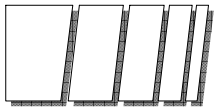


Grundkonzepte paralleler Programmierung

=> MPD (Multi-threaded Parallel Distributed)

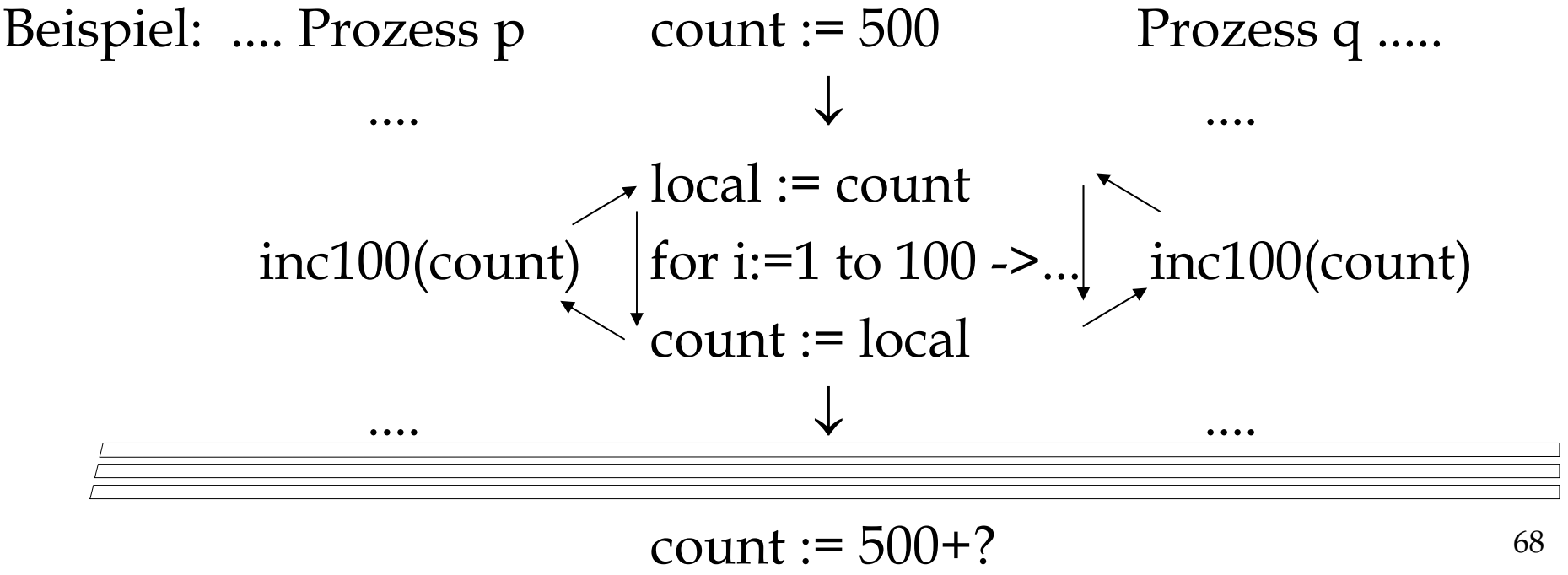
Andrews (Arizona University)

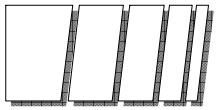
http://www.cs.arizona.edu/mpd

Synchronisation von Speicherzugriffen

Problem:

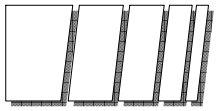
- Speicherzugriffe auf gemeinsame Variablen müssen koordiniert werden
- Nicht-atomare Operationen erfordern exklusiven Zugriff auf globale Daten





Beispiel: MPD-Programm counter.mpd

```
resource counter()
  int count = 0
  # increment count  by 100
  process inc1000 [id = 1 to 10] {
    int local;     int top = 100
    for [j=1 to id] {
      # increment global counter - critical section
      local = count;   writes(id,": get counter = ", local);  write()
      for [k = 1 to top] {   local += 1   };     nap(int(random()*100))
      count = local;   writes(id,": put counter = ", local); write()
      # wait for some time - non-critical section
      nap(int(random()*100))
  } }
  final
    { write(count) }
end  counter
```



Synchronisationsformen

1. Wechselseitiger Ausschluss (mutual exclusion)

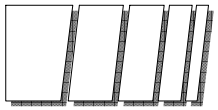
   Bsp: geschützter Zugriff auf gemeinsame Ressourcen (s.o.)

2. einseitige Synchronisation
   Bedingungssynchronisation (condition synchr.)
   Ereignissynchronisation (event synchr.)
   Ein Prozess muss auf eine Bedingung oder ein Ereignis warten, das von einem anderen Prozess ausgelöst wird.

   Bsp: Erzeuger-Verbraucher mit (un-)beschränktem Puffer ((un-)bounded buffer)

3. Barrierensynchronisation (Verallgemeinerung von 2.)
   Eine Gruppe von Prozessen muss an einer sog. Barriere warten, bis alle Prozesse der Gruppe die Barriere erreicht haben.

Bsp: Synchronisation von Schleifendurchläufen paralleler Prozesse

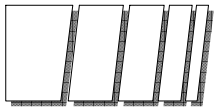


Race Conditions

- Situationen, in denen das „Wettrennen“ (race) der Prozesse beim Zugriff auf gemeinsame Ressourcen Auswirkungen auf das Ergebnis eines Programmlaufs hat.

- Durch Synchronisationen werden „race conditions“ vermieden. Das Wettrennen unter den Prozessen wird eingeschränkt.

- Gefahren durch unpassende Synchronisationen
  - Verklemmungen (deadlocks)
  - Aushungerungen (starvations)



Synchronisationskonstrukte: Semaphore (Dijkstra 1965)

Ein Semaphor ist ein abstrakter Datentyp mit

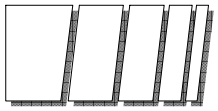
- nicht-negativer Integer-Variablen (Semaphorzähler)
- zwei unteilbaren (atomaren) Operationen P und V
  P => Passieren, V => Verlassen

Bei der Initialisierung wird dem Semaphor ein nicht-negativer Wert zugewiesen. Anschließend ist nur noch eine Manipulation mit den Operationen P und V möglich.

P(S): atomar
Wenn S>0,
dann S := S-1
sonst wird der Prozess, der P(S) ausführt, suspendiert

V(S): atomar
Wenn Prozesse bei Ausführung von P(S) suspendiert wurden
dann reaktiviere einen Prozess
sonst S := S+1

binäres Semaphor: Initialisierung mit 1



Semaphore in MPD

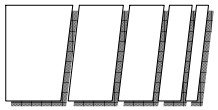
- Semaphor-Deklaration: sem sem_def, sem_def
- Semaphor-Definition: sem id [subscripts] = expr
  - einzelnes Semaphor
  - Feld von Semaphoren

- Operationen: P(sem_id [subscripts])
  V(sem_id [subscripts])

Wechselseitiger Ausschluss

```
resource     Critical_Section ()
  const N := 20   # Anzahl Prozesse
  var   X := 0    # gemeinsame Variable
  sem mutex := 1  # Semaphor zum Schutz von X
  process p (i:= 1 to N)
      # non-critical section
      ...
      # critical section
      P(mutex)
      X:=X+1
      V(mutex)
      # non-critical section
      ...
  end
end
```

Warteschlange ist als first-come-first-served FIFO-Schlange organisiert.



Beispiel: MPD-Programm counter.mpd mit Semaphor

```
resource counter()
  int count = 0; sem mutex := 1;
  # increment count  by 100
  process inc1000 [id = 1 to 10] {
    int local;     int top = 100
    for [j=1 to id] {
      # increment global counter - critical section
      P(mutex)
      local = count;   writes(id,": get counter = ", local);  write()
      for [k = 1 to top] {   local += 1   };     nap(int(random()*100))
      count = local;   writes(id,": put counter = ", local); write()
      V(mutex)
      # wait for some time - non-critical section
      nap(int(random()*100))
  } }
  final      { write(count) }
end  counter
```

Einseitige Synchronisation: Erzeuger/Verbraucher-Problem

Erzeuger produziert ➔ **Verbraucher konsumiert**

Annahme: unbeschränkter Puffer

```
resource Producer-Consumer ();
...
sem full = 0; sem mutex = 1;
...
process producer ()
   var item : int
   while (true) {
      produce (item)
      P(mutex)
      enter (item)
      V(mutex)
      V(full) }
```

```
...
process consumer ()
   var item : int
   while (true) {
      P(full)
      P(mutex)
      remove (item)
      V(mutex)
      consume (item)}
```

Zyklische Synchronisation: P1 > P2 > P3 ...

```
resource     Critical_Section ()
  const N := 20   # Anzahl Prozesse
  var   X := 0    # gemeinsame Variable
  sem mutex[N] := (1, [N-1] 0)  # Semaphorfeld
  process p (i:= 1 to N)
      # non-critical section
      ...
      # critical section
      P(mutex[i])
      X:=X+1
      V(mutex[(i mod N)+1])
      # non-critical section
      ...
  end
end
```

Fallstudie: Leser-/Schreiber-Problem

Mehrere Prozesse arbeiten auf gemeinsamem Speicherbereich.
Gleichzeitige Lesezugriffe sind erlaubt. → CREW (concurrent read
Schreibzugriffe müssen exklusiv erfolgen. exclusive write)

Lösungsansatz [Courtois, Heymans, Parnas 71, ACM]

Idee: Verwalte Zähler readcount für Leseranzahl

→ 2 Semaphore: sem rcount_mutex := 1 (Schutz für readcount)
sem writing := 1 (Sperre für exklusiven Schreibzugriff)

Schreibprozesse:

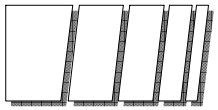
```
process writer ()
  while (true) {
   <Daten erzeugen>
   P(writing)
   <Daten schreiben>
   V(writing) }
end
```

Leseprozesse

Leseranmeldung:
Erster Leser besorgt
Speicherfreigabe.

Leserabmeldung:
Letzter Leser gibt
Speicher frei.

```
process reader ()
   while (true) {
      P(rcount_mutex)
      if readcount == 0 { P(writing) }
      readcount += 1
      V(rcount_mutex)
      <Daten lesen>
      P(rcount_mutex)
      readcount -= 1
      if readcount == 0 { V(writing) }
      V(rcount_mutex)
   }
end
```

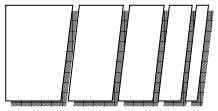


Korrektheit

Das Semaphor writing schützt den Speicherbereich.

- Schreiber aktiv => writing ist gesetzt
  - Kein weiterer Schreiber kann passieren.
  - Erster Leser blockiert bei P(writing).
  - Weitere Leser blockieren bei P(rcount_mutex).

- Leser aktiv => writing ist durch ersten Leser gesetzt.
  - Schreiber wird bei P(writing) blockiert.
  - Weitere Leser passieren writing nicht, sondern erhöhen nur den Lesezähler.

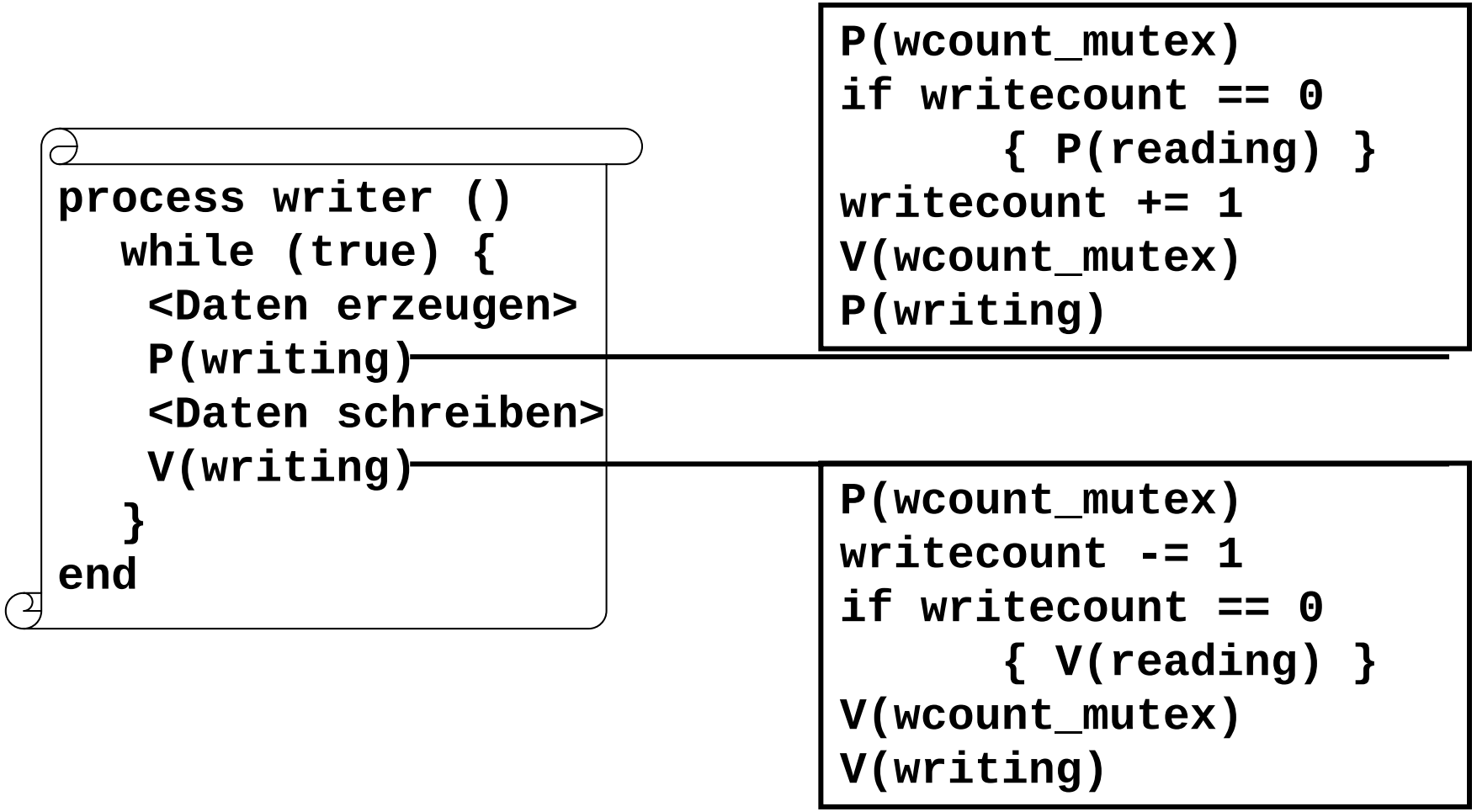


Problem: Schreiberaushungerung

- Wenn ständig neue Leser hinzukommen, wird readcount nie Null und die Schreiber erhalten keinen Zugang zum kritischen Bereich.
- Besser: Blockiere neue Leser, sobald ein Schreiber wartet.
- Führe neuen Zähler writecount für Leser und (wartende) Schreiber ein.
- neue Semaphore:
  - sem wcount_mutex = 1 (Schutz für Schreiberzähler)
  - sem reading = 1 (Sperre für Leser, falls Schreiber warten)

=>zusätzliche Anmeldung für Schreiberprozesse, um neue Leser zu stoppen

Schreiberan- und -abmeldung



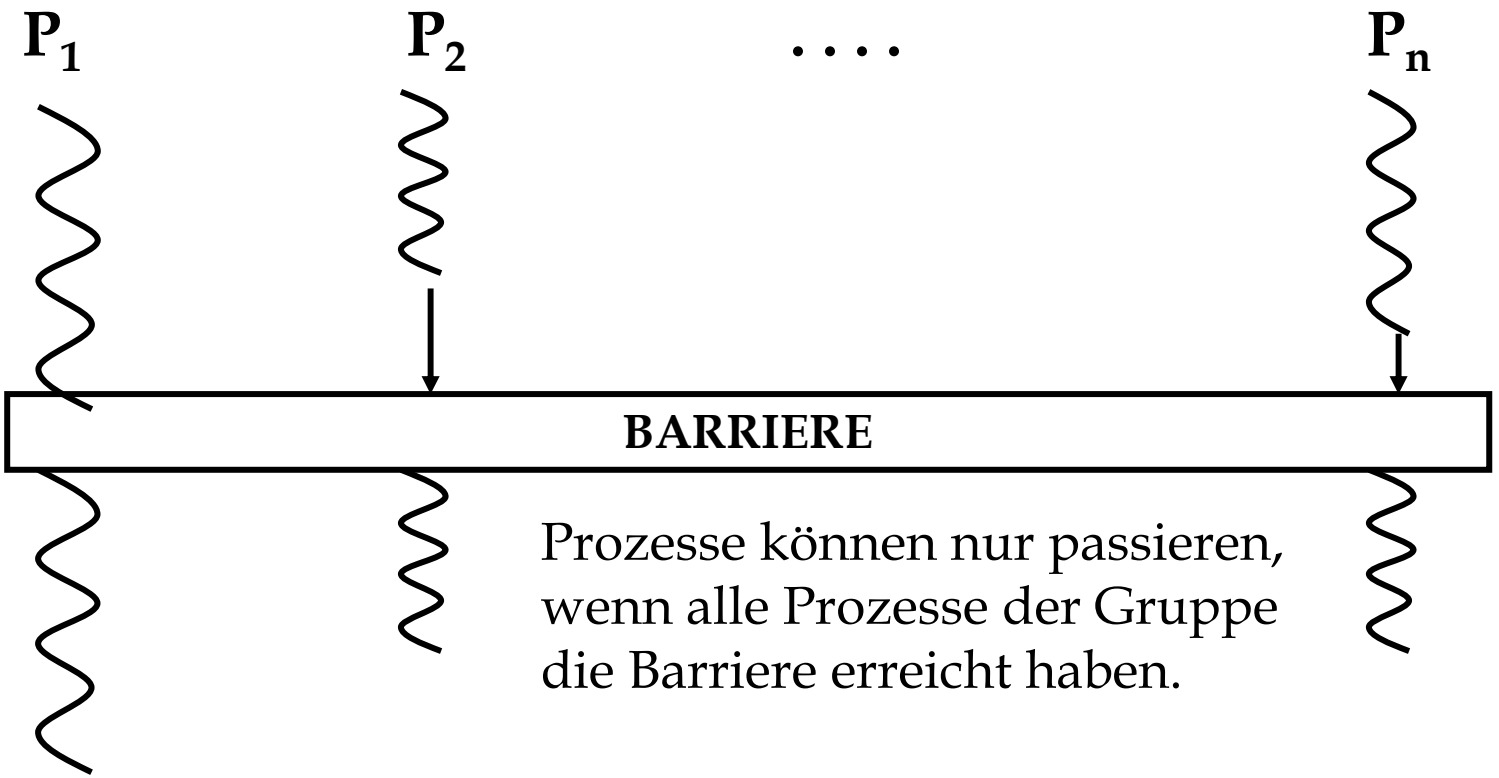
Modifikation der Leseranmeldung

```
process reader ()
  while (true) {
    P(reading)
    P(rcount_mutex)
    if readcount == 0 { P(writing) }
    readcount += 1
    V(rcount_mutex)
    V(reading)
    <Daten lesen>
    P(rcount_mutex)
    readcount -= 1
    if readcount == 0 { V(writing) }
    V(rcount_mutex)
  }
end
```

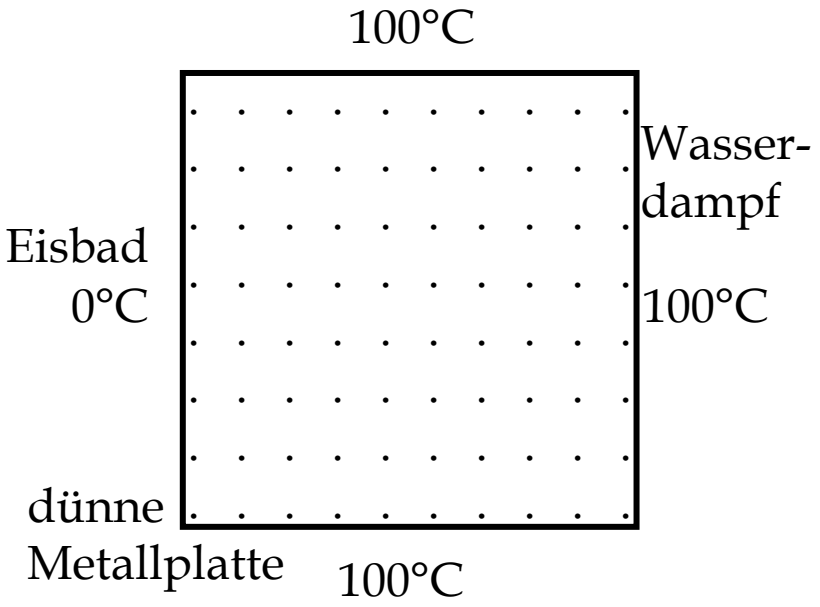
blockiert neue Leser, falls
- Schreiber wartet oder
- Leser bei Anmeldung

damit erster Schreiber nicht blockiert wird

Barrierensynchronisation



Bsp: Iterationsverfahren zum Lösen partieller Differentialgleichungen



zweidimensionales Temperaturverteilungsproblem:
Ermittle die Temperaturverteilung an den Gitterpunkten im stabilen Zustand

Im stabilen Zustand gilt:

$$\varphi_{x,y} = 1/4\,(\varphi_{x-1,y} + \varphi_{x+1,y} + \varphi_{x,y-1} + \varphi_{x,y+1})$$

Iterationsverfahren nach Jacobi (1845) – Gesamtschrittverfahren:

$\varphi_{x,y}^{0}$ = **geschätzter Wert, Anfangswert**

$$\varphi_{x,y}^{i+1} = 1/4\,(\varphi_{x-1,y}^{i} + \varphi_{x+1,y}^{i} + \varphi_{x,y-1}^{i} + \varphi_{x,y+1}^{i})$$

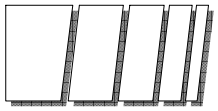
$fn^2/2$ Iterationen garantieren Fehler unter 10^{-f} bei Problemgröße n (=#Gitterpunkte)

Paralleles MPD-Programm mit co ... oc

```
resource Jacobi()
  const int n = 5; const int f = 3; int numiter
  real a [0:n+1,0:n+1], b [0:n+1,0:n+1]
  ...
   # Iteration zur Bestimmung der Werte im stabilen Zustand
  for [k=1 to numiter]  {
       # 1. Phase : Neuberechnung der Werte
       co [i=1 to n]  compute_row(i) oc

       # 2. Phase : Update
       co  [i=1 to n]  update_row(i) oc
   }
   ...
end
```

Analyse:
neue Prozesse für jede Iteration und jede der beiden Phasen
besser:
Prozesse nur einmal erzeugen; Synchronisation mittels Barriere



Lineare Barriere mittels Zählvariable

- einfachste Form der Barriere
- Verwaltung eines Zählers
- Sperren aller aufrufenden Prozesse, bis Zähler Maximalwert erreicht, danach Freigabe aller Prozesse
- mit Semaphoren:
  - sem arrival = 1 (Freigabe des Barrierenzugangs)
  - sem departure = 0 (Sperren des Barrierenabgangs)
  - var count : int = 0

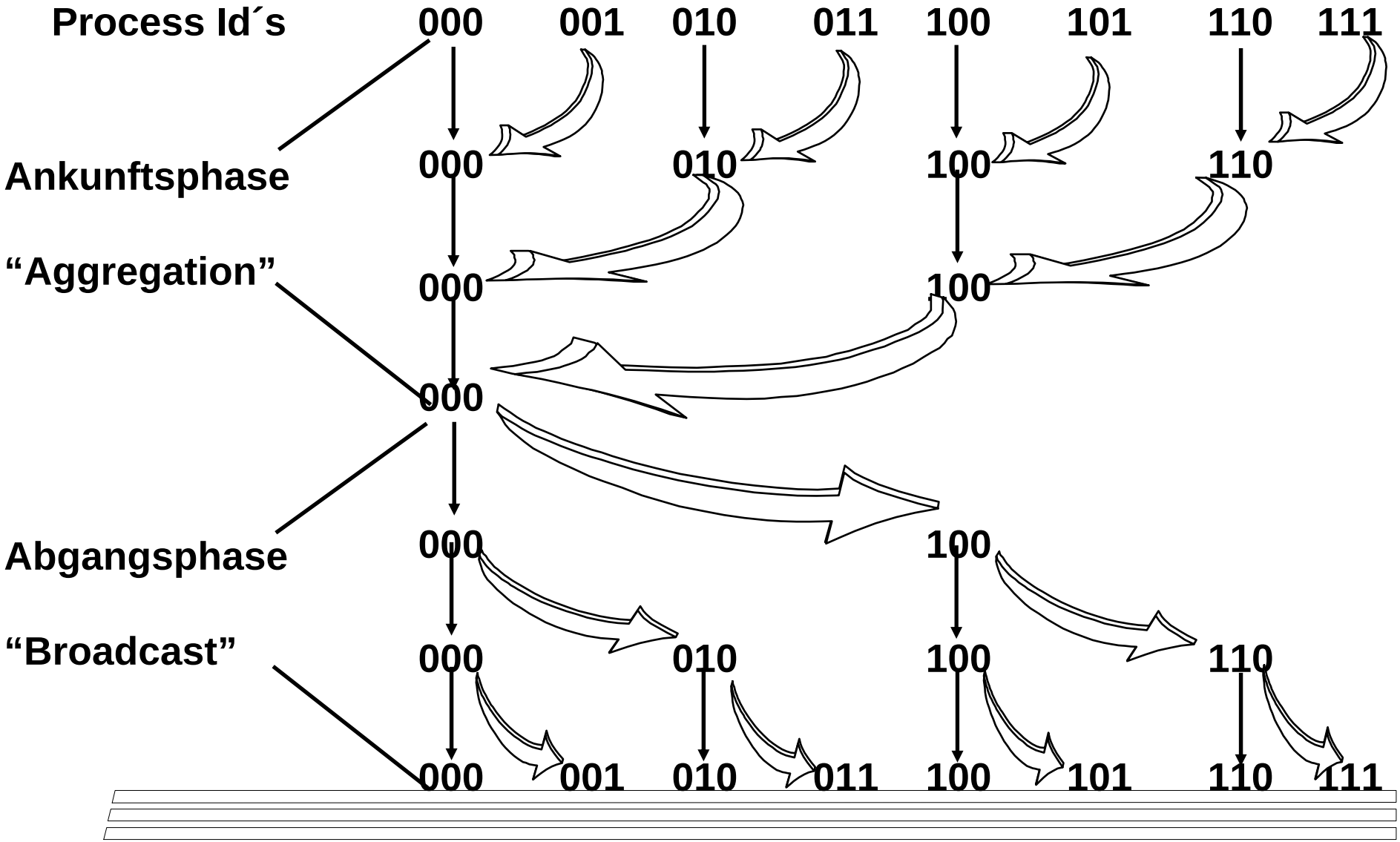
```
global barriere
  sem arrival = 1, departure  = 0;
  int count = 0
  op barrier(val int n)

body barriere
proc barrier (n) {
   # Ankunftsphase
   P (arrival);  count += 1
   if (count < n)  { V(arrival) }
               else { V(departure) }
   # Abgangsphase
   P(departure);  count -= 1
   if (count > 0)  { V(departure) }
               else { V(arrival) }
  }
end barriere
```

Modifiziertes MPD-Programm

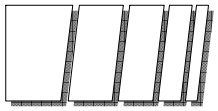
```
resource Jacobi()
  const int n = 5; const int f = 3; int numiter
  real a [0:n+1,0:n+1], b [0:n+1,0:n+1]
  ...
  process row [i=1 to n] {
  # Iteration
  for [k=1 to numiter]  {
      # 1. Phase : Neuberechnung der Werte
      compute_row(i)
      call barriere.barrier(n)  # SYNCHRONISATION
      # 2. Phase : Update
      update_row(i)
      call barriere.barrier(n)  # SYNCHRONISATION
  } }
  ...
end
```

Turnierbarriere mit Aufwand O(log n)



Turnierbarriere in MPD

```
global barriere2
   const int n = 8 # Anzahl Prozesse
   sem b[0:n-1] = ([n] 0) # Feld von Barrierensemaphoren
   op barrier(val int myid, val int n)
body barriere2
proc barrier (myid,n) {
int pos = 1 # Bitwertigkeiten
# Ankunftsphase
while ((myid / pos) mod 2 == 0 & pos < n) {
       P (b[myid])      # Warte auf Partner
       pos *= 2
}
if (myid != 0) {
       V(b[myid-pos])  # Melde Ankunft an Partner
       P(b[myid])
}
# Abgangsphase
while ( pos > 1 ) { pos = pos / 2; V(b[myid+pos]) }
}
end barriere2
```



Symmetrische Barriere

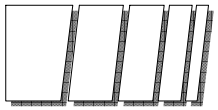
- hier nur für 2 Prozesse mit 2 Semaphoren

  sem b1 = 0; sem b2 = 0

  P1: ..... V(b2); P(b1) .....

  P2: ..... V(b1); P(b2) .....

- allgemein: Butterfly-Schema, Aufwand O(log n)



Vor- und Nachteile von Semaphoren

Vorteile:

- einfach
- effizient zu realisieren

Nachteile:

- niedriges Abstraktionsniveau
- fehleranfällige Programmierung
  - fehlende P-Operation => kein wechselseitiger Ausschluß
  - fehlende V-Operation => mögliche Verklemmung
- unstrukturierte Programmierung
  - P- und V-Operationen über das ganze Programm verstreut, z.B. bei bedingter Synchronisation