Verteilte Systeme

7. Verteilter wechselseitiger Ausschluß

Inhalt

I. Was ist wechselseitiger Ausschluß
II. Wechselseitiger Ausschluß mit State-Variablen
III. Wechselseitiger Ausschluß mit Nachrichten

Wechselseitiger Ausschluß (mutual exclusion)

„Mehrere Prozesse möchten etwas tun, wobei garantiert sein muß, daß es zu einem Zeitpunkt nur einer darf."

- exklusive Betriebsmittel
- „abstrakte" exklusive Betriebsmittel, z.B. Termin in einem verteilten TVS

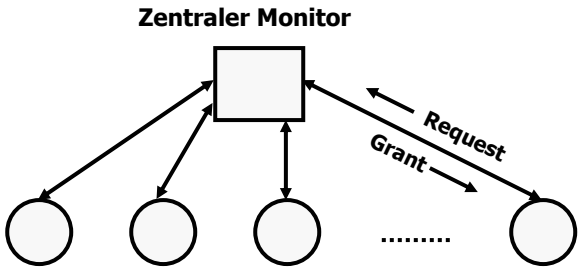
Im „shared memory"-Fall:

Mit Hilfe von z.B. Semaphoren einfach lösbar!

$\Longrightarrow$ Interessiert uns hier nicht!

Wechselseitiger Ausschluß (mutual exclusion)

Was uns auch nicht interessiert:



➔ 2 Nachrichten

- **Fehlertoleranz:**

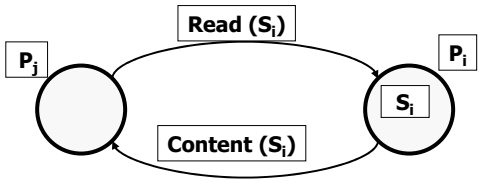
II. STATE-VARIABLEN

Jeder Prozeß P_i hat zwei Arten von Variablen:

Lokale Variablen: Nur P_i kann auf sie zugreifen (lesend und schreibend)

State-Variablen:
P_i darf lesend und schreibend zugreifen.
$\forall j : j \neq i : P_j$ darf lesend zugreifen.

➔ Prinzipiell auch in Nachrichten-basierten Systemen möglich:



„**Bäckerei-Algorithmus**" (Lamport'74)

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

number [i] ← 1 + max (number[0], ..., number [n-1]);

for j=0 to n-1, i ≠ j
begin
wait (number [j] ≠ 0) ⇒
(number [i], i) < (number [j], j);
end;
<kritischer Abschnitt>

number [i] ← 0;

(a,b) < (c,d) ⇔ (a < c) ∨ ((a = c) ∧ (b < d))

„**Bäckerei-Algorithmus**" (Lamport'74)

1) Algorithmus garantiert wechselseitigen Ausschluß:

   number [i] = n_i
   • $(n_1, ..., n_n)$ besitzt bezgl. der Ordnung ein eindeutiges Minimum!
   • Sei n_i dieses Minimum
   $\Rightarrow$ Für P_i : $\forall_j : j \neq i$: wait (true)
   $\Rightarrow$ Für P_j : $j \neq i : \exists$ wait (false);

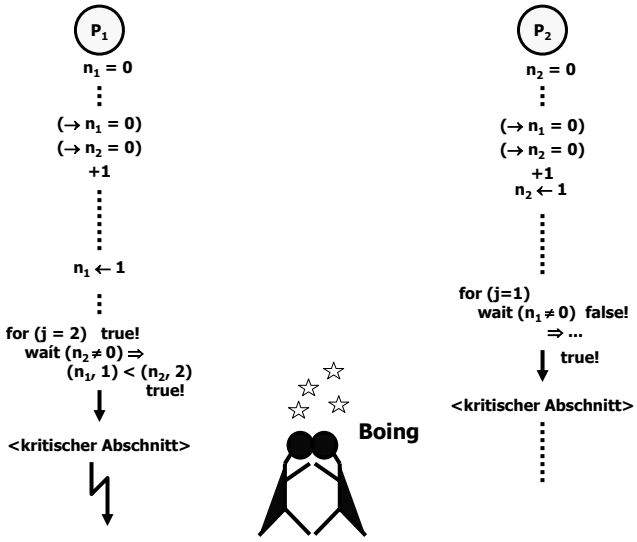
2) Algorithmus ist fair!
   Prozeß P_i muß maximal (n-1) mal warten

3) Algorithmus ist korrekt?

   Fast!
   Problem: number [i] $\leftarrow$ 1 + max (number [0], ..., number [n-1])

   ist **nicht** atomar!

Szenario: Ein wechselseitiger Nicht-Ausschluß



„**Bäckerei-Algorithmus**" (Lamport'74)

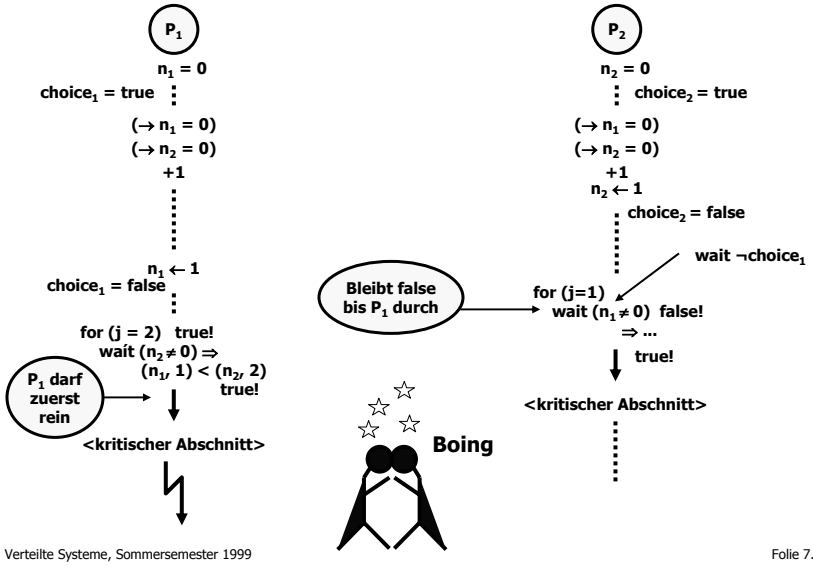

```
var number : array [0 ... n-1] of integer;
    choice : array [0 ... n-1] of boolean;

choice [i]  ← true;
number [i] ← 1 + max (number[0], ..., number [n-1]);
choice [i]  ← false;

for  j=0  to  n-1, i ≠ j
     begin
          wait ¬choice [j];
          wait (number [j] ≠ 0) ⇒ (number [i], i) < (number [j], j);
     end;
          <kritischer Abschnitt>

number [i] ← 0;
```


Szenario: Ein wechselseitiger Nicht-Ausschluß

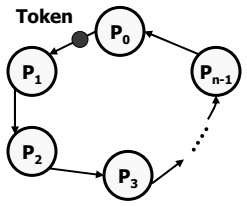


Bemerkungen:

1) Algorithmus ist symmetrisch
2) Fehlertoleranz:
   Es muß „bloß" garantiert sein, daß bei Absturz von P_i jeder andere Prozeß P_j davon ausgeht: P_i ist nicht im kritischen Abschnitt!
   Deadlock-Gefahr:
   Prozeß P_i stürzt immer bei der Maximumbildung ab!
   $\Rightarrow$ choice; = true
   $\Rightarrow$ Alle Prozesse warten an wait ¬choice;
3) Nachrichtenaufwand: Hängt davon ab, wie man die State-Variablen realisiert ($\sim 2n + 4 \cdot (n - 1)$)

Verteilte Systeme, Sommersemester 1999 Folie 7.11

Dijkstra's selbst-stabilisierender Algorithmus (1974)



Sei k>1:

var flag: array [0 ... n-1] of 0...k;

P_0: wait flag [0] = flag [n-1];
<kritischer Abschnitt>
flag [0] $\leftarrow$ (flag [0] + 1) mod k;

P_1(i $\neq$ 0):
wait flag [i] $\neq$ flag [i-1];
<kritischer Abschnitt>
flag [i] $\leftarrow$ flag [i-1];

Verteilte Systeme, Sommersemester 1999 Folie 7.12

Dijkstra's selbst-stabilisierender Algorithmus (1974)

Sei flag[i] = 0 $\forall i$

Bemerkungen:

1) Algorithmus garantiert wechselseitigen Ausschluß:
   true!
   P_0 : wait flag [0] = flag [n-1]
   P_1 : wait flag [i] $\neq$ flag [i-1]
   false! $\forall i : i \neq 0$
   $\Rightarrow P_0$ betritt kritischen Abschnitt
   ... danach flag [0] = 1
   false!
   P_0 : wait flag [0] = flag [n-1]
   P_1 : wait flag [1] $\neq$ flag [0] true!
   P_i : wait flag [i] $\neq$ flag [i-1]
   false! $\forall i : i \neq 0, i \neq 1$
   $\Rightarrow P_1$ betritt kritischen Abschnitt
2) Algorithmus verhindert Deadlocks ✓
3) Algorithmus ist fair

Verteilte Systeme, Sommersemester 1999 Folie 7.13

Dijkstra's selbst-stabilisierender Algorithmus (1974)

<u>Aber:</u>

1) Algorithmus ist nicht symmetrisch
2) Prozesse bekommen das Privileg, obwohl sie gar nicht in den kritischen Abschnitt wollen!
   $\Rightarrow$ müssen „Token" weitergeben
3) Fehlertoleranz:
   Abstürzende Prozesse müssen aus dem Ring entfernt werden!
   Wie man entfernt, hängt vom Prozeß P ab
   (P_0 $\not\rightarrow$ Anderer wird P_0)
   (P_i $\not\rightarrow$ Raus aus Ring)

   $\Rightarrow$ Zusätzliche Informationen für den Ringaustrag

Verteilte Systeme, Sommersemester 1999 Folie 7.14

Dijkstra's selbst-stabilisierender Algorithmus (1974)

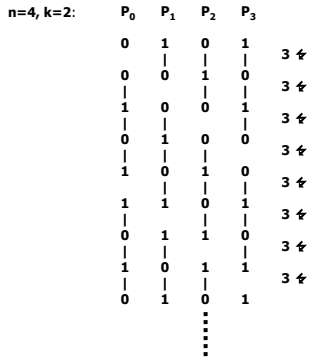
Interessante Eigenschaft:

Man kann die initiale Bedingung flag [i] = 0 $\forall i$ fallen lassen

$\Rightarrow$ Selbst – Stabilisierung

Hängt von k ab:

1) $k \leq n$: Wechselseitiger Ausschluß kann nicht garantiert werden!



Dijkstra's selbst-stabilisierender Algorithmus (1974)

k > n : Selbst-Stabilisierung

P_o inkrementiert (wenn er darf)
P_i schieben Token nach rechts

Da $k > n \Rightarrow$
informell: „Müll" wird nach rechts geschoben
und durch sinnvolle Inhalte in P_o ergänzt

!Man kann erkennen, wann wechselseitiger Ausschluß garantiert ist!

Abschließende Bemerkungen

- Einfache, verhältnismäßig fehlertolerante Algorithmen sind möglich!

Nachteile:

- 2 Ebenen werden durcheinander gebracht:
  - Logik
  - Implementierung

  $\Rightarrow$ schwer beschreibbar & schwer verständlich
- Hohe Anzahl an Nachrichten
- „Blindes" Testen von Statevariablen
  $\Rightarrow$ Nachrichten an andere, wenn sich Statevariablen in P_i ändern:
  $\Rightarrow$ Nachrichten-basierte Algorithmen

➔ Viele unnütze Nachrichten

Verteilte Systeme, Sommersemester 1999 Folie 7.17

Abschließende Bemerkungen

Aber:

Effiziente Realisierung in „shared-memory"-System

Vorteil: (gegenüber z.B. Semaphor)

- keine Globalen Variablen
- Erhöhte Fehlertoleranz

Verteilte Systeme, Sommersemester 1999 Folie 7.18

III. Wechselseitiger Ausschluß
- Nachrichten-basiert -

„STATE-Variablen"-Algorithmen können in einer nachrichten-basierten Umgebung realisiert werden!
⇒ „Request Informationen"-Nachrichten
⇒ Viele z.T. unnötige Nachrichten

➔ „Send Informationen"-Nachrichten

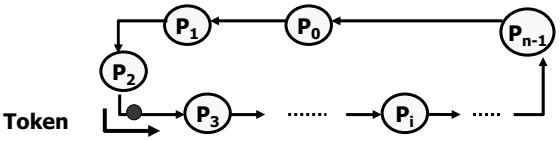
- Anzahl der Nachrichten wird minimiert
- Keine unnützen Nachrichten
- Alle Nachrichten signalisieren Zustandswechsel

Nachrichten-Basierte Algorithmen sind verteilter als die mit State-Variablen!

III. Wechselseitiger Ausschluß
- Nachrichten-basiert -

... natürlich gibt es auch hier eine Token-Ring Lösung:

a) Le Lann's Algorithmus (1977)



P_i : Warte auf Token von P_{i-1} ;
<Kritischer Abschnitt>
Sende Token an P_{i+1} ;

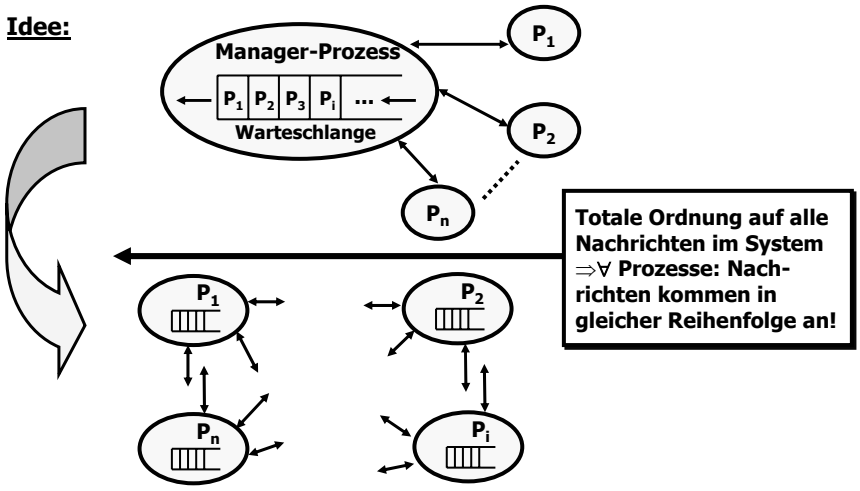
Klar:
- Garantiert wechselseitigen Ausschluß
- Verhindert Deadlocks
- Fair

Nachteile:
- Anzahl Nachrichten: 1 ... + ∞
- Prozeß bekommt Privileg, obwohl er es nicht braucht!
- Fehlertoleranz
  1) Token-Verlust ⇒ Genau 1 neues Token!
  2) Prozeß-Absturz ⇒ Ring verkürzen

III. Wechselseitiger Ausschluß
- Nachrichten-basiert -

b) Verteilte Warteschlange (Lamport 1978)

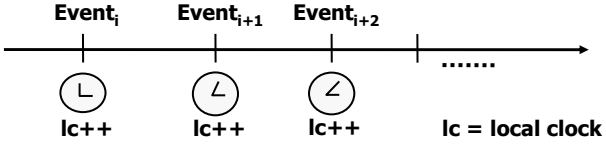
Idee:



Verteilte Systeme, Sommersemester 1999 Folie 7.21

Timestamp - Mechanismus

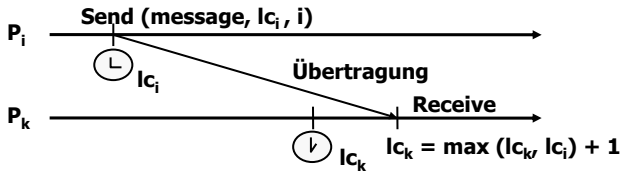
- Jeder Prozeß (Knoten) besitzt eine lokale Uhr, die unabhängig von anderen Uhren ist!
  ⇒ Anpassung der Uhren!



Verteilte Systeme, Sommersemester 1999 Folie 7.22

Timestamp - Mechanismus

- Uhren-Synchronisation beim Nachrichtenverschicken:



d.h. bei Empfang von (message, lc_m , m) bei P_k:

if $lc_k < lc_m$ then $lc_k \leftarrow lc_m$ endif;
$lc_k \leftarrow lc_k + 1$;

Timestamp – Mechanismus

- Erweiterung: Für zwei empfangene Nachrichten
  ($Daten_1$, k_1, i) und
  ($Daten_2$, k_2, j)
  → **Prozeß (Knoten)-Nummer**

  kann gelten: $lc_1 = lc_2$!

  Dann gilt:

  $(Daten_1, lc_1, i) < (Daten_2, lc_2, j)$

  $\Leftrightarrow$

  $(lc_1 < lc_2) \vee ((lc_1 = lc_2) \wedge (i < j))$

- Das ist nur eine mögliche Art, Nachrichten mit Zeitstempeln zu versehen!

  Es gibt auch andere!

Lamport's Algorithmus

n Prozesse: P_0 , ... , P_{n-1}

Lokale Variablen für P_i

var: $clock_i$: 0 ... + ∞

$queue_i$:

0	1		j		n-1
REL	REL		REL		REL
0	0	...	0	...	0
0	1		j		n-1

Letzte Nachr.
Clock von P_j
j

Ist Req, Ack, oder Rel.

Lamport's Algorithmus

P_i möchte in den kritischen Abschnitt:

1. Sende (req, $clock_i$, i) an alle; /*Broadcast*/
   $queue_i$ [i] := $clock_i$ + 1;
2. Warte bis ∀j : j ≠ i gilt:
   Die Nachricht in $queue_i$ [i] ist kleiner
   als die Nachricht in $queue_i$ [j]
3. <kritischer Abschnitt>
4. Sende (rel, $clock_i$, i) an alle;
   $queue_i$ [i] := (rel, $clock_i$, i);
   $clock_i$, := $clock_i$ + 1;

! Voraussetzung: Nachrichten überholen sich nicht!

Lamport's Algorithmus

P_i empfängt Nachricht (Daten, clock, Nr):

1) /* Uhr synchronisieren */
   wenn $clock_i < clock$ dann
   $clock_i := clock$;
   $clock_i := clock_i + 1$;

2) Falls
   Daten = req: $queue_i [Nr] := (req, clock, Nr)$;
   Sende $(ack, clock_i, i)$ zu P_{Nr};
   Daten = rel: $queue_i [Nr] := (rel, clock, Nr)$;
   Daten = ack: wenn Nachricht in $queue_i [Nr] \neq req$
   dann
   $queue_i [Nr] := (ack, clock, Nr)$;

Algorithmus ist
- Fair
- Deadlock frei (Totale Ordnung enthält keine Zyklen)
- Korrekt (Garantiert wechselseitigen Ausschluß)

Nachrichten-Komplexität:

$3 \cdot (n-1)$

Ein optimaler Algorithmus

c) Ricart & Agrawala's Algorithmus (1981)
   - Optimierung von Lamport's Algorithmus

Prozess P_i möchte in den kritischen Abschnitt:
   1) Sende (Request, $clock_i$, i) an alle;
   2) Warte auf n-1 Reply's;
   3) <kritischer Abschnitt>

Prozess P_i empfängt Request:
- Sendet Reply sofort, wenn
  - I) P_i nicht in den kritischen Abschnitt will!
  - II) P_i zwar in den kritischen Abschnitt will, aber der andere höhere Rechte hat!
- Verzögert das Reply, wenn
  - III) P_i in den kritischen Abschnitt will und höhere Rechte hat!

Ein optimaler Algorithmus

c) Ricart & Agrawala's Algorithmus (1981)
- Optimierung von Lamport's Algorithmus

Prozess P_i möchte in den kritischen Abschnitt:
1) Sende (Request, $clock_i$, i) an alle;
2) Warte auf n-1 Reply's;
3) <kritischer Abschnitt>
4) Gib alle verzögerten Reply's raus!

Prozess P_i empfängt Request:
- Sendet Reply sofort, wenn
  - I) P_i nicht in den kritischen Abschnitt will!
  - II) P_i zwar in den kritischen Abschnitt will, aber der andere höhere Rechte hat!
- Verzögert das Reply, wenn
  - III) P_i in den kritischen Abschnitt will und höhere Rechte hat!

Nachrichten-Komplexität

$2 \cdot (n-1)$

Ein optimaler Algorithmus

In dieser Richtung sind dann einige Verbesserungen vorgenommen worden:

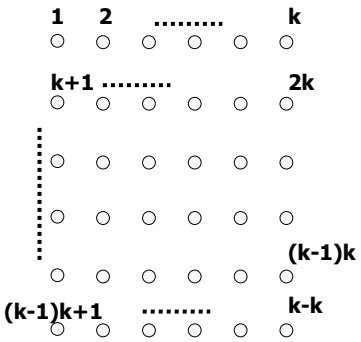
Le Lann (1977)	1 ... + ∞
Lamport (1978)	3 (n-1)
Ricart & Agrawala (1981)	2 (n-1)
Carvalho & roucairol	0 ... (2(n-1))
Suzuki & Kasami (1982)	
✓ Ricart & Agrawala (1983)	n

0 oder 42?

Ein optimaler Algorithmus

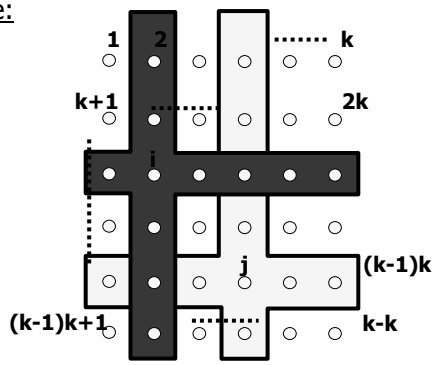
d) Maekawa's Algorithmus (1985)

Idee:



Ein optimaler Algorithmus

d) Maekawa's Algorithmus (1985)
Idee:



⇒ **Prozess i hat eine Menge von Prozessen S_i, die er um Erlaubnis fragen muß!**
Es gilt: $\forall i, j \quad S_i \cap S_j \neq \emptyset$
(Wichtig, um wechselseitigen Ausschluß zu garantieren)

Ein optimaler Algorithmus

Nachrichten:

- REQUEST
- LOCKED
- RELEASE

} **für wechselseitigen Ausschluß**

- Inquire
- Relinquish
- Failed

} **Vermeidung von Deadlocks**

Algorithmus

P_i möchte kritischen Abschnitt betreten:
1) Sende REQUEST an alle Elemente von S_i;
2) Warte auf LOCKED von allen Prozessen aus S_i;
3) <kritischer Abschnitt>
4) Sende RELEASE an alle Prozesse aus S_i;

$P_j \varepsilon S_i$ empfängt REQUEST-Nachricht:
1) Wenn frei dann
   Zustand ← locked;
   Sende LOCKED-Nachricht an P_i;
   Sonst /* Abschnitt ist von $P_k \varepsilon S_j$ belegt */
   wenn P_j von einem aussichtsreicheren REQUEST-Kandidaten weiß
   dann
   Sende FAILED-Nachricht an P_i;
   sonst
   Sende INQUIRE-Nachricht an P_k;
   /* Warte auf RELINQUISH oder RELEASE von P_k */

Algorithmus

$P_k \varepsilon S_i$ empfängt INQUIRE-Nachricht von P_j:
1) wenn ein Prozess εS_k ein FAILED an P_k geschickt hat
dann
Hebe bereits angelaufene LOCK's auf;
Sende RELINQUISH an P_j;
sonst
Sende RELEASE nach Verlassen des kritischen Abschnitts;

... usw. usw.

Algorithmus
- garantiert wechselseitigen Ausschluß
- verhindert Deadlocks
- ist fair!

Algorithmus

Nachrichten-Komplexität:

$$\mathbf{c \cdot \sqrt{n} \quad c \varepsilon [6 \ldots 10]}$$

Algorithmus kann verbessert werden:

Die „REQUEST – GRANTING"-Mengen S_i sind im Gitter nicht minimal!

Lösung:
- Problem aus der projektiven Geometrie
- Für bestimmte n sind minimale S_i möglich ($|S_i|$ = Potenz einer Primzahl)

Im minimalen Fall ist die Nachrichten-Komplexität

$$\mathbf{c \cdot \sqrt{n} \quad c \varepsilon [3 \ldots 5]}$$

Algorithmus

Fehlertoleranz:

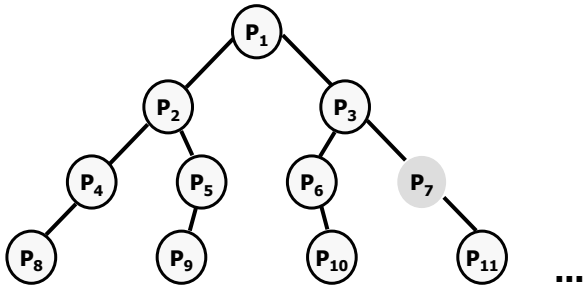
- Bei Ausfall von P_i muß die Request-Grant-Menge S_i von einem anderen Rechner übernommen werden!

  Interessant: Das wissen von P_i ist in $\sigma(\sqrt{n})$-Nachrichten wieder herleitbar!

Ein optimaler Algorithmus

c) Raymond's Algorithmus (1989)

Idee: Topologie ist ein spannender Baum

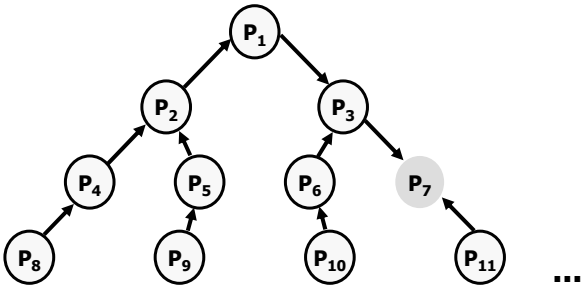


- Ein Prozess P_i besitzt das Privileg („Token"), den kritischen Abschnitt zu betreten P_i

Ein optimaler Algorithmus

c) Raymond's Algorithmus (1989)

Idee: Topologie ist ein spannender Baum



- Ein Prozess P_i besitzt das Privileg („Token"), den kritischen Abschnitt zu betreten P_i
- Die anderen Prozesse wissen in welcher Richtung es zu finden ist! ➔

Verteilte Systeme, Sommersemester 1999 Folie 7.39

Nachrichten-Arten:

- REQUEST : Ein Prozess will das Token! (Privileg)
- PRIVILEG : Hier ist das Privileg!

Lokale Variablen für P_i:

HOLDER = ‚self' oder Name eines direkten Nachbarn in dessen Richtung das Privileg liegt.
USING : bool = true wenn P_i im kritischen Abschnitt ist (USING ⇒ Holder – self)
REQUEST-QUEUE = FIFO–Schlange angekommener Request-Nachrichten von Prozessen, die das Privileg noch nicht hatten
ASKED : bool = true ⇔ Request nach Privileg in Richtung HOLDER ist bereits verschickt worden.

Enthält nur ‚self' und Namen direkter Nachbarn!

Verteilte Systeme, Sommersemester 1999 Folie 7.40

Algorithmus

Prozess P_i will Privileg:
1) Füge ‚self' an REQUEST-QUEUE an;
2) wenn ¬ ASKED dann
   sende REQUEST an HOLDER;
   ASKED ← true;

Prozess P_i emfängt Request von P_j:
1) Füge „P_j" an REQUEST-QUEUE an;
2) wenn ¬ ASKED dann
   sende REQUEST an HOLDER;
   ASKED ← true;

Prozess P_k hat Privileg und will es loswerden:
1) wenn |REQUEST-QUEUE| > 0 dann
   X ← Kopf von REQUEST-QUEUE;
   Lösche Kopf der REQUEST-QUEUE;
   Holder ← X;
   sende PRIVILEGE an HOLDER;
   asked ← false;
   using ← false;

Algorithmus

Prozess l empfängt Privileg von P_m:
1) X ← Kopf von REQUEST-QUEUE;
2) Lösche Kopf von REQUEST-QUEUE;
3) wenn X = self then
   Using ← true;
   <kritischer Abschnitt>
   sonst
   HOLDER ← X; sende Privilege an HOLDER;

Algorithmus

- Garantiert wechselseitigen Ausschluß
- Verhindert Deadlock's:
  spannender Baum ist azyklisch
  $\Rightarrow$ Privileg trifft nach endlicher Zeit auf einen Request.
- Fair! (FIFO-Queue's)

Nachrichten-Komplexität:

$$\sigma\ (\log_k n)$$

k = Grad des Baums

- Verbesserungen
- Initialisierung:
  I) Ein Prozeß wird privilegiert
  II) Sendet INITIALIZE an alle Nachbarn;
  III) Alle Prozesse, die INITIALIZE empfangen setzen
  HOLDER $\leftarrow$ INIT-Sender;
  Sende INITIALIZE an alle Nachbarn;
- Fehlertoleranz

Algorithmus

VERGLEICH

Le Lann (1977)	1 ... + ∞
Lamport (1978)	3 (n-1)
Ricart & Agrawala (1981)	2 (n-1)
Carvalho & Roucairol	0 ... (2(n-1))
Ricart & Agrawala (1983)	n
Maekawa (1985)	$c \cdot \sqrt{n}$
Raymond (1989)	$\log_k n$

Verteilte und Symmetrische Algorithmen

Algorithmus

Problem: Es ist nicht einfach zu entscheiden, ob es einen besten Algorithmus gibt!

Güte eines Algorithmus hängt nicht allein vom „worst-case"-Verhalten ab!

⇒ Empirische Untersuchungen, da „mittlere"-Verhalten fast nie theoretisch bestimmt werden kann

z.B. Ausführliche Simulation aller Algorithmen bis 1983 (CNET : Lannion)

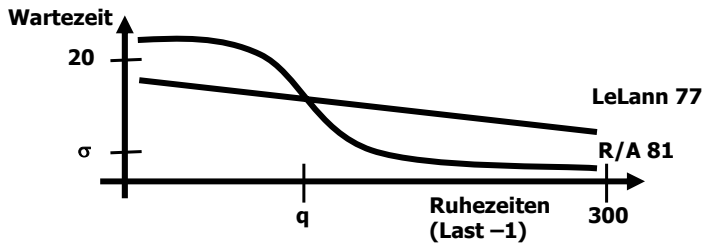
Parameter:

≠ Knoten
Ruhezeiten
Belegtzeiten

} **mittlere Wartezeit ≠ Nachrichten**

Verteilte Systeme, Sommersemester 1999 Folie 7.45

Algorithmus



⇒ **Es gibt keinen besten Algorithmus**

Verteilte Systeme, Sommersemester 1999 Folie 7.46

Algorithmus

... nochmal:

Wechselseitiger Ausschluß

Idee der „Request-Grant"-Mengen kann erweitert werden!

$\Rightarrow$ B. A. Sanders (1987)

2 Mengen/Prozess:
Inform-Menge: Alle Prozesse, die informiert werden müssen, wenn ein bestimmter Prozeß den kritischen Abschnitt verläßt.

Request-Menge: Alle Prozesse, die gefragt werden müssen, wenn ein bestimmter Prozeß in den kritischen Abschnitt will.

- Anpassbar an bestimmte Anwendungen
- Bestimmte Anforderungen an die Inform- und Request-Mengen, um wechselseitigen Ausschluß zu garantieren.

Literatur:

1) **Michel Raynal:**
Algorithmus for mutual Exclusion, 1986
State-Variablen Algorithmen, Nachrichten-basierte Algorithmen:
LeLann, Lamport, Ricart & Agrawala

2) **Mamoru Maekawa:**
A $\sqrt{n}$ Algorithm for Mutual Exclusion in Decentralized Systems,
ACM Transactions on Computer Systems, Vol. 3, No. 2, May 1985,
pp. 145 – 159

3) **Kerry Raymond:**
A Tree-Based Algorithm for Distributed Mutual Exclusion, ACM Transactions on Computer Systems, Vol. 7, No. 1, February 1989,
pp. 61 – 77
log n - Algorithmus

4) **Beverly A. Sanders:**
The Information Structure of Distributed Mutual Exclusion Algorithms,
ACM Transactions on Computer Systems, Vol. 5, No. 3, August 1987,
pp. 284 – 299
Verallgemeinerung