Mutual Exclusion & Condition Synchronisation

Peter Puschner
Institut für Technische Informatik
peter@vmars.tuwien.ac.at

Beispiel 1

Prozess A

a = a + 100;

```
mov A, a
add A, 100
mov a, A
```

Prozess B

a = a – 100;

```
mov B, a
sub B, 100
mov a, B
```

„parallele“ Ausführung von *A* u. *B*

Fall 1

```
mov A, a
add A, ...
mov a, A
```

```
mov B, a
sub B, ...
mov a, B
```

Fall 2

```
mov A, a
```

```
mov B, a
sub B, ...
mov a, B
```

```
add A, ...
mov a, A
```

Fall 3

```
mov B, a
sub B, ...
```

```
mov A, a
add A, ...
mov a, A
```

```
mov a, B
```

a_{neu}: a_{alt} $a_{alt} + 100$ $a_{alt} - 100$

Beispiel 2

Memory

```
shared ShM;
```

Prozess A

```
t := 0;
loop
   ShM := t;
   t := (t+1)
         mod 10;
end loop
```

Prozess B

```
loop
   t := ShM;
   print(t);
end loop
```

„parallele“ Ausführung von *A* u. *B*

ShM: 0	ShM: 0	ShM: 0
ShM: 1	print: 0	print: 0
print: 1	ShM: 1	ShM: 1
print: 1	ShM: 2	print: 1
ShM: 2	print: 2	ShM: 2
print: 2	print: 2	print: 2

Ausgabe:

1 1 2 | 0 2 2 | 0 1 2

Gemeinsame Ressourcen

BS und Prozesse, parallel laufende Prozesse

- verwenden die selben Daten bzw. Ressourcen (Puffer, Shared Memory, Files, Geräte)
- tauschen Daten aus und/oder kooperieren

daher brauchen wir:

- „disziplinierten“ Zugriff auf gemeinsame Daten/Ressourcen
- geordnete Abarbeitung verteilter Aktionsfolgen

sonst Fehlverhalten:

- Inkonsistenz
- Nicht eindeutige Ereignisfolge

Race Condition

Ergebnis einer Operation hängt ab vom

- zeitlichen Verhalten oder
- der Reihenfolge

von bestimmten Einzeloperationen oder Ereignissen in der Umgebung.

Fehler: ein oder mehrere der möglichen Ergebnisse sind falsch bzw. führen zu falschem Verhalten.

Überblick

- Begriffe und Ziele
- Programmierung v. Mutual Exclusion und Bedingungssynchronisation
  - Mechanismen (Semaphore, Nachrichten, etc.)
  - Beispiele

Interaktion von Prozessen

- *Wettstreit um Ressourcen (Competition)*
- *Wechselseitiger Ausschluss (Mutual Exclusion)*
  - Verhindern des “gleichzeitigen” Ressourcenzugriffs
  - Ununterbrechbarkeit einer Aktionsfolge (*Atomizität*)
  - Ziel: Konsistenthalten der Daten
- *Bedingungssynchronisation (Condition Synchronisation)*
  - Erreichen einer definierten Abfolge von Operationen
  - Ziel: Vermeidung von *Race Conditions* (Abhängigkeit von Ergebnissen von relativem Prozessfortschritt)

Mutual Excl. und Cond. Sync.

- Orthogonale Konzepte
- Vier mögliche Kombinationen

MutEx.	*CondSync.*	*Ziel*
–	–	unabhängige Aktionen
–	+	vorgegebene Abfolge
+	–	Konsistenz
+	+	Konsistenz und Abfolge

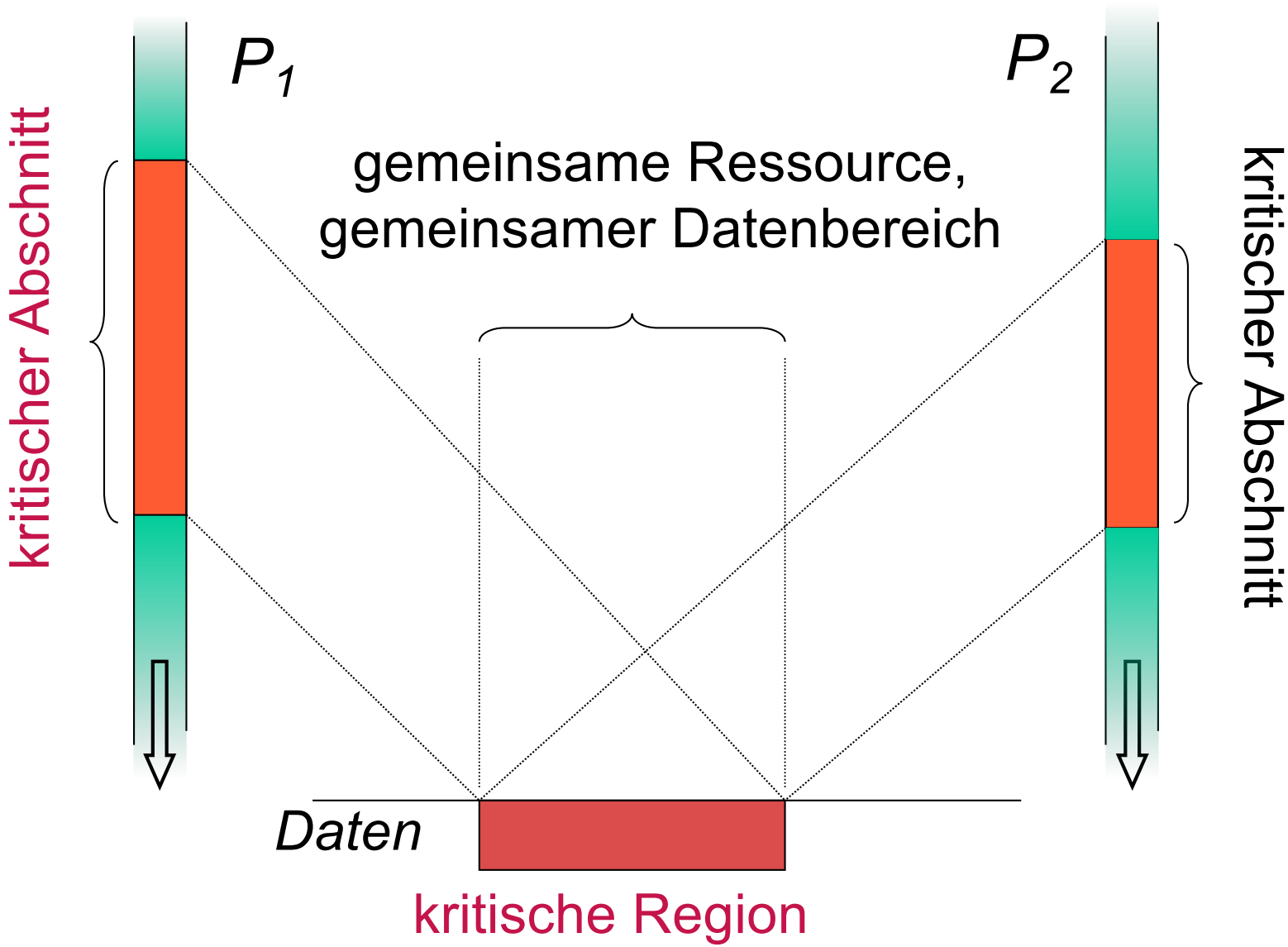
BS-Kontrollaufgaben

- Mutual Exclusion
- Datenkonsistenz
- Regelung der Abfolge der Ressourcenvergabe
- Verhinderung von Deadlock

  Bsp.: P1 besitzt R1 und braucht R2,
  P2 besitzt R2 und braucht R1
- Verhinderung von Starvation

  Bsp.: P1 und P2 laufen abwechselnd auf CPU,
  P3 bekommt die CPU nicht

Begriffe Mutual Exclusion

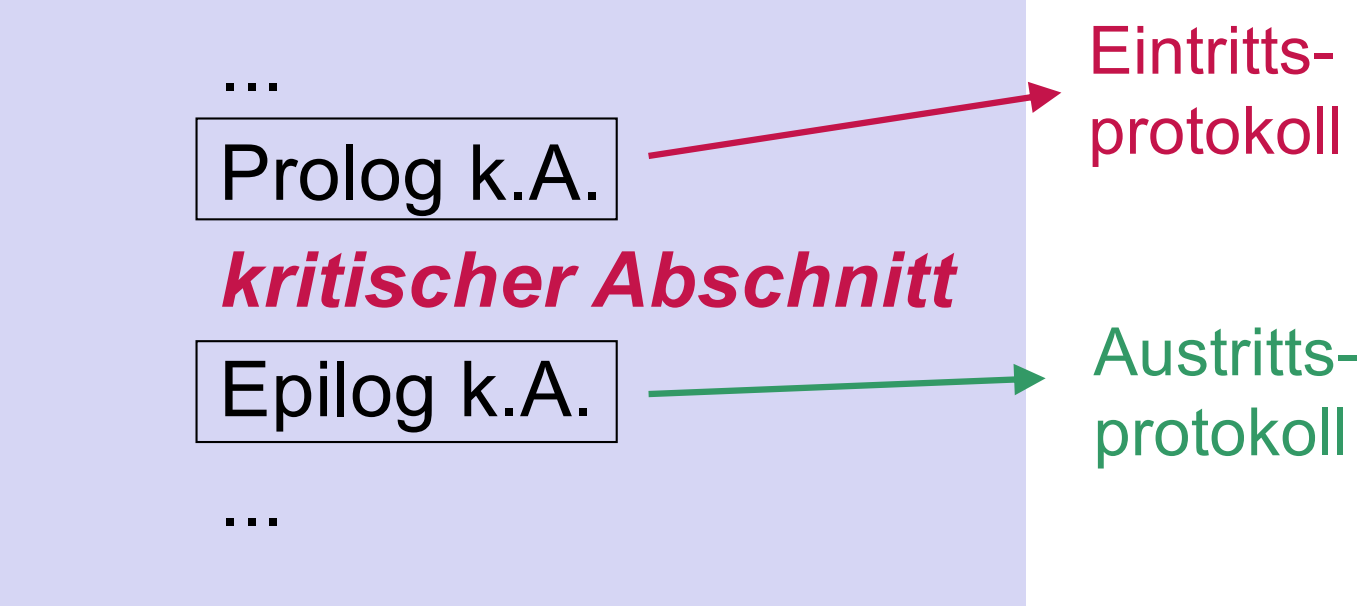


Kritischer Abschnitt

- Ein Prozess befindet sich im *kritischen Abschnitt* (k.A.), wenn er auf gemeinsame Daten bzw. Ressourcen zugreift
- *Wechselseitiger Ausschluss (mutual exclusion)*: zu jedem Zeitpunkt darf sich höchstens ein Prozess in seinem kritischen Abschnitt befinden
- Eintritt in k.A. muss geregelt erfolgen

Prozessstruktur für krit. Abschnitt

- kein Prozess darf in seinen kritischen Abschnitt eintreten, wenn sich bereits ein Prozess in seinem k.A. befindet
- Prozessstruktur



Anforderungen für k.A. Lösung

- Mutual Exclusion
- Progress
  - wenn sich kein Prozess im k.A. befindet und Prozesse in den k.A. eintreten möchten, darf die Entscheidung über den nächsten Prozess nicht unendlich lange verzögert werden
    (kein Lifelock, „After You“)
- Bounded Waiting
  - Nachdem ein Prozess einen Request für den k.A. abgegeben hat, ist die Anzahl der Eintritte in den k.A. durch andere Prozesse abgeschrankt
    (keine Starvation)

Arten von Lösungen

- Softwarelösungen
  - Annahmen: Atomizität einer Lese- bzw. Schreiboperation auf dem Speicher
- Hardwareunterstützte Lösungen
  - Mächtigere atomare Maschineninstruktionen
- Höhere Synchronisationskonstrukte
  - Vom API des Betriebssystems zur Verfügung gestellt
  - stellen dem Programmierer Funktionen und Datenstrukturen zur Verfügung
  - Semaphore, Monitor, Message Passing, …

Mutual Exclusion in Software

Softwarelösungen

- Zunächst 2 Prozesse, *P0* und *P1*
- dann Generalisierung auf *n* Prozesse, *Pi*, wobei *Pi* <> *Pj* für *i* <> *j*
- HW: atomares Lesen/Schreiben eines Wertes
- Synchronisation über globale Variable
- *Busy Waiting*: Warten auf das Eintreten einer Bedingung durch ständiges Testen (busy … benötigt CPU-Zeit)

while *Bedingung* ≠ true **do** *nothing*

Dekker-Algorithmus, 1. Versuch

global:

```
var turn: 0 .. 1;
```

Prozess *Pi*:

```
while turn ≠ i do nothing;
   critical section
turn = j;
   remainder section
```

- Mutual Exclusion gegeben
- Prozesse können nur abwechselnd in k.A.
- Wenn ein Prozess terminiert, wird der andere Prozess endlos blockiert

Dekker-Algorithmus, 2. Versuch

global:

```
var flag: array [0..1] of boolean;
```

Prozess *Pi*:

```
while flag[j] do nothing;
flag[i] := true;
   critical section
flag[i] := false;
   remainder section
```

Initialisierung:

```
flag[i] := false;
flag[j] := false;
```

- Wenn ein Prozess terminiert, kann der andere Prozess weiterarbeiten
- Mutual Exclusion nicht garantiert

Dekker-Algorithmus, 3. Versuch

global:

```
var flag: array [0..1] of boolean;
```

Prozess *Pi*:

```
flag[i] := true;
while flag[j] do nothing;
   critical section
flag[i] := false;
   remainder section
```

- Mutual Exclusion gesichert
- aber: gegenseitiges Blockieren (*Deadlock*) möglich

Dekker-Algorithmus, 4. Versuch

Prozess *Pi*:

```
flag[i] := true;
while flag[j] do
begin
   flag[i] := false;
   wait for a short time;
   flag[i] := true;
end;
   critical section
flag[i] := false;
   remainder section
```

- Mutual Exclusion gesichert, kein Deadlock
- aber: *Lifelock,* Prozessoraktivität ohne Produktivität

Dekker-Algorithmus

```
flag[i] := true;
while flag[j] do
   if turn = j then
   begin
      flag[i] := false;
      while turn = j do nothing;
      flag[i] := true;
   end;
      critical section
turn := j;
flag[i] := false;
   remainder section
```

flag … Zustand der beiden Prozesse

turn … bestimmt die Reihenfolge des Eintretens in den k.A. im Falle von völligem Gleichlauf
(siehe 4. Versuch)

Peterson-Algorithmus

eleganter, leichter zu durchschauen als Dekker Alg.

Prozess *Pi*:

```
loop
   flag[i] := true;
   turn := j;
   while flag[j] and turn = j do nothing;
      critical section
   flag[i] := false;
      remainder section
end loop
```

Initialisierung:

```
flag[i] := false;
flag[j] := false;
```

Peterson-Algorithmus

- Mutual Exclusion
  - *P0* setzt *flag[0]* auf *true* → *P1* blockiert
  - *P1* im k.A. → *flag[1]* = *true* → *P0* blockiert
- kein endloses gegenseitiges Blockieren

  Annahme: *P0* blockiert in while-Schleife
  (d.h. *flag[1]* = *true* und *turn* = *1*)
  - *P1* will nicht in k.A. >< *flag[1]* = *true*
  - *P1* wartet auf k.A. >< *turn* = 1
  - *P1* monopolisiert k.A. >< *P1* muss *turn* vor neuerlichem Eintritt in k.A. auf 0 setzen

Bakery Algorithm

Lösung des k.A.-Problems für *n* Prozesse

- Nummernvergabe für k.A.; Prozess mit der niedrigsten Nummer (>0) tritt als erstes ein
- Nummer 0 … keine Anforderung für k.A.
- *Pi* und *Pj* mit gleichen Nummern:
  Pi kommt vor *Pj* in k.A., wenn *i* < *j*
- es gilt stets: neu vergebene Nummer >= vorher vergebene Nummern

```
var     choosing: array [0..n-1] of boolean;
        number: array [0..n-1] of integer;
```

```
loop
    choosing [i] := true;
    number [i] := 1 + max(number [0], … , number [n-1]);
    choosing [i] := false;
    for j := 0 to n-1 do
        begin
          while choosing [ j] do nothing;
          while number [ j] ≠ 0 and
                    (number [ j],j) < (number [i],i) do nothing;
        end;
          critical section
    number [i] := 0;
          remainder section
end loop
```

Hardwareunterstützte Lösungen

- Interrupt Disabling während k.A.
  - für Uniprozessorsystem geeignet
  - aber: Einschränkung des BS beim Dispatching anderer Tasks
  - Rückgabe der Kontrolle an das BS?
  - Schutz von Instruktionssequenzen im BS
- Mächtigere atomare Maschineninstruktionen
  - Test and Set
  - Exchange (Swap)

Test and Set

- Hardwareinstruktion, die zwei Aktionen atomar (d.h. ununterbrechbar) ausführt
  → Mutual Exclusion

```
function testset (var i: integer): boolean;
begin
        if i = 0 then
        begin
            i := 1;
            return testset := true;
        end;
        else return testset := false;
end.
```

Test and Set für k.A.

globale Var.:

```
var b: integer;
b := 0;
```

Prozess *Pi*:

```
while not testset(b) do nothing;
   critical section
b := 0;
   remainder section
```

Sicherung des k.A. mit „exchange“

globale Var.:

```
var b: integer;
b := 0;
```

Prozess *Pi*:

```
var key: integer;
key := 0;
```

```
key := 1;
do exchange(key, b) while key = 1;
   critical section
exchange (key, b);
   remainder section
```

Test and Set bzw. *Exchange*:

- einfacher Prolog für den k.A. für beliebig viele Prozesse
- *Starvation* von Prozessen möglich

Busy Waiting

- Bisherige Lösungen verwenden Busy Waiting: Iterieren in einer Schleife, bis eine Synchronisationsbedingung erfüllt ist
- Prozess verschwendet CPU-Zeit mit Warten
- *Kein* geeigneter Mechanismus zur Synchronisation für Benutzer-Code

⇨ Blockieren der wartenden Prozesse in *Blocked Queues* (siehe Abschnitt „Prozesse“)

Semaphore

Semaphor

- Synchronisationskonstrukt ohne Busy Waiting (vom BS zur Verfügung gestellt)
- Semaphor S: Integer-Variable, auf die nach der Initialisierung nur mit zwei atomaren Funktionen, *wait* und *signal,* zugegriffen werden kann

Semaphore

Ziel: Sichern eines kritischen Abschnitts auf “einfache Art”

```
wait (S);
   critical section
signal (S);
   remainder section
```

Semaphore - Datenstruktur

- Semaphor ist ein Record:

```
type semaphore = record
                     value: integer;
                     queue: list of process;
                 end;
```

- Auf einen Semaphor S wartende Prozesse werden in die *Blocked Queue* von S (S.queue) gestellt

Semaphore - Operationen

```
init (S, val):    S.value := val; S.queue := empty list;

wait (S):         S.value := S.value - 1;
                  if S.value < 0
                  then   add this process to S.queue
                         and block it;

signal (S):       S.value := S.value + 1;
                  if S.value <= 0
                  then   remove a process P from S.queue
                         and place P on ready queue;
```

Semaphore müssen auf einen nicht-negativen Wert (für Mutual Exclusion auf 1) initialisiert werden

Wechselseitiger Ausschluss

Initialisierung:

```
init (S, 1)
```

Prozess *Pi*:

```
wait (S);
   critical section
signal (S);
   remainder section
```

- Maximal ein Prozess im k.A.
- Reihenfolge, in der Prozesse in den k.A. eintreten?
  → FIFO Queue → Bounded Waiting, Fairness

Semaphore - Implementierung

- *wait* und *signal* müssen als atomare Operationen ausgeführt werden
- Sichern der Semaphoroperationen (= k.A.) mit Test and Set
  - zusätzliche Record-Komponente flag in der Semaphordatenstruktur
  - Vor k.A.: Busy Waiting mit Test von flag
  - da *wait* und *signal* sehr kurz sind, kommt es kaum zum Busy Waiting; daher ist der Overhead vertretbar

Semaphore - Bemerkungen

- S.count >= 0: Anzahl der Prozesse, die hintereinander ein *wait* ausführen können, ohne zu blockieren
- Initialisierung von S mit Wert *n*
  → *n* Prozesse können vor dem ersten Blockieren die wait-Operation passieren
  → allgemeinere Synchronisationsaufgaben (z.B. *n* Ressourcen verfügbar)
- S.count < 0: |S.count| Prozesse in der Queue von S blockiert (in gezeigter Implementierung)

Semaphore - Bemerkungen

- *Binary Semaphore*: nimmt nur die Werte 0 oder 1 an
- *Counting Semaphore*: kann beliebige (ggf. nicht negative) ganzzahlige Werte annehmen - implementierungsabhängig
- Operationen *wait* und *signal* werden in der Literatur auch als

  P (*proberen*/*passeren*) bzw.

  V (*vrijgave*/*verhogen*) bezeichnet

Bedingungssynchronisation

- Prozesse *P1* und *P2*
- Codeabschnitt *C1* in *P1* muss vor Abschnitt *C2* in *P2* ausgeführt werden
- Semaphor für *Bedingungssynchronisation (Condition Synchronisation)*

Initialisierung:

```
init (S, 0)
```

P1:

```
C1;
signal (S);
```

P2:

```
wait (S);
C2;
```

Abwechselnder Zugriff

- Beispiel: Datentransfer von *P1* an *P2*
- *P1* schreibt, *P2* liest Shared Memory
- kein Duplizieren bzw. Verlust von Daten

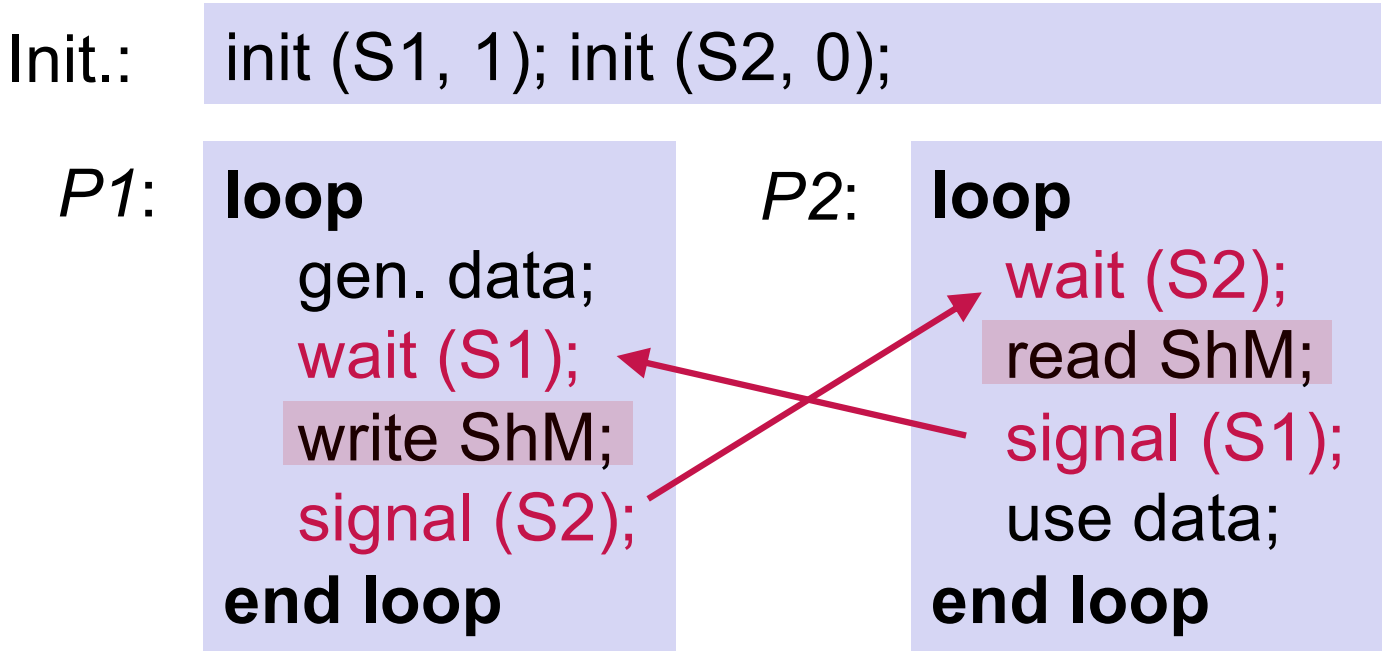
Init.:
```
init (…
```

P1:
```
loop
   gen. data;

   write ShM;

end loop
```

P2:
```
loop

   read ShM;

   use data;
end loop
```

Abwechselnder Zugriff

- Beispiel: Datentransfer von *P1* an *P2*
- *P1* schreibt, *P2* liest Shared Memory
- kein Duplizieren bzw. Verlust von Daten



Aufgabe

- Gegeben: 3 zyklische Prozesse, *A*, *B*, *C*
- *A* produziert Daten und schreibt sie auf ShM
- *B* und *C* lesen die Daten vom ShM (ohne Einschränkung der Parallelität)
- Jeder von *A* geschriebene Datensatz soll von *B* und *C* genau einmal gelesen werden

???

Aufgabe

Init.:
```
init (…
```

A:
```
loop
   gen. data;

   write ShM;

end loop
```

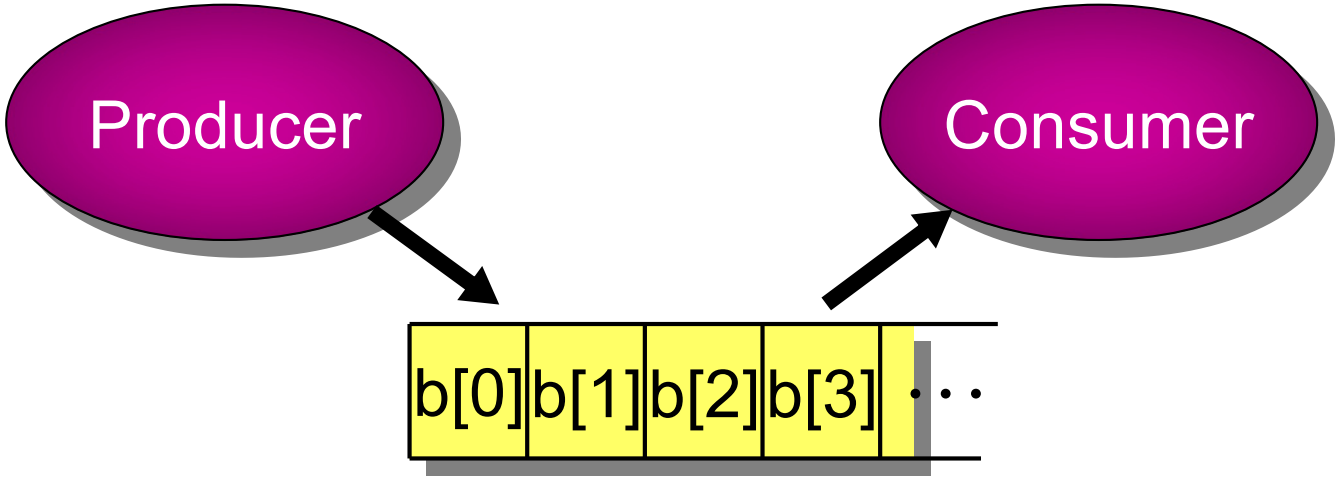
B:
```
loop

   read ShM;

   use data;
end loop
```

C:
```
loop

   read ShM;

   use data;
end loop
```

Semaphore – Beispiele

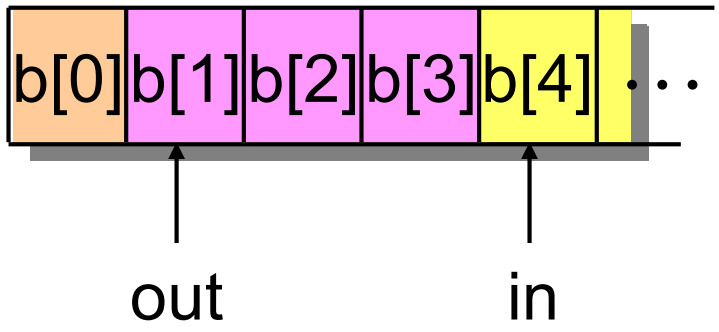
Producer-Consumer Problem

- Produzenten generieren Informationen, die von Konsumenten gelesen werden
  - Beispiel: Druckaufträge an einen Drucker
- Einzelne Datensätze werden über einen Puffer an Konsumenten übergeben



P-C mit unbegrenztem Puffer

- Produzent kann jederzeit ein Datenelement schreiben
- Konsument muss auf Daten warten
- “in” zeigt auf nächstes freies Puffer-Element
- “out” zeigt auf nächstes zu lesendes Element



P-C: Semaphore

- Mutual Exclusion: zu jedem Zeitpunkt darf nur ein Prozess auf den Puffer zugreifen (Semaphore “S”)
- Bedingungssynchronisation: ein Konsument darf nur dann lesen, wenn mindestens ein ungelesenes Datenelement im Puffer steht (Counting Semaphore “N” spiegelt Anzahl der Elemente im Puffer wieder)

P-C: Implementierung

Initialisierung: `init (S, 1);  init(N, 0);  in := out := 0;`

```
append (v):
  b[in] := v;
  in := in + 1;

take ( ):
  w := b[out];
  out := out + 1;
  return w;
```

Producer:

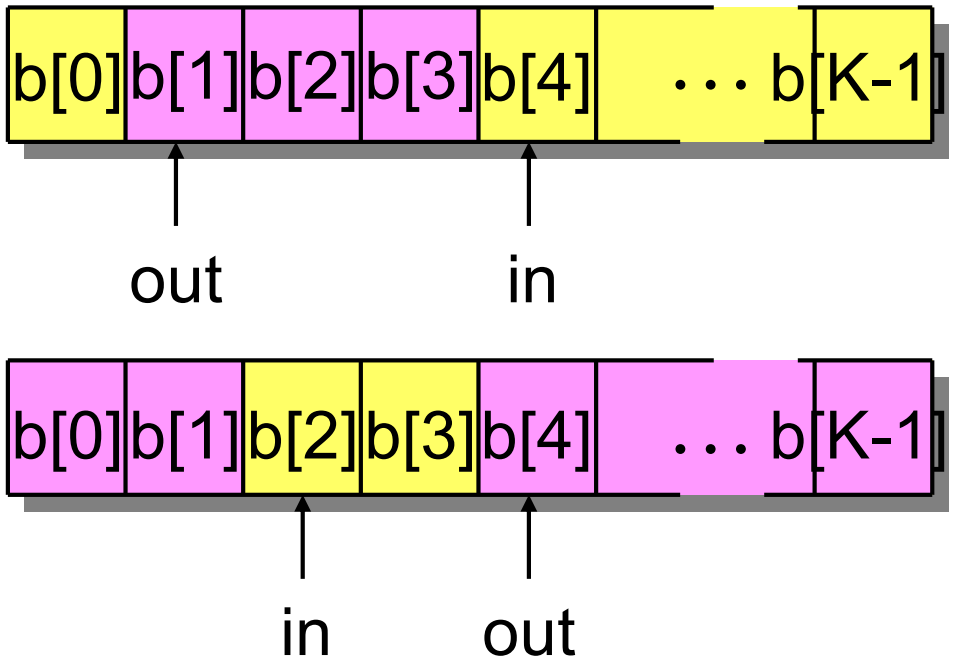
```
loop
    produce (v);
    P (S);
    append (v);
    V (S);
    V (N);
end loop
```

Consumer:

```
loop
    P (N);
    P (S);
    w := take ( );
    V (S);
    consume (w)
end loop
```

P-C mit Ringpuffer

- Begrenzter Puffer mit *K* Elementen
- Lesen: mind. ein “neuer” Wert notwendig
- Schreiben: mind. ein Element frei



Ringpuffer: Semaphore

- Semaphore wie bei unbegrenztem Puffer
  - Mutual Exclusion: zu jedem Zeitpunkt darf nur ein Prozess auf den Puffer zugreifen (S)
  - Bedingungssynchronisation: der Konsument darf nur dann lesen, wenn mindestens ein ungelesenes Datenelement im Puffer steht (N)
- Bedingungssynchronisation: ein Produzent darf nur dann schreiben, wenn mindestens ein leerer Speicherbereich vorhanden ist (E)

P-C Ringpuffer-Implementierung

Initialisierung:

```
init (S, 1);   init (N, 0);   init (E, K);
in := out := 0;
```

```
append (v):
  b[in] := v;
  in := (in + 1)
        mod K;

take ( ):
  w := b[out];
  out := (out + 1)
         mod K;
  return w;
```

Producer:

```
loop
    produce (v);
    P (E);
    P (S);
    append (v);
    V (S);
    V (N);
end loop
```

Consumer:

```
loop
    P (N);
    P (S);
    w := take ( );
    V (S);
    V (E);
    consume (w)
end loop
```

Reihenfolge von P und V

Initialisierung:

```
init (S, 1);   init (N, 0);   init (E, K);
in := out := 0;
```

```
append (v):
  b[in] := v;
  in := (in + 1)
        mod K;

take ( ):
  w := b[out];
  out := (out + 1)
         mod K;
  return w;
```

Producer:

```
loop
    produce (v);
    P (E);
    P (S);
    append (v);
    V (S);
    V (N);
end loop
```

Consumer:

```
loop
    P (S);
    P (N);
    w := take ( );
    V (S);
    V (E);
    consume (w)
end loop
```

Reihenfolge von P und V

Reihenfolge von V-Operationen “beliebig”
Achtung: Abfolge von P-Operationen ist relevant!

Producer:

```
produce (v);
P (E);
P (S);
append (v);
```

Consumer:

```
…
P (S);
P (N);
w := take ();
```

vertauscht!
falsch!!!

Systemverklemmung (Deadlock), wenn Consumer bei leerem Puffer (N=0) in k.A. eintritt.

Reader-Writer Problem

- Es gibt Lese- und Schreibeprozesse, die auf eine gemeinsame Ressource zugreifen
- Beliebig viele Leser dürfen parallel lesen
- Schreiber benötigen exklusiven Zugriff

Reader-Writer Problem

```
init (x, 1); init (y, 1); init (z, 1); init (wsem, 1); init (rsem, 1);
rc := 0; wc := 0;
```

Reader:

```
loop
    P (x);
    rc := rc + 1;
    if rc = 1 then P (wsem);
    V (x);
      read;
    P (x);
    rc := rc - 1;
    if rc = 0 then V (wsem);
    V (x);
end loop
```

Writer:

```
loop
    P (wsem);
      write;
    V (wsem);
end loop
```

Reader haben Priorität

Reader-Writer Problem

Reader:

```
loop
    P (z);
    P (rsem);
    P (x);
    rc := rc + 1;
    if rc = 1 then P (wsem);
    V (x);
    V (rsem);
    V (z);
       read;
    P (x);
    rc := rc - 1;
    if rc = 0 then V (wsem);
    V (x);
end loop
```

Writer:

```
loop
    P (y);
    wc := wc + 1;
    if wc = 1 then P (rsem);
    V (y);
    P (wsem);
       write;
    V (wsem);
    P (y);
    wc := wc - 1;
    if wc = 0 then V (rsem);
    V (y);
end loop
```

Writer haben Priorität

Reader-Writer Problem

Reader:

```
loop
    P (z);
    P (rsem);
    P (x);
    rc := rc + 1;
    if rc = 1 then P (wsem);
    V (x);
    V (rsem);
    V (z);
       read;
    P (x);
    rc := rc - 1;
    if rc = 0 then V (wsem);
    V (x);
end loop
```

Writer:

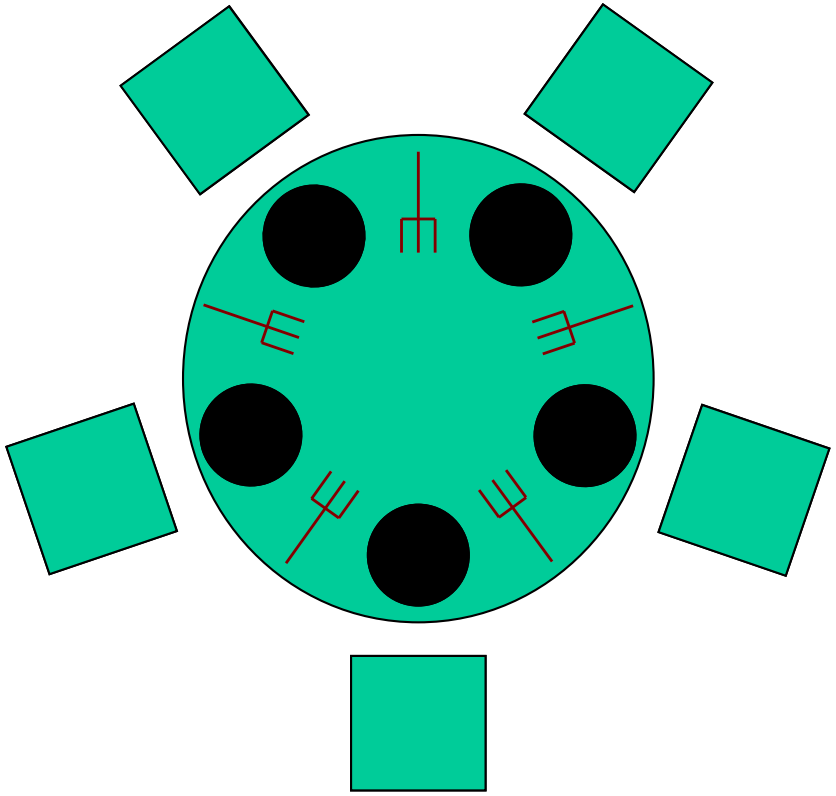
```
loop
    P (rsem);
    P (wsem);
       write;
    V (wsem);
    V (rsem);
end loop
```

Writer haben Priorität,
Keine Starvation der *Reader*

Vergleich mit voriger Lösung?

Dining Philosophers Problem

- klassisches Problem zur Synchronisation (Lehre)
- 5 Philosophen denken und essen
- jeder braucht zum Essen zwei Gabeln
- gesucht: Lösung ohne Deadlock und Starvation



Dining Philosophers Problem

- Jedem Philosophen entspricht ein Prozess
- Ein Semaphor pro Gabel

```
fork: array[0..4] of
        semaphore;

foreach i in [0..4]
    init (fork[i], 1);
```

- Erster Versuch:

Prozess Pi:

```
loop
    think;
    P (fork[i]);
    P (fork[(i+1) mod 5]);
    eat;
    V (fork[(i+1) mod 5]);
    V (fork[i]);
end loop
```

Erster Versuch: Deadlock, wenn alle Philosophen gleichzeitig die erste Gabel aufnehmen

Dining Philosophers: Lösungen

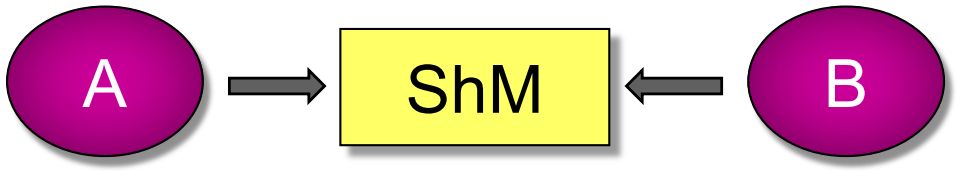
- Zusätzlicher Semaphor, der maximal 4 Philosophen gleichzeitig das Aufnehmen einer Gabel erlaubt
- Mindestens ein Prozess nimmt die Gabeln in umgekehrter Reihenfolge auf
- atomare P-Operation für mehrere Semaphore

```
mP (fork[i], fork[(i+1) mod 5]);
```

P und V für mehrere Semaphore

- *mP*, *mV*: atomare Operationen *P* und *V* für eine Menge von Semaphoren
- *mP (S1, S2, …, Sn)*: blockiert solange, bis alle Semaphore *S1* bis *Sn* größer als 0 sind
  - Löst Problem der Reihung von *P*-Operationen
- *mV (S1, S2, …, Sn)*: erhöht alle Semaphore *S1* bis *Sn* um eins

Aufgabe



- Gegeben: zwei Typen von Prozessen: *A*, *B*
- Prozesse von Typ *A* und *B* greifen exklusiv auf das ShM zu
- Prozesse vom Typ *A* haben Priorität gegenüber Prozessen vom Typ *B*

???

Probleme mit Semaphoren

- Semaphoroperationen sind über Prozesse verteilt, daher unübersichtlich
- Operationen müssen in allen Prozessen korrekt verwendet werden
- Ein fehlerhafter Prozess bewirkt Fehlverhalten aller Prozesse, die zusammenarbeiten

⇨ Monitore

Monitor, Nachrichten

und andere Synchronisationsmechanismen

Monitor

- Softwaremodul, bestehend aus:
  - Prozeduren
  - Lokalen Daten
  - Initialisierungscode
- Eigenschaften
  - Zugriff auf lokale Variable: Monitorprozeduren
  - Eintritt von Prozessen in den Monitor über Monitorprozeduren
  - max. 1 Prozess zu jedem Zeitpunkt im Monitor

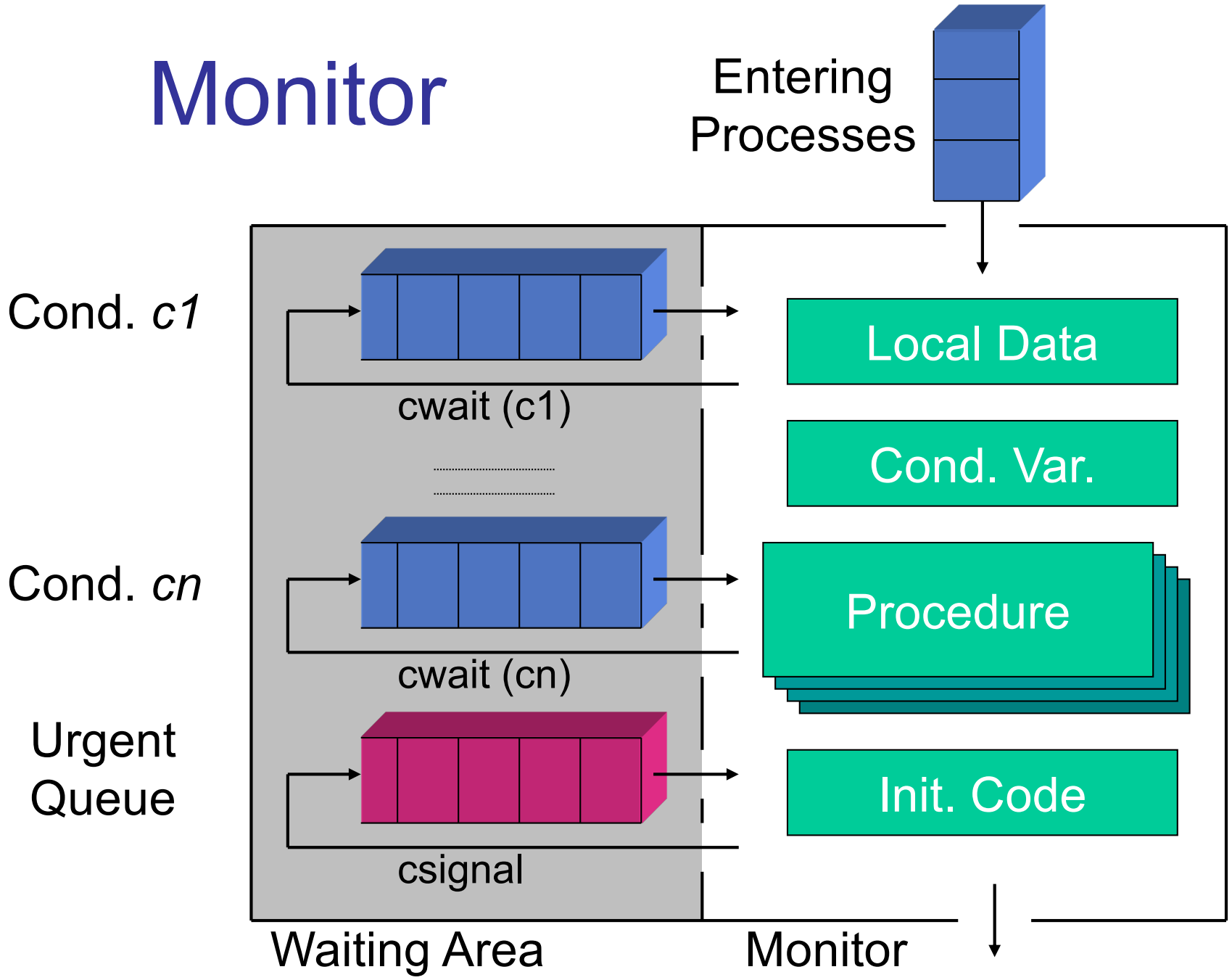
Monitor

- Monitor sorgt für Mutual Exclusion, kein explizites Programmieren
- Gemeinsamer Speicher ist im Monitorbereich anzulegen
- Bedingungssynchronisation über Monitorvariable
  - Programmierung von Wartebedingungen für Bedingungsvariable *(Condition Variables)*
  - Monitoreintritt, wenn Wartebedingungen der Bedingungsvariablen erfüllt sind

Condition Variables

- Lokal im Monitor (Zugriff nur im Monitor)
- Zugriff nur über die Zugriffsfunktionen
  - *cwait (c)*: blockiert aufrufenden Prozess bis Bedingung(svariable) *c* den Wert *true* annimmt
  - *csignal (c)*: Setze einen Prozess, der auf die Bedingung *c* wartet, fort
    - wenn mehrere Prozesse warten, wähle einen aus
    - wenn kein Prozess wartet, keine Aktion
    - keine speichernde Wirkung (≠ Semaphor-Wait)

Monitor


Entering Processes
Cond. c1
cwait (c1)
Cond. cn
cwait (cn)
Urgent Queue
csignal
Local Data
Cond. Var.
Procedure
Init. Code
Waiting Area
Monitor

Monitor: Producer-Consumer

```
monitor ProducerConsumer

    b: array[0..K-1] of items;
    In := 0,  Out := 0,  cnt := 0 : integer;
    notfull, notempty: condition;

append (v):
    if (cnt = K) cwait (notfull);
    b[In] := v;
    In := (In + 1) mod K;
    cnt := cnt + 1;
    csignal (notempty);

take (v):
    if (cnt = 0) cwait (notempty);
    v := b[Out];
    Out := (Out + 1) mod K;
    cnt := cnt - 1;
    csignal (notfull);
```

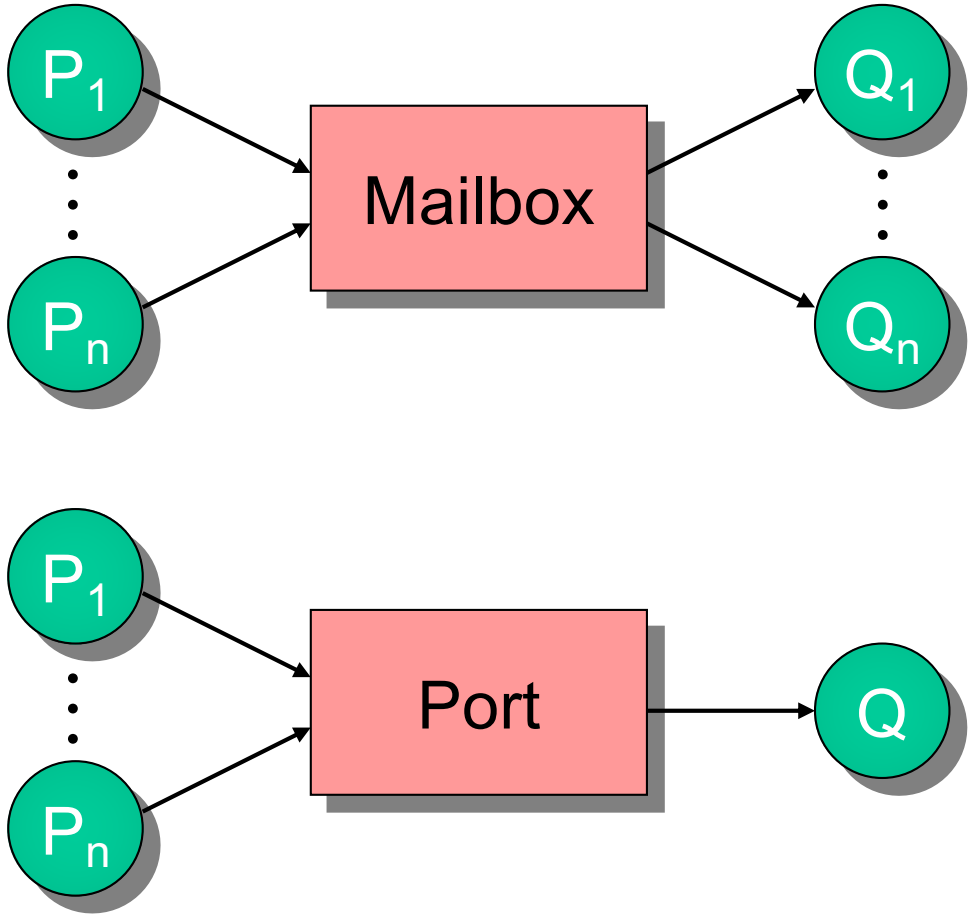
Message Passing

- Methode zur Interprozesskommunikation (IPC)
  - in zentralen und verteilten Systeme
- Eine Nachricht ist eine atomare Datenstruktur
  - ⇨ Datenkonsistenz
- Verschiedene Synchronisationssemantiken
- Funktionen zum Senden und Empfangen
  - *send (destination, message)*
  - *receive (source, message)*

Charakteristika von Nachrichten

- Synchronisation
  - send: blocking, non-blocking
  - receive: blocking, non-blocking, test for arrival
- Datenverwaltung
  - Ereignissemantik: Warteschlange (Queue)
  - Zustandssemantik: empfangene Daten überschreiben alte Werte
- Adressierung
  - 1:1 vs. 1:N
    physikalisch vs. logisch
    direkt vs. indirekt (Mailbox, Portadressierung)

Mailbox und Portadressierung



Ereignisnachricht für MutEx

- Prozesse verwenden eine gemeinsame Mailbox *mutex* und eine Nachricht (Token)
- Blocking *receive*, non-blocking *send*

init:

```
send (mutex, msg);
```

Prozess *Pi*:

```
loop
    receive (mutex, msg);
        critical section;
    send (mutex, msg);
        remainder section;
end loop
```

Locks

- Einfacher Sync.-Mechanismus
  - *enter (lock)*: blockiert nachfolgende *enter*-Aufrufe
  - *release (lock)*: gibt Lock frei

Sequencer und Eventcounts

Eventcount E: ganzzahliger Ereigniszähler

- *advance (E)*: erhöht E um 1 (Anfangswert: 0)
- *await (E, val)*: blockiert bis *E* >= *val*

Sequencer S: ganzzahliger "Nummernserver"

- *ticket (S)*: retourniert Wert von *S* und inkrementiert *S* (Anfangswert: 0)

Bsp. k.A.:

```
myticket = ticket (S));
await (E, myticket);
    critical section;
advance (E);
```

Zusammenfassung

- Anforderungen paralleler Prozesse
  - Konsistenter Zugriff auf gemeinsame Daten
  - Vorgegebene Reihenfolge von Aktionen
- Mutual Exclusion → Konsistenz
- Condition Synchronization → Reihenfolge
- Kritischer Abschnitt
  - Aktionen, die gemeinsame Daten manipulieren
  - mit Konstrukten zur Synchronisation gesichert

Zusammenfassung

- Sicherung des kritischen Abschnitts
  - Lösungen abhängig vom Instruction Set
  - Unterstützung durch Betriebssystem
    - ➪ kein Busy Waiting!
- Semaphor
  - *init*, *wait* bzw. *P* und *signal* bzw. *V*
  - Achtung: Reihenfolge der *wait* Operationen
- Monitor
- Nachrichten