

Gegenseitiger Ausschluss

Problemstellung

- ▶ Mehrere Prozesse greifen auf gemeinsame Daten/Ressourcen zu; dieser Zugriff muss koordiniert werden: Immer nur ein Prozess darf den kritischen Abschnitt (k. A.) betreten.

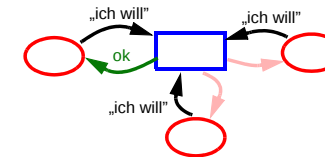
Literatur

- ▶ M. Singhal, N. Shivaratri: Advanced Concepts in Operating Systems
- ▶ Original-Publikationen zu den einzelnen Algorithmen
- ▶ Ein guter Überblick über die meisten wichtigen Forschungsarbeiten auf diesem Gebiet liefert insbesondere: G. Cao, M. Singhal: A Delay-Optimal Quorum-Based Mutual Exclusion Algorithm for Distributed Systems, IEEE TPDS 12, Dec. 2001

Gegenseitiger Ausschluss: Überblick

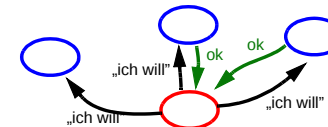
Zentraler Koordinator:

- ▶ Alle fragen einen



Vollständig verteilt:

- ▶ Jeder fragt jeden



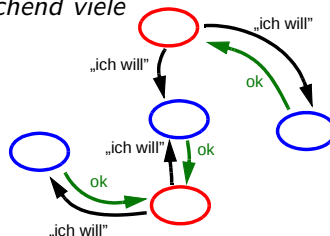
Beispiel

- ▶ Lamport: Vollständige Replikation des Systemzustandes
- ▶ Ricart und Agrawala: Reduktion der Nachrichtenzahl

Gegenseitiger Ausschluss: Überblick

Quorumbasierte Algorithmen:

- ▶ Dezentral, aber reduzierter Aufwand
- ▶ Jeder fragt ausreichend viele



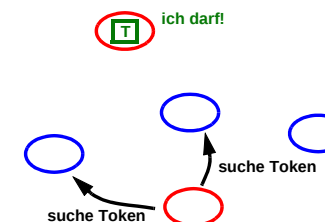
Beispiel

- ▶ Maekawa: Quorumbasierter gegenseitiger Ausschluss
- ▶ Sanders: Universeller erlaubnisbasierter Algorithmus

Gegenseitiger Ausschluss: Überblick

Tokenbasierte Algorithmen

- ▶ Wer Token besitzt, darf in k.A.!
- ▶ Wie bekommt man das Token?



Beispiel

- ▶ Perpetuum-Mobile-Algorithmus: Token wandert alleine
- ▶ Algorithmus von Suzuki und Kasami: Broadcast-Suche
- ▶ Algorithmus von Raymond: Suche auf Baumstrukturen
- ▶ Algorithmus von Singhal: „Intelligenter“ Broadcast

Gegenseitiger Ausschluss: Überblick

Systemmodell

- ▶ Prozesse wickeln zeitlich sequentiell eine Folge von Aktionen ab
- ▶ Es gibt drei Arten von Aktionen:
 - ▶ lokale Aktionen (keinerlei Auswirkungen auf andere Prozesse)
 - ▶ Aktionen zum Senden von Nachrichten
 - ▶ Aktionen zum Empfangen von Nachrichten

Gegenseitiger Ausschluss: Überblick

Weitere wesentliche Gesichtspunkte

- ▶ Fairness
 - ▶ Die Zulassung erfolgt in der Reihenfolge der Anforderungen.
 - ▶ Eine globale Ordnung nach physikalischer Zeit ist i. d. R. nicht möglich. Üblich daher „Reihenfolge“ gemäß Ordnung durch logische Uhren
 - ▶ Fairness impliziert Aushungerungsfreiheit (aber nicht umgekehrt)
- ▶ Fehlertoleranz
- ▶ Erzeugte Netzlast (Nachrichtenanzahl)
- ▶ Erzielte Performanz

Gegenseitiger Ausschluss: Überblick

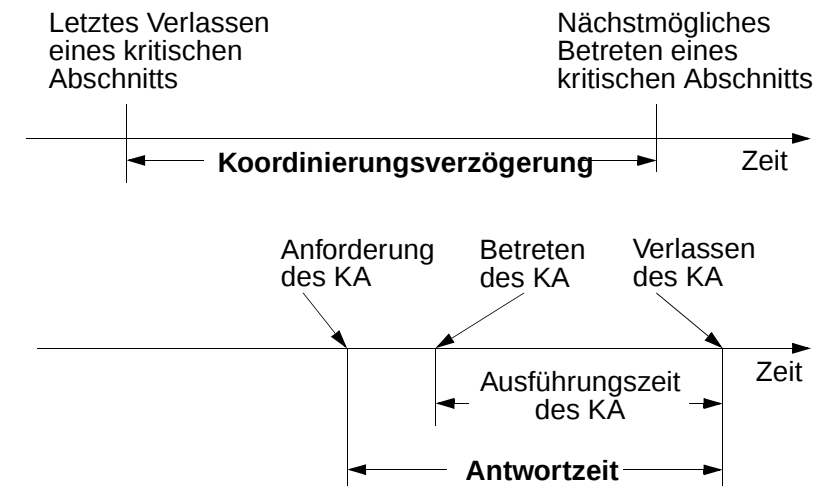
Formale Anforderungen

- GA1: Gegenseitiger Ausschluss:** Zu keinem Zeitpunkt hat mehr als ein Prozess die Erlaubnis, den kritischen Abschnitt auszuführen.
- GA2: Lebendigkeit (bzw. Verklemmungsfreiheit):** Wenn sich kein Prozess im k. A. befindet und mindestens ein Prozess diesen betreten möchte, erhält ein Prozess in endlicher Zeit die Erlaubnis, den kritischen Abschnitt zu betreten.

Strengere Anforderung

- GA2': Aushungerungsfreiheit:** Ein Prozess, der den kritischen Abschnitt betreten möchte, erhält innerhalb endlicher Zeit die Erlaubnis dazu.

Gegenseitiger Ausschluss: Überblick

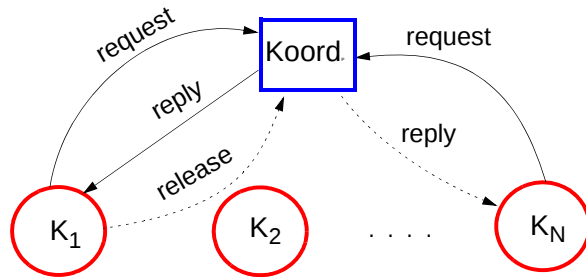


Durchsatz: Anzahl KA pro Zeit; $1/(\text{Koord. Verz.} + \text{Ausführungszeit})$

Zentraler Koordinator

Nachrichtentypen

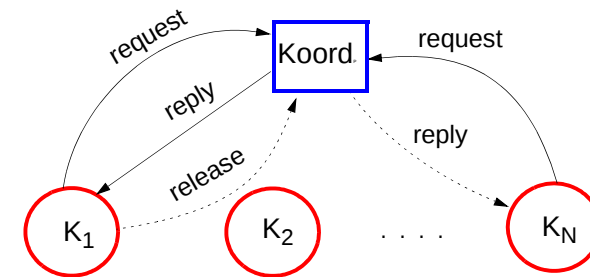
- ▶ REQUEST Anforderung des kritischen Abschnitts
- ▶ REPLY Zuteilung des kritischen Abschnitts
- ▶ RELEASE Freigabe des kritischen Abschnitts



Zentraler Koordinator

Eigenschaften

- ▶ Anzahl der Nachrichten: 3 pro kritischem Abschnitt
- ▶ Koordinierungsverzögerung: 2 Nachrichtenlaufzeiten (RELEASE, REPLY)



Algorithmus von Lamport

Systemmodell

- ▶ Nachrichtenkanäle: zuverlässig, FIFO
- ▶ Lamports logische Uhren zum Erzeugen von Zeitstempeln (t)

Algorithmus

- ▶ Eintrittswunsch des Knotens P_i :
 - ▶ sende REQUEST(t, i) an alle (einschließlich sich selbst)
- ▶ Knoten P_i verlässt den kritischen Abschnitts:
 - ▶ sende RELEASE(t, i) an alle

Algorithmus von Lamport

- ▶ Bearbeitung von REQUEST-Nachrichten:
 - ▶ Jeder Prozess sammelt bei ihm angekommene REQUEST-Nachrichten und ordnet sie nach steigendem Zeitstempel in lokaler REQUEST-Warteschlange.
 - ▶ Wenn P_j eine REQUEST-Nachricht von P_i erhält, sendet P_j eine Bestätigungsnachricht ACK(t, j) an P_i

Algorithmus von Lamport

► Betreten kritischer Abschnitte:

P_i kann kritischen Abschnitt betreten, wenn diese beiden Bedingungen erfüllt sind:

- P_i hat von allen anderen Knoten eine (beliebige) Nachricht mit größerem Zeitstempel als sein eigener REQUEST erhalten (FIFO-Kanal $\Rightarrow$ keine Nachrichten mit älteren Zeitstempel unterwegs $\Rightarrow$ es kann kein REQUEST mit kleinerem Zeitstempel mehr ankommen!)
- Der eigene REQUEST ist am Kopf der Warteschlange

Algorithmus von Lamport

Bezeichnungen

- T : mittlere Nachrichtenlaufzeit;
- N : Anzahl der Knoten
- E : mittlere Ausführungszeit für kritischen Abschnitt

Eigenschaften des Algorithmus

- Antwortzeit: $2T + E$
- Nachrichten: $3(N - 1)$ pro k. A.
- Koordinierungsverzögerung: T

Fehlertoleranz

- Keine! (Sobald ein beliebiger Knoten ausfällt, kann kein Knoten mehr den kritischen Abschnitt betreten)

Algorithmus von Lamport

► Verlassen kritischer Abschnitte

- P_i entfernt bei Verlassen seines k. A. die eigene Anforderung aus der lokalen Warteschlange und sendet eine RELEASE-Nachricht an alle Knoten
- Wenn P_j eine RELEASE-Nachricht von P_i bekommt, entfernt es die zugehörige REQUEST-Nachricht aus seiner Warteschlange

Algorithmus von Ricart und Agrawala

Systemmodell

- Nachrichtenkanal: zuverlässig (FIFO-Eigenschaft nicht mehr notwendig!)
- Reduktion der Nachrichtenanzahl durch Kombination von ACK und RELEASE zu einer REPLY-Nachricht

Beschreibung des Algorithmus

- ▶ Eintrittswunsch des Knotens P_i : REQUEST(t, i) an alle
- ▶ Bearbeitung von REQUEST:
 - ▶ Wenn P_j eine REQUEST-Nachricht von P_i erhält, sendet er eine REPLY-Nachricht an P_i , falls er
 - ▶ weder selbst seinen kritischen Abschnitt betreten will noch ihn gerade betreten hat, oder
 - ▶ er selbst eine Anforderung gestellt hat, aber der Zeitstempel von P_i 's Anforderung kleiner ist als der seiner eigenen Anforderung.
 - ▶ In allen anderen Fällen wird das Senden der REPLY-Nachricht aufgeschoben (Eintrag in Warteschlange!).

Algorithmus von Ricart und Agrawala

Bezeichnungen

- ▶ T mittlere Nachrichtenlaufzeit
- ▶ E mittlere Ausführungszeit für k. A.

Eigenschaften des Algorithmus

- ▶ Antwortzeit: $2T + E$
- ▶ Nachrichten: $2(N - 1)$ pro k. A.
- ▶ Koordinierungsverzögerung: T

Beschreibung des Algorithmus

- ▶ Betreten kritischer Abschnitte
 - ▶ P_i kann k. A. betreten, wenn er auf seine Anforderung von allen anderen eine REPLY-Nachricht erhalten hat.
- ▶ Verlassen des kritischen Abschnitts durch P_i
 - ▶ P_i versendet zu allen aufgeschobenen REQUESTs eine REPLY Nachricht.

Optimierung von Carvalho und Roucairol

Ausgangssituation

- ▶ Bisheriger Algorithmus von Ricart und Agrawala; P_i hat von P_j ein REPLY erhalten;

Optimierungs-idee:

- ▶ Implizites „Weiterverwenden“ der durch das REPLY erteilten Erlaubnis für den k. A.
- ▶ Weiterverwendung, solange P_j nicht selbst in den k. A. will (d.h. P_i hat seit dem REPLY von P_j keinen REQUEST empfangen und folglich keinen REPLY an P_j gesendet)

Anzahl der Nachrichten pro k.A.: Variabel $0 \dots 2(N - 1)$

- ▶ Besonders effizient, wenn ein kleiner Teil der Knoten den k. A. mehrfach betreten will

Quorenbasierte verteilte Algorithmen

Überlegung zur weiteren Reduktion der Nachrichtenzahl

- ▶ Alle bisher betrachteten verteilten Algorithmen benötigen im ungünstigsten Fall $O(N)$ Nachrichten pro k. A.
- ▶ Quorenbasierte Algorithmen verfolgen die Idee, nur mit einer Teilmenge aller Knoten (einem Quorum) zu interagieren, um den kritischen Abschnitt betreten zu können
- ▶ Quorum-Mengen sollen so gebildet werden, dass
 - ▶ der gegenseitige Ausschluss sichergestellt ist
 - ▶ der Kommunikationsaufwand möglichst weit reduziert wird
 - ▶ eine möglichst gleichmäßige Verteilung der Last erreicht wird

Quorenbasierte verteilte Algorithmen

Nicht alle Quoremsysteme sind „gleich gut“

- ▶ Evtl. unnötiger Aufwand, wenn sich zwei Quoren in mehreren Knoten überschneiden
- ▶ Ungleiche Lastverteilung
 - ▶ falls manche Knoten häufiger in Quoren verwendet als andere
 - ▶ bei Fehlertoleranz-Anwendungen: Ausfall mancher Knoten kann mehr Schaden anrichten

Von Interesse sind daher vor allem Quoren mit bestimmten Minimalitätseigenschaften!

Quorenbasierte verteilte Algorithmen

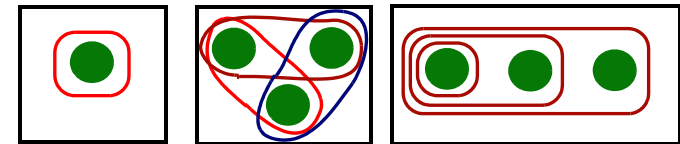
Allgemeine Definition von „Quorensystem“:

- ▶ Menge von Knotenmengen („Quoren“), von denen sich je zwei in mindestens einem Element überschneiden

Vielfältige Anwendungsgebiete in verteilten Systemen

- ▶ Gegenseitiger Ausschluss, fehlertolerante Einigungs- und Commit-Protokolle, gemeinsame verteilte Variablen,...

Beispiele



Quorenbasierte verteilte Algorithmen

Satz (hier nicht bewiesen)

- ▶ Ein ideales Quorensystem (alle Quoren sind gleich groß und überschneiden sich in nur einem Element) existiert nur, wenn sich die Knotenanzahl N darstellen lässt in der Form $N = p^m(p^m + 1) + 1$, wobei p eine Primzahl ist.
- ▶ Beispiel: $N = 13 = 3^1(3^1 + 1) + 1$

In anderen Fällen müssen die angegebenen Bedingungen etwas gelockert werden.

Quorenbasierte verteilte Algorithmen

Einfache Quorensysteme

- ▶ $\{\{A\}\}$: Ein zentraler Koordinator
- ▶ $\{\{K_1, K_2, \dots, K_n\}\}$: Alle Knoten
- ▶ Mehrheitsquoren: Alle Mengen von Knoten mit mehr als der Hälfte der Knoten
- ▶ Gewichtete Mehrheitsquoren:
 - ▶ Jeder Knoten K_i hat Gewicht G_i , $G_{ges} = \sum_i G_i$
 - ▶ Quoren sind alle Knotenmengen mit Gesamtgewicht $> G_{ges}/2$

Quorenbasierte verteilte Algorithmen

„Crumbling Wall“-Quoren

- ▶ Knoten sind in unterschiedlich große Reihen von Blöcken aufgeteilt
- ▶ Jedes Quorum besteht aus einer kompletten Reihe und aus je einem Knoten jeder darunter liegenden Reihe

Auch Bäume sind geeignet um Quorensystem aufzubauen.

Quorenbasierte verteilte Algorithmen

Quadratgitterkonstruktion von Quoren

- ▶ Eintragen der Knoten-IDs in ein Quadratgitter
- ▶ Quorum: Alle Elemente in selber Zeile oder Spalte wie Knoten selbst
 - ▶ Die Quoren überschneiden sich i. d. R. in zwei Elementen
- ▶ Quorengröße ist $2\lceil\sqrt{N}\rceil - 1$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Algorithmus von Maekawa

Verteilter Algorithmus, der Quoren zur Reduktion des Kommunikationsaufwands verwendet

Datenstrukturen eines Knoten P_i

- ▶ **Anforderungsmenge**: Menge von Knoten, von denen P_i eine Anforderung für den kritischen Abschnitt empfangen hat
- ▶ **Erlaubnismenge**: Menge von Antworten auf eine eigene Anforderung für den kritischen Abschnitt
- ▶ **Belegungskennung**: belegt für P_k , falls P_k von P_i die Erlaubnis für den kritischen Abschnitt erhalten hat
- ▶ **Logische Uhren** nach Lamport werden verwendet, um eine totale Ordnung auf Anforderungen zu definieren

Algorithmus von Maekawa

Grundidee des Algorithmus

- ▶ Knoten P_i fordert kritischen Abschnitt an: REQUEST(i , t) an alle Knoten des Quorums von P_i
- ▶ P_j empfängt REQUEST(i , t): P_j in Anforderungsmenge aufnehmen und je nach Zustand von P_j :
 - ▶ *frei*: Zustand „belegt für P_i “; Nachricht LOCKED an P_i
 - ▶ *belegt für r* : Senden von LOCKED wird verzögert
- ▶ P_k empfängt LOCKED von P_i :
 - ▶ P_i in die Erlaubnismenge aufnehmen
 - ▶ Falls alle Quorummitglieder in der Erlaubnismenge sind, darf der **kritische** Abschnitt betreten werden

Algorithmus von Maekawa

Grundidee des Algorithmus (2)

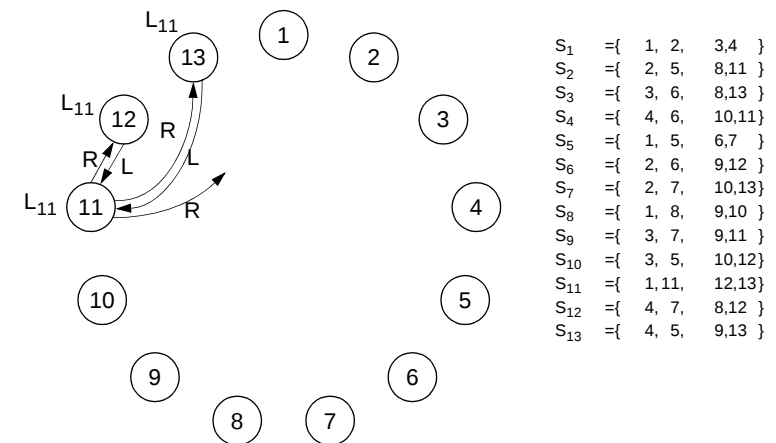
- ▶ Knoten P_j verlässt den kritischen Abschnitt:
 - ▶ Erlaubnismenge leeren; Nachricht RELEASE an alle Quorummitglieder
- ▶ Knoten P_i empfängt von P_j die Nachricht RELEASE:
 - ▶ P_j aus Anforderungsmenge entfernen
 - ▶ Falls die Anforderungsmenge weitere Anforderungen enthält: Für älteste Anforderung von P_r aus der Anforderungsmenge: Zustand „belegt für r “; Nachricht LOCKED an den Knoten P_r

Algorithmus von Maekawa

- ▶ Bisher beschriebener Algorithmus gewährleistet den gegenseitigen Ausschluss
- ▶ Es kann allerdings zu Verklemmungen kommen!

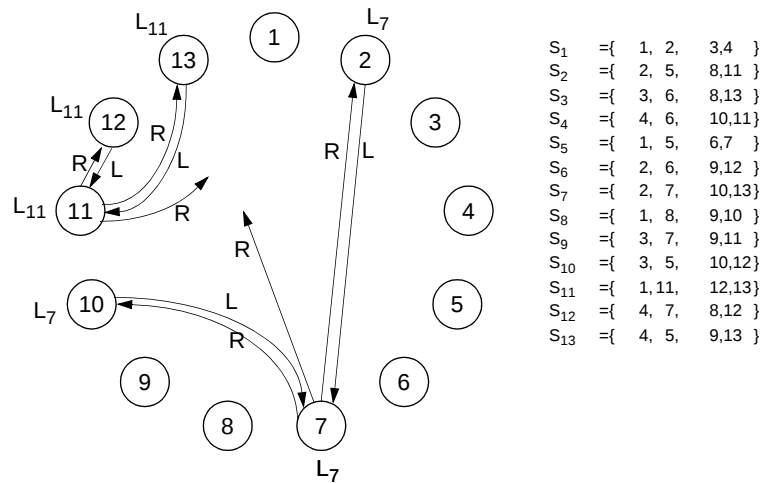
Algorithmus von Maekawa

Beispiel-Ablauf für Verklemmung (1)



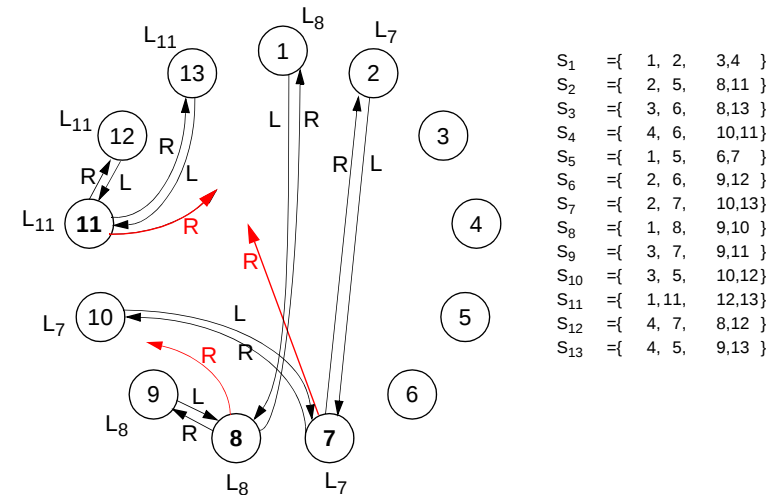
Algorithmus von Maekawa

Beispiel-Ablauf für Verklemmung (2)



Algorithmus von Maekawa

Beispiel-Ablauf für Verklemmung (3)



Algorithmus von Maekawa

Vollständiger Algorithmus trifft Maßnahmen zum Erkennung und Auflösen von Verklemmungen:

- Im Konfliktfall bekommt eine Anforderung mit kleinerem Zeitstempel Vorrang (außer, wenn die Anforderung mit größerem Zeitstempel bereits die erforderliche Erlaubnis für den k. A. erhalten hat)
- Dazu drei weitere Nachrichten:
 - FAILED: Zeigt an, dass Knoten derzeit für eine Anforderung mit kleinerem Zeitstempel belegt ist
 - INQUIRE: Bittet um die Rückgabe einer Zuteilung (LOCKED)
 - RELINQUISH: Gibt Zuteilung zurück

Algorithmus von Maekawa

Erweiterungen zur Deadlock-Behandlung

- P_j empfängt REQUEST(i, t) von P_i , hat P_i in die Anforderungsmenge aufgenommen, und ist „belegt für P_r “:
 - Falls P_i ältestes Element in der Anforderungsmenge: P_j sendet INQUIRE an P_r
 - Falls nicht: P_j sendet FAILED an P_i
- P_r empfängt INQUIRE von P_j :
 - Falls P_r bereits FAILED empfangen hat: RELINQUISH an P_j , P_j aus Erlaubnismenge entfernen
 - Ansonsten: Empfang von INQUIRE merken (bei nachfolgenden Empfang von FAILED muss das Senden von RELINQUISH nachgeholt werden)

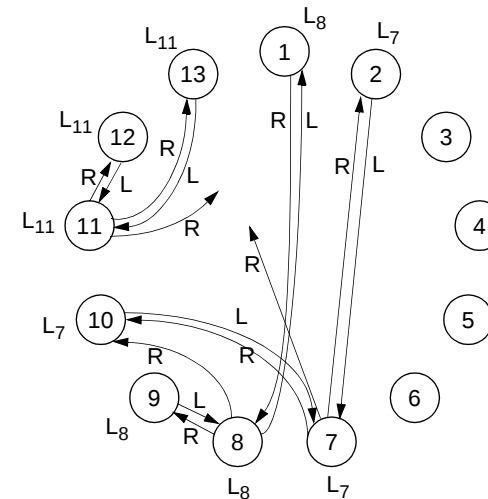
Algorithmus von Maekawa

Erweiterungen zur Deadlock-Behandlung

- ▶ Wenn P_j die Nachricht RELINQUISH von P_i empfängt:
Belegung für P_i aufheben; für ältestes Element P_r in der Anforderungsmenge „belegt für P_r “ vermerken und LOCKED an P_r senden
- ▶ Wenn P_i von P_j FAILED empfängt:
 - ▶ Falls P_i bereits INQUIRE von einem Knoten P_k empfangen hat: RELINQUISH an P_k , P_k aus Erlaubnismenge entfernen
 - ▶ Ansonsten: Empfang von FAILED merken (bei nachfolgenden Empfang von INQUIRE muss das Senden von RELINQUISH nachgeholt werden)

Algorithmus von Maekawa

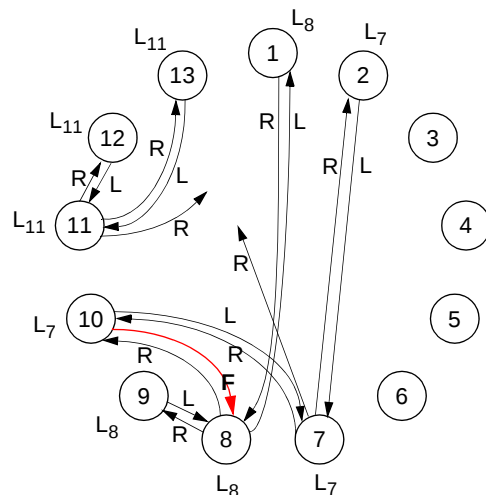
Beispiel-Ablauf (4)

Quorum von P_i ist S_i :

S_1	= { 1, 2, 3, 4 }
S_2	= { 2, 5, 8, 11 }
S_3	= { 3, 6, 8, 13 }
S_4	= { 4, 6, 10, 11 }
S_5	= { 1, 5, 6, 7 }
S_6	= { 2, 6, 9, 12 }
S_7	= { 2, 7, 10, 13 }
S_8	= { 1, 8, 9, 10 }
S_9	= { 3, 7, 9, 11 }
S_{10}	= { 3, 5, 10, 12 }
S_{11}	= { 1, 11, 12, 13 }
S_{12}	= { 4, 7, 8, 12 }
S_{13}	= { 4, 5, 9, 13 }

Algorithmus von Maekawa

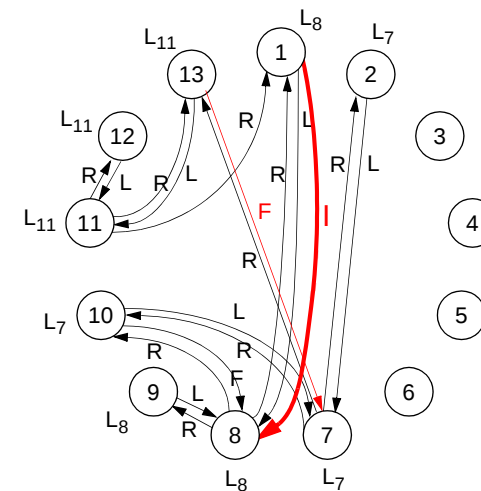
Beispiel-Ablauf (5)

Quorum von P_i ist S_i :

S_1	= { 1, 2, 3, 4 }
S_2	= { 2, 5, 8, 11 }
S_3	= { 3, 6, 8, 13 }
S_4	= { 4, 6, 10, 11 }
S_5	= { 1, 5, 6, 7 }
S_6	= { 2, 6, 9, 12 }
S_7	= { 2, 7, 10, 13 }
S_8	= { 1, 8, 9, 10 }
S_9	= { 3, 7, 9, 11 }
S_{10}	= { 3, 5, 10, 12 }
S_{11}	= { 1, 11, 12, 13 }
S_{12}	= { 4, 7, 8, 12 }
S_{13}	= { 4, 5, 9, 13 }

Algorithmus von Maekawa

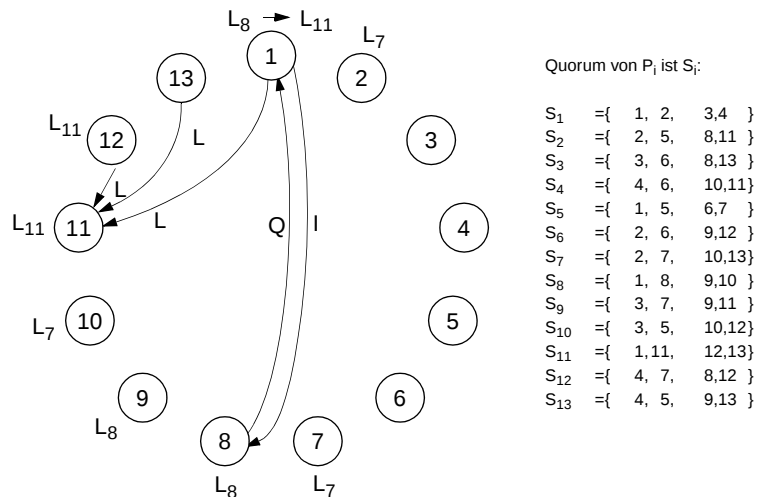
Beispiel-Ablauf (6)

Quorum von P_i ist S_i :

S_1	= { 1, 2, 3, 4 }
S_2	= { 2, 5, 8, 11 }
S_3	= { 3, 6, 8, 13 }
S_4	= { 4, 6, 10, 11 }
S_5	= { 1, 5, 6, 7 }
S_6	= { 2, 6, 9, 12 }
S_7	= { 2, 7, 10, 13 }
S_8	= { 1, 8, 9, 10 }
S_9	= { 3, 7, 9, 11 }
S_{10}	= { 3, 5, 10, 12 }
S_{11}	= { 1, 11, 12, 13 }
S_{12}	= { 4, 7, 8, 12 }
S_{13}	= { 4, 5, 9, 13 }

Algorithmus von Maekawa

Beispiel-Ablauf (7)



Algorithmus von Maekawa

Bezeichnungen:

- ▶ T mittlere Nachrichtenlaufzeit
- ▶ E mittlere Ausführungszeit für k . A.
- ▶ K mittlere Größe eines Quorums

Eigenschaften des Algorithmus:

- ▶ Antwortzeit: $2T + E$
 - ▶ Koordinierungsverzögerung: $2T$
 - ▶ Nachrichten:
 - ▶ keine Konflikte: $3(K - 1)$ pro k.A.
 - ▶ Konfliktfall, normal: $4(K - 1)$ pro k.A. (FAILED)
 - ▶ ungünstigster Fall: $5(K - 1)$ pro k.A. (INQUIRE, RELINQ.)
- ⇒ $O(\sqrt{N})$ Nachrichten bei geeigneten Quoren

Tokenbasierte Algorithmen

Prinzip

- ▶ Im System existiert genau ein Token
- ▶ Ein Prozess kann einen kritischen Abschnitt nur betreten, wenn er über das Token verfügt

Passive Verfahren

- ▶ „Perpetuum Mobile”

Aktive Verfahren

- ▶ Algorithmus von Suzuki/Kasami: Vollständige Suche im ganzen Netz
- ▶ Algorithmus von Raymonds: Gezielte Suche nach vordefinierter Baumstruktur
- ▶ Algorithmus von Singhal: Reduktion der Nachrichtenzahl durch intelligente, selbstadaptierende Suche nach dem Token

Perpetuum Mobile

Vorgehen

- ▶ Token wird automatisch auf (virtuellem) Ring weitergeschickt
- ▶ Knoten, der k. A. betreten will, wartet einfach auf das Token

Eigenschaften

- ▶ (Fast) alle Knoten wollen (fast) immer in den k. A.:
 - ▶ (Meist) nur 1 Nachricht zwischen allen k.A., nahezu optimal
- ▶ Ansonsten:
 - ▶ Durchschnittliche Wartezeit auf Token $N/2$ Nachrichtenlaufzeiten
 - ▶ Unbeschränkte Anzahl von Nachrichten, selbst wenn kein Knoten den k.A. betreten will

Aktive Tokenbasierte Algorithmen

Generelles Prinzip

- ▶ Ein Prozess, der in seinen kritischen Abschnitt eintreten will, aber nicht über das Token verfügt, versendet REQUEST-Anforderungen.
- ▶ Ein Prozess, der REQUEST-Anforderungen erhält, übergibt das Token, sobald es frei ist, an einen Anforderer.

Probleme

- ▶ Erzeugung genau eines Tokens
- ▶ Bestimmung des Prozesses, der den Token erhält
- ▶ Welche REQUEST-Anforderungen sind veraltet, welche aktuell?

Algorithmus von Suzuki-Kasami

Anforderung des Tokens

- ▶ Wenn Prozess P_i das Token benötigt, erhöht er seine Sequenznummer $RN[i]$ und sendet eine Nachricht $REQUEST(i, RN[i])$ an alle anderen
- ▶ Wenn ein Prozess P_j eine Nachricht $REQUEST(i, sn)$ empfängt, setzt er $RN[i] = \max(RN[i], sn)$.
Falls P_j über das Token verfügt und nicht im kritischen Abschnitt ist, sendet er es an P_i , wenn $RN[i] == LN[i] + 1$ ist.

Algorithmus von Suzuki-Kasami

Datenstrukturen

- ▶ Lokale Datenstruktur des Prozesses P_i

```
int RN[n] // Sequenznummern, soweit bekannt
```

- ▶ Datenstruktur des Tokens

```
struct {
    int LN[n];
    IntFifo queue;
}
```

$LN[i]$ enthält die durch P_i ausgeführten REQUEST-Operation.

Algorithmus von Suzuki-Kasami

Ausführung eines kritischen Abschnitts

- ▶ Prozess P_i führt den kritischen Abschnitt aus, sobald er über das Token verfügt

Verlassen eines kritischen Abschnitts

- ▶ $LN[i] := RN[i]$
- ▶ Jeder Prozess P_j , der nicht in queue vermerkt ist und für den gilt, dass $RN[j] == LN[j] + 1$ ist, wird an queue angefügt
- ▶ Falls queue nicht leer ist, wird der erste Prozess aus queue entfernt und das Token an ihn verschickt

Algorithmus von Suzuki-Kasami

Bezeichnungen:

- ▶ T : mittlere Nachrichtenlaufzeit; N : Anzahl der Knoten
- ▶ E : mittlere Ausführungszeit für kritischen Abschnitt

Eigenschaften des Algorithmus

- ▶ Antwortzeit: $2T + E$
- ▶ Nachrichten: N pro k. A.
- ▶ Koordinierungsverzögerung: T

Fehlertoleranz

- ▶ Knoten, die selber REQUESTS ausgesendet haben, oder die das Token besitzt, dürfen nicht ausfallen

Weitere tokenbasierte Algorithmen

Algorithmus von Raymond

- ▶ Ziel: Reduktion der Nachrichten-Anzahl
- ▶ Idee: Anordnung der Prozesse als logischem Baum (eine solche Struktur kann z.B. durch einen Wahlalgorithmus automatisch gewonnen werden)

Algorithmus von Singhal

- ▶ Ziel: Beibehaltung der guten Koordinierungsverzögerung von Suzuki-Kasami
- ▶ Aber Reduktion der Nachrichtenzahl durch geschickte Vorhersage, welche Knoten das Token besitzen können

Gegenseitiger Ausschluss

Zusammenfassung

- ▶ Bewertungskriterien für die verschiedenen Algorithmen

