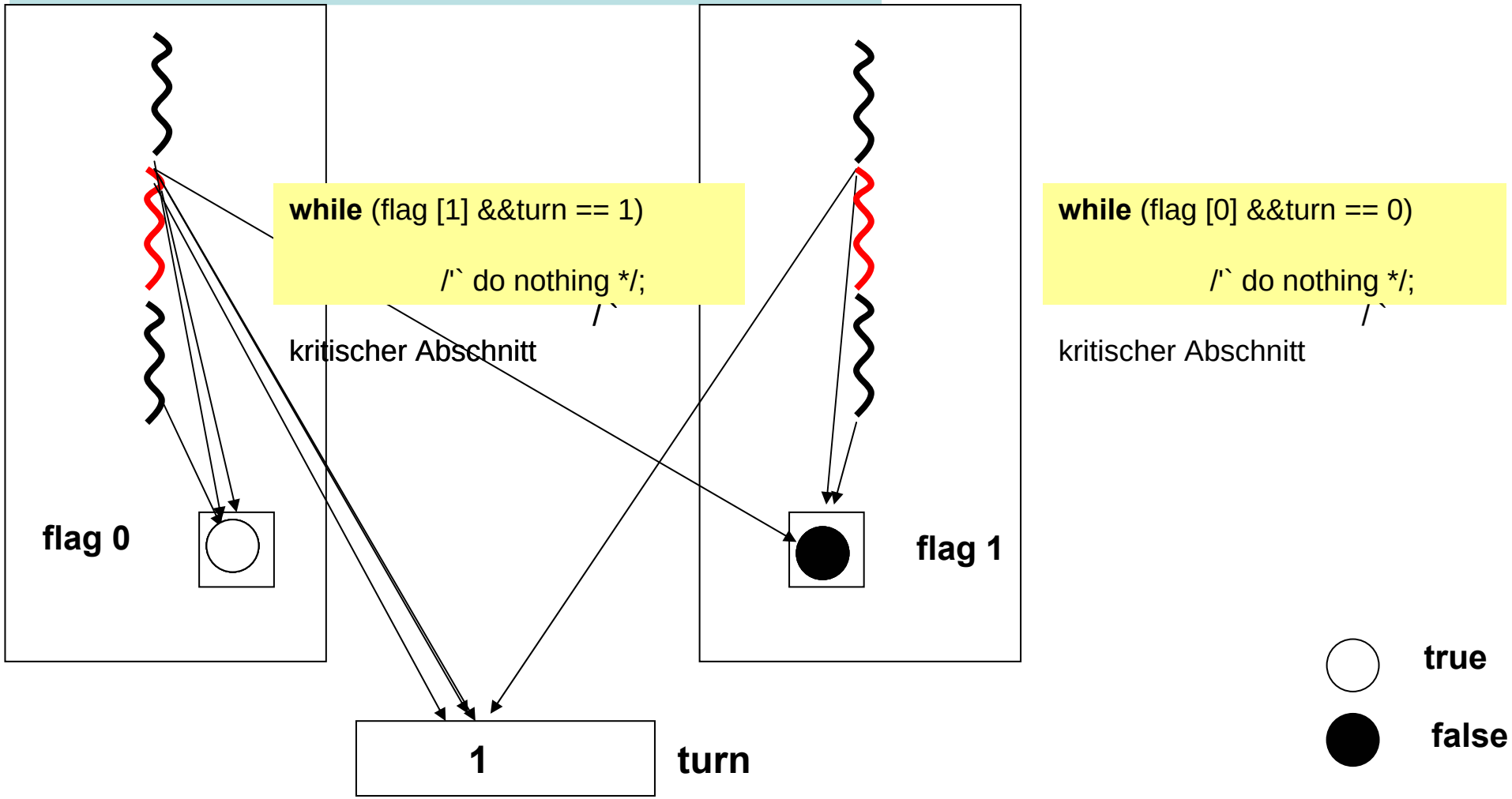
```
boolean flag [2];
int turn;
```

```
void P0()
{
      while(true)
                {
            flag [0] = true;
            turn = 1;
            while (flag [1] &&turn ==1)
                        / * do nothing * /;
                /* kritischer Abschnitt */
            flag [0] = false;
            /" Restliche Programmzeilen */;
                }
             }
```

```
void P1()
 {
       while (true)
       {
            flag [1] = true;
            turn = 0;
            while (flag [0] &&turn == 0)
                              /'` do nothing */;
            / ` kritischer Abschnitt
            flag [1] = false;
            / * Restliche Programmzeilen
       }
}
```

```
void main() t
flag [0] = false; flag [1] = false;
parbegin (P0, P1);
}
```

Peterson's Algorithmus



Vergleich von Dekker's und Peterson's Algorithmus

```
void P0()
{
        while (true)
        {
                flag [0] = true;
                while (flag [1])
                        if (turn == 1)
                        {
                                flag [0] = false;
                                while (turn == 1)
                                        / * do nothing */
                                flag [0] = true;
                        }
                /* kritischer Abschnitt
                turn = 1;
                flag [0] = false;
                / * Restliche Programmzeilen * /;
        }
}
```

```
void PO()
{
        while(true)
        {
                flag [0] = true;
                turn = 1;
                while (flag [1] && turn ==1)
                        / * do nothing * /;
                /* kritischer Abschnitt */
                flag [0] = false;
                /" Restliche Programmzeilen */;
        }
      }
```

Peterson's Algorithmus

- eleganter als Dekker's Algorithmus
- wechselseitiger Ausschluss ist gewährleistet
- gegenseitige Blockierung und Monopolisierung des kritischen Abschnitts durch einen Prozess wird verhindert.

Problem: Erweiterung auf n Prozesse

Wechselseitiger Ausschluss mit Hardwareunterstützung

Befehl: Test and Set:

```
boolean testset (int i)
{
        if (i==0)
        {
                i= 1;
                return true;
        }
        else
        {
                return false;
        }
}
```

```
/* program mutualexclusion */

const int n =  /* number of processes*/
int bolt;
void P(int i)
{
        while (true)
        {
                while (!testset (bolt))
                        /* do nothing */;
                /* critical section*/
                bolt = 0;
                /* remainder*/
        }

void main()
{
                bolt = 0;
                parbegin (P(1), P(2),. . . ,P(n));
}
```

Was wurde bisher erreicht?

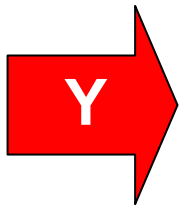
Ziel: Erhaltung der Konsistenz von Daten trotz nebeläufigen Zugriffs, sei es durch Multiprogramming oder Multiprocessing.

Zur Manipulation der Daten müssen Befehlssequenzen (und nicht nur einzelne Befehle) atomar ausgeführt werden.

Die Algorithmen von Dekker und Peterson bieten korrekte Lösungen.

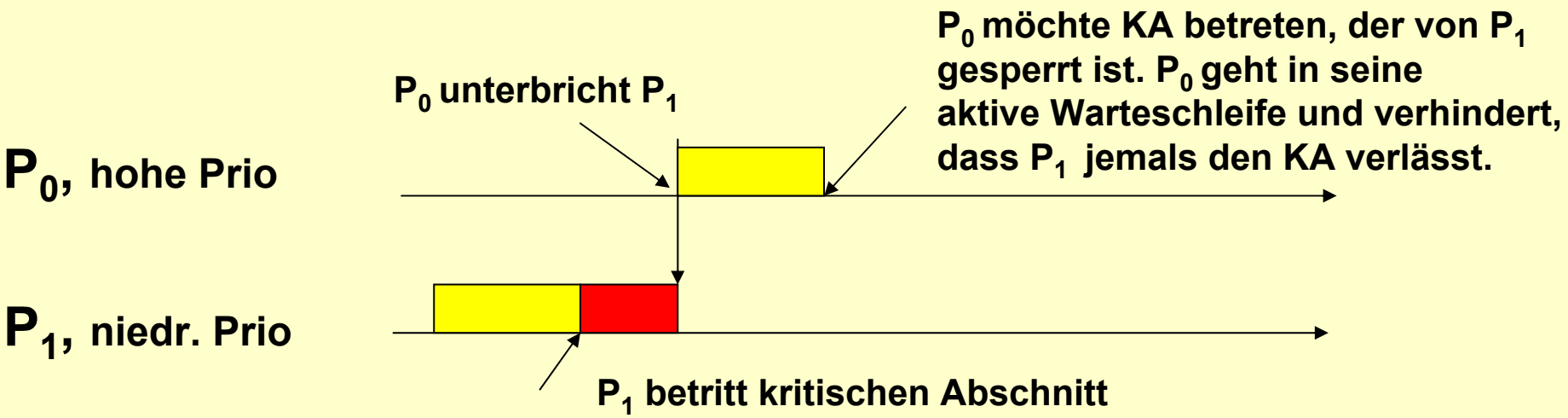
Hardwareunterstützung vereinfacht und verbessert die reinen Softwarelösungen.

Probleme ?



Probleme mit den bisherigen Lösungen

- Aktives Warten
- Ein vollst. auf Anwendungsebene konzipierter Mechanismus
- Keine Garantie gegen Verhungern
- Verklemmungen können auftreten



Was soll erreicht werden?

- Einbettung des Konzepts in das Betriebssystem oder (zumindest) in die Programmiersprache.
- Allgemeiner Mechanismus für die "Kooperation sequentieller Prozesse"
- Kein aktives Warten mehr.
- Leicht zu verstehen und zu handhaben

- **Semaphore (1965, Edsger W. Dijkstra, 1930–2002)**

- **Monitore (1974, Sir Charles Antony Richard (C.A.R.) Hoare, 1934)**

Semaphore

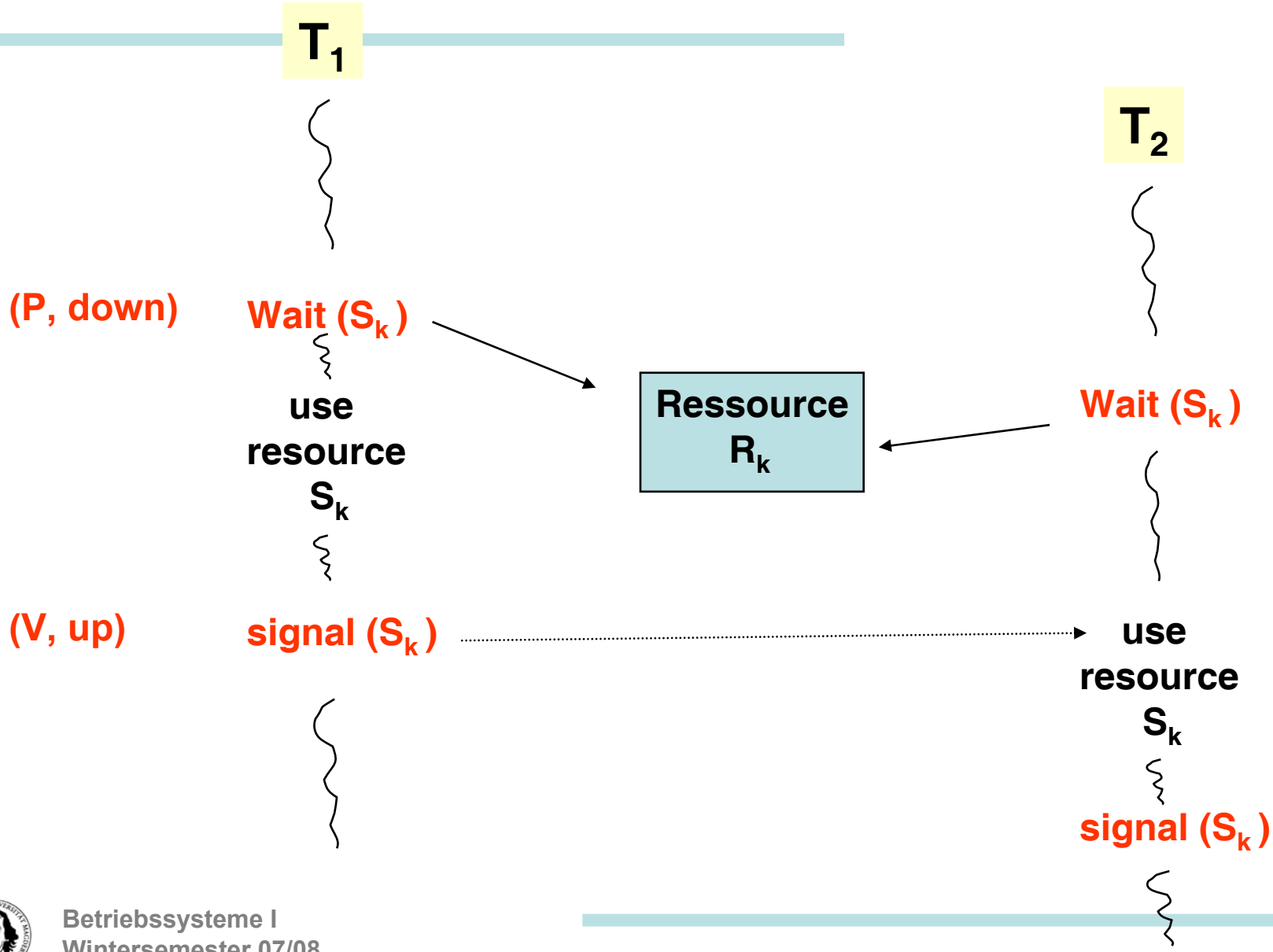
"Semaphor" aus dem Griech. von sema (Zeichen) und pherein (tragen)

Winkeralphabeth

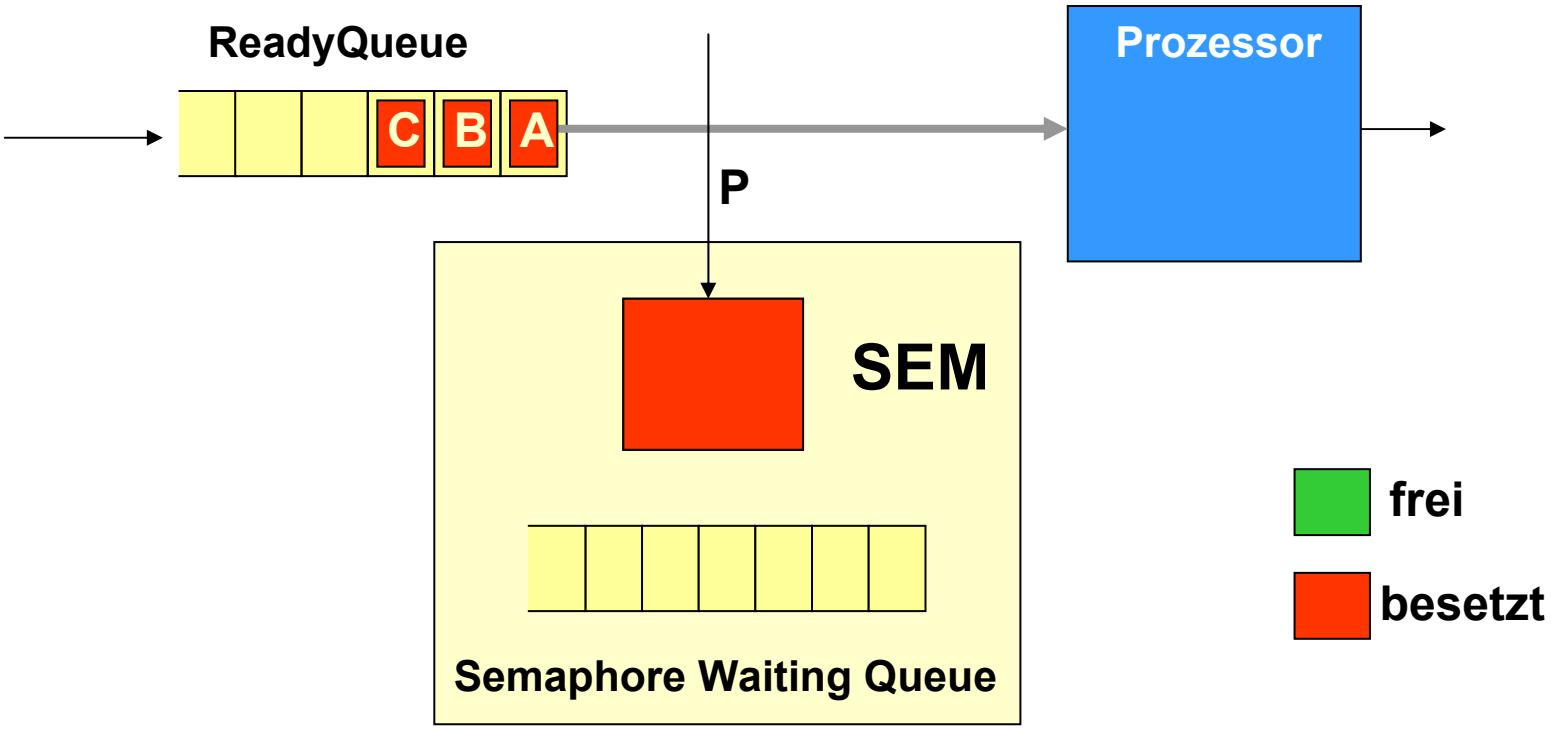
Tour du telegraphe Chappe Saverne

Formsignale Eisenbahn

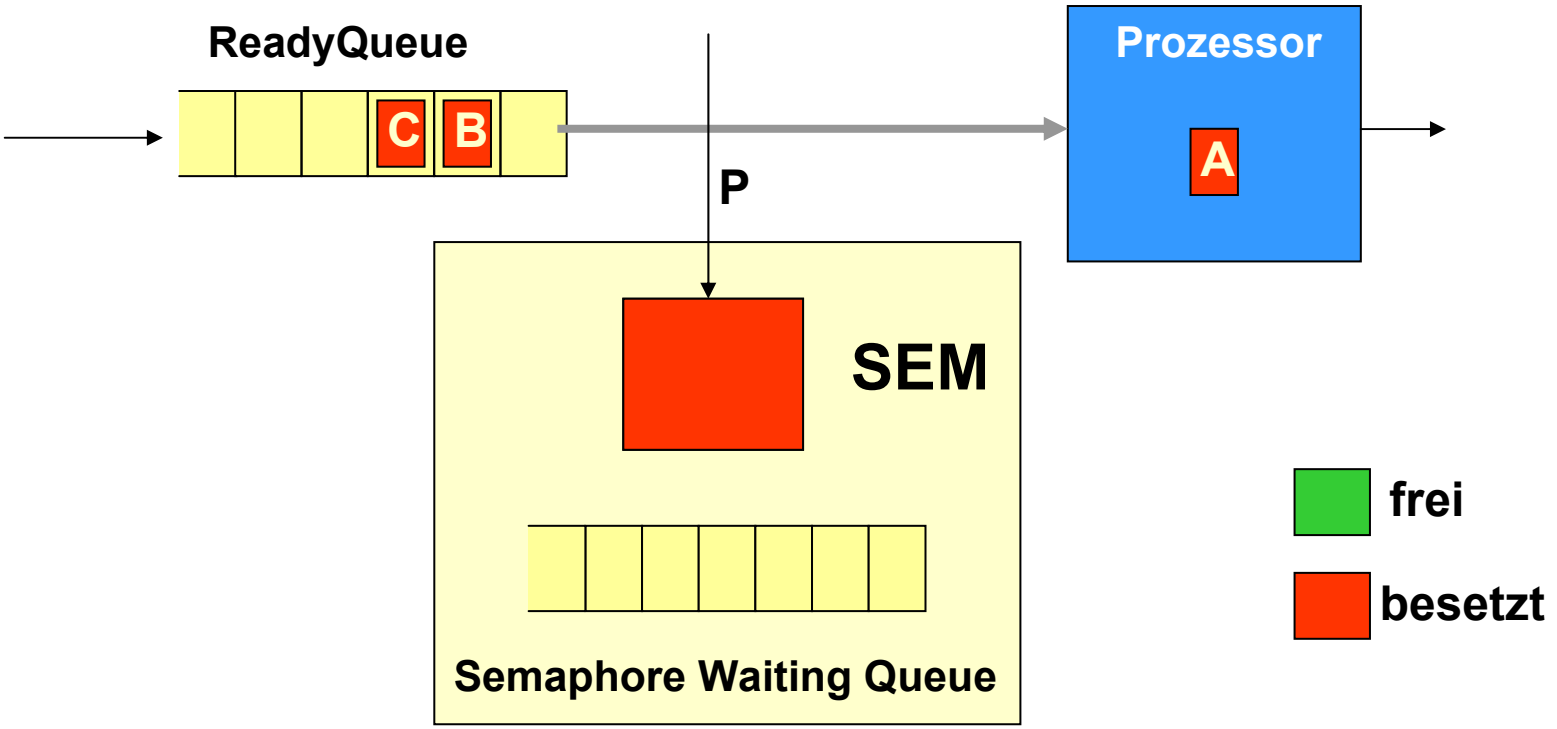
Semaphore


T1
T2
(P, down)
Wait (Sk)
use resource Sk
Ressource Rk
Wait (Sk)
(V, up)
signal (Sk)
use resource Sk
signal (Sk)

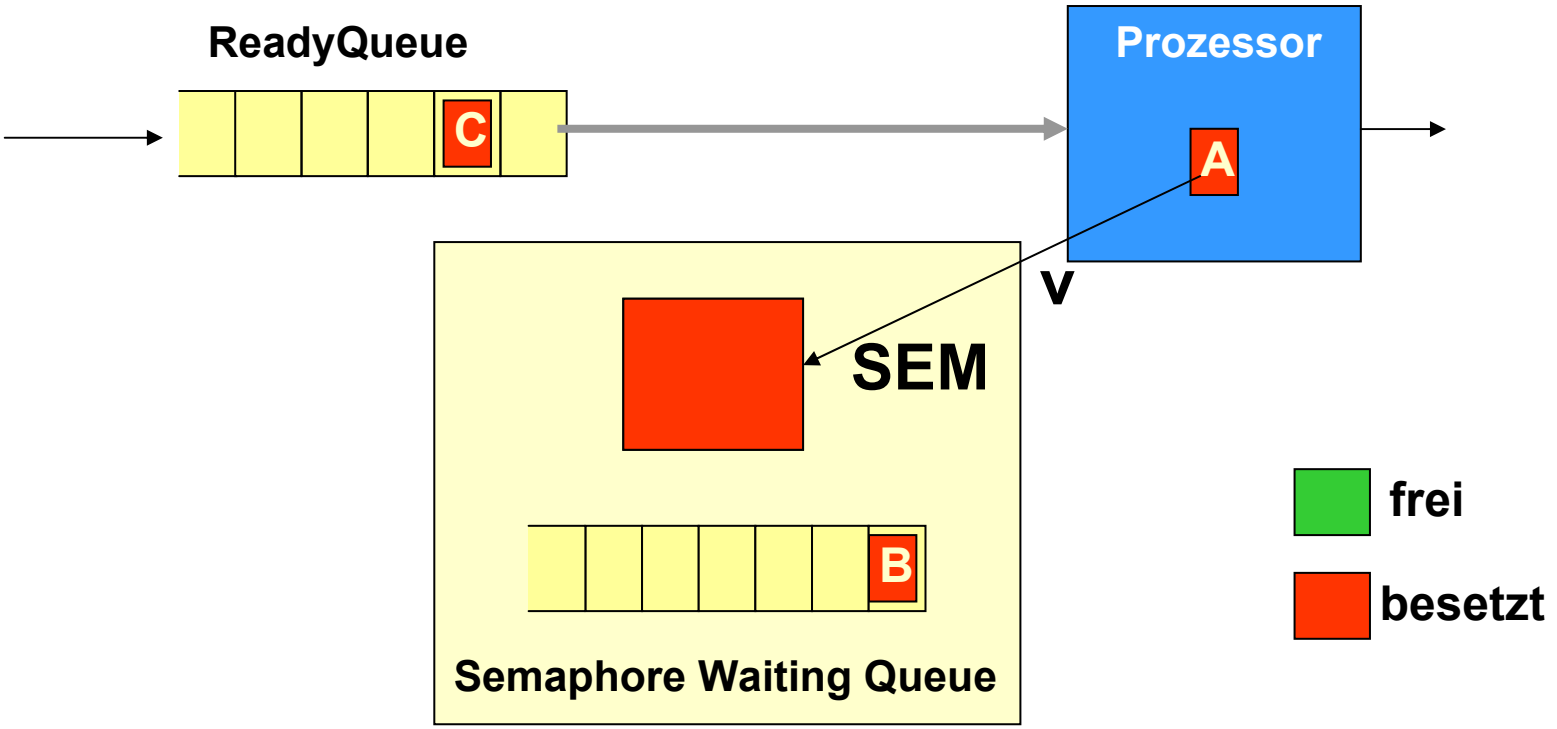
Semaphore


ReadyQueue
C
B
A
Prozessor
P
SEM
Semaphore Waiting Queue
frei
besetzt

Semaphore


ReadyQueue
C
B
P
Prozessor
A
SEM
Semaphore Waiting Queue
frei
besetzt

Semaphore


ReadyQueue
C
Prozessor
A
V
SEM
B
Semaphore Waiting Queue
frei
besetzt

Semaphore

Ein Semaphor ist eine gemeinsam benutzte Variable, der eine Warteschlange zugeordnet ist. Folgende Operationen sind auf einem Semaphor definiert:

	Beisp.
Initialisierung mit einem Wert ≥ 0:	**S=1**
wait (P): verringert den Wert eines Semaphors. Ist S < 0, wird der Prozess blockiert.	**S= S-1**
signal (V): erhöht den Wert des Semaphors. Ist S ≤ 0, wird ein durch wait blockierter Prozess freigegeben.	**S= S+1**

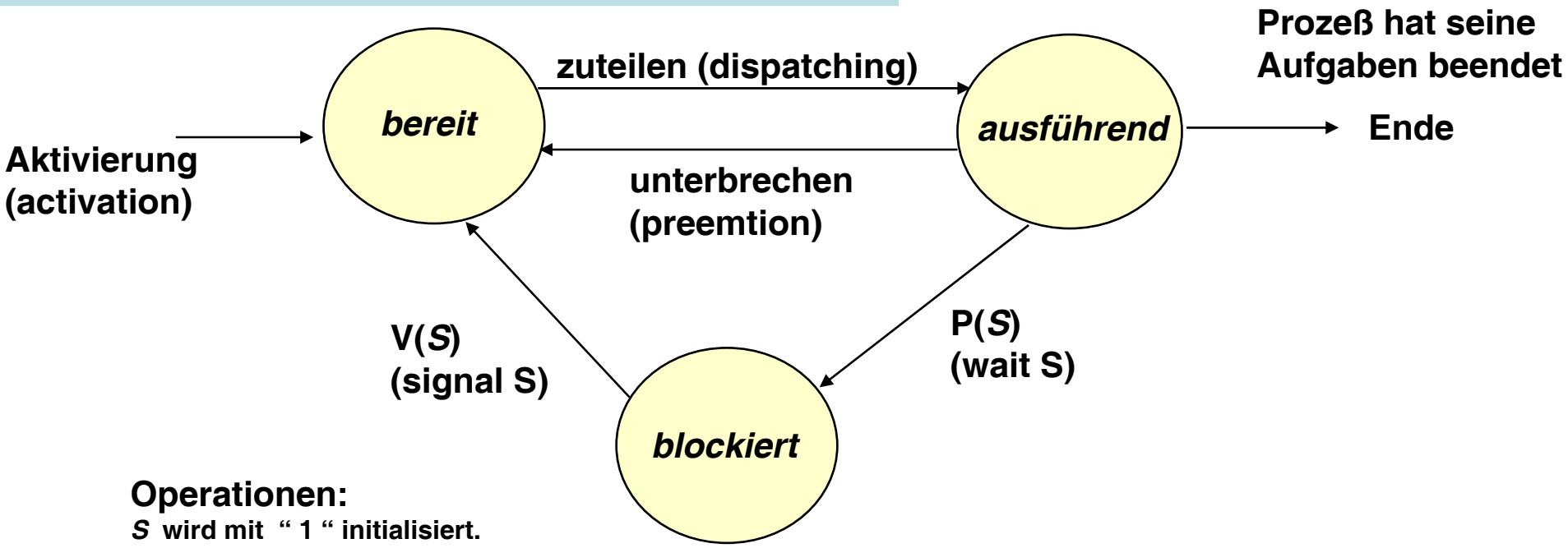
- **signal und wait sind atomare Operationen.**
  Wert eines Semaphors:
- **positiver Wert: Anzahl der zur Verfügung stehenden Betriebsmittel.**
- **negativer Wert: Anzahl der wartenden Prozesse.**

Semaphor-Varianten

Binäre Semaphore: Genau 1 Prozess darf in den KA eintreten. Alle anderen Prozesse werden in eine WS eingeordnet. Semaphor wird mit 1 initialisiert.

Zählende Semaphore: Mehrere Prozesse dürfen in einen KA eintreten. Semaphor wird mit n initialisiert. Wird n überschritten, werden Prozesse in eine WS eingeordnet.

Binäres Semaphor



Operationen:
S **wird mit " 1 " initialisiert.**

wait(S)

```
P(S) {
     S = S - 1;
     if (S < 0)  blockiere P;  /* Prozess wird in den Zustand blockiert überführt und in die S- Warteschlange
     }                         /* eingereiht falls S<0
```

signal(S)

```
V(S) {
     S = S + 1;
     if (S ≤ 0) wecke P;      /* Prozess, der am Anfang der S-Warteschlenge steht
     }                        /* kann in den KA eintreten. Falls S > 0, kein Prozess in der WS.
```

Semaphore

Wechselseitiger Ausschluss durch Verwendung von Semaphoren

```
/* Programm WechselseitigerAusschluss */
const int n = /* Anzahl der Prozesse */;
semaphore s = 1;
void P(int i)
{
        while (true)
            {
            wait(s);
            / * kritischer Abschnitt' /;
            signal(s);
            /* Restliche Programmzeilen */;
            }
}
void main ()
{
        parbegin (P(1), P(2),..., P(n));
}
```

Semaphore

Primitive für allgemeines Semaphor

```
struct semaphore
{
    int count;
    queueType queue;
}

void wait(semaphore s)
{
        s.count--;
        if (s.count < 0)
            {
            diesen Prozess in s.queue ablegen;
            diesen Prozess blockieren
            }
}
void signal(semaphore s)
{
        s.count++;
        if (s.count <= 0)
        {
            einen Prozess P aus s.queue entfernen;
            Prozess P in Liste bereiter Prozesse ablegen;
        }
}
```

Primitive für binäres Semaphor

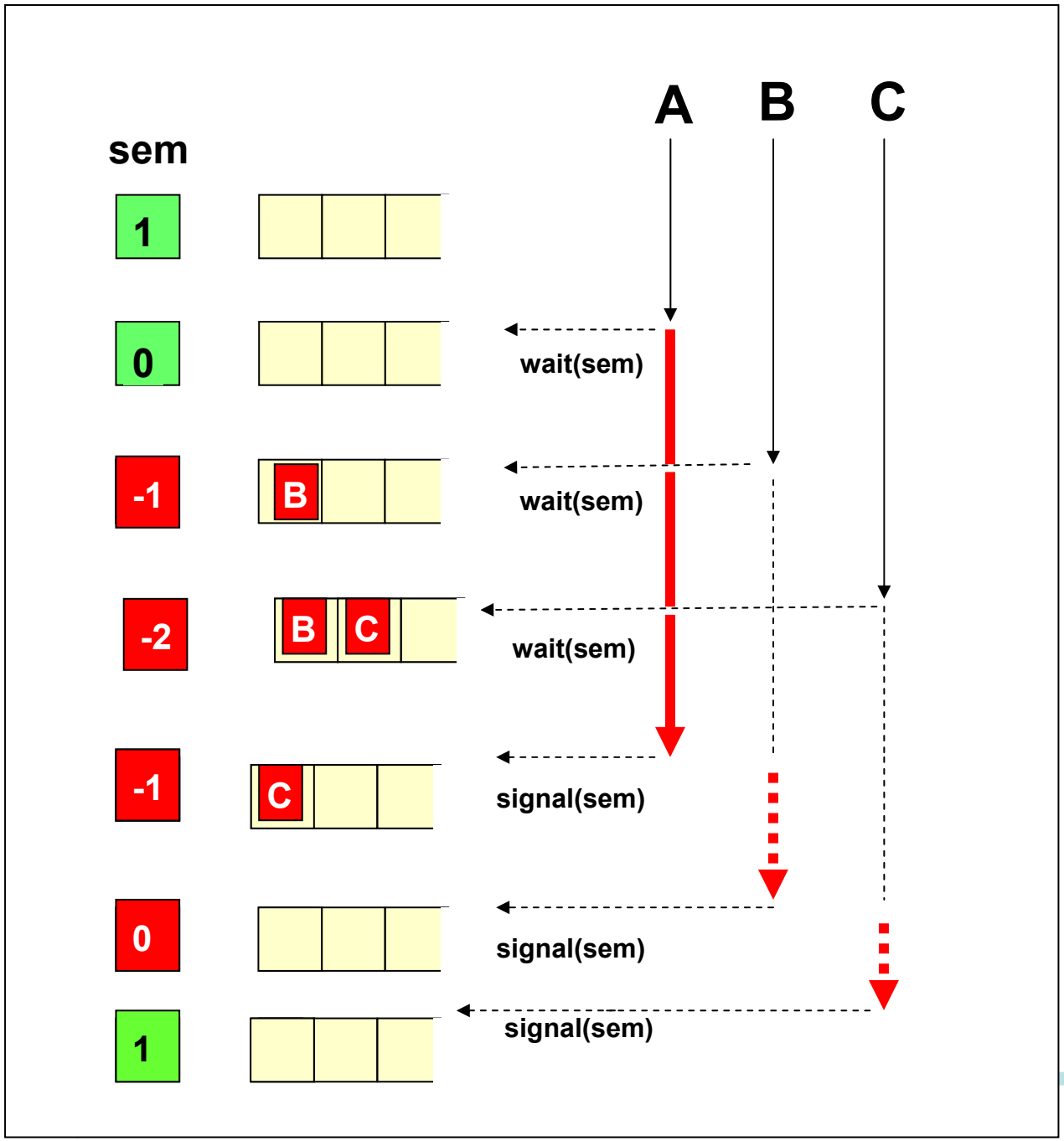
```
struct binary_semaphore
{
    enum (zero, one) value;
    queueType queue;
};

void waitB(binary_semaphore s)
{
        if (s.value == 1)
        s.value = 0;
        else
        {
            diesen Prozess in s.queue ablegen;
            diesen Prozess blockieren;
        }
}
void signalB(semaphore s)
{
        if (s.queue.is empty())
        s.value = 1;
        else
        {
            einen Prozess P aus s.queue entfernen;
            Prozess P in Liste bereiter Prozesse ablegen;
        }
}
```

Implementierung von Semaphoren

Blockierung der Interrupts

```
wait(s)
{
        inhibit interrupts;
        s.count--;
        if (s.count < 0)
        {
                diesen Prozess in s.queue ablegen;
                diesen Prozess blockieren und
                Interrupts zulassen
        }
                else
                allow interrupts;
```

```
signal(s)
{
        inhibit interrupts;
        scount++;
        if (s.count <= 0)
        {
                remove a process P from s.queue;
                place process P on ready list
        }
        Interrupts wieder zulassen;
}
```

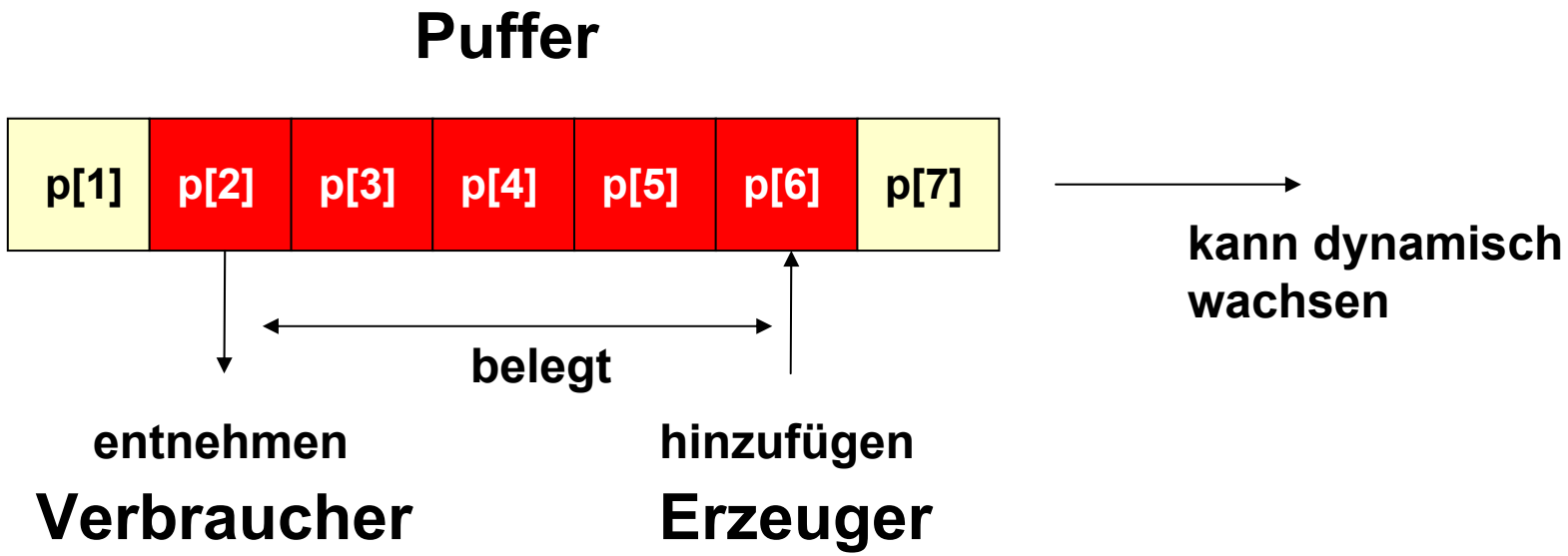


Wechselseitiger Ausschluss mit einem Semaphor

Für jedes Betriebsmittel gibt es genau 1 Semaphor !

Anwendung von Synchronisationsmechanismen

Das Erzeuger-Verbraucher Problem (producer-consumer)



1. Prozesse dürfen nicht gleichzeitig auf dasselbe Element zugreifen.
2. Verbraucher kann nur dann ein Element entnehmen, wenn der Puffer nicht leer ist.
3. Erzeuger kann nur in den Puffer schreiben, wenn der Puffer nicht voll ist.

Das Erzeuger-Verbaucher (Producer - Consumer) Problem

Aspekte:

1. Wechselseitiger Ausschluss

2. Synchronisation der Aktivitäten.

Erzeuger und Verbraucher sind voneinander abhängig in Bezug auf ihre Verarbeitungsbeschwindigkeit.

Das Erzeuger-Verbaucher (Producer - Consumer) Problem

```
Erzeuger:
while (true)
{
        /* erzeuge Element v*/;
        b[in] = v;
        in++;
}
```

```
Verbraucher:
while (true)
{
        while (in <= out)
        /* tu nichts */;
w = b [out] ;
    out++;
/* verbrauche Element w */
```

Beispiel: Das Erzeuger-Verbaucher Problem

```
/* Programm ErzeugerVerbraucher */
int n;
binary_semaphore s =1;
binary_semaphore delay = 0;
```

```
void producer()
{
        while (true)
        {
        produce();
        waitB(s);
        append();
        n++;
        if (n==1)
        signalB(delay);
        signalB(s);
        }
}
```

```
void main
{
n =0;
parbegin (producer, consumer);
}
```

```
void consumer()
{
   waitB(delay);
      while (true)
     {
            waitB(s);
            take();
            n--;
            signalB(s);
            consume();
            if (n==0) waitB(delay);
      }
   }
```

inkorrekte Lösung mit binären Semaphoren

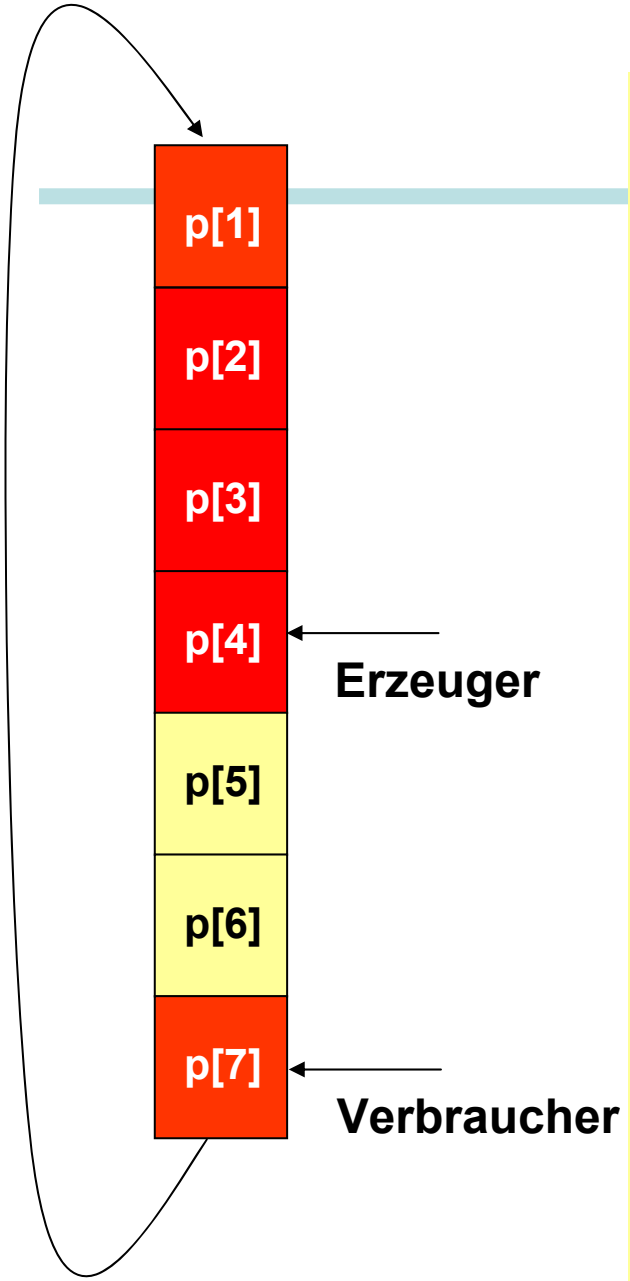
Das Erzeuger-Verbaucher (Producer - Consumer) Problem

```
/ * Programm
ErzeugerVerbraucher*/
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
void producer()
{
        while (true)
        {
            produce();
            waitB(s);
            append();
            n++;
            if (n==1)
            signalB(delay);
            signalB(s);
        }
}
```

```
void consumer()
{
        int m;  / *eine lokale Variable */
        waitB(delay);
        while (true)
        {
            waitB(s);
            take();
            n--;
            m = n;
            signalB(s);
            consume();
            if (m==0)
            waitB(delay);
        }
}

void main()
{
n=0;
parbegin (producer, consumer);
}
```

korrekte Lösung mit binären Semaphoren

der Verbraucher merkt sich un-abhängig vom Erzeuger den Zustand von n




```
/ * Programm PufferFesterGröße `/
const int sizeofbuffer = /* Puffergröße */;
semaphore s = 1; /* wechselseitiger Ausschluss */;
semaphore n= 0;  /* Indikator für leeren Puffer */
semaphore e= sizeofbuffer; /Indikator für vollen Puffer /
void producer()
{
        while (true)
        {
                produce();
                wait(e);
                wait(s);
                append();
                signal(s);
                signal (n)
        }
void consumer()
{
        while (true)
        {
                wait(n);
                wait(s);
                take();
                signal(s);
                signal(e);
                consume();
        }
}
void main() .....
```

(Ring-) Puffer fester Größe

Semaphore

Allgemeines, leistungsfähiges Konzept zur Koordinierung nebenläufiger Prozesse.

Kontrolle über die Anzahl von Prozesse, die eine Ressource nutzen wollen.

Spezialfall: wechselseitiger Ausschluss (MUTEX).

Kritik:

wait und signal sind low level Programmierkonstrukte.

wait und signal können über das ganze Programm verteilt sein.

- **schwer zu kontrollieren**
- **Korrektheit schwer nachzuweisen**

Wünschenswerte Eigenschaften für einen Mechanismus zur Koordination nebenläufiger Prozesse:

- Besser an die Programmierung angepasster Mechanismus.
- Semantik der Kontrolle besser formulierbar.
- Einbettung in eine Programmiersprache.

Konzept des Monitors

C.A.R. Hoare (1974), Brinch Hansen (1975)

Ein Monitorbeispiel

```
monitor erzeugerverbraucher
        integer i;
        condition c;

        procedure erzeuger ();
        .
        .
        end;

        procedure verbraucher ();
        .
        .
        end;

end monitor;
```

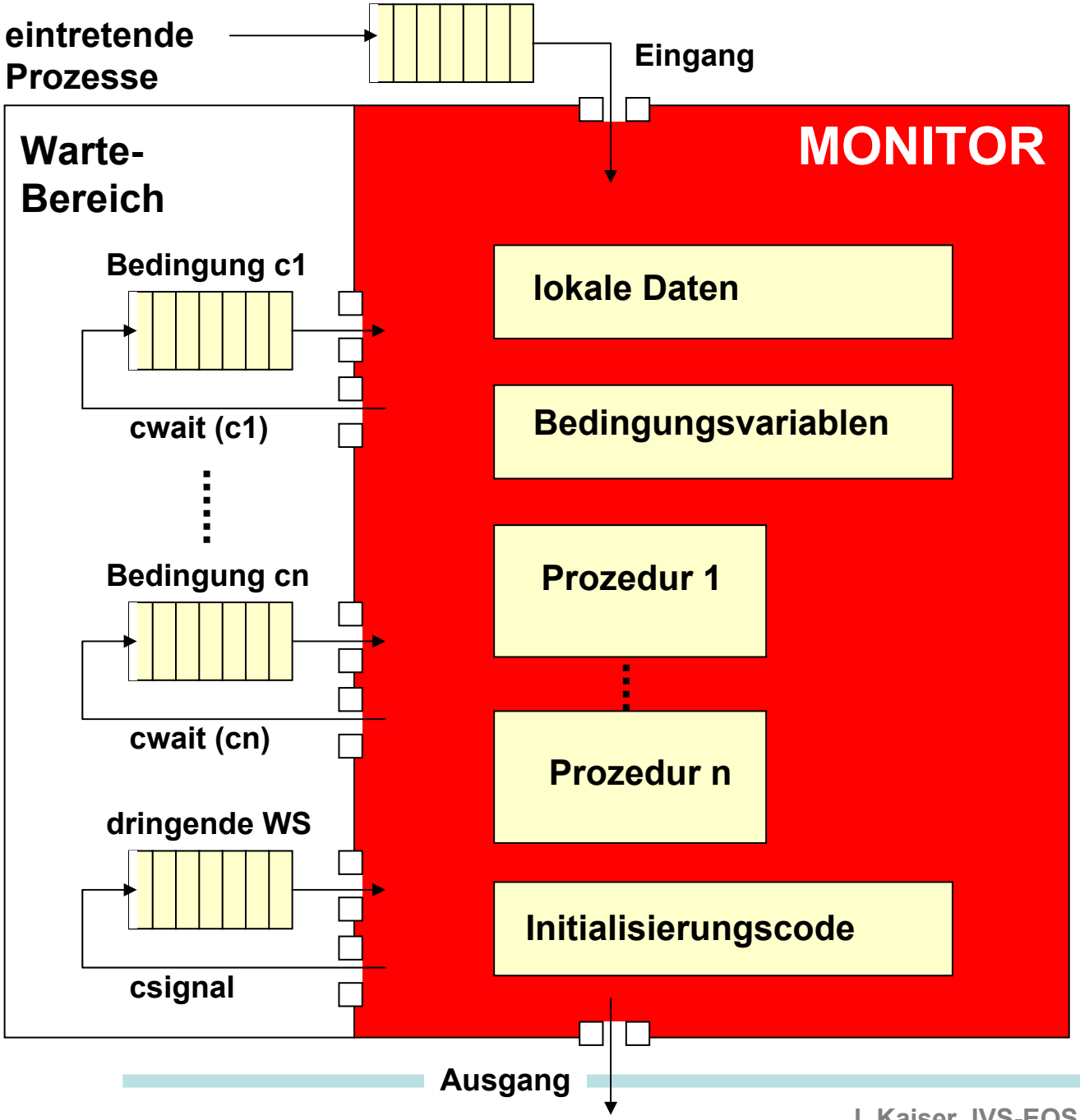
Organisation eines Monitors

- nur 1 Prozess kann zu jedem Zeitpunkt im Monitor sein.

- Monitor ist ein Sprachkonstrukt, d.h. Compiler überwacht den korrekten gegenseitigen Ausschluss

- Programmierer des Monitors muss sich nicht um gegenseitigen Ausschluss kümmern.

- Synchronisation ist aus den Monitor beschränkt. Korrektheit ist deshalb leichter zu überprüfen.



```
/* Programm ErzeugerVerbraucher*/
monitorboundedbuffer;
char buffer [N];          /* Platz für N Elemente */
int nextin, nextout;      /* Pufferzeiger */
int count;                /* Anzahl der Elemente im Puffer */
int notfull,notempty;     /* zur Synchronisierung */
```

```
void append (char x)
{
if (count == N)
        cwait(notfull);          /* Puffer ist voll; Überlauf vermeiden */
buffer[nextin] = x;
nextin = (nextin + 1)% N;
count++;                         /* ein Element mehr im Puffer */
csignal(notempty);               /* Verbraucher wieder aufnehmen */
}
```

```
void take (char x)
{
if (count == 0)
        cwait(notempty);     /* Puffer ist leer; Unterlauf vermeiden */
x = buffer[nextout];
nextout = (nextout + 1)% N;
count-;                          /* ein Element weniger im Puffer */
csignal(notfull);                /* wartenden Erzeuger wieder aufnehmen */
}
```

```
 /* Programmcode des Monitors */
nextin = 0;
nextout = 0;
count = 0;      / * Puffer ursprünglich leer */
}
```

```
void producer()
char x;
{
        while(true)
        {
                produce(x);
                append(x);
        }

}
```

```
void consumer()
{
char x;
while(true)
        {
        take(x);
        consume(x);
        }
}
```

```
void main()
{
parbegin(producer, consumer);
}
```

Diskussion: Monitore

Sprachkonstrukt, d.h. muss in eine Sprache "eingebaut" sein.

Semaphore kann man "leicht" selber realisieren, wenn das Betriebssystem solche Konstrukte kennt.

Monitore und Semaphore erfordern die Existenz eines gemeinsamen Speichers.

Semaphore und Monitore können nur zur Synchronisation des Datenaustauschs verwendet werden aber nicht zum Datenaustausch selbst.

Nicht geeignet für verteilte Systeme.

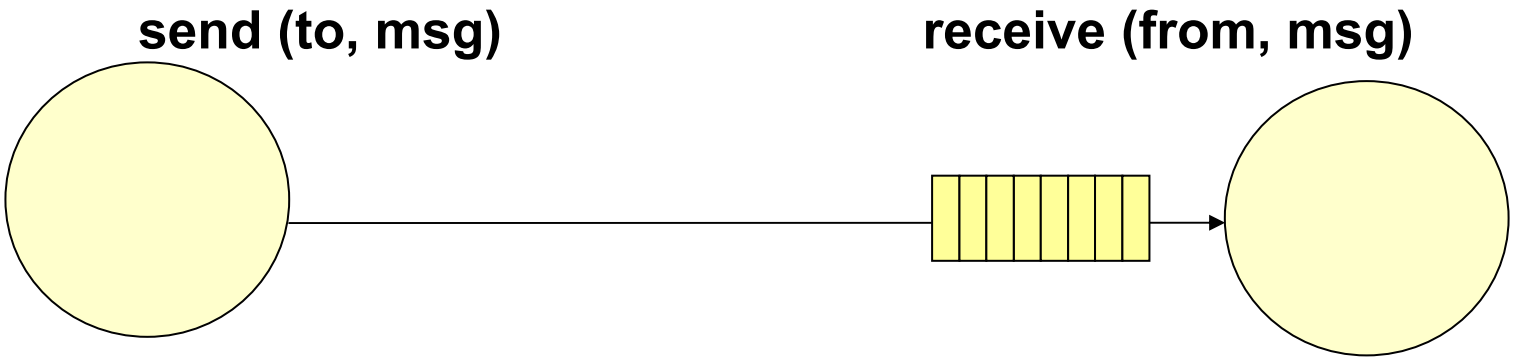
Interprozesskommunikation IPC

Transfer von Daten von einem Prozess zu einem oder mehreren anderen Prozessen.

- über gemeinsamen Speicher "Shared memory"
Lesen und Schreiben gemeinsamer Variablen.

- über einen gemeinsamen Kommunikationskanal:
explizites Senden und Empfangen von Nachrichten.

Nachrichtenaustausch



integriert Datentransfer + Synchronisation

Ist der Empfänger bereit?
Was tun, wenn der Empfänger nicht bereit ist?

Wann kommt eine Nachricht?
Was tun, wenn eine Nachricht nicht kommt?

Wie wird adressiert?
Wann muss man wissen, mit wem man kommunizieren will?
Wie ist das Nachrichtenformat?
Wie werden ankommende Nachrichten geordnet?

Nachrichtenaustausch

Synchronisationsaspekt:

Koordination zwischen Sender und Empfänger erforderlich:

- **Sender darf Nachrichten nicht schneller erzeugen, als der Empfänger sie verarbeiten kann.**
  - **Empfänger bestimmt die Geschwindigkeit der Kommunikation!**

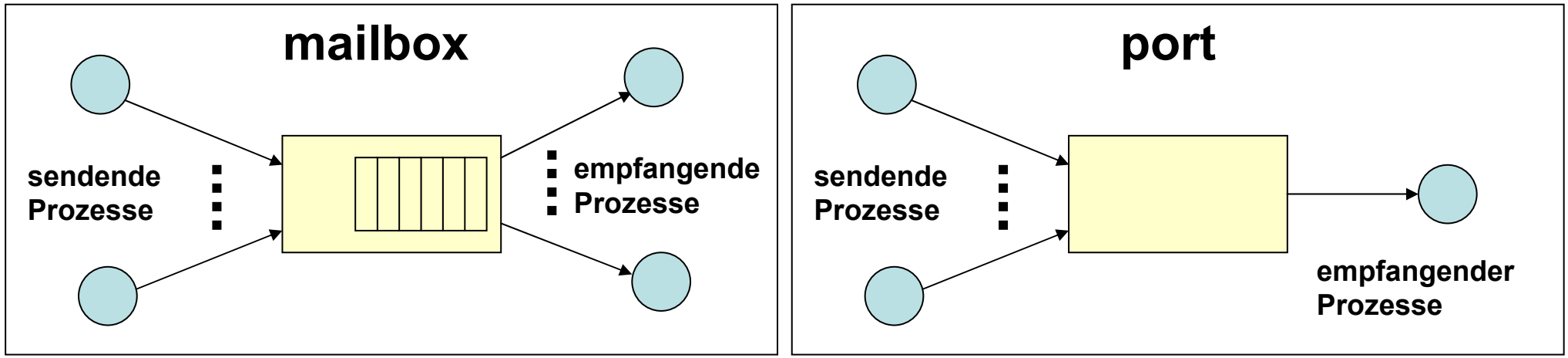
- **blockierendes Senden:** Sender wartet bis Empfänger die Nachricht empfangen kann
- **blockierendes Empfangen:** Empfänger wartet auf Nachricht vom Sender.

Blockierendes Senden und blockierendes Empfangen ohne Zwischenpuffer → "Rendevous".

Nachrichtenaustausch

Direkte Adressierung:
In den send/receive-Primitiven wird ein Prozess direkt spezifiziert.

Indirekte Adressierung:
Nutzung einer gemeinsamen Datenstruktur.



Das Erzeuger-Verbraucher-Problem mit beschränktem Puffer.

Lösung über Nachrichtenaustausch

```
const int
    capacity = /* Pufferkapazität */
    null = /* Leernachricht*/
int i;
void producer()
{ message pmsg;
    while (true)
    {
    receive (mayproduce, cmsg);
    pmsg = produce();
    send (mayconsume, pmsg);
     }
}
void consumer()
{   message cmsg;
    while (true)
    {
        receive (mayconsume, pmsg);
        consume (pmsg);
        send (mayproduce, null);
    }
void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++)
        send (mayproduce, null);
    parbegin (producer, consumer);
}
```

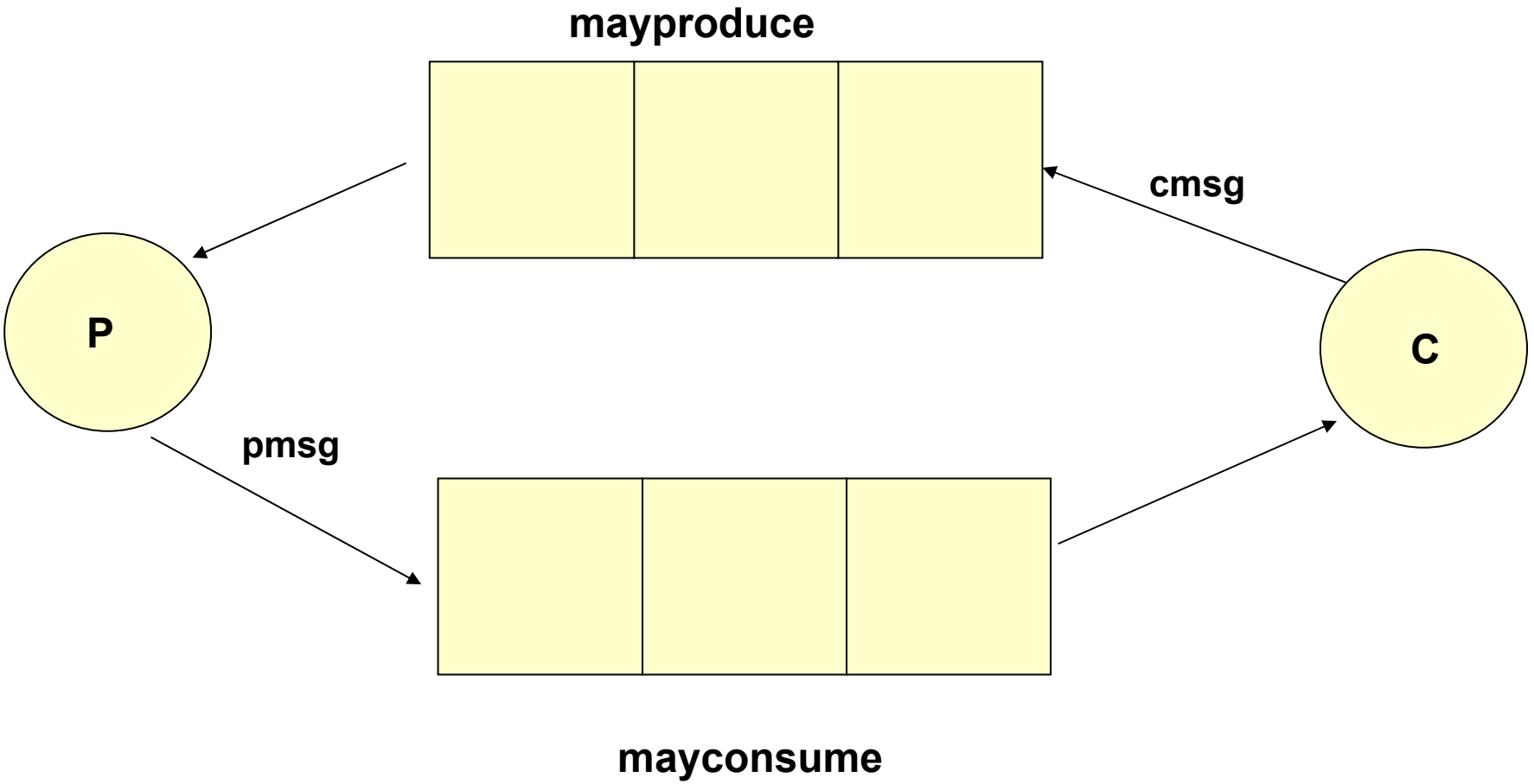
mayconsume: Puffer für die tatsächlichen Nutzdaten

mayproduce: "Zähler" für die Anzahl der belegten Plätze.

pmsg: produce msg (blockiert, wenn mayproduce = 0)
cmsg: consume msg (blockiert, wenn mayconsume = 0)

Initialisierung des Puffers.
Nach Initialisierung ist der Puffer mit 0-msg gefüllt.

Das Erzeuger-Verbraucher-Problem mit beschränktem Puffer.



Klassisches Problem der Interprozesskommuniktion: Das Leser/Schreiber Problem

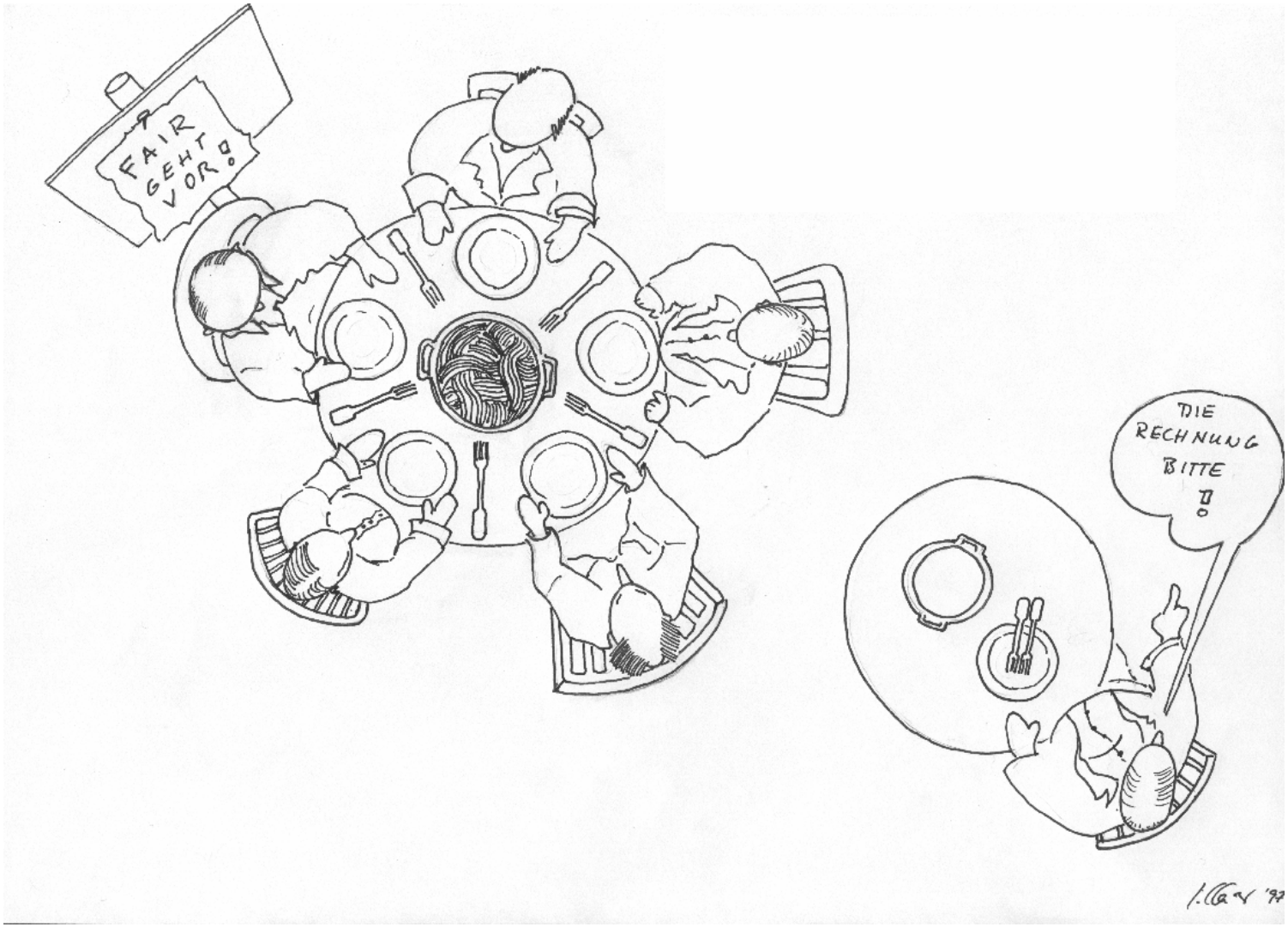
Ein Schreiber - mehrere Leser

- Schreiber Priorität

- Leser Priorität

Sonderform des Erzeuger Verbraucher Problems???

Klassisches Problem der Interprozesskommuniktion: Die speisenden Philosophen



Probleme:

Verklemmung (Deadlock)
Verhungern (Starvation)

Bedingungen für Deadlocks:

Entwurfsentscheidungen
lokale Funktion

1 Wechselseitiger Ausschluss

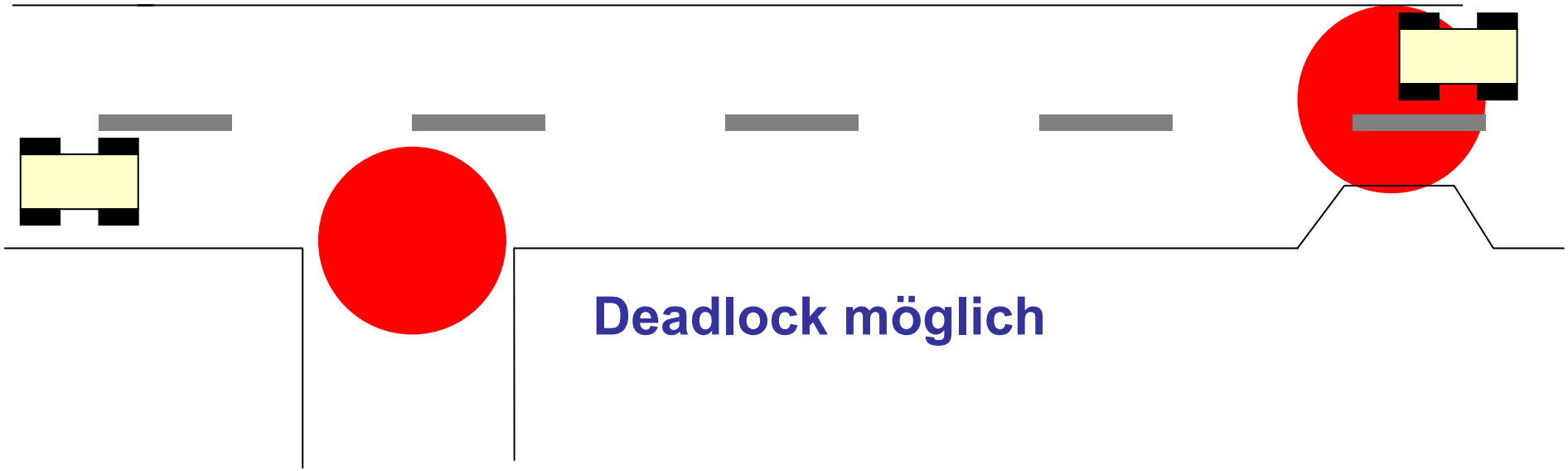
2 Behalten und Warten (Hold & Wait)

3 Kein Entzug von Ressourcen

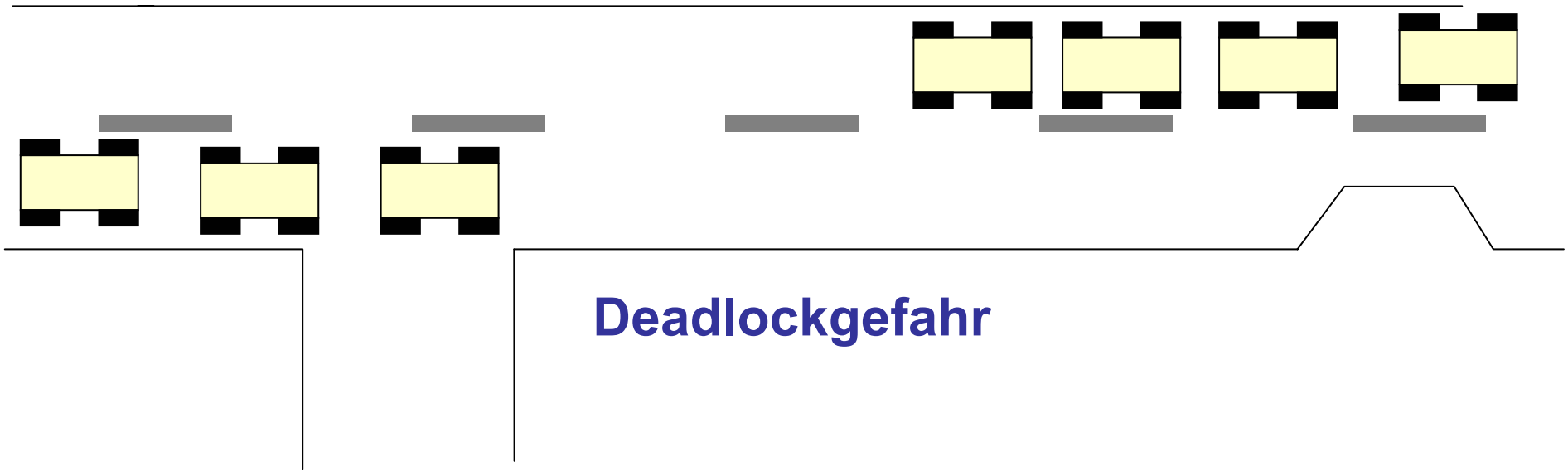
4 Zyklisches Warten

dynamisch entstehende Situation

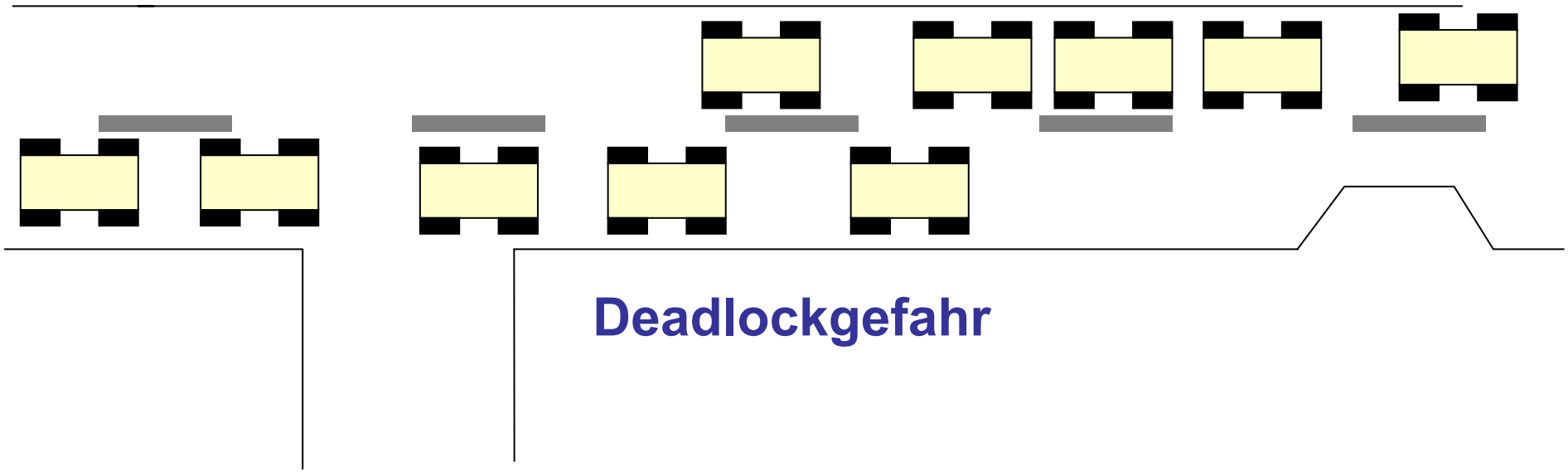
Typische Deadlock Situation im Strassenverkehr



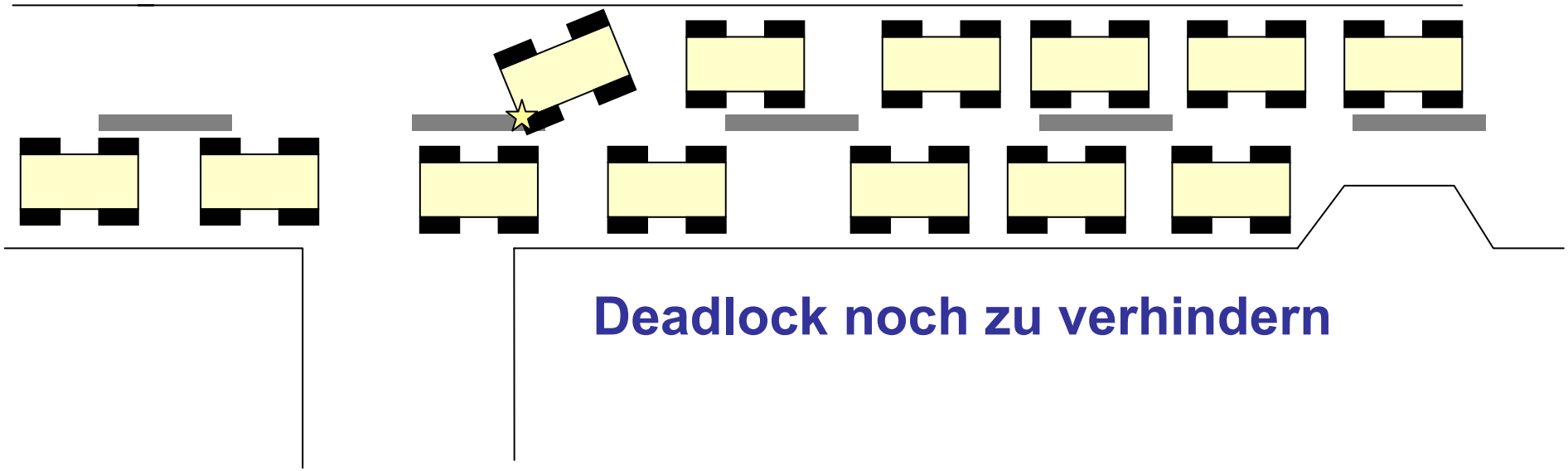
Typische Deadlock Situation im Strassenverkehr



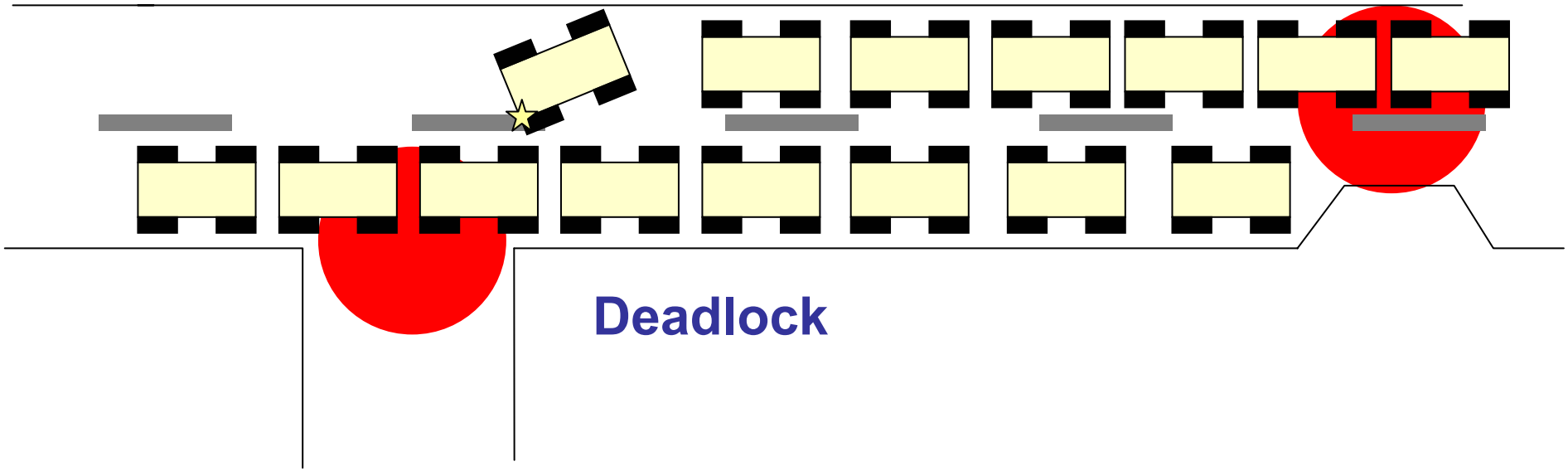
Typische Deadlock Situation im Strassenverkehr



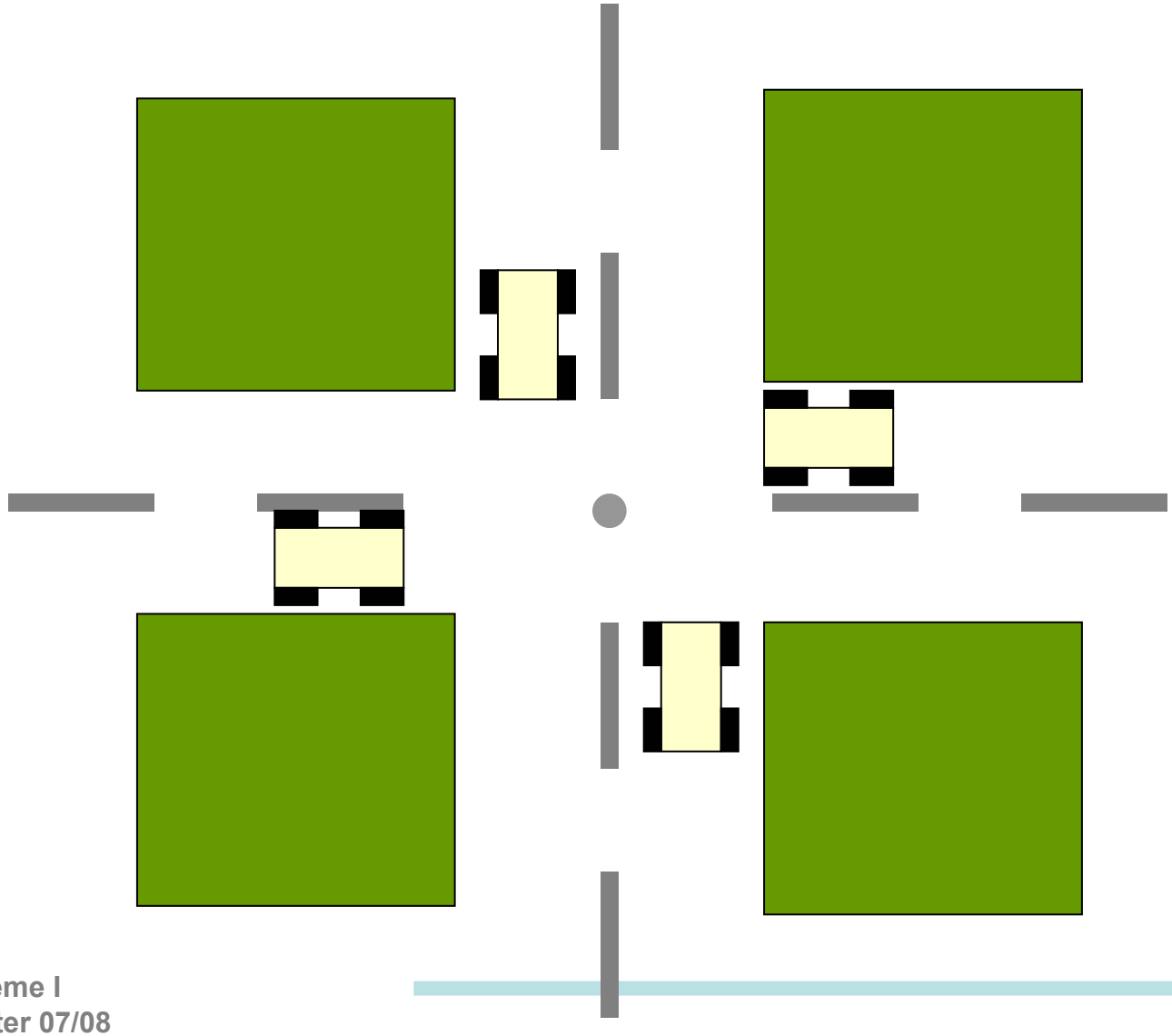
Typische Deadlock Situation im Strassenverkehr



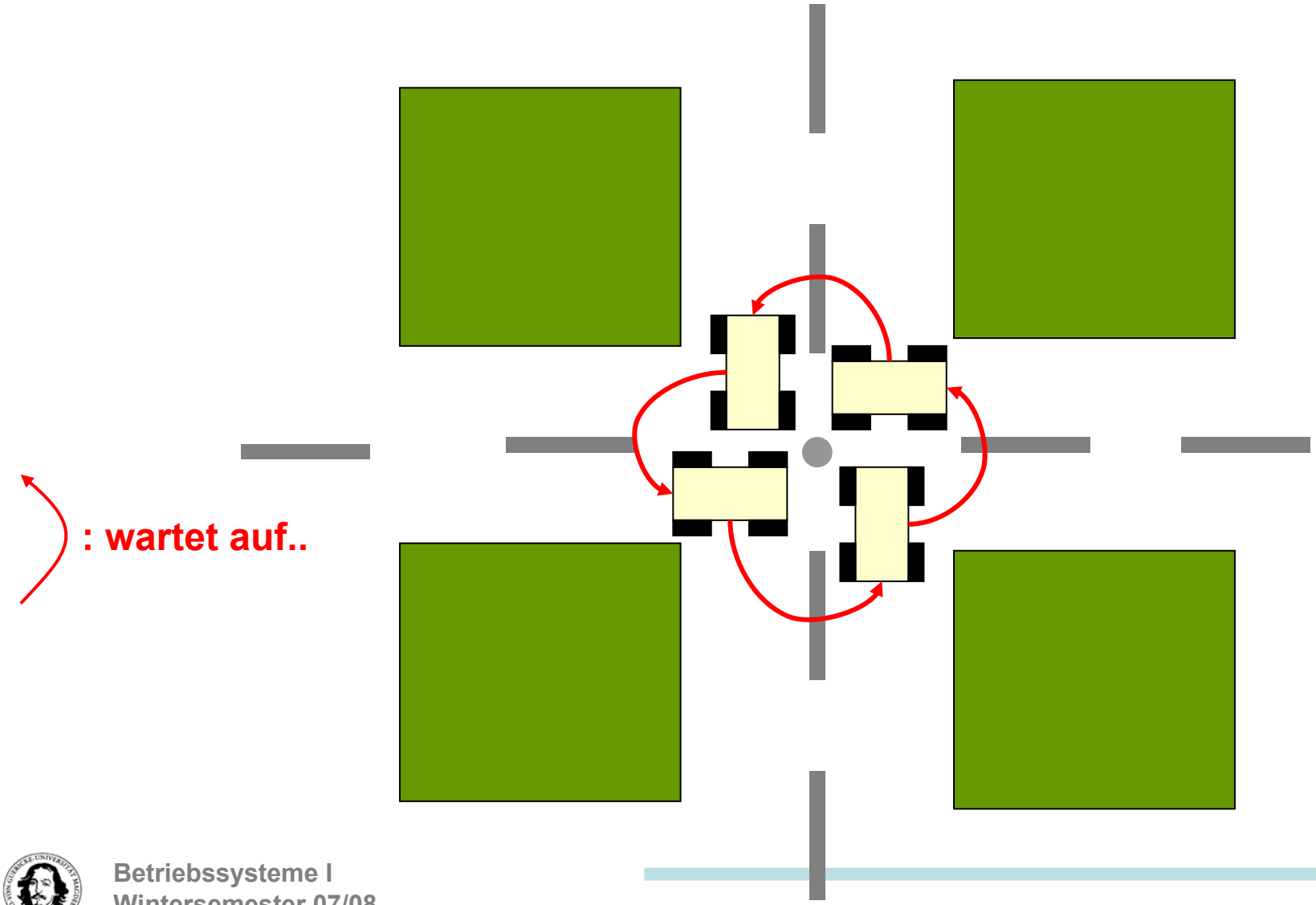
Typische Deadlock Situation im Strassenverkehr



Typische Deadlock Situation im Strassenverkehr



Typische Deadlock Situation im Strassenverkehr



Strategien zur Behandlung von Deadlocks

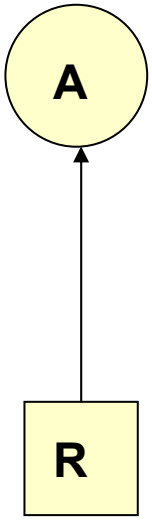
1. Ignorieren

2. Erkennen und Behandeln.

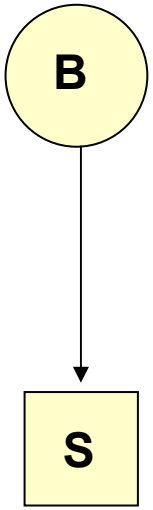
3. Dynamisch vermeiden durch vorsichtige Belegung von Ressourcen.

4. Verhindern indem man erreicht, dass eine der 4 Bedingungen nicht eintreten kann.

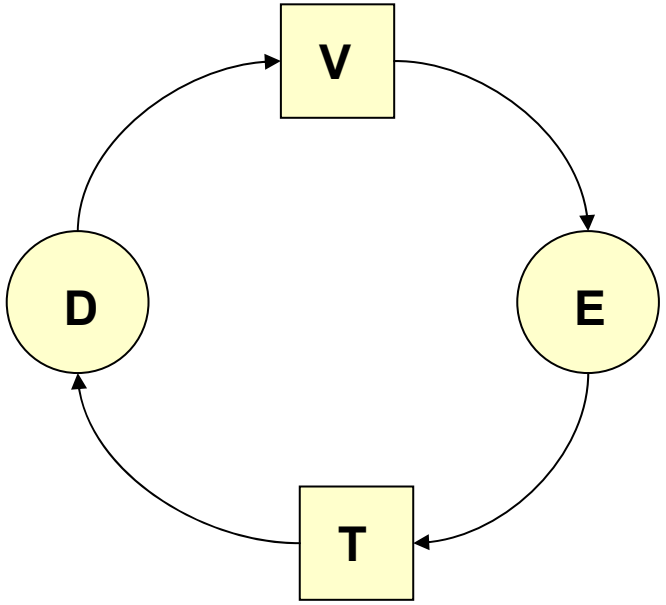
Belegungsgraphen zur Beschreibung von Betriebsmittelanforderungen



A belegt R

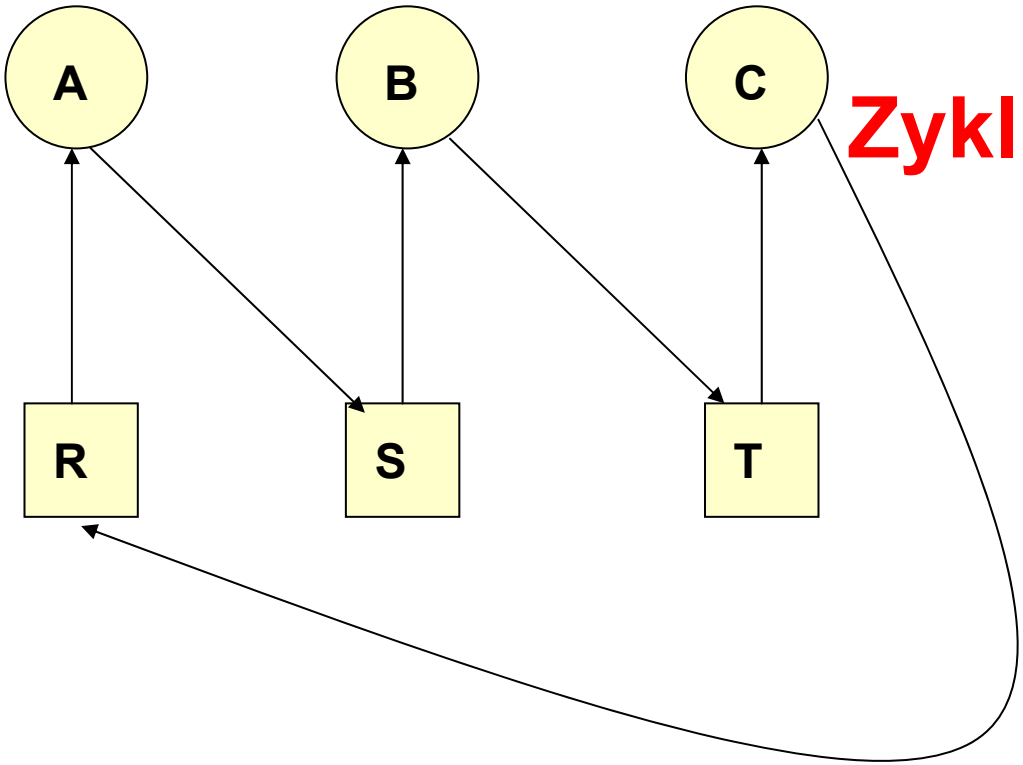


B wartet auf R



zyklische Wartesituation
Deadlock

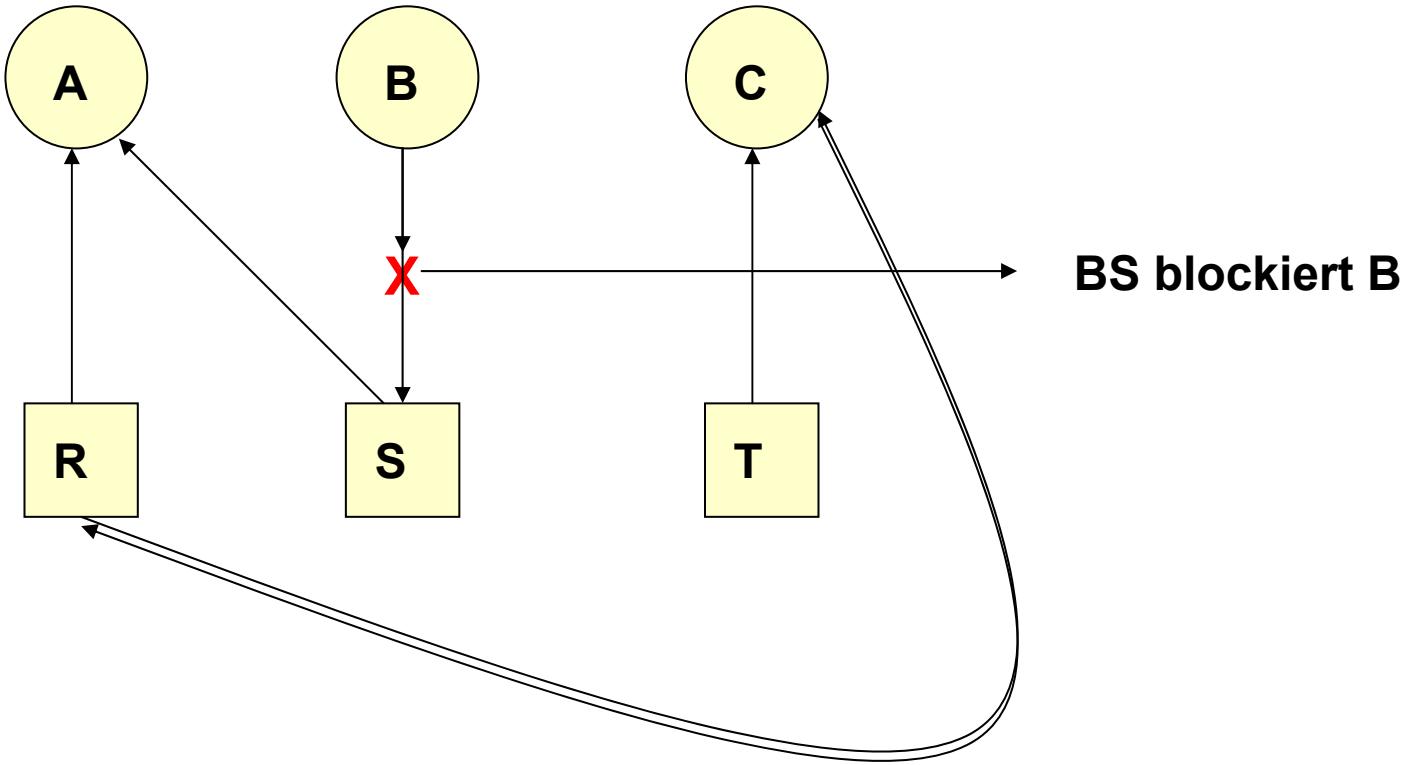
Zyklisches Warten



Zyklisches Warten!

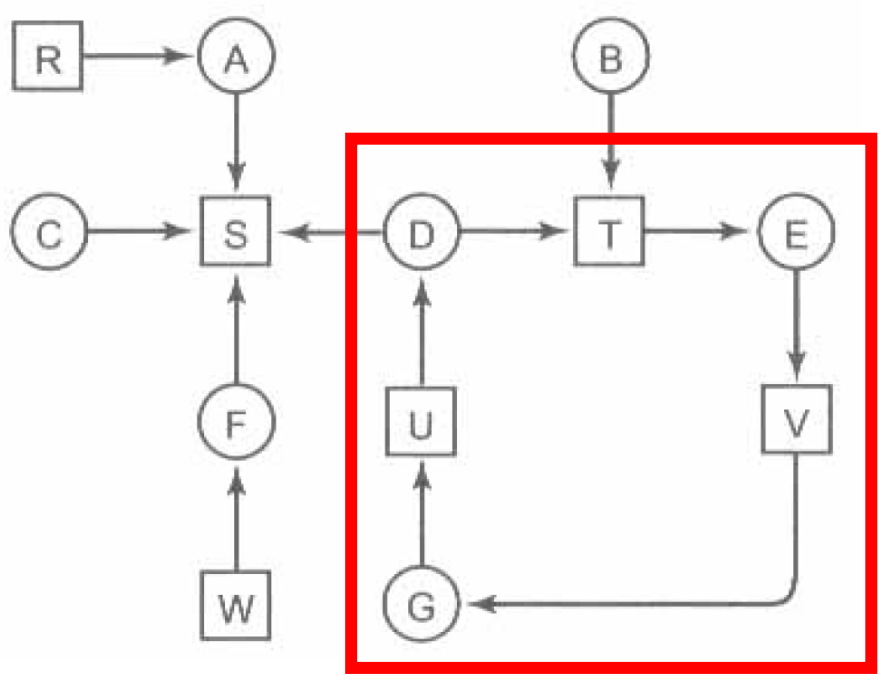
Deadlock

Vermeidung zyklisches Wartens



Deadlocks erkennen

1. A belegt R und verlangt S.
2. B belegt nichts, verlangt aber T.
3. C belegt nichts, verlangt aber S.
4. D belegt U und verlangt S und T.
5. E belegt T und verlangt V.
6. *F* belegt W und verlangt S.
7. G belegt *V* und verlangt *U.*



Zyklus

A. Tanenbaum: Moderne BS, S.189

Deadlocks erkennen

Algorithmus:

1. Für alle Knoten K im Graphen führe die folgenden Schritte aus. Benutze K als Startknoten

2. Lösche alle Kantenmarkierungen und L. (L: Liste von Knoten)

3. Hänge den aktuellen Knoten an das Ende von L und überprüfe, ob er in L zweimal vorkommt. Wenn ja, enthält L den gesuchten Zyklus und der Algorithmus terminiert.

4. Überprüfe, ob vom aktuellen Knoten unmarkierte Kanten wegführen. Wenn ja, gehe zu Schritt 5, sonst gehe zu Schritt 6.

5. Wähle zufällig eine wegführende Kante und markiere sie. Folge der Kante zum neuen aktuellen Knoten und gehe zu Schritt 3.

6. Wir sind in einer Sackgasse. Wenn der aktuelle Knoten der Startknoten ist, enthält der Graph keine Zyklen und der Algorithmus terminiert. Ansonsten lösche den aktuellen Knoten aus L, gehe zurück zum vorherigen Knoten, d.h. zu dem, der zuletzt der aktuelle Knoten war, mache ihn wieder zum aktuellen Knoten und gehe zu Schritt 3.

Deadlocks erkennen

Annahmen bisher: Eine Ressource / Ressourcentyp, d.h. nur jeweils 1 Prozess kann eine Ressource belegen.

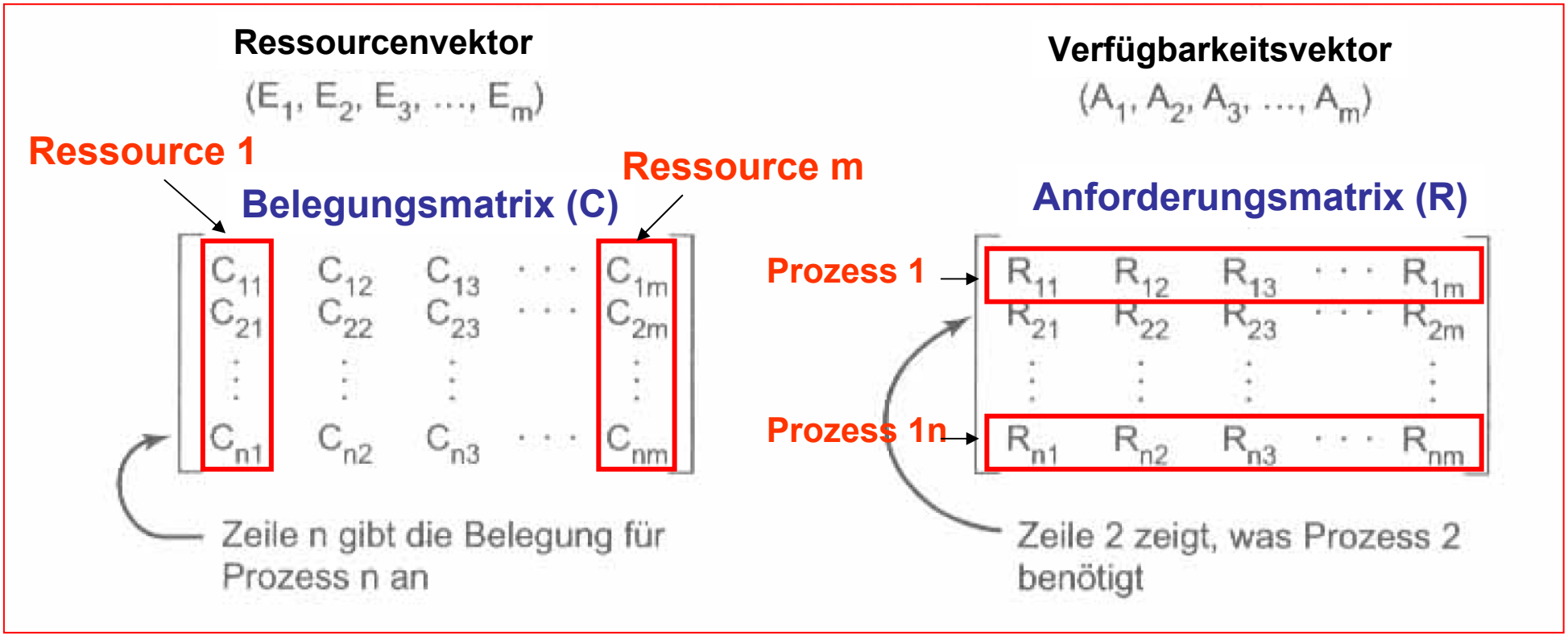
Neue Annahmen: n Prozesse $P_1, \ldots, P_n$
m Ressourcenklassen

Datenstrukturen:

E_i ,$1 \leq i \leq m$: Ressourcenvektor E (Existing resources vector)
A_i ,$1 \leq i \leq m$: Verfügbarkeitsvektor A (Available resources vector)
C : Belegungsmatrix (Allocation Matrix)
R: Anforderungsmatrix (Request Matrix)

Deadlocks erkennen

Datenstrukturen:



Invariante für die Ressourcenverteilung:

$$\left[\sum_{i=1}^{n} C_{ij}\right] + A_j = E_j$$

A. Tanenbaum: Moderne BS, S.189

Deadlocks erkennen

Belegungsmatrix

	R1	R2	R3
P1	1	4	0
P2	2	2	2

Anforderungsmatrix

	R1	R2	R3
P1	2	1	1
P2	0	2	0

Ressourcenvektor (E)

4	6	2

Verfügbarkeitsvektor (A)

1	0	0

Zuteilungsvektor

3	6	2

$$\left[\sum_{i=1}^{n} C_{ij}\right] + A_j = E_j$$

i: Prozess
j: Ressource

$\Sigma C_{i,1} + A_1 = 3 + 1 = E_1 = 4$
$\Sigma C_{i,2} + A_2 = 6 + 0 = E_2 = 6$
$\Sigma C_{i,3} + A_3 = 2 + 0 = E_3 = 2$

Deadlock-Erkennungsalgorithmus

Sei die Relation ≤ auf der Menge der Vektoren definiert als:
A ≤ B genau dann wenn $A_i \leq B_i$ für alle i mit $1 \leq i \leq n$

Der Deadlock-Erkennungsalgorithmus läuft folgendermaßen ab:

1. **Suche einen unmarkierten Prozess P_i, für den die i-te Zeile von R kleiner oder gleich A ist.**
2. **Wenn ein solcher Prozess existiert, addiere die i-te Zeile von C zu A, markiere den Prozess und gehe zu Schritt 1.**
3. **Andernfalls beende den Algorithmus.**

Wenn der Algorithmus beendet ist, sind alle nicht markierten Prozesse an einem Deadlock beteiligt.

Erkennen von Deadlocks

Belegungsmatrix (C)

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Anforderungsmatrix (R)

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

Ressourcenvektor (E)

9	3	6

Verfügbarkeitsvektor (A)

0	1	1

Erkennen von Deadlocks

Belegungsmatrix (C)

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Anforderungsmatrix (R)

	R1	R2	R3
P1	2	2	2
P2	1	0	1
P3	1	0	3
P4	4	2	0

Ressourcenvektor (E)

9	3	6

Verfügbarkeitsvektor (A)

0	1	1

Deadlock !!

Deadlocks behandeln

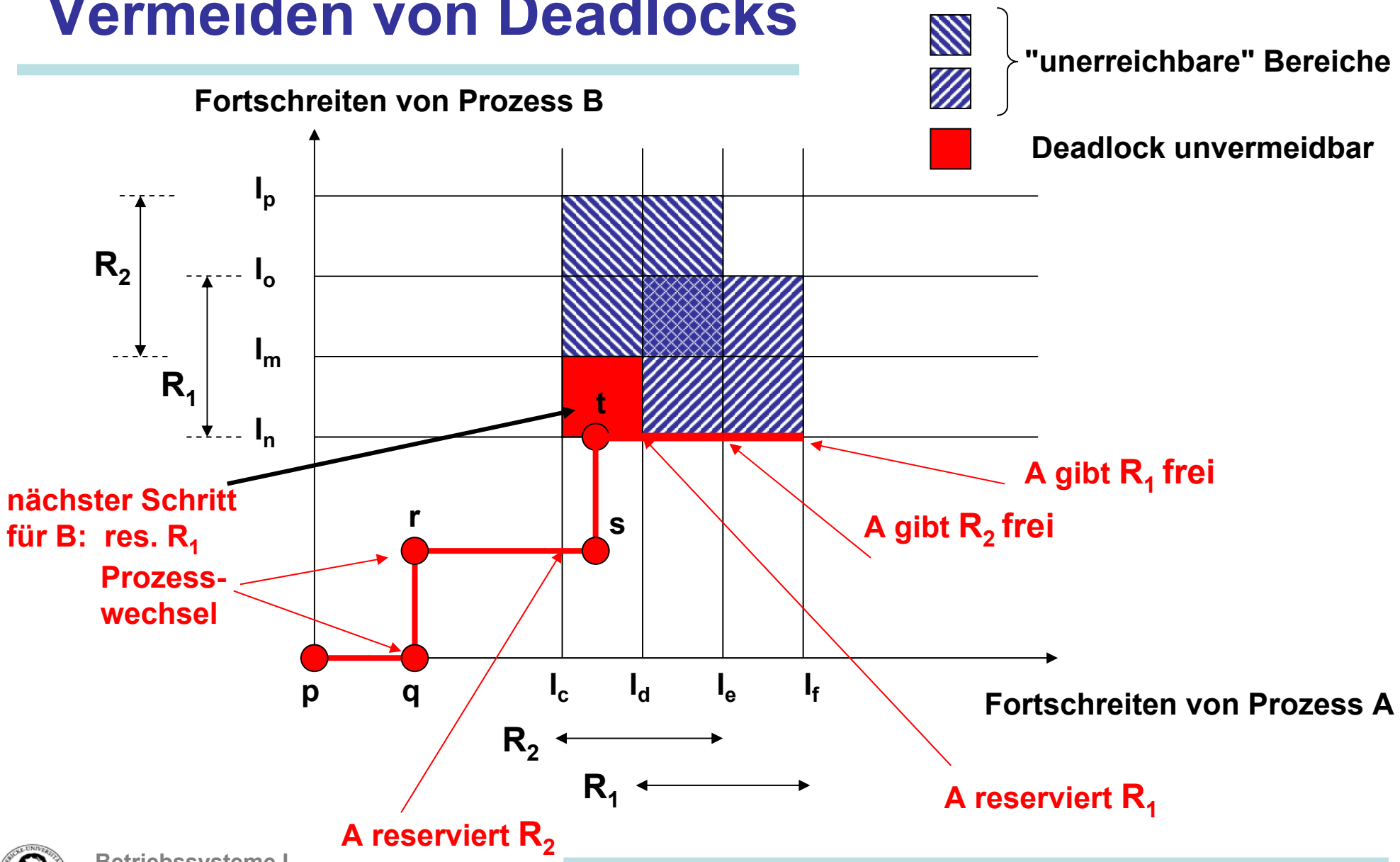
Was tun, wenn ein Deadlock erkannt wurde?

Unterbrechung eines Prozesses und zeitweiser Entzug von Ressourcen.

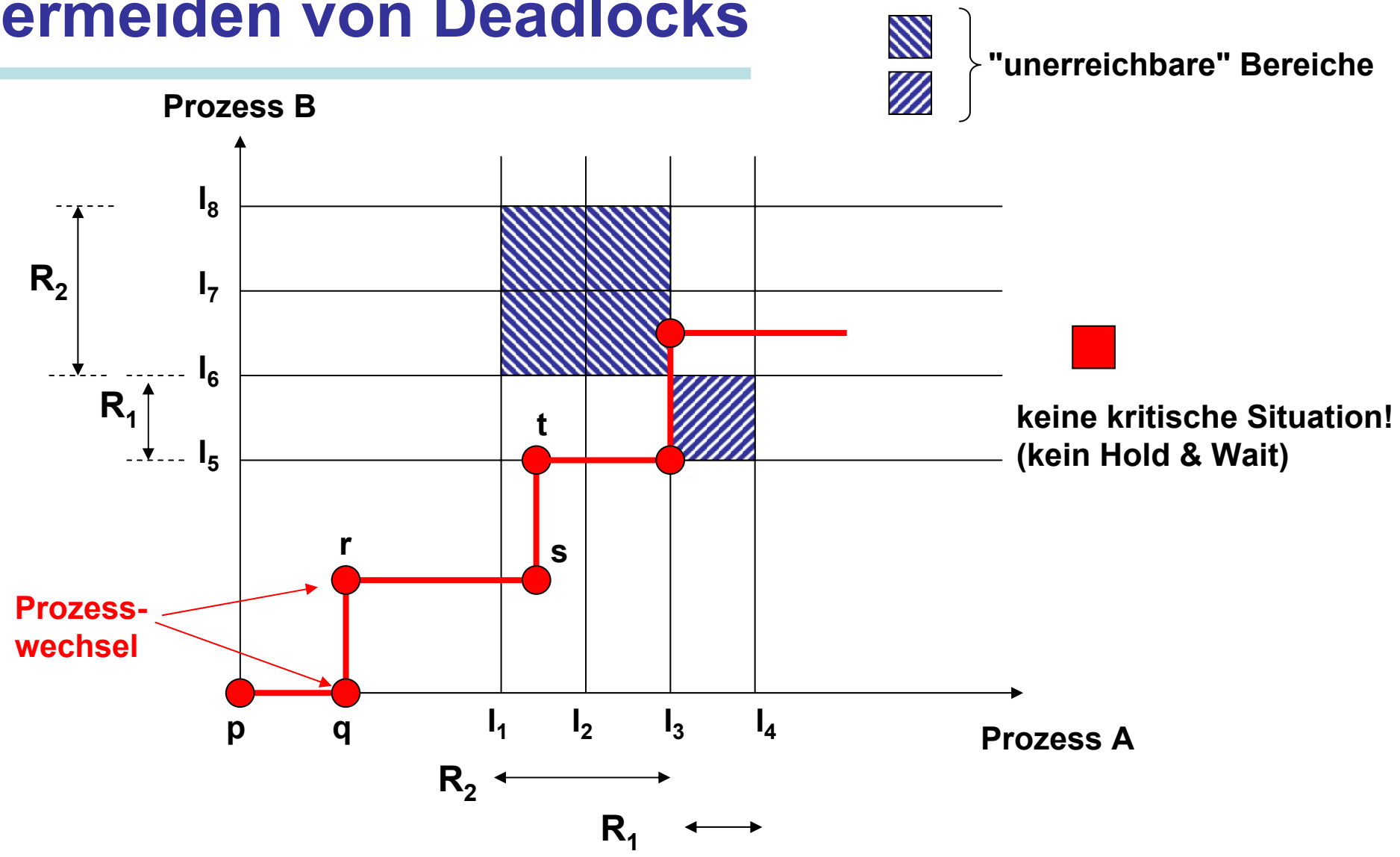
Zurücksetzen und Wiederholen (Rollback)

Abbruch eines oder mehrerer Prozesse

Vermeiden von Deadlocks



Vermeiden von Deadlocks



Vermeiden von Deadlocks

Erkennung sicherer und unsicherer Zustände

Def.: Ein sicherer Zustand ist ein Zustand, von dem aus mindestens eine Ausführungsreihenfolge existiert, die nicht zu einem Deadlock führt, d.h. alle Prozesse können bis zum Abschluss ablaufen.

Belegungsmatrix

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Anforderungsmatrix

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

Ressourcenvektor

R1	R2	R3
9	3	6

Verfügbarkeitsvektor

R1	R2	R3
0	1	1

Welcher Prozess kann bei der gegebenen Situation bis zur Terminierung laufen ?

Sicherer Zustand

Belegungsmatrix

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Anforderungsmatrix

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	2
P4	4	2	0

Ressourcenvektor

R1	R2	R3
9	3	6

Verfügbarkeitsvektor

R1	R2	R3
1	1	2

Unsicherer Zustand

Belegungsmatrix

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Anforderungsmatrix

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	2
P4	4	2	0

Verfügbarkeitsvektor

R1	R2	R3
0	1	1

Der Bankier-Algorithmus zur Vermeidung von Deadlocks

Ausgangspunkt ist ein sicherer Zustand der Ressourcenbelegung.

Bei jeder Anforderung wird geprüft, ob dadurch ein Übergang in einen sicheren oder unsicheren Zustand erfolgt.

Bei Übergang in den unsicheren Zustand wird die Ressourcenanforderung verweigert.

Bankier-Algorithmus (Dijkstra 1965):

1. **Suche eine Zeile in R, die kleiner oder gleich A ist. Wenn es keine solche Zeile gibt, kann kein Prozess beendet werden --> unweigerlich Deadlock.**
2. **Prozess kann alle notwendigen Ressourcen reservieren und terminieren. Markiere den Prozess als beendet und addiere Ressourcen zu A.**
3. **Wiederholung von 1 und 2 bis alle Prozesse markiert sind oder Deadlocksituation erkannt wird.**

Deadlock Verhinderung

Problem	Versuch der Problemlösung
Wechselseitiger Ausschluss	Keine generelle Lösung!
Hold & Wait	Ressourcen alle auf einmal anfordern
Ununterbrechbarkeit	Ressourcen wegnehmen
Zyklisches Warten	Ressourcen nummerieren

Kontrollmechanismen für Nebenläufigkeit unter Unix

Erich Ehses, Lutz Köhler, Petra Riemer, Horst Stenzel, Frank Victor: "Betriebssysteme", Pearson Studium, 2005

Shared Memory: Ein Teil des virtuellen Adressraums wird in mehrere Prozess-adressräume eingeblendet. Für Synchronisation muss der Programmierer sorgen.

Semaphore: Erweiterung des Semaphor- Konzeptes.

Nachrichten: Mit *msgsnd* und *msgrcv* können Nachrichten an Prozesse geschickt werden. Nachrichten sind typisiert. Prozesse können Nachrichten nach FIFO oder nach Typ geordnet annehmen. Senden an eine volle und Lesen aus einer leeren Warteschlange führt zur Suspendierung des Prozesses.

Signale: "Nachrichten ohne Inhalt", die einen Prozess über das Auftreten eines asynchronen Ereignisses informieren.

Pipes: Puffer(dateien) für die Kommunikation zwischen ZWEI Prozessen. Das Betriebssystem sorgt für die Synchronisation beim Zugriff.

Kontrollmechanismen für Nebenläufigkeit unter Unix

Semaphore unter Unix:

- **Mit eine einzigen Aufruf kann eine Gruppe von Semaphoren angelegt werden.**
- **Semaphore können um größerer Werte als "1" erhöht, bzw. erniedrigt werden.**
- **Für eine Gruppe von Semaphoren kann eine Folge von Operationen definiert werden. Diese Folge wird atomar ausgeführt.**
- **Semaphore registrieren Eigentümer, Zustandsmodi und Zeitstempel**

Systemkommandos auf Semaphoren:

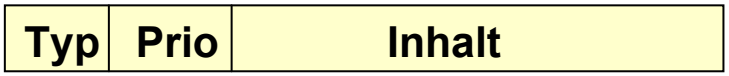
semget: *id = semget (key, nsems, flag)*
legt eine neue Semaphorgruppe an oder greift auf eine bestehende zu.

semop: *result = semop (id, sops, nsops)*
ändert die Werte der Semaphore einer Gruppe, indem eine oder mehrere *up*- oder *down*-Operationen ausgeführt werden. Die Folge wird atomar ausgeführt.

semcntl: *result = semctl (id, nsem, cmd, arg)*
führt Steuerungsfunktionen wie Setzen, Auslesen, Löschen von Semaphoren durch.

Kontrollmechanismen für Nebenläufigkeit unter Unix

Nachrichten:



Systemaufrufe für Nachrichten:

msgget: *id = msgget (key, flag)*
legt eine neue Nachrichtenwarteschlange an oder greift auf eine bestehende zu.

msgsnd: *result = msgsnd (id, ptr, size, flag)*
fügt eine neue Nachricht in eine Nachrichtenwarteschlange ein.

msgrcv: *result = msgrcv (id, ptr, size, type, flag)*
entnimmt eine Nachricht aus einer Nachrichtenwarteschlange.

Kontrollmechanismen für Nebenläufigkeit unter Unix

Solaris erweitert die Kontrollmechanismus um:

- **Sperren für den wechselseitigen Ausschluss (MUTEX-Lock)**
  aktives Warten (Spin Lock)
  Angabe einer Warteschlange (Turnstile-ID)
  Primitive: *mutex_enter(), mutex_tryenter(), mutex_exit()*

- **Leser/Schreiber Sperre (Readers/Writer Lock)**
  unterstützt Lese/Schreibsemantik mit einem Schreiber und mehreren Lesern.
  Primitive: *rw_enter, rw_exit, rw_tryenter, rw_downgrade, rw_tryupgrade*

- **Bedingungsvariablen**
  unterstützt die Realisierung von Monitoren. Bedingungsvariablen müssen immer im Zusammenhang mit MUTEX-Lock eingesetzt werden.
  Primitive: *cv_wait(), cv_signal(), cv_broadcast()*

Zusammenfassung: Nebenläufigkeit und Synchronisation

- Interprozesskommunikation (IPC) erfordert Datentransfer und Synchronisation.
- Wechselseitiger Ausschluss und kritischer Abschnitt.
- Dekker's und Peterson's Algorithmen als Softwarelösungen und Unterstützung durch unteilbare (Maschinen-) Operationen. Mechanismus auf der Anwendungsebene.
- Das Semaphorkonzept von Djikstra als leistungsfähiger Synchronisationsmechanismus. Eingebettet in ein Betriebssystem.
- Monitorkonzept von Hoare bettet Synchronisation in eine Sprache ein. Der Funktionsaufruf ist mit der Synchronisation verknüpft.
- Nebenläufigkeitsprobleme Deadlock (Verklemmung) und Starvation (Verhungern).
- Auslöser: Wechselseitiger Ausschluss, Besitzen und Warten, kein Ressourcenentzug, zyklisches Warten.
- Erkennung, Vermeidung und Verhinderung von Deadlocks. Bankiersalgorithmus und sichere Zustände.