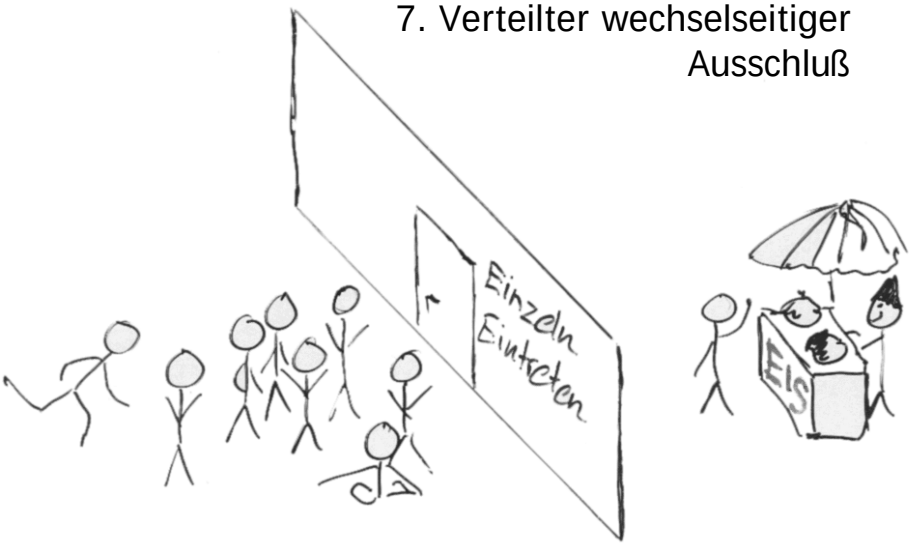
Verteilte Systeme

7. Verteilter wechselseitiger Ausschluß



Wechselseitiger Ausschluß (mutual exclusion)

- Aufgabenstellung

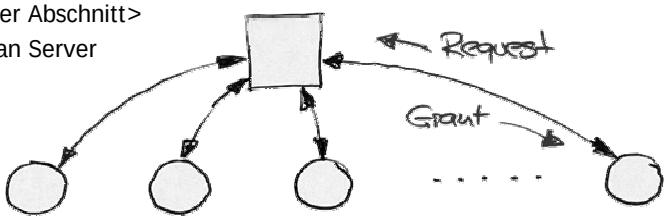
  „Mehrere Prozesse möchten etwas tun, wobei garantiert sein muß, daß es zu einem Zeitpunkt nur einer darf."

- Voraussetzungen
  - exklusive Betriebsmittel
  - „abstrakte" exklusive Betriebsmittel (z.B. Termin in einem verteilten TVS)

- Lösung bei gemeinsamen Speicher
  - Semaphore ⇒ Interessiert uns hier nicht!

Zentralisierter Ansatz

- Zentraler Server verwaltet Zustand
- Client
  - Request an Server
  - Warten auf Grant
    <Kritischer Abschnitt>
  - Release an Server



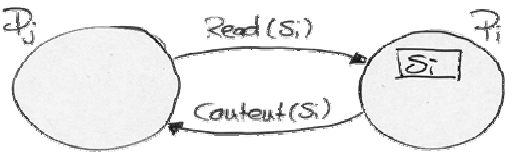
- Bemerkungen
  - 3 Nachrichten
  - SPOF

Verteilte Systeme, Winter 2003 Folie 7.3

7.1 Lösungen mittels State-Variablen

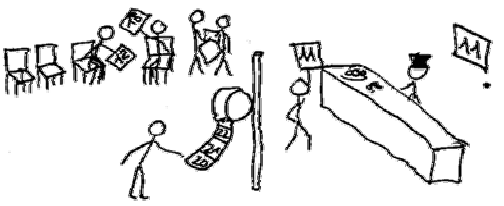
State-Variable

- Jeder Prozeß Pi hat zwei Arten von Variablen
  - Lokale Variablen
    - Nur Pi kann auf sie zugreifen (lesend und schreibend)
  - State-Variablen
    - Pi darf lesend und schreibend zugreifen
    - $\forall j : j \neq i$: Pj darf lesend zugreifen
- Einfache Umsetzbarkeit in nachrichten-basierten Systemen



Verteilte Systeme, Winter 2003 Folie 7.5

„Bäckerei-Algorithmus“ (Lamport ’74)



- Algorithmus für Pi

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

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

for j=0 to n-1 (i ¹ j) {
    wait (number[j] ¹ 0) Þ
           (number[i],i) < (number[j],j);
}

<kritischer Abschnitt>

number[i] ¬ 0;
```

(a,b) < (c,d) Û
(a < c) Ú ((a = c) Ù (b < d))

Verteilte Systeme, Winter 2003 Folie 7.6

Bemerkungen

- Berechnung der Reihenfolge durch
  - **number[i] ¬ 1+max(number[0],...,number[n-1]);**
- Algorithmus garantiert wechselseitigen Ausschluß
  - number[i] besitzt eindeutiges Minimum
  - Sei Pi Prozeß mit diesem Minimum
    - ⇒ Für Pi : $\forall j : j \neq i$: wait (true)
    - ⇒ Für Pj : $j \neq i : \exists$ wait (false);
- Algorithmus ist fair
  - Prozeß Pi muß maximal (n-1) mal warten
- Algorithmus ist korrekt?
  - Fast
  - Problem: Berechnung von number[i] ist nicht atomar!

Szenario: Ein wechselseitiger Nicht-Ausschluß

Initialisierung: **number[i] = 0**

P_1

```
number[0] == 0
number[1] == 0
     ⋮
number[0] = 1 + max(0,0);

for …
    wait (number[1] != 0) true
    Þ (number[0],0) < number[1],1) true
     ⋮
```

Im kritischen Abschnitt

P_2

```
number[0] == 0
number[1] == 0
number[1] = 1 + max(0,0);
     ⋮
for …
    wait (number[0] != 0) false
     ⋮
```

Im kritischen Abschnitt

„Bäckerei-Algorithmus“, iterierte Fassung

- Algorithmus für Prozeß i

```
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, i¹j {
    wait ¬choice[j];
    wait (number[j] ¹ 0)
           Þ (number[i],i) < (number[j],j);
    }

<kritischer Abschnitt>

number[i] ¬ 0;
```


Szenario: Ein wechselseitiger Ausschluß

Initialisierung:
`number[i] = 0`
`choice[i] = false`

P_1

```
choice[0] = true
number[0] == 0
number[1] == 0
    ⋮
number[0] = 1 + max(0,0);
choice[0] = false

for …
    wait ¬choice[1] true
    wait (number[1] != 0) true
    Þ (number[0],0) < number[1],1) true
    ⋮
```

Im kritischen Abschnitt

P_2

```
choice[1] = true
number[0] == 0
number[1] == 0
number[1] = 1 + max(0,0);
choice[1] = false
    ⋮
for …
    wait ¬choice[0] false
    wait (number[0] != 0) false
```

P2 blockiert

Bemerkungen

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

Dijkstra‘s selbst-stabilisierender Algorithmus (1974)

- Sei k>1

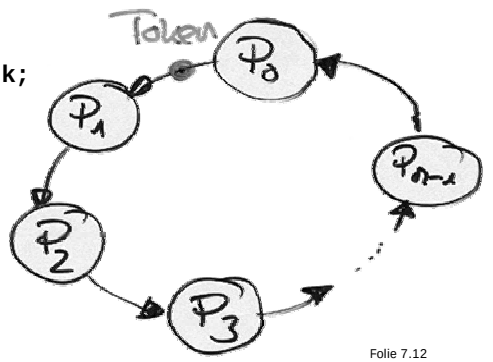
```
var flag : array [0...n-1] of 0...k;
```

- P0

```
wait flag[0] = flag[n-1];
<kritischer Abschnitt>
flag[0] ¬ (flag[0]+1) mod k;
```

- Pi (i ≠ 0)

```
wait flag[i] ¹ flag[i-1];
<kritischer Abschnitt>
flag[i] ¬ flag[i-1];
```



Bemerkungen

- Annahme: sei ∀i : **`flag[i] = 0`**
- Algorithmus garantiert wechselseitigen Ausschluß
  - Schritt 1
    - P0: **`wait (flag[0] = flag[n-1])`** true
    - Pi: **`wait (flag[i] ¹ flag[i-1])`** false ∀i : i ≠ 0
    - P0 betritt kritischen Abschnitt
    - danach **`flag[0] = 1`**
  - Schritt 2
    - P0 : **`wait (flag[0] = flag[n-1])`** false
    - P1 : **`wait (flag[1] ¹ flag[0])`** true
    - Pi : **`wait (flag[i] ¹ flag[i-1]`** false ∀i : i ≠ 0, i ≠ 1
    - P1 betritt kritischen Abschnitt
  - ...
- Algorithmus verhindert Deadlocks ✓
- Algorithmus ist fair

... aber

- Algorithmus ist nicht symmetrisch
- Prozesse bekommen das Privileg, obwohl sie gar nicht in den kritischen Abschnitt wollen!
  ⇒ Prozesse müssen Token immer weitergeben
- Fehlertoleranz
  - Abstürzende Prozesse müssen aus dem Ring entfernt werden!
  - Wie man entfernt, hängt vom Prozeß P ab
    - P0 ↯ → Anderer wird P0
    - Pi ↯ → Raus aus Ring

    ⇒ Zusätzliche Informationen für den Ringaustrag

Selbst-Stabilisierung

- Initiale Bedingung
  ∀i: flag [i] = 0 unnötig
- Korrektheit hängt von k ab
  - k ≤ n: Kein WA garantiert
    - Beispiel: n = 4, k = 2
  - k > n: WA garantiert
    - P0 inkrementiert (wenn er darf)
    - Pi schieben Token nach rechts
    - „Müll“ wird nach rechts geschoben und durch sinnvolle Inhalte in P0 ergänzt
    - Initialisierungsdauer berechenbar

P_0	P_1	P_2	P_3	
0	1	0	1	
	|	|	|	3 ↯
0	0	1	0	
|		|	|	3 ↯
1	0	0	1	
|	|		|	3 ↯
0	1	0	0	
|	|	|		3 ↯
1	0	1	0	
	|	|	|	3 ↯
1	1	0	1	
|		|	|	3 ↯
0	1	1	0	
|	|		|	3 ↯
1	0	1	1	
|	|	|		3 ↯
0	1	0	1	

Bemerkungen: State-Variablen

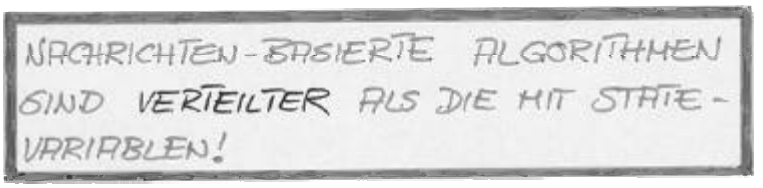
- Einfache, verhältnismäßig fehlertolerante Algorithmen
- 2 Ebenen werden durcheinander gebracht
  - Logik
  - Implementierung

  ⇒ schwer beschreibbar & schwer verständlich
- Zugriff auf State-Variablen
  - Blindes Testen von Statevariablen
  - Hohe Anzahl an unnötigen Nachrichten
- Effiziente Realisierung in „shared-memory“-System
- Vorteil gegenüber z.B. Semaphor
  - keine globalen Variablen
  - Erhöhte Fehlertoleranz

7.2 Nachrichten-basierte Lösungen

Nachrichten-basierte Ansätze

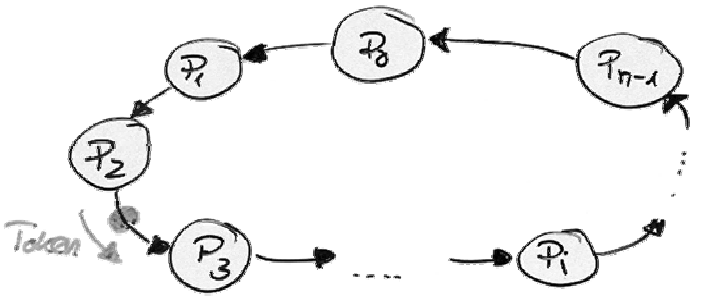
- State-Variablen können in einer nachrichten-basierten Umgebung realisiert werden
  - „Request Informationen“-Nachrichten (Pull)
  - Viele z.T. unnötige Nachrichten
- Push-Ansätze
  - „Send Informationen“-Nachrichten
  - Anzahl der Nachrichten wird minimiert
  - Keine unnützen Nachrichten
  - Alle Nachrichten signalisieren Zustandswechsel



Verteilte Systeme, Winter 2003 Folie 7.18

Le Lann's Algorithmus (1977)

- Natürlich gibt es auch hier eine Token-Ring Lösung

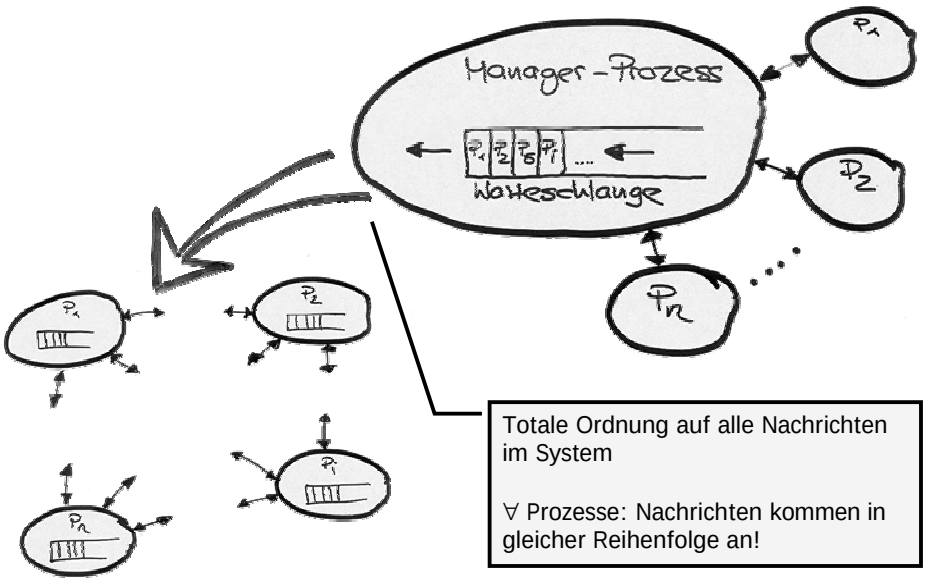


- Prozeß i
  - **Warte auf Token von Pi-1**
  - **<Kritischer Abschnitt>**
  - **Sende Token an Pi+1**

Bemerkungen

- Klar
  - Garantiert wechselseitigen Ausschluß
  - Verhindert Deadlocks
  - Fair
- Nachteile
  - Anzahl Nachrichten: 1 ... + ∞
  - Prozeß bekommt Privileg, obwohl er es nicht braucht!
  - Fehlertoleranz
    - Token-Verlust ⇒ Genau 1 neues Token
    - Prozeß-Absturz ⇒ Ring verkürzen

Verteilte Warteschlange (Lamport 1978)



Totale Ordnung auf alle Nachrichten im System

∀ Prozesse: Nachrichten kommen in gleicher Reihenfolge an!

Implementierungsansatz

- Verteilung der Nachrichten
  - Request: Multicast an alle Teilnehmer
  - Grant: Unicast Jedes Teilnehmers zurück an Sender
  - Release: Multicast an alle Teilnehmer
- Nachrichten enthalten Zeitstempel
  - Totale Ordnung ⇒ Erweiterte Lamport-Zeit
  - Einsortieren gemäß Zeitstempel
- Annahmen
  - FIFO-Eigenschaft jeweils zwischen Pi und Pk
- Lamport-Algorithmus
  - Implementierung?

Bemerkungen

- Algorithmus ist
  - Fair
  - Deadlock frei (Totale Ordnung enthält keine Zyklen)
  - Korrekt (Garantiert wechselseitigen Ausschluß)
- Nachrichten-Komplexität
  - 2 mal Multicast + (n-1) mal Unicast
  - Multicast = n Unicast ~ 3(n-1) Nachrichten
  - Multicast = 1 Unicast ~ n+1 Nachrichten
- Allgemein O(n)

Ein optimaler Algorithmus

- Ricart & Agrawala's Algorithmus (1981)
  - Optimierung von Lamport's Algorithmus
- Prozess Pi möchte in den kritischen Abschnitt
  - **`Sende (Request, clocki, i) an alle`**
  - **`Warte auf n-1 Grants`**
  - **`<kritischer Abschnitt>`**
- Prozess Pk empfängt Request von Pi
  - Sendet Reply sofort, wenn
    - Pk nicht in den kritischen Abschnitt will!
    - Pk zwar in den kritischen Abschnitt will, aber Pi höhere Rechte hat
  - Verzögert das Reply, wenn
    - Pk in den kritischen Abschnitt will und höhere Rechte als Pi hat

Ein optimaler Algorithmus

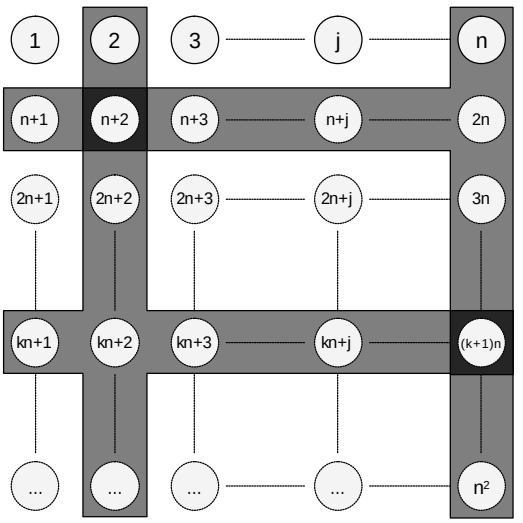
- Ricart & Agrawala's Algorithmus (1981)
  - Optimierung von Lamport's Algorithmus
- Prozess Pi möchte in den kritischen Abschnitt
  - **`Sende (Request, clocki, i) an alle`**
  - **`Warte auf n-1 Grants`**
  - **`<kritischer Abschnitt>`**
  - **`Gib alle verzögerten Replies raus`**
- Nachrichtenkomplexität: 2(n-1)

Ein optimaler Algorithmus

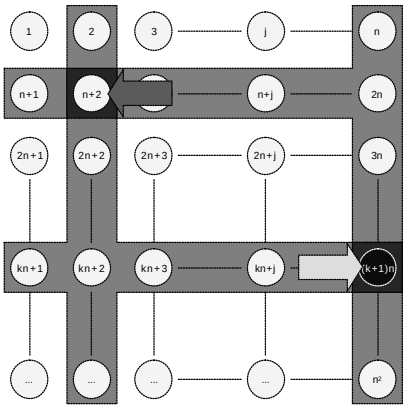
- In dieser Richtung sind dann einige Verbesserungen vorgenommen worden
  - Le Lann (1977) 1 … + ∞
  - Lamport (1978) 3 (n-1)
  - Ricart & Agrawala (1981) 2 (n-1)
  - Carvalho & Roucairol 0 … (2(n-1))
  - Suzuki & Kasami (1982) n
  - Ricart & Agrawala (1983) n

Ein optimaler Algorithmus: Maekawa (1985)

- Idee
  - Anordnung in Gitter
  - Prozeß i fragt Gruppe Si von anderen Prozessen um Erlaubnis
  - Si ∩ Sj ? ∅ ∀ i? j



Ein optimaler Algorithmus



- Deadlock-Gefahr
- Nachrichten für wechselseitigen Ausschluß
  - REQUEST, LOCKED, RELEASE
- Nachrichten zur Deadlock-Vermeidung
  - INQUIRE, RELINQUISH, FAILED

Algorithmus

- Pi möchte kritischen Abschnitt betreten
  - **Sende REQUEST an alle Elemente von Si;**
  - **Warte auf LOCKED von allen Prozessen aus Si;**
  - **<kritischer Abschnitt>**
  - **Sende RELEASE an alle Prozesse aus Si;**
- Pj ∈ Si empfängt REQUEST-Nachricht

```
if (frei) {
    Zustand ¬ locked;
    Sende LOCKED-Nachricht an Pi;
}
else { /* Abschnitt ist von Pk Î Sj belegt */
    if (Pj kennt aussichtsreicheren REQUEST-Kandidaten)
        Sende FAILED-Nachricht an Pi;
    else
        Sende INQUIRE-Nachricht an Pk;
        /* Warte auf RELINQUISH oder RELEASE von Pk */
}
```


Algorithmus (contd.)

- Pk ∈ Si empfängt INQUIRE-Nachricht von Pj

```
if (Prozess Î Sk hat FAILED an Pk geschickt) {
    Hebe bereits angelaufene LOCK's auf;
    Sende RELINQUISH an Pj;
}
else
    Sende RELEASE nach Verlassen des KA;
```

- weitere Funktionen?
- Eigenschaften
  - Algorithmus garantiert wechselseitigen Ausschluß
  - Auflösung von Deadlocks
  - Fairness
  - Fehlertoleranz
    - Übernahme der RG-Menge eines Prozesses Pi
    - Das Wissen von Pi ist in O(√n) Nachrichten wieder herleitbar

Nachrichtenkomplexität

- Grundversion

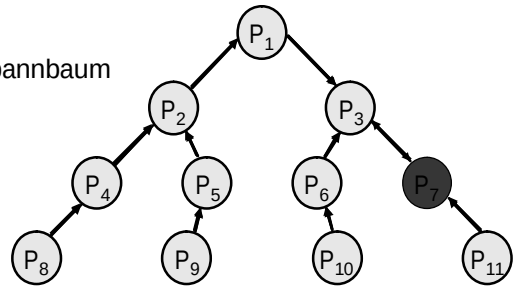
$$c \cdot \sqrt{n} \quad c \in [6 \ldots 10]$$

- Verbesserungen
  - „Request-Grant“-Mengen im Gitter sind nicht minimal
  - Problem aus der projektiven Geometrie
    - Für bestimmte n sind minimale Si möglich (|Si| = Potenz einer Primzahl)
- Im minimalen Fall ist die Nachrichten-Komplexität

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

Ein optimaler Algorithmus: Raymond (1989)

- Idee: Topologie ist ein Spannbaum
- Ein Prozess Pi besitzt das Privileg („Token“)
  - kann den kritischen Abschnitt betreten
- Andere Prozesse kennen Richtung zum Token



- Beim Tokenwechsel werden Richtungen aktualisiert
  - Einfache Pointeroperationen beim Wechseln
- Requests werden in Tokenrichtung resendet
  - Letztendlich kommt Token an
  - Token kann dem Request nicht ewig „davonrennen“

Nachrichten und Datenstrukturen

- Nachrichten
  - REQUEST : Ein Prozess will das Token
  - PRIVILEG : Hier ist das Privileg!
- Lokale Variablen für Prozeß Pi
  - HOLDER: „self“ oder Name eines direkten Nachbarn in dessen Richtung das Privileg liegt
  - USING : bool = true wenn Pi im kritischen Abschnitt ist (USING = true ⇒ Holder = self)
  - REQUEST-QUEUE = FIFO–Schlange angekommener Request-Nachrichten von Prozessen, die das Privileg noch nicht hatten
    - Enthält nur „self“ und Namen direkter Nachbarn
  - ASKED : bool = true ⇔ Request nach Privileg in Richtung HOLDER ist bereits verschickt worden.

Verteilte Systeme, Winter 2003 Folie 7.33

Algorithmus

- Prozess Pi will Privileg
  - **Füge ‚self‘ an REQUEST-QUEUE an;**
  - **if (Ø ASKED) {**
    sende REQUEST an HOLDER;
    ASKED ¬ true;
    }
- Prozess Pk emfängt Request von Nachbar Pi
  - **Füge „Pi“ an REQUEST-QUEUE an;**
  - **if (Ø ASKED) {**
    sende REQUEST an HOLDER;
    ASKED ¬ true;
    }

Verteilte Systeme, Winter 2003 Folie 7.34

Algorithmus (contd.)

- Prozess Pk hat Privileg und will es loswerden
  -
```
if (|REQUEST-QUEUE| > 0) {
    X ¬ Head(REQUEST-QUEUE);
    Holder ¬ X;
    sende PRIVILEGE  an HOLDER;
    asked ¬ using ¬ false;
}
```
- Prozess Pk empfängt Privileg von Pm
  -
```
X ¬ Head(REQUEST-QUEUE);
```
  -
```
if (X = self) {
    Using ¬ true;
    <kritischer Abschnitt>
}
else {
    HOLDER ¬ X;
    sende Privilege an HOLDER;
}
```


Bemerkungen

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

$$O(\log_k n)$$

  - k = Grad des Baumes

Bemerkungen (contd.)

- Im Artikel werden weitere Verbesserungen vorgeschlagen
- Initialisierung
  - Ein Prozeß wird privilegiert
  - Sendet INITIALIZE an alle Nachbarn;
  - Alle Prozesse, die INITIALIZE empfangen
    - Setzen HOLDER ← INIT-Sender;
    - Senden INITIALIZE an alle Nachbarn;
- Fehlertoleranz

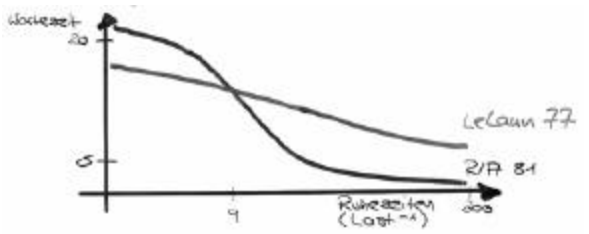
Vergleich der vorgestellten Algorithem

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)	$\log_k n$

- Unterschiede
  - Symmetrische und asymmetrische Algorithmen

Welchen Algorithmus soll man nehmen

- DEN optimalen Algorithmus gibt es nicht!
- Güte hängt nicht allein vom „Worst-Case“-Verhalten ab
  - Empirische Untersuchungen, da „mittleres“ Verhalten fast nie theoretisch bestimmt werden kann
  - z.B. Ausführliche Simulation aller Algorithmen bis 1983
- Simulationsparameter
  - Anzahl der Knoten, Ruhezeiten, Belegtzeiten



Nochmal wechselseitigen Ausschluß

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

- Michel Raynal
  Algorithmus for mutual Exclusion
  MIT Press 1986
  (State-Variablen und nachrichten-basierte Algorithmen)
- Mamoru Maekawa
  A Ön Algorithm for Mutual Exclusion in Decentralized Systems
  ACM Transactions on Computer Systems, Vol. 3, No. 2, May 1985, pp. 145 – 159
- Kerry Raymond
  A Tree-Based Algorithm for Distributed Mutual Exclusion
  ACM Transactions on Computer Systems, Vol. 7, No. 1, February 1989, pp. 61 – 77
- 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