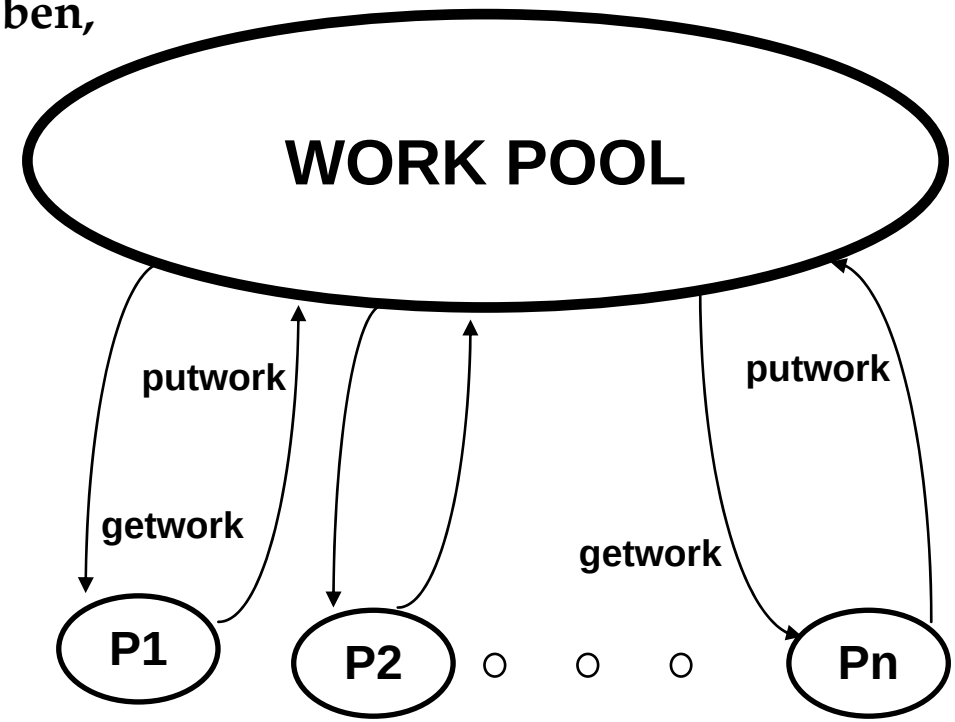
Dynamische Aufgabenverwaltung

Merkmale:

- **zentrale Aufgabensammlung (work pool)**
- **identische Arbeitsprozesse auf verschiedenen Prozessoren**
- **Operation "getwork" zur Anforderung neuer Aufgaben**
- **dynamische Generierung neuer Aufgaben, die mit "putwork" in den Workpool geschrieben werden**
- **Termination, falls alle Prozesse ohne Arbeit sind und der Workpool leer ist**



Probleme:

- **» Organisation des Aufgabenkatalogs**
  - **– Vermeidung von Zugriffskonflikten**
  - **– gleichmäßige Auslastung aller Prozesse**
- **» Terminationserkennung**

Beispiel: Bestimmung kürzester Wege

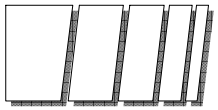
sequentieller Algorithmus:

- **Verwalte in einem Feld mindist den bisher gefundenen kürzesten Abstand vom Quellknoten zu anderen Knoten, Initialisierung: [0, ∞, ∞, ..., ∞]**
- **Verwalte in einer Warteschlange die noch zu untersuchenden Knoten, Initialisierung: [1]**

```
solange die Warteschlange nicht leer ist:
    entnimm Knoten aus Warteschlange, etwa i
    für jede Kante (i,j) im Graphen {
        newdist  = w(i,j) + mindist[i]
        if newdist < mindist[j]  then {
            mindist[j] = newdist
            if j nicht in Warteschlange then  füge j zu Warteschlange hinzu } }
```

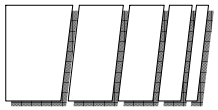
Parallelisierung: Warteschlange => Workpool, Master-Worker-Schema

- **Zentraler Workpool im gemeinsamen Speicher**
- **gemeinsame Daten über Semaphore sichern**



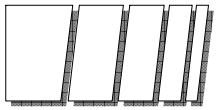
Nebenläufiges MPD Programm

```
resource  ShortPath
import workpool
const int n = ... # number of vertices
const int infinity = ... # maxint
int  weight [n,n]; int mindist [n]; sem save[n] = ([n] 1)
bool  inflag [n]; int start
... # Einlesen der Eingabedaten
for [i=1 to n] { mindist[i] = infinity;  inflag[i] = false }
mindist[start] = 0;  inflag[start] = true;

process worker [me = 1 to p] { ... }
final
... # Ausgabe des Ergebnisses
end ShortPath
```



Arbeitsprozesse

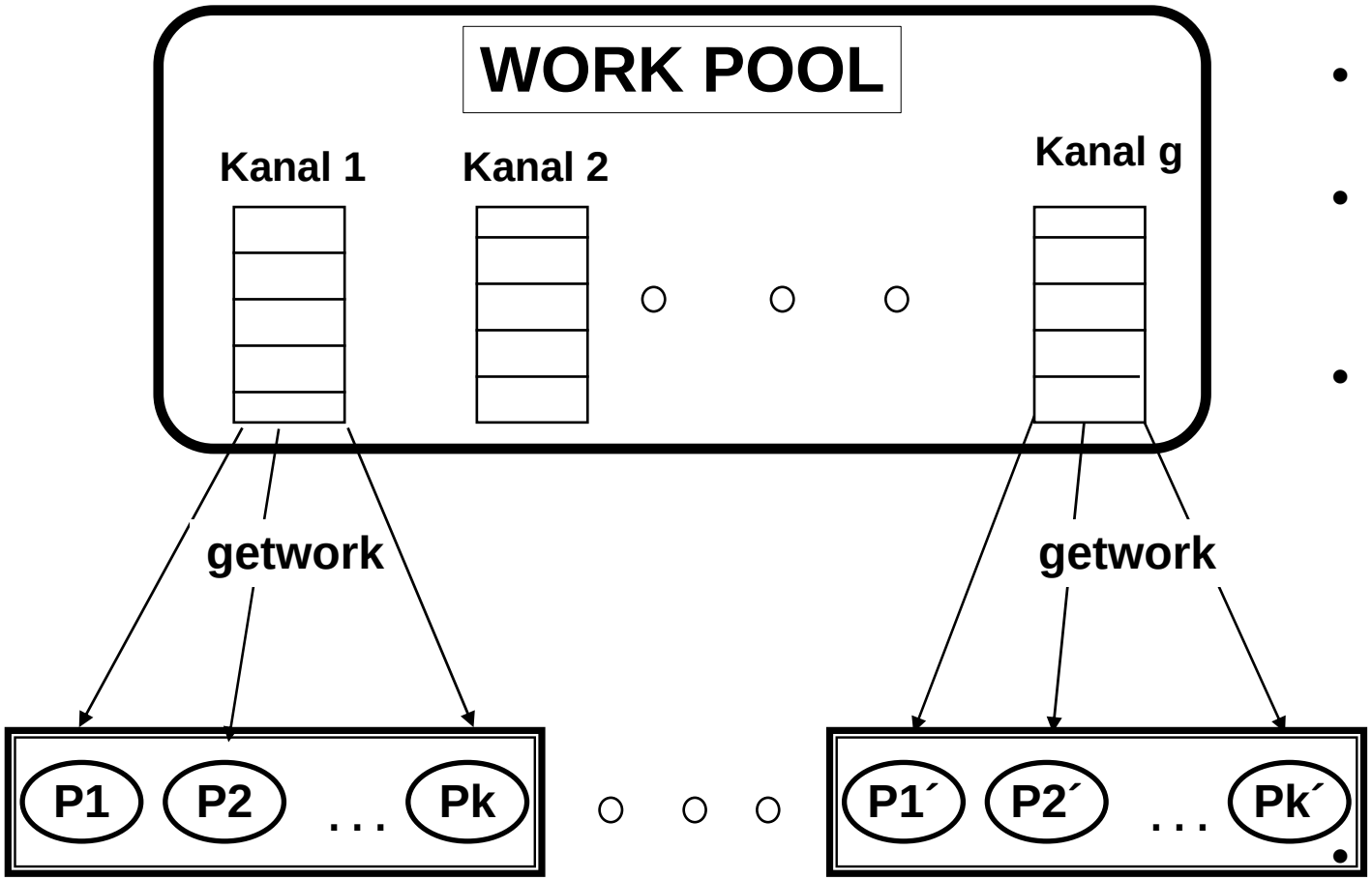
```
process  worker [me = 1 to p] {
work vertex; int  newdist
getwork(me,vertex);
while (vertex != -1) { # solange Termination nicht vorliegt
   inflag[vertex] = false
   for [i=1 to n] {
    if  (weight[vertex,i] < infinity)  # Kante vorhanden?
      { newdist = mindist[vertex] + weight[vertex,i];
        P(save[i]);
        if (newdist < mindist[i])
          { mindist[i] = newdist; V(save[i])
            if  (not inflag[i]) # i noch nicht im Pool?
                {inflag[i]= true; putwork(me,i)} }
        else { V(save[i]) } }
       };    getwork(me,vertex) # neuen Knoten anfordern
   }
}
```



Zentraler Workpool als m:n Kanal

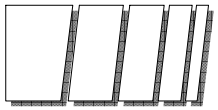
```
global workpool
const  p := ... # number of processes
type work = int;
op work pool {send};  int count = 0; sem  mutex =1;

proc putwork (me:int; item:work) {
P(mutex); count++; V(mutex); send pool(item)
}

proc getwork (me:int; var item:work) {
int workcount  # Warum lokaler Zähler?
P(mutex); workcount = count-1; count = workcount; V(mutex)
if workcount = -p  {  # Termination
   item = -1; for [i=1 to p-1] { send pool(item) }
else  {  receive pool(item) }
}
end
```

Dezentraler Workpool



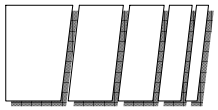
- **mehrere Kanäle für Workpool**
- **Prozessgruppen fordern Arbeit von einem Kanal**
- **Balancierung der Arbeitslast**
  - **Prozesse verteilen neue Arbeit round robin auf alle Kanäle**
  - **einfach und effizient**

zweistufige Terminationserkennung



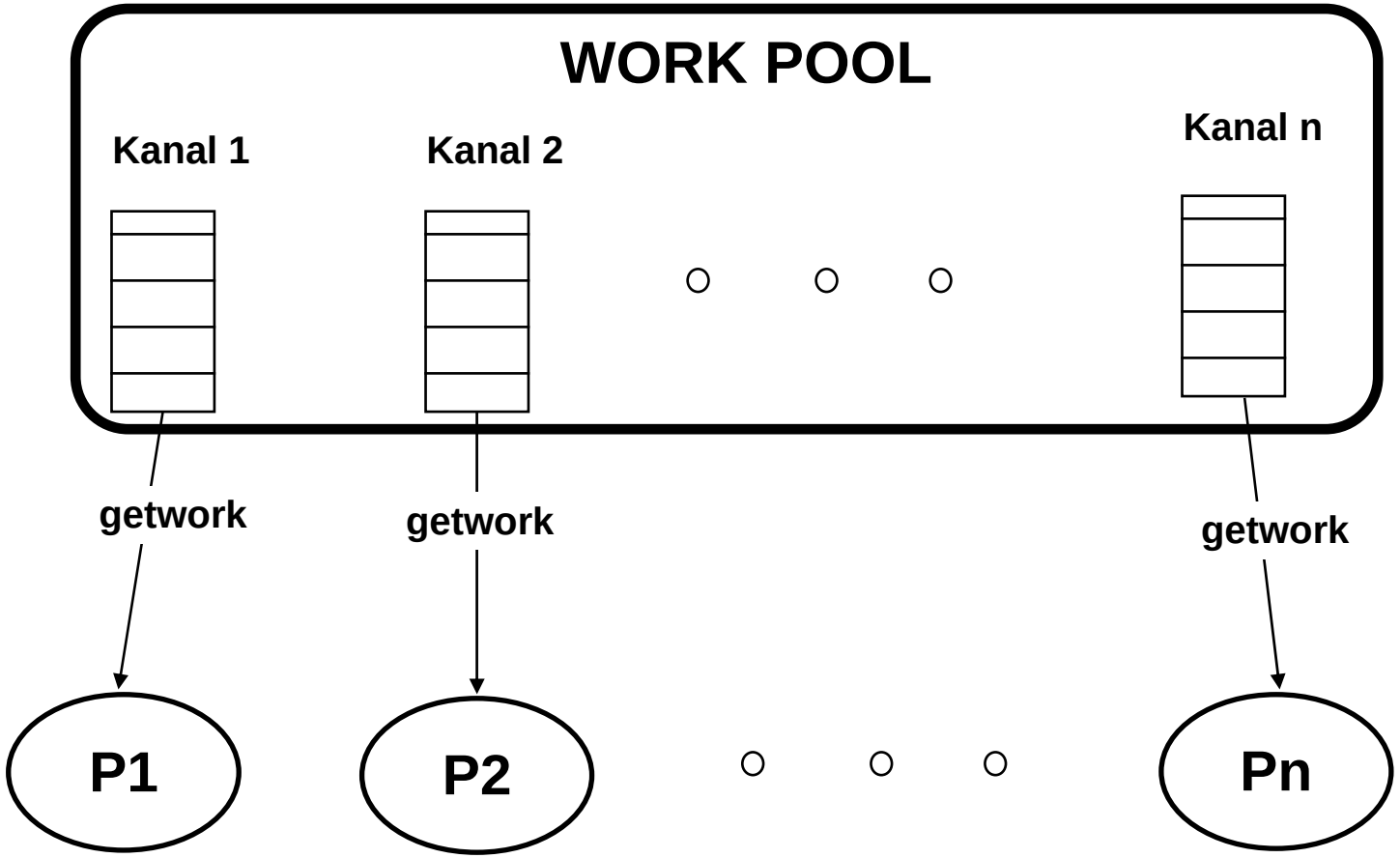
Dezentraler Workpool

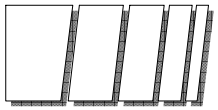
```
global workpool
const int p = ...   # Prozesse
const int g = ...   # Gruppen, g | p
int gsize = p / g
type work = int;
op work pool [g] {send}; int count[g] = ([g] 0); sem mutex[g]
int gcount; sem gmutex; int nextchan [p]; ...

proc putwork (me:int, item:work) {
   int wcount, next
   next = nextchan[me]
   P(mutex[next])
   wcount= count[next]+1; count[next]= wcount;
   V(mutex[next])
   if (wcount == - gsize + 1)
       { P(gmutex); gcount = gcount -1; V(gmutex) }
   send pool[next](item); nextchan[me] = next mod g + 1
}
```



```
proc getwork (int me, work item) {
int workcount, lcount, mychan

mychan = (me-1) / gsize + 1
P(mutex[mychan])
workcount := count[mychan]-1; count[mychan] := workcount
V(mutex[mychan])
if (workcount = - gsize) { # lokale Termination
   P(gmutex); lcount := gcount +1; gcount := lcount; V(gmutex)
   if (lcount = -g) { item = -1;
                      for [i=1 to g] {
                         for [j=1 to gsize] {
                            send pool[i](item) } } }
receive pool[mychan] (item)
}
end   workpool
```

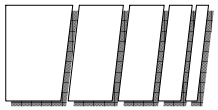
Verteilter Workpool





"Kürzeste Wege" Programm (verteilte Version)

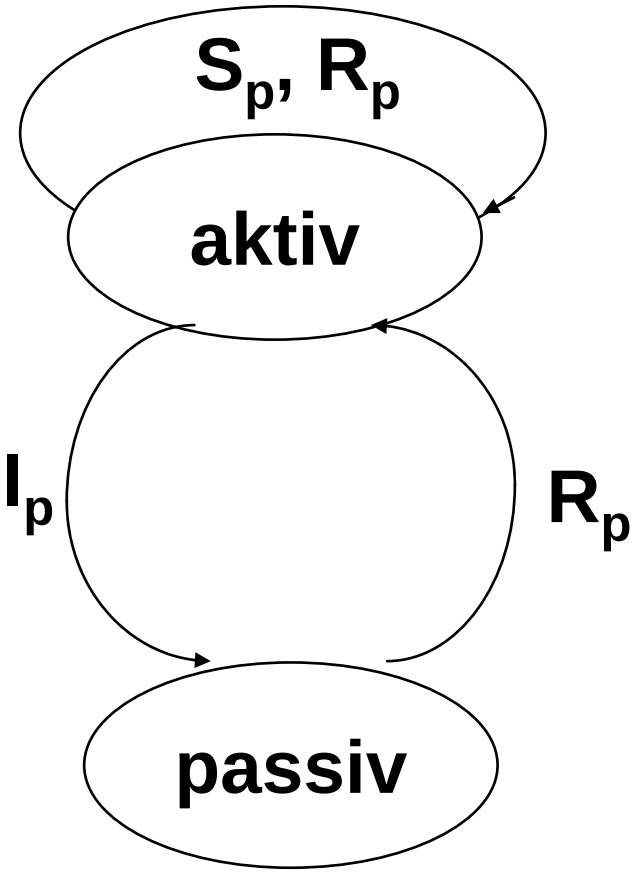
```
resource  ShortPath () # Entwurf für mehrere virtuelle Maschinen
import worker
...
cap vm vmcap[p];
optype workpool(work); op workpool pool [0:p];  op int chan[p]
weightrow weight[n]; int finaldist[n]; int start

procedure createworker(int i, weightrow weighti,
      cap(work) pool[0:p], cap(int) chani, cap vm machine) {
      create worker(i,weighti,pool,chani) on machine       }
...
co [i=1 to p] createworker(i,weight[i],pool,chan[i],vmcap[i]) oc
for [i=1 to p] {    receive chan[i](finaldist[i]);
                    write(finaldist[i],"   ") }
write()

end ShortPath
```



Arbeitsprozesse in verteilter Version (ohne Terminationserkennung)

```
resource worker
const int n = ...; const int p = n  # Anzahl Knoten = Anzahl Prozesse
const int infty = ...;
type weightrow = [n] int; type workpool = [p] cap(int);
int mindist; int newdist
body worker (me: int, myweight: weightrow,
             pool: workpool, answer: cap(int))
op getwork (int distance); op putwork(int vertex,int distance)
proc getwork (distance) { receive pool[me] (distance) }
proc putwork (vertex,distance) { send pool[vertex] (distance) }
mindist = infinity; getwork(newdist)
while (newdist != -1) {
    if (newdist < mindist) {
        mindist = newdist  # kürzere Distanz gefunden
        for [w=1 to n] {
           if (myweight[w] < infty) { putwork(w, mindist+myweight[w])}}
    }; getwork(newdist) }
send answer(mindist)
end worker
```

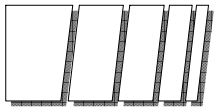
Abstraktes Prozessmodell



- **Axiome:**
  - **A1: Prozesse sind aktiv oder passiv.**
  - **A2: Nur aktive Prozesse versenden Nachrichten.**
  - **A3: Aktive Prozesse können passiv werden.**
  - **A4: Passive Prozesse werden nur durch den Erhalt von Nachrichten aktiv.**
- **Prozessaktionen:**
  - **S_p: {$status_p$=aktiv}**
    send M to Q
  - **R_p: receive M; $status_p$ = aktiv**
  - **I_p: {$status_p$=aktiv}**
    $status_p$:=passiv

- **Termination liegt vor, wenn**
  - **alle Prozesse passiv**
  - **keine Nachricht unterwegs**

  => stabiler Zustand



Terminationserkennung nach Dijkstra/Scholten (1980)

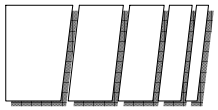
- **Aufbau eines Aktivierungskontrollbaums:**
  - **Aktivität diffundiert entlang Aktivierungsbaum**
  - **Passivität in Gegenrichtung**
- **Jeder Prozess merkt sich, von wem er aktiviert wurde.**
  => Quellenangabe in Nachrichten erforderlich
- **Alle Nachrichten werden quittiert.**
- **Die Aktivierungsnachricht wird erst quittiert, wenn der Prozess ohne Arbeit ist und von allen gesendeten Nachrichten Bestätigungen erhalten hat.**
- **Termination liegt vor, wenn der Wurzelknoten passiv wird.**

Formale Spezifikation

- **zusätzliche Kontrollvariablen pro Prozess:**
  - **father :: processname „Aktivator“**
  - **count :: int „Anzahl gesendeter Nachrichten“**

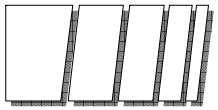
- **Prozess- und Kontrollaktionen**

S_p: {statusp = aktiv}
send <M,p> to q; count = count + 1 *„Zähle Nachricht“*
R_p: receive <M,q>;
if $status_p$ = aktiv then send <ack> to q
else $status_p$ = aktiv, father = q fi
I_p:{$status_p$ = aktiv, count = 0} *„alle gesendeten Nachrichten bestätigt“*
send <ack> to father; $status_p$ = passiv
$Rack_p$: receive <ack>; count = count -1

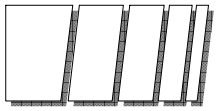


Workpool mit Terminationserkennung nach Dijkstra/Scholten

```
type work = rec( int source, int distance)
body worker(me:int;myweight:weightrow;
                    pool:[0:p]cap(work);answer:cap(int))
int mindist = infty; int newdist;
int ackcount = 0; int parent; bool active = false

op getwork(int distance); op putwork(int vertex, int distance)

proc putwork (vertex,distance) {
work outwork
ackcount = ackcount + 1; outwork = work(me,distance)
send pool[vertex](outwork)
}
```

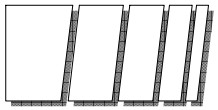


```
proc getwork (distance) {
 int source; work inwork, ackwork
 inwork.source = -2; inwork.distance = 0;
 ackwork.source = -2; ackwork.distance = 0
 while (inwork.source == -2) {
   if active && (ackcount == 0) && ?pool[me] == 0  {
      active = false;  send pool[parent](ackwork)
      write(me," announces termination to ", parent);
      write() }
   receive pool[me](inwork)
   if (inwork.source == -2) { ackcount = ackcount-1 }
 }
 if active { send pool[inwork.source](ackwork) }
 else { active = true; parent = inwork.source }
 distance = inwork.distance
}
```



Anweisungsteil Worker

```
getwork(newdist);
write(me," got distance  ", newdist)
while (newdist != -1) {
  if (newdist < mindist) {
     mindist = newdist;
     for [w=1 to n] {
       if (w != me) {
        if myweight[w] < infinity {
           putwork(w,mindist+myweight[w])
           write(me," puts  work ", w,„ ",mindist+myweight[w])
    } } } }
  getwork(newdist); write(me, "got distance", newdist)
}
send answer(mindist)
end worker
```



Wächterprozess

```
process monitor
work item
receive pool[0](item)      # Warte auf Termination
item.source = -1; item.distance = -1
                      # erzeuge  Terminationsnachricht
for [i=1 to p] { send pool[i] (item) }
                      # und sende diese an alle Prozesse
}
```

Kreditmethode nach Mattern (1989)

- **Ziel:**
  - **geringeres Nachrichtenaufkommen**
  - **keine Bestätigung von Nachrichten**
- **Idee:**
  - **zentraler Kontrollprozess, der über die im System vorhandene Aktivität buchführt und schließlich die Termination signalisiert**
  - **Kreditanteile für Prozesse und Nachrichten**
  - **Kontrollprozess verwaltet Gesamtkredit (=Summe aller Kreditanteile), der nur dekrementiert wird**

Formale Spezifikation

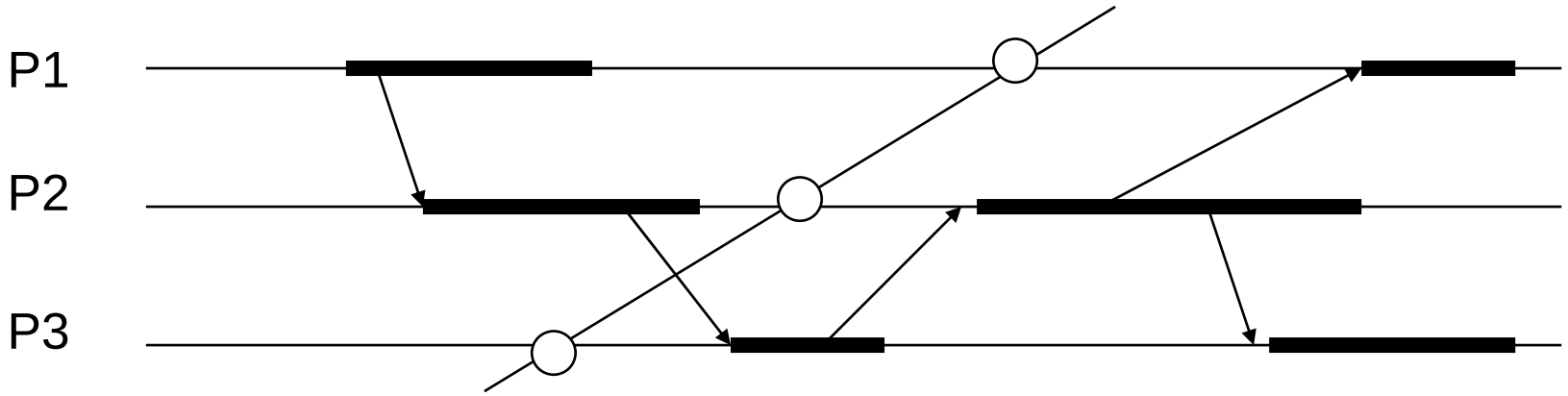
- **Kontrollvariable in jedem Prozess:** K_p: real **# Kreditanteil**
- **Prozess- und Kontrollaktionen**

S_p: {$status_p$ = aktiv}
send <M, K_p/2> to q; K_p := K_p/2 # *Kreditsumme bleibt gleich*
R_p: receive <M, K>;
$status_p$:=aktiv; K_p := K_p + K # *Absorption des N.-kredites*
I_p: {$status_p$=aktiv}
$status_p$:=passiv; send <K_p> to P_0; K_p := 0
$Rcredit_{P0}$: receive <K>; K_0 := K_0 - K;
if K_0 = 0 then "announce termination"

- **Invarianten:**
  - **$status_p$ = aktiv ⇔ K_p > 0**
  - **Jede Nachricht M hat positiven Kredit K_M >0**
  - **K_0 = $\Sigma_P K_P + \Sigma_M K_M$**

Unabhängige Kontrollmethoden

Visualisierung durch Zeitdiagramme:



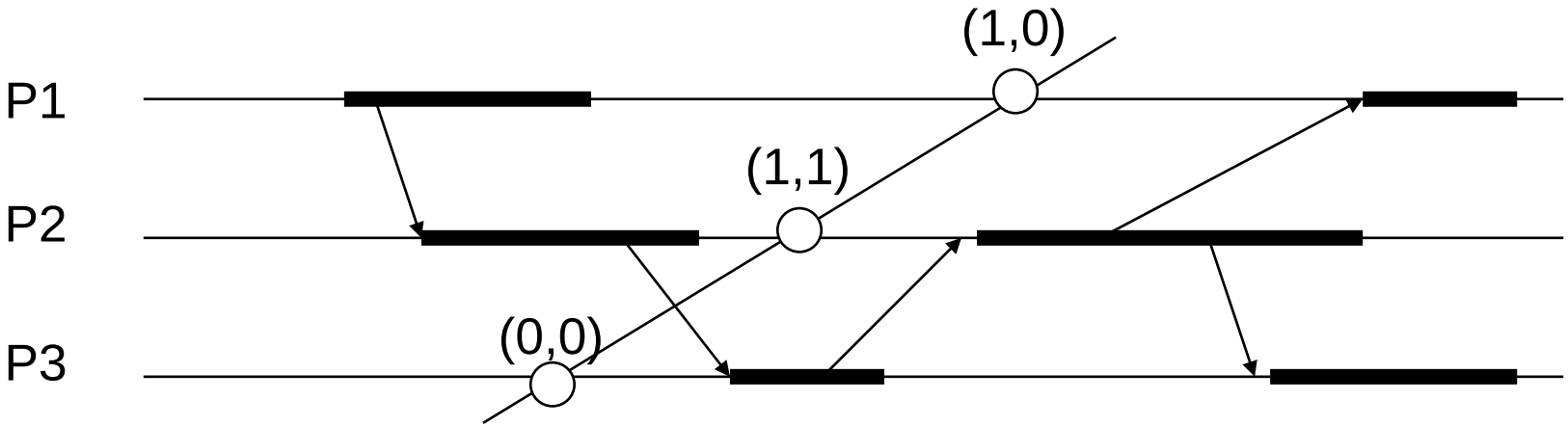
Kontrollwelle: alle Prozesse passiv

=> Prozesse müssen über die Zahl der empfangenen und gesendeten Nachrichten buchführen.

Kontrollwellenverfahren

Visualisierung durch Zeitdiagramme:

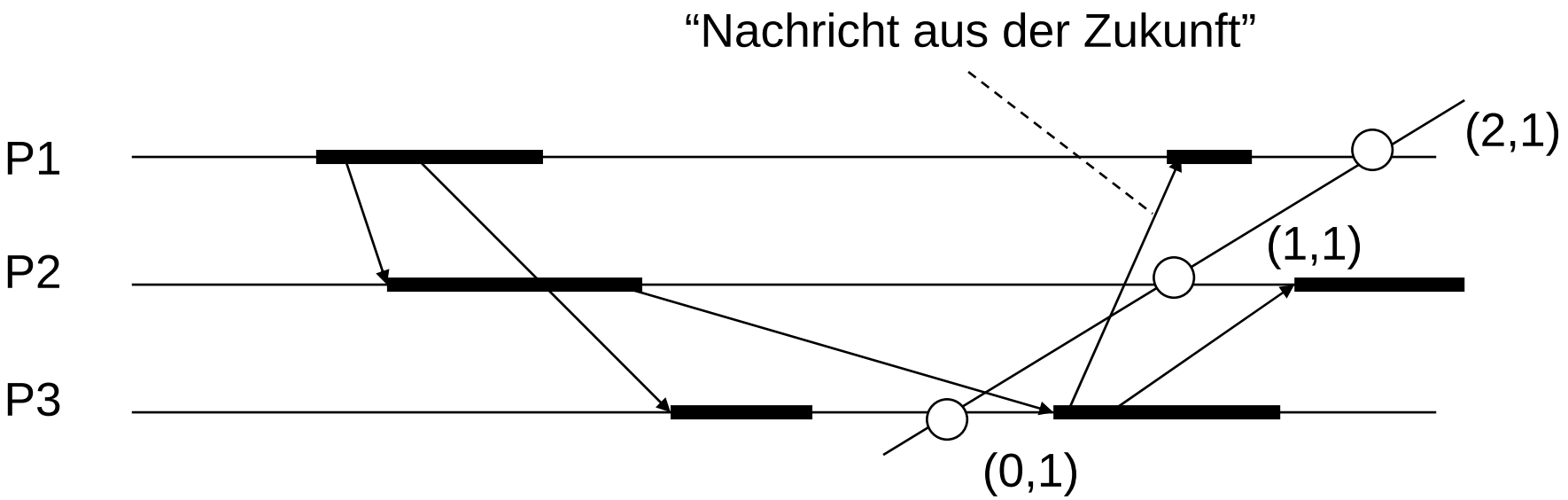
Bilanz: 1 Nachricht noch unterwegs



Kontrollwelle: alle Prozesse passiv

=> **Prozesse müssen über die Zahl der empfangenen und gesendeten Nachrichten buchführen.**

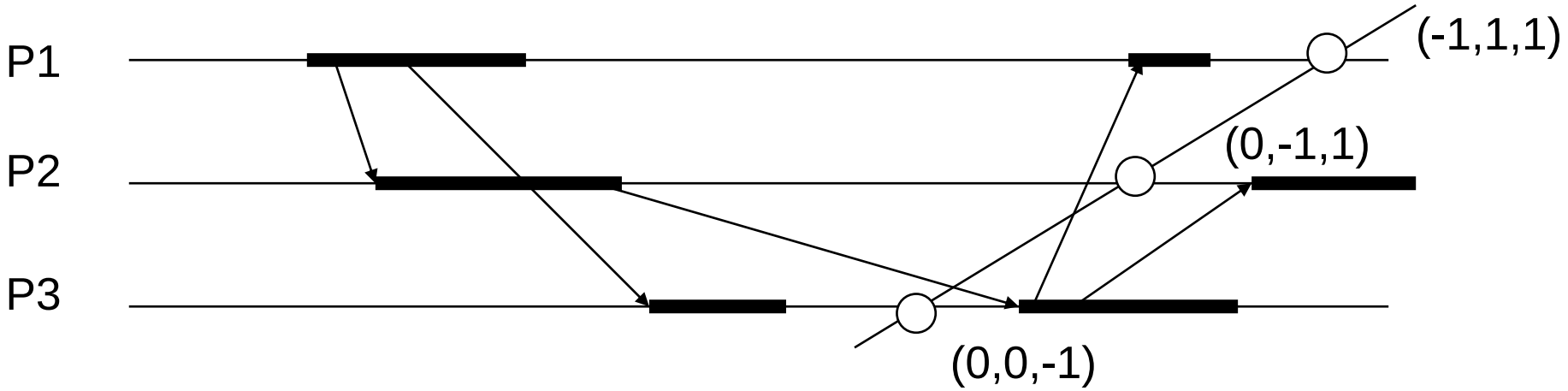
Gegenbeispiel



Kontrollwelle: alle Prozesse passiv und Bilanz ausgeglichen

Vektorzählmethode

- **Jeder Prozess verwaltet Vektor von Zahlen:**
  - **zähle für jeden anderen Prozess separat, wieviele Nachrichten an ihn gesendet wurden (+)**
  - **zähle in eigener Vektorkomponente die empfangenen Nachrichten (-)**
- **Die Kontrollwelle addiert die Vektoren.**



Kontrollwellensumme: (-1,0,1)

Spezifikation der Vektorzählmethode

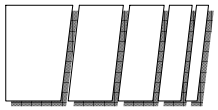
- **Variablen in Prozess P_i ($1 \leq i \leq p$):**
  - **$C_i[p]$: integer := [p] 0 # Zähler**
  - **$flag_i$: boolean := i==1 # Kontrollflag**
  - **$status_i$: {aktiv, passiv} := if i==1 then aktiv else passiv**

- **Prozessaktionen & Kontrollaktionen:**

S_i : {$status_i$ = aktiv} send M to P_j ; $C_i[j] := C_i[j] + 1$

R_i : receive M; $status_i$:=aktiv; $C_i[i] := C_i[i] - 1$

I_p: {$status_i$ =aktiv} $status_i$:=passiv;
if $flag_i$ then send <control, ($C_i[1]$, ..., $C_i[p]$)>
fa j:=1 to n -> $C_i[j] := 0$ af; $flag_i$:= false



Rcontrol$_i$: receive <control, (C1, ..., Cp)>

fa j:= 1 to p -> $C_i[j] := Cj + C_i[j]$ af

if status$_i$=passiv

then if " j: $C_i[j] = 0$ then "announce termination"

else send <control, ($C_i[1]$, ..., $C_i[p]$)> to $P_{(i \bmod p) + 1}$

fa j:=1 to n -> $C_i[j] := 0$ af

else $flag_i$:= true

Doppelzählverfahren

- **einfache Zähler für gesendete und empfangene Nachrichten**
- **starte zweite Kontrollwelle nach Kontrollwelle mit ausgeglichener Bilanz**
- **Termination liegt vor, wenn das Ergebnis beider Kontrollwellen gleich ist**