



BP 1	Gegenseitiger Ausschluß: Überblick
4	Gegenseitiger Ausschluß
4.1	Erlaubnisbasierte Algorithmen
4.1.1	Der Algorithmus von Lamport Ein Algorithmus basierend auf vollständiger Replikation des relevanten Systemzustandes
4.1.2	Der Algorithmus von Ricart und Agrawala Reduktion der Nachrichtenzahl
4.1.3	Der Algorithmus von Maekawa <i>Maekawa, M.: A \sqrt{n} Algorithm for Mutual Exclusion in Decentralized Systems. ACM Transactions on Computer Systems, Vol. 3, No. 2, May 1985, pp. 145-159.</i> Verteilte Darstellung des Systemzustandes
4.1.4	Der Algorithmus von Sanders <i>Sanders, B. A.: The Information Structure of Distributed Mutual Exclusion Algorithms. ACM Transactions on Computer Systems, Vol. 5, No. 3, Aug. 1987, pp. 284-299.</i> Universeller erlaubnisbasierter Algorithmus
02.11.01	<small>Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig</small> 4.1-1

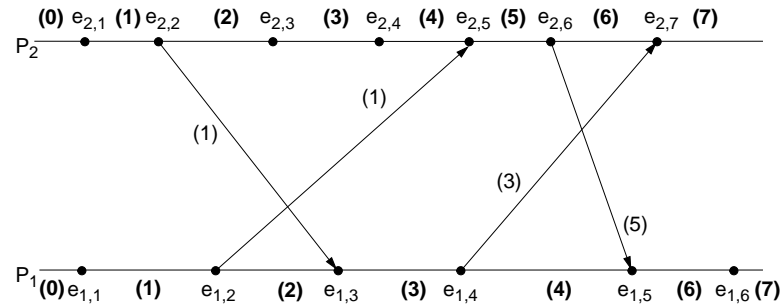
BP 1	Gegenseitiger Ausschluß: Überblick
4.2	Token-basierte Algorithmen
4.2.1	Der Algorithmus von Suzuki und Kasami
4.2.2	Der Algorithmus von Singhal <i>Singhal, M.: A Heuristically-Aided Algorithm for Mutual Exclusion in Distributed Systems. IEEE Transactions on Computers, Vol. 38, No. 5, May 1989, pp. 651-662.</i>
4.2.3	Der Algorithmus von Raymond
02.11.01	<small>Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig</small> 4.2-2

BP 1	Gegenseitiger Ausschluß: Systemmodell, logische Uhren
	Systemmodell <ul style="list-style-type: none"> • Prozesse (Knoten, 'sites') wickeln zeitlich sequentiell eine Folge von Aktionen ab • Es gibt drei Arten von Aktionen: <ul style="list-style-type: none"> - lokale, die keinerlei Auswirkungen auf andere Prozesse haben, - Aktionen zum Senden von Nachrichten, - Aktionen zum Empfangen von Nachrichten.
02.11.01	<small>Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig</small> 4.2-3

BP 1	Gegenseitiger Ausschluß: Systemmodell, logische Uhren
	Logische Uhren nach Lamport <ul style="list-style-type: none"> • Jeder Prozeß verfügt über eine Uhr C_i, die in folgender Weise zählt: <ul style="list-style-type: none"> - Bei Ausführung einer lokalen Aktion und bei Ausführung einer Sende-Aktion durch Prozeß i, wird seine Uhr C_i um 1 erhöht. - Jede Nachricht m trägt als Zeitstempel t_m den Stand der Uhr des Senders mit sich, den sie unmittelbar vor der Sendeaktion hatte. - Nach Ausführung einer Empfangsaktion durch Prozeß i bei Uhrenstand C_i und empfangener Nachricht m wird die lokale Uhr auf $\max(C_i, t_m) + 1$ gestellt. • Ergänzung zu vollständiger Ordnung Zeitstempel bestehend aus Prozessornummer i und lokaler Zeit C_i. Anordnung: $(C_i, i) < (C_k, k)$ genau dann, wenn $C_i < C_k$ oder $C_i = C_k$ und $i < k$.
02.11.01	<small>Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig</small> 4.2-4

BP 1 Gegenseitiger Ausschuß: Systemmodell, logische Uhren

- Beispiel

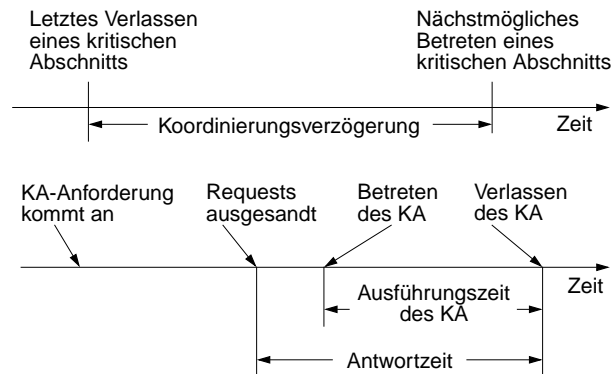


BP 1 Gegenseitiger Ausschuß: Systemmodell, logische Uhren

- Wesentliche Gesichtspunkte
 - Gewährleistung des gegenseitigen Ausschlusses
 - Verklemmungsfrei
 - Aushungerungsfrei
 - D. h. kein Teilnehmer wird auf Dauer am Betreten des k. A. gehindert, während andere Teilnehmer wiederholt ihren k. A. ausführen.
 - Fair
 - D. h. die Zulassung erfolgt in der Reihenfolge der Anforderungen.
 - 'livelock'-frei
 - Fehlertolerant

BP 1 Gegenseitiger Ausschuß: Systemmodell, logische Uhren

- Erzielte Performanz



BP 1 Gegenseitiger Ausschuß: Algorithmus von Lamport

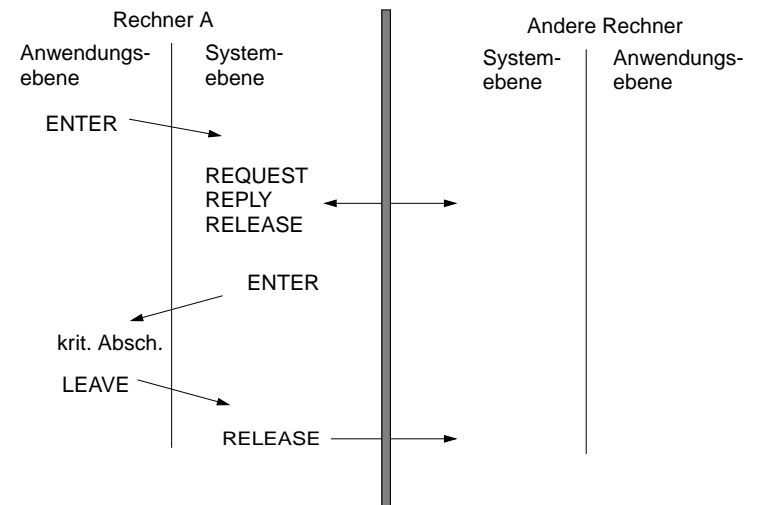
- ### 4.1.1 Der Algorithmus von Lamport
- ◆ Nachrichtenkanal: zuverlässig, FIFO
 - ◆ Eintrittswunsch des Knotens S_i (site i)
 - REQUEST(ts, i) an alle
 - ◆ Verlassen eines kritischen Abschnitts
 - RELEASE(ts, i) an alle
 - ◆ Bearbeitung von REQUESTs:
 - Jeder Prozeß sammelt bei ihm angekommene REQUEST-Nachrichten und ordnet sie sortiert nach steigendem Zeitstempel in lokaler REQUEST-Warteschlange.
 - Wenn S_j REQUEST-Nachricht von S_i erhält, sendet S_j eine REPLY-Nachricht an S_i .

BP 1 Gegenseitiger Ausschuß: Algorithmus von Lamport

- ◆ Betreten kritischer Abschnitte
 - S_i kann k. A. betreten, wenn folgende Bedingungen gemeinsam erfüllt sind:
 - S_i hat von allen anderen eine Nachricht mit Zeitstempel größer (ts, i) erhalten.
 - S_i 's REQUEST ist am Kopf der Warteschlange.
- ◆ Verlassen kritischer Abschnitte
 - S_i entfernt bei Verlassen seines k. A. die eigene Anforderung aus der lokalen Warteschlange und sendet RELEASE-Nachricht an alle Knoten.
 - Wenn S_j eine RELEASE-Nachricht von S_i bekommt, entfernt es die zugehörige REQUEST-Nachricht aus seiner Warteschlange
- ◆ Verhalten
 - T mittlere Nachrichtenlaufzeit; E mittlere Ausführungszeit für k. A.
 - Antwortzeit: $2T + E$
 - Nachrichten: $3(N - 1)$ pro k. A.
 - Koordinierungsverzögerung: T

BP 1 Gegenseitiger Ausschuß: Algorithmus von Lamport

• Implementationsarchitektur



BP 1 Gegenseitiger Ausschuß: Die Ablaufumgebung für die Darstellung der Algorithmen

Die Ablaufumgebung

```
class BasicMessage {
    static final int DATALEN = 10;

    SimProcess source, destination;
    String type;
    double timestamp;
    int creationTime, deliveringTime; // For simulation purposes only

    public BasicMessage(SimProcess f_source,
                        SimProcess f_destination,
                        String f_type,
                        int latency) { ... }

    public void print(String string) { ... }
}
```

BP 1 Gegenseitiger Ausschuß: Die Ablaufumgebung für die Darstellung der Algorithmen

```
public abstract class SimProcess extends Thread {
    Scheduler scheduler;
    int uniqueID = 1;
    int nodeNumber;
    Vector outgoingConnections = new Vector();
    Vector incomingConnections = new Vector();
    Semaphore messageArrived;
    int receiveTurn; // Number of next connection to be searched
                    // for messages
    double time; // Local time

    public SimProcess(String name, Scheduler scheduler,
                      int nodeNumber) {
        ...
        time = nodeNumber / 100.0;
        ...
    }
}
```

BP 1 Gegenseitiger Ausschluß: Die Ablaufumgebung für die Darstellung der Algorithmen

```
public void addConnection(SimProcess destination) { ... }
// Add outgoing channel; at destination add as incoming channel

Message receive() { ... }
// Waiting for any message

public void run () { ... }

public abstract void runProcess();

public double incrementTime() { return time++; }

public double getTime() { return time; }

public void setTime(double f_time) { time = f_time; }
}
```

BP 1 Gegenseitiger Ausschluß: Die Ablaufumgebung für die Darstellung der Algorithmen

```
public class BasicConnection {
    SimProcess source, destination;
    Semaphore empty, full, mutex;
    Semaphore messageArrived = null;
    Vector buffer = new Vector(); // Message buffer

    public BasicConnection(SimProcess f_source,
                           SimProcess f_destination) { ... }

    public BasicConnection(SimProcess f_source,
                           SimProcess f_destination,
                           Semaphore messageArrived) { ... }

    public void send(String type) { ... }

    public BasicMessage receive() { ... }

    public synchronized void xmit(BasicMessage message) { ... }

}
```

BP 1 Gegenseitiger Ausschluß: Der Algorithmus von Lamport

Der Algorithmus von Lamport

```
public class Message extends BasicMessage {
    double requestTimestamp;

    public Message(SimProcess source, SimProcess destination,
                   String type, double requestTimestamp, int latency) {
        super(source, destination, type, latency);
        this.requestTimestamp = requestTimestamp;
    }
}
```

BP 1 Gegenseitiger Ausschluß: Der Algorithmus von Lamport

```
public class Connection extends BasicConnection {

    public Connection(SimProcess source, SimProcess destination) {
        super(source, destination);
    }
    public Connection(SimProcess source, SimProcess destination,
                     Semaphore messageArrived) {
        super(source, destination, messageArrived);
    }
    public void send(String type) {
        Message message
            = new Message(source, destination, type, 0, 2);
        xmit(message);
    }
    public void send(String type, double requestTimestamp) {
        Message message = new Message(source, destination, type,
                                       requestTimestamp, 2);

        xmit(message);
    }
}
```

BP 1 Gegenseitiger Ausschuß: Der Algorithmus von Lamport

```
class QueueEntry {
    String type;           // REQUEST, REPLY, OUTSIDE, INSIDE
    double requestTimestamp;
    double latestTimestamp;

    public QueueEntry() {
        type = "OUTSIDE";
        requestTimestamp = latestTimestamp = -1;
    }

    public QueueEntry(Message packet) {
        type = packet.type;
        requestTimestamp = packet.timestamp;
        latestTimestamp = packet.timestamp;
    }
}
```

BP 1 Gegenseitiger Ausschuß: Der Algorithmus von Lamport

```
public class SystemThread extends SimProcess {
    QueueEntry[] queue = new QueueEntry[Lamport.NODES];

    public SystemThread(String name, Scheduler scheduler,
        int nodeNumber) {
        super(name, scheduler, nodeNumber);
        for (int i = 0; i < Lamport.NODES; i++) {
            queue[i] = new QueueEntry();
        }
    }
}
```

BP 1 Gegenseitiger Ausschuß: Der Algorithmus von Lamport

```
public void runProcess() {
    int i;

    while (true) {
        Message message = receive();
        int sender = message.source.nodeNumber;
        queue[sender].latestTimestamp = message.timestamp;
    }
}
```

BP 1 Gegenseitiger Ausschuß: Der Algorithmus von Lamport

```
if (message.type.equals("ENTER")) {
    for (i = 0; i < Lamport.NODES; i++) {
        ((Connection)(outgoingConnections.elementAt(i)))
            .send("REQUEST", message.timestamp);
    }

} else if (message.type.equals("LEAVE")) {
    for (i = 0; i < Lamport.NODES; i++) {
        ((Connection)(outgoingConnections.elementAt(i)))
            .send("RELEASE");
    }
}
```

BP 1 Gegenseitiger Ausschuß: Der Algorithmus von Lamport

```

    } else if (message.type.equals("REQUEST")) {
        queue[sender] = new QueueEntry(message);
        ((Connection)(outgoingConnections.elementAt(sender)))
            .send("REPLY");

    } else if (message.type.equals("REPLY")) {
        if (check()) {
            ((Connection)(outgoingConnections
                .elementAt(Lamport.NODES)))
                .send("ENTER");
        }
    }

```

BP 1 Gegenseitiger Ausschuß: Der Algorithmus von Lamport

```

    } else if (message.type.equals("RELEASE")) {
        queue[sender].type = "OUTSIDE";
        if (check()) {
            ((Connection)(outgoingConnections
                .elementAt(Lamport.NODES)))
                .send("ENTER");
        }
    } else {
        SystemPrinter.println("Error: unknown message type");
        System.exit(0);
    }
}
}

```

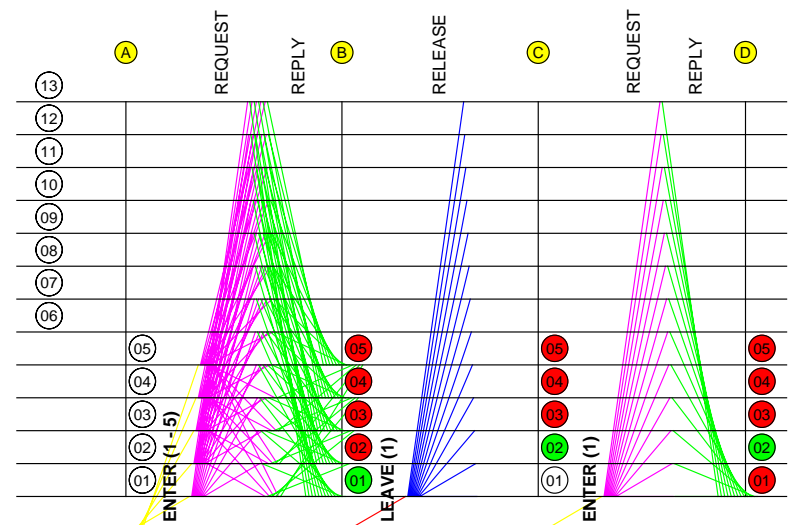
BP 1 Gegenseitiger Ausschuß: Der Algorithmus von Lamport

```

boolean check() {
    // checking if I can enter
    if (queue[nodeNumber].type.equals("REQUEST")) {
        boolean allowed = true;
        for (int i = 0; i < Lamport.NODES; i++) {
            if (queue[i].latestTimestamp
                < queue[nodeNumber].requestTimestamp
                || (queue[i].type.equals("REQUEST")
                    && queue[i].requestTimestamp
                        < queue[nodeNumber].requestTimestamp)) {
                allowed = false; break;
            }
        }
        if (allowed) { queue[nodeNumber].type = "INSIDE"; }
        return allowed;
    } else {
        return false;
    }
}

```

BP 1 Gegenseitiger Ausschuß: Der Algorithmus von Lamport



BP 1	Gegenseitiger Ausschluß: Algorithmus von Ricart und Agrawala
4.1.2	<p>Der Algorithmus von Ricart und Agrawala</p> <ul style="list-style-type: none"> ◆ Nachrichtenkanal: zuverlässig (FIFO-Eigenschaft nicht notwendig!) ◆ Eintrittswunsch des Knotens S_i (site i) <p>REQUEST(ts, i) an alle</p> <ul style="list-style-type: none"> ◆ Bearbeitung von REQUESTs: <ul style="list-style-type: none"> • Wenn S_j eine REQUEST-Nachricht von S_i erhält, sendet er eine REPLY-Nachricht an S_i, falls er <ul style="list-style-type: none"> - weder selbst seinen kritischen Abschnitt betreten will noch ihn ausführt, oder - Knoten S_j seinerseits eine Anforderung gestellt hat und der Zeitstempel von S_i's Anforderung kleiner ist als derjenige der Anforderung von S_j. • In allen anderen Fällen wird das Senden der REPLY-Nachricht aufgeschoben.
02.11.01	<small>Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig</small> 4.2-25

BP 1	Gegenseitiger Ausschluß: Algorithmus von Ricart und Agrawala
	<ul style="list-style-type: none"> ◆ Betreten kritischer Abschnitte S_i kann k. A. betreten, wenn er auf seine Anforderung von allen anderen eine REPLY-Nachricht erhalten hat. ◆ Verlassen des kritischen Abschnitts durch S_i <ul style="list-style-type: none"> • S_i versendet zu allen aufgeschobenen REQUESTs eine REPLY-Nachricht. <p>□ Verhalten T mittlere Nachrichtenlaufzeit; E mittlere Ausführungszeit für k. A.</p> <ul style="list-style-type: none"> • Antwortzeit: $2T + E$ • Nachrichten: $2(N - 1)$ pro k. A. • Koordinierungsverzögerung: T
02.11.01	<small>Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig</small> 4.2-26

BP 1	Gegenseitiger Ausschluß: Algorithmus von Ricart und Agrawala
	<p>Der Algorithmus von Ricart und Agrawala</p> <pre> public class Message extends BasicMessage { double requestTimestamp; public Message(SimProcess source, SimProcess destination, String type, double requestTimestamp, int latency) { super(source, destination, type, latency); this.requestTimestamp = requestTimestamp; } } </pre>
02.11.01	<small>Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig</small> 4.2-27

BP 1	Gegenseitiger Ausschluß: Algorithmus von Ricart und Agrawala
	<pre> public class Connection extends BasicConnection { public Connection(SimProcess source, SimProcess destination) { super(source, destination); } public Connection(SimProcess source, SimProcess destination, Semaphore messageArrived) { super(source, destination, messageArrived); } public void send(String type) { Message message = new Message(source, destination, type, 0, 2); xmit(message); } public void send(String type, double requestTimestamp) { Message message = new Message(source, destination, type, requestTimestamp, 2); xmit(message); } } </pre>
02.11.01	<small>Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig</small> 4.2-28

BP 1 Gegenseitiger Ausschuß: Algorithmus von Ricart und Agrawala

```
public class SystemThread extends SimProcess {
    String state; // OUTSIDE, INSIDE, REQUEST
    double requestTimestamp;
    boolean[] deferredReply;
    int numberOfReplies;

    public SystemThread(String name, Scheduler scheduler,
        int nodeNumber) {
        super(name, scheduler, nodeNumber);
        deferredReply
            = new boolean[scheduler.system.getNumberOfProcesses()];
        requestTimestamp = -1;
        for (int i=0; i<scheduler.system.getNumberOfProcesses(); i++)
            deferredReply[i] = false;
        state = "OUTSIDE";
    }
}
```

BP 1 Gegenseitiger Ausschuß: Algorithmus von Ricart und Agrawala

```
public void runProcess() {
    int i;

    while (true) {
        Message message = receive();
        int sender = message.source.nodeNumber;
```

BP 1 Gegenseitiger Ausschuß: Algorithmus von Ricart und Agrawala

```
    if (message.type.equals("ENTER")) {
        state = "REQUEST";
        requestTimestamp = message.timestamp;
        for (i=0; i<scheduler.system.getNumberOfProcesses(); i++){
            ((Connection)(outgoingConnections.elementAt(i)))
                .send("REQUEST", message.timestamp);
        }
        numberOfReplies = 0;

    } else if (message.type.equals("LEAVE")) {
        state = "OUTSIDE";
        for (i=0; i<scheduler.system.getNumberOfProcesses(); i++){
            if (deferredReply[i])
                ((Connection)(outgoingConnections.elementAt(i)))
                    .send("REPLY");
        }
    }
```

BP 1 Gegenseitiger Ausschuß: Algorithmus von Ricart und Agrawala

```
    } else if (message.type.equals("REQUEST")) {
        if (state.equals("OUTSIDE")
            || (state.equals("REQUEST")
                && message.requestTimestamp<=requestTimestamp)) {
            ((Connection)(outgoingConnections.elementAt(sender)))
                .send("REPLY");
        } else {
            deferredReply[sender] = true;
        }

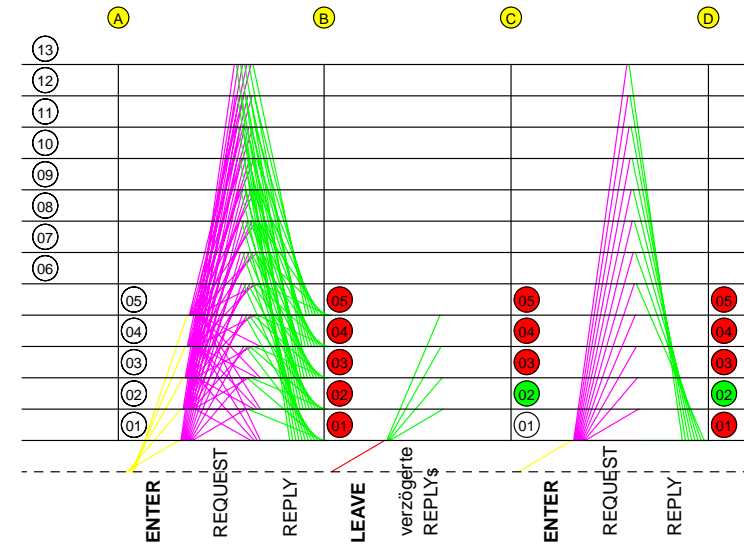
    } else if (message.type.equals("REPLY")) {
        numberOfReplies++;
        check();

    } else {
        SystemPrinter.println("Error: Unknown message type");
        System.exit(0);
    }
}
```


BP 1 Gegenseitiger Ausschuß: Algorithmus von Ricart und Agrawala

```
void check() {
// checking if I can enter
if (state.equals("REQUEST")) {
    if (numberOfReplies
        == scheduler.system.getNumberOfProcesses()) {
        state = "INSIDE";
        scheduler.sleep(10000);
        ((Connection)(outgoingConnections.elementAt(nodeNumber)))
            .send("LEAVE");
    }
}
}
```

BP 1 Gegenseitiger Ausschuß: Algorithmus von Ricart und Agrawala



BP 1 Gegenseitiger Ausschuß: Algorithmus von Maekawa

4.1.3 Algorithmus Maekawa

- **Vorstellung:** Die Verantwortung soll auf alle möglichst gleichmäßig verteilt werden, wobei jeder nur soviel Verantwortung wie unbedingt nötig tragen soll.
- **Lösungsidee:** Bildung von Quorum-Mengen, wobei jeder Beteiligte zum Eintritt in den kritischen Abschnitt die Zustimmung seines Quorums benötigt.

◆ Bedingungen an Quorum-Mengen

Jedem Prozeß i wird ein Quorum S_i ($1 \leq i \leq n$) zugeordnet derart, daß folgende Eigenschaften erfüllt sind:

1. $\forall i, j (1 \leq i < j \leq n \Rightarrow (S_i \cap S_j \neq \emptyset))$
2. $\forall i (1 \leq i \leq n \Rightarrow i \in S_i)$
3. $\forall i (1 \leq i \leq n \Rightarrow (|S_i| = \text{Konstante } 1))$
4. $\forall i (1 \leq i \leq n \Rightarrow (|\{j | (i \in S_j)\}| = \text{Konstante } 2))$

BP 1 Gegenseitiger Ausschuß: Algorithmus von Maekawa

□ Satz (aus der endlichen projektiven Geometrie)

Eine Quorum-Menge, die obige Bedingungen erfüllt, existiert, wenn sich 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 Bedingungen (3) und (4) gelockert werden.

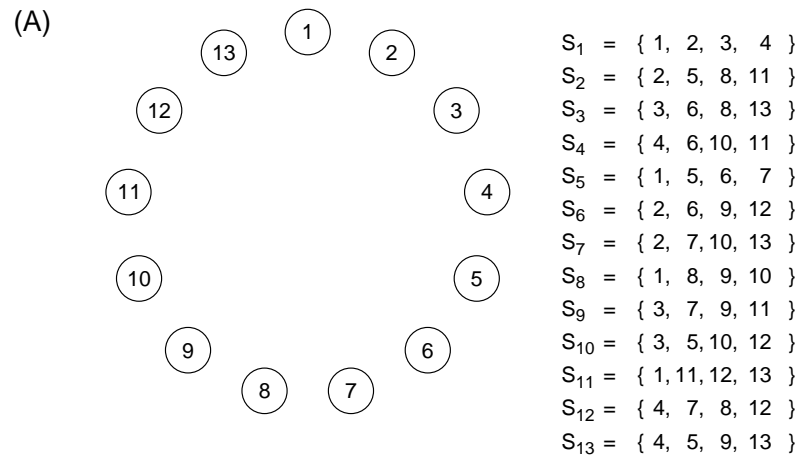
BP 1	Gegenseitiger Ausschuß: Algorithmus von Maekawa
	<ul style="list-style-type: none"> ◆ Nachrichtenformat: (source, target, timestamp, message) ◆ Übergangsfunktion <ol style="list-style-type: none"> 1. Zugang anfordern durch Knoten i: $\text{send}(i, s, T, \text{REQUEST}), \text{ an alle } s \in S_i$ 2. Bei Empfang von Anforderung $r = (i, s, T', \text{REQUEST})$ durch s: <div style="margin-left: 40px;"> <i>frei:</i> r in Anforderungsmenge aufnehmen <i>belegt für r</i> notieren; $\text{send}(s, i, T, \text{LOCKED});$ </div> <div style="margin-left: 40px;"> <i>belegt für r':</i> r in Anforderungsmenge aufnehmen; r nicht ältestes Element der Anf.-Menge: $\text{send}(s, i, T, \text{FAILED});$ r ältestes Element der Anf.-Menge: $\text{send}(s, r'.\text{source}, T, \text{INQUIRE}),$ falls noch nicht geschehen; </div>
02.11.01	Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig 4.2-37

BP 1	Gegenseitiger Ausschuß: Algorithmus von Maekawa
	<ol style="list-style-type: none"> 3. Bei Empfang von $(s, i, T', \text{INQUIRE})$ durch i: <div style="margin-left: 40px;"> Eine der letzten Anforderungen von i wurde von einem Quorums-Mitglied mit FAILED beantwortet: $\text{send}(i, s, T, \text{RELINQUISH});$ $\text{LOCKED}(s)$ aus der Erlaubnismenge entfernen; </div> <div style="margin-left: 40px;"> sonst: ignorieren, aber vormerken!!!!; </div> 4. Bei Empfang von RELINQUISH: <div style="margin-left: 40px;"> Bestehende Belegung (<i>belegt für r</i>) aufheben; Für älteste Anforderung r' aus der Anforderungsmenge <i>belegt für r'</i> notieren; $\text{send}(s, r'.\text{source}, T, \text{LOCKED});$ </div>
02.11.01	Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig 4.2-38

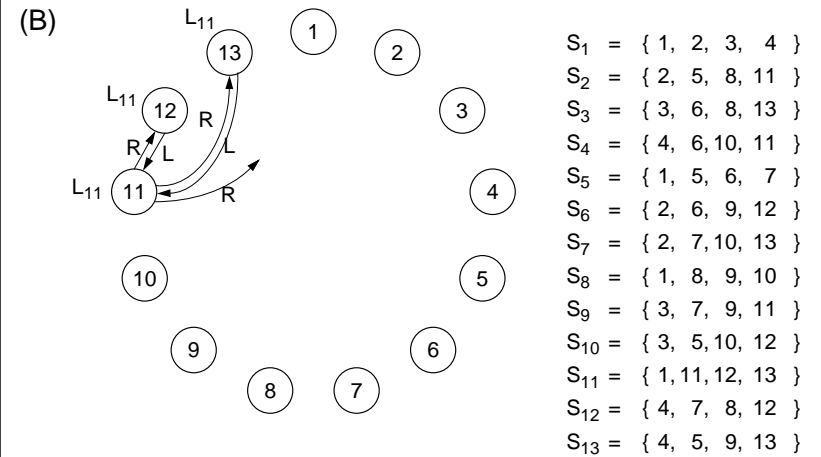
BP 1	Gegenseitiger Ausschuß: Algorithmus von Maekawa
	<ol style="list-style-type: none"> 5. Bei Empfang von $(s, i, T', \text{LOCKED})$ <div style="margin-left: 40px;"> $\text{LOCKED}(s)$ in die Erlaubnismenge aufnehmen Von allen Mitgliedern des Quorums Erlaubnis (LOCKED) erhalten: Kritischen Abschnitt bearbeiten; </div> 6. Beenden des kritischen Abschnitts <div style="margin-left: 40px;"> Erlaubnismenge leeren $\text{send}(i, s, T, \text{RELEASE})$ für alle $s \in S_i$ </div>
02.11.01	Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig 4.2-39

BP 1	Gegenseitiger Ausschuß: Algorithmus von Maekawa
	<ol style="list-style-type: none"> 7. Bei Empfang von RELEASE: <div style="margin-left: 40px;"> belegt für r: r aus Anforderungsmenge entfernen Anforderungsmenge enthält neben r weitere Anforderungen: Für älteste Anforderung $r' \neq r$ aus der Anforderungsmenge <i>belegt für r'</i> notieren; $\text{send}(s, r'.\text{source}, T, \text{LOCKED});$ </div> 8. Bei Empfang von $(s, i, T', \text{FAILED})$ durch i <div style="margin-left: 40px;"> Eine der letzten Anforderungen von i wurde vom Quorums-Mitglied t mit INQUIRE beantwortet: $\text{send}(i, t, T, \text{RELINQUISH});$ $\text{LOCKED}(t)$ aus der Erlaubnismenge entfernen; </div> <div style="margin-left: 40px;"> sonst: ignorieren, aber vormerken!!!! </div>
02.11.01	Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig 4.2-40

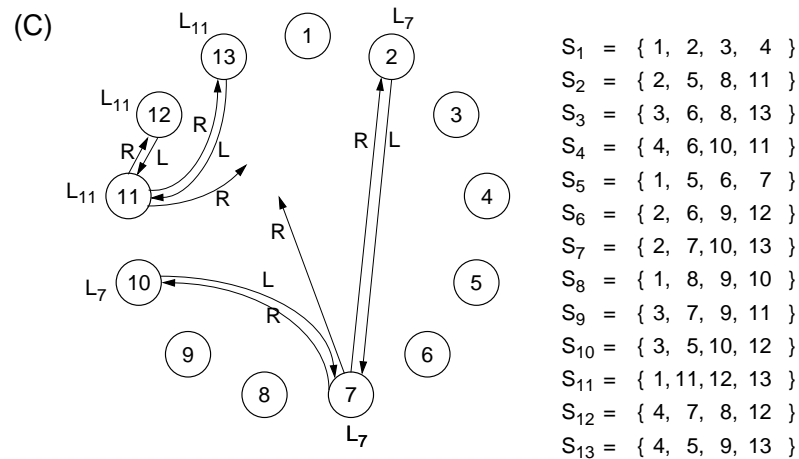
BP 1 Gegenseitiger Ausschuß: Algorithmus von Maekawa



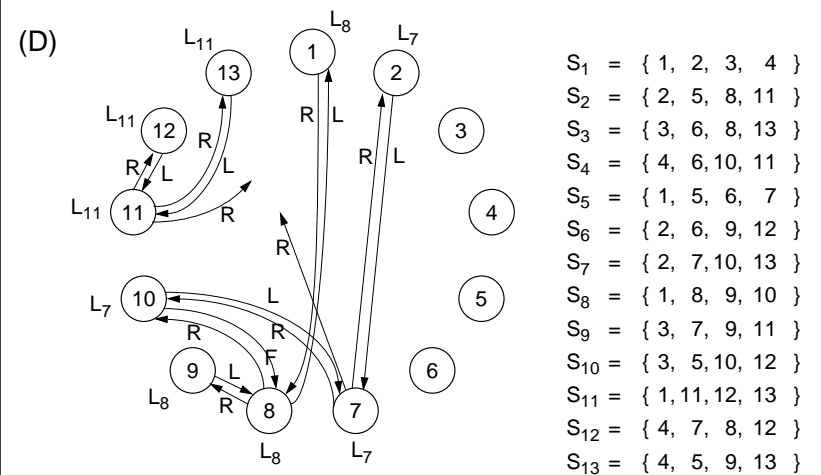
BP 1 Gegenseitiger Ausschuß: Algorithmus von Maekawa



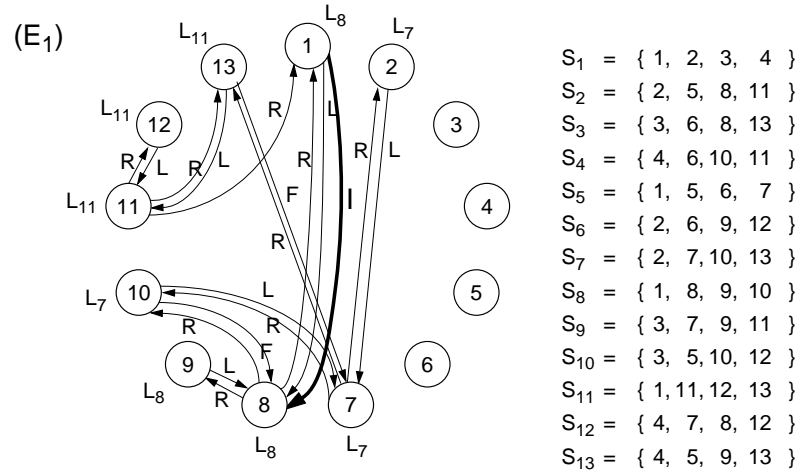
BP 1 Gegenseitiger Ausschuß: Algorithmus von Maekawa



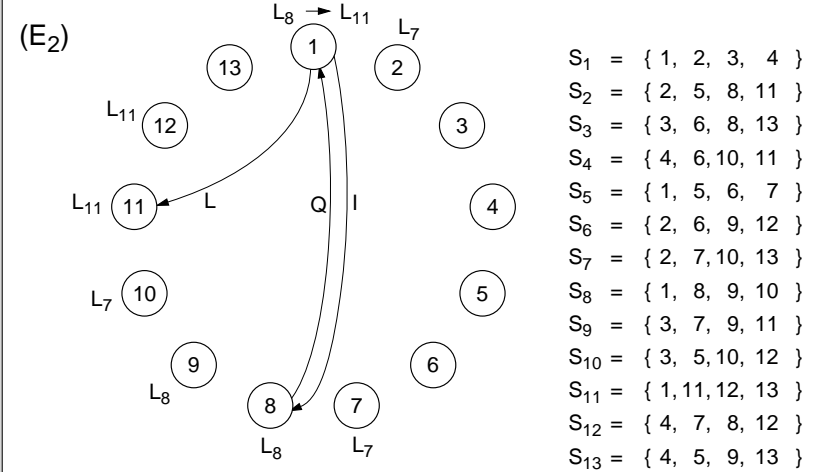
BP 1 Gegenseitiger Ausschuß: Algorithmus von Maekawa



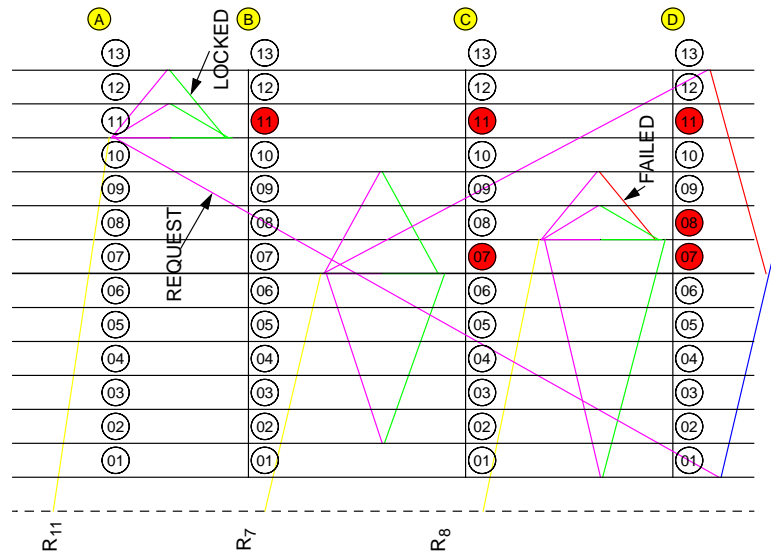
BP 1 Gegenseitiger Ausschuß: Algorithmus von Maekawa



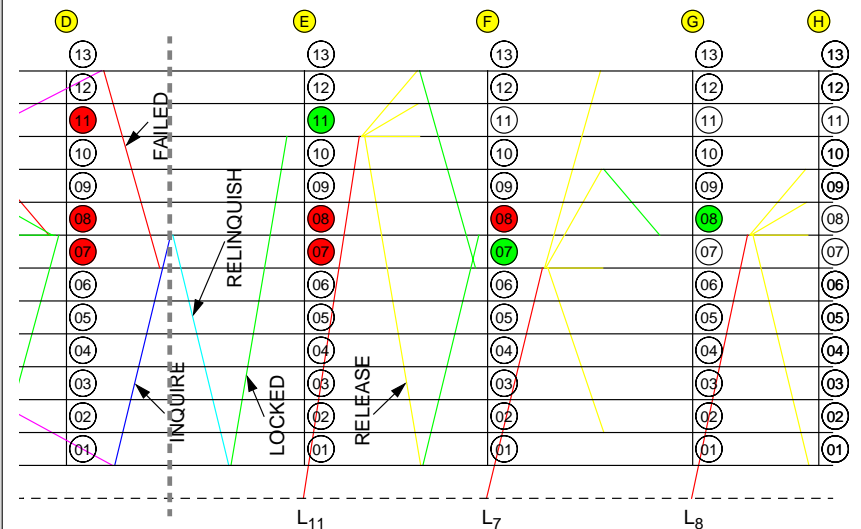
BP 1 Gegenseitiger Ausschuß: Algorithmus von Maekawa



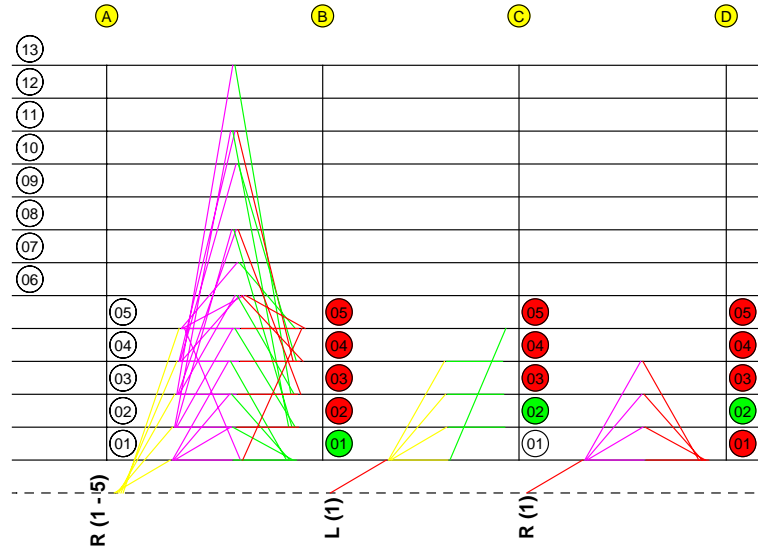
BP 1 Gegenseitiger Ausschuß: Algorithmus von Maekawa



BP 1 Gegenseitiger Ausschuß: Algorithmus von Maekawa



BP 1 Gegenseitiger Ausschuß: Algorithmus von Maekawa



BP 1 Gegenseitiger Ausschuß: Algorithmus von Sanders

4.1.4 Verallgemeinerung: Algorithmus von Sanders

- R_i Menge der Prozesse, bei denen Prozeß i Erlaubnis einholen muß, wenn er seinen kritischen Abschnitt betreten will.
- I_i Menge der Prozesse, die informiert werden müssen, wenn Prozeß i seinen kritischen Abschnitt verläßt.
- $St_i = \{ j \mid i \in I_j \}$ Menge aller Prozesse, die Prozeß i bei Verlassen ihres kritischen Abschnitts informieren müssen.

S4.1 Satz

Wenn $\forall i (1 \leq i \leq N \Rightarrow i \in I_i)$ gilt, dann genügen die beiden folgenden Bedingungen zur Sicherstellung des gegenseitigen Ausschlusses:

- $\forall i (1 \leq i \leq N \Rightarrow I_i \subseteq R_i)$
- $\forall i, j (1 \leq i, j \leq N \Rightarrow (I_i \cap I_j \neq \emptyset) \vee (i \in R_j \wedge j \in R_i))$

BP 1 Gegenseitiger Ausschuß: Algorithmus von Sanders

Algorithmus

◆ Zugang anfordern durch Prozeß i

send($i, j, T, REQUEST$) an alle $j \in R_i$
warten_auf_LOCKED von allen $j \in R_i$

◆ Verlassen des kritischen Abschnitts durch Prozeß i

send($i, j, -, RELEASE$) an alle $j \in I_i$

◆ Empfang von $r = (i, s, T, REQUEST)$

r in Anforderungsmenge aufnehmen;
status == frei:
send($s, r'.source, -, LOCKED$)
für älteste bestehende Anforderung r'
status = belegt(r'), falls $r'.source \in St_s$
 r' aus der Anforderungsmenge entfernen

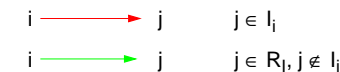
BP 1 Gegenseitiger Ausschuß: Algorithmus von Sanders

◆ Empfang von $r = (r, s, -, RELEASE)$

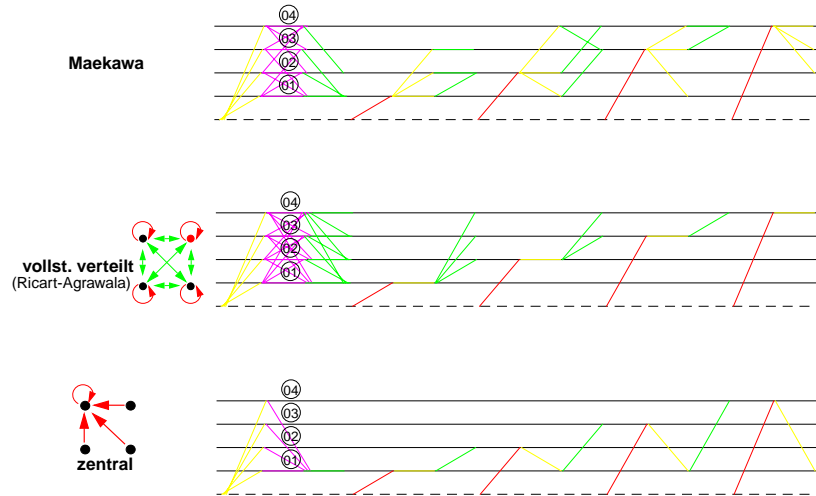
status = frei
solange status == frei und Anforderungsmenge != \emptyset
send($s, r'.source, -, LOCKED$)
für älteste bestehende Anforderung r'
status = belegt(r'), falls $r'.source \in St_s$
 r' aus der Anforderungsmenge entfernen

Nicht verklemmungsfrei! Lösung analog Maekawa

◆ Graphische Darstellung



BP 1 Gegenseitiger Ausschuß: Algorithmus von Sanders



BP 1 Gegenseitiger Ausschuß: Algorithmus von Sanders

Zahl der Nachrichten pro kritischem Abschnitt

NM_i sei die Zahl der Nachrichten pro kritischem Abschnitt des Prozesses P_i . Dann ist

$$|I_i - \{i\}| + 2(|R_i - \{i\}|) \leq NM_i$$

Der erste Summand ist die Zahl der Informationsnachrichten beim Verlassen eines k. A., der zweite die der Request- und Locked-Nachrichten.

Weiter ist

$$NM_i \leq |I_i - \{i\}| + 2(|R_i - \{i\}|) + DM \text{ mit}$$

$$DM = \sum_{j \in R_i} r_j \text{ und } r_j = \begin{cases} 1 & \text{falls } St_j = \{j\} \text{ oder } = \{i, j\} \text{ (FAIL)} \\ 4 & \text{sonst (INQUIRE, RELIN., LOCKED, FAIL)} \end{cases}$$

BP 1 Gegenseitiger Ausschuß: Algorithmus von Sanders

Spezialisierungen

Lamport

$$I_i = R_i = \text{alle}, |I_i| = |R_i| = N$$

$$3(N-1) \leq NM_i \leq 3(N-1) + 4N$$

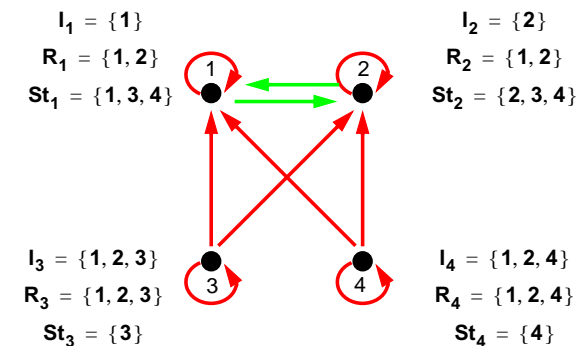
Ricart-Agrawala

$$I_i = \{i\}, R_i = \text{alle}$$

$$2(N-1) \leq NM_i \leq 2(N-1) + N$$

BP 1 Gegenseitiger Ausschuß: Algorithmus von Sanders

Ein neuer Algorithmus



$x \xrightarrow{\text{green}} y$ x muß y fragen, aber nicht informieren
 $x \xrightarrow{\text{red}} y$ x muß y fragen und informieren

BP 1

Gegenseitiger Ausschuß: Algorithmus von Sanders

♦ Zahl auszutauschender Nachrichten

$$2 \leq NM_i \leq 2 + 8 \text{ für } i \in \{1, 2\}$$

$$6 \leq NM_i < 6 + 9 \text{ für } i \in \{3, 4\}$$

♦ Koordinierungsverzögerung

$$D_{i,j} = \begin{cases} 2 & \text{falls } I_i \cap I_j \neq \emptyset \\ 0 & \text{falls } i = j \\ 1 & \text{sonst} \end{cases}$$

i \ j	1	2	3	4
1	0	1	2	2
2	1	0	2	2
3	2	2	0	2
4	2	2	2	0

02.11.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
 Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig

4.2-57

BP 1

Gegenseitiger Ausschuß: Token-basierte Algorithmen

4.2 Token-basierte Algorithmen

□ Prinzip

- Im System existiert genau ein Token
- Ein Prozeß kann einen kritischen Abschnitt nur betreten, wenn er über das Token verfügt.
-
- 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
- Erkennung veralteter REQUEST-Anforderungen
- Bestimmung des Prozesses, dem das Token übergeben werden soll.

02.11.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
 Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig

4.2-58

BP 1

Gegenseitiger Ausschuß: Algorithmus von Suzuki-Kasami

4.2.1 Algorithmus von Suzuki-Kasami für n kooperierende Prozesse P_i ($0 \leq i < n$)

□ Datenstrukturen

- Lokale Datenstruktur des Prozesses P_i

```
int RN[n] // Sequenznummern, soweit bekannt
```
- Datenstruktur des Tokens


```
struct {int LN[n]; int_fifo queue(n); }
```

LN[i] enthält die Sequenznummer der zuletzt durch P_i ausgeführten REQUEST-Operation.

□ 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.

02.11.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
 Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig

4.2-59

BP 1

Gegenseitiger Ausschuß: 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

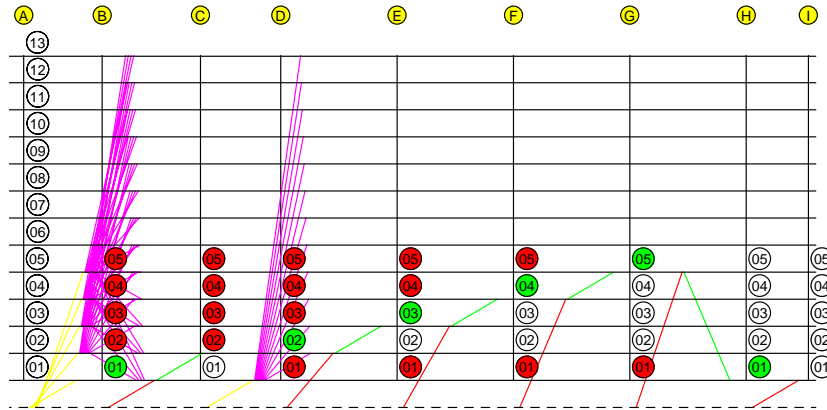
- $LN[i] = RN[i]$
- Jeder Prozeß P_j , der nicht in queue vermerkt ist und für den $RN[j] == LN[j] + 1$ ist, wird an queue angefügt.
- Falls queue nicht leer ist, wird der erste Prozeß aus queue entfernt und das Token an ihn verschickt.

02.11.01

Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann
 Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig

4.2-60

BP 1 Gegenseitiger Ausschuß: Algorithmus von Suzuki-Kasami



BP 1 Gegenseitiger Ausschuß: Algorithmus von Singhal

4.2.2 Algorithmus von Singhal



Datenstrukturen

• Lokale Datenstruktur des Prozesses P_i

Zustände:

- R Prozeß will kritischen Abschnitt betreten (requesting)
- E Prozeß ist in kritischem Abschnitt (executing)
- H Prozeß hält unbenutztes Token (holding free token)
- N Prozeß ist unbeteiligt (none of the above)

```
/* 1 */ enum t_state {R, E, H, N};
/* 1 */ struct { t_state ST; // state
/* 2 */ int SN; // sequence number
/* 3 */ } P[MAX_TEILNEHMERZAHL + 1]; // processes are numbered from 1(!) to
// MAX_TEILNEHMERZAHL
```

BP 1 Gegenseitiger Ausschuß: Algorithmus von Singhal



Datenstruktur des Tokens

```
/* 4 */ struct {
/* 5 */ struct {
/* 6 */ t_state ST = N;
/* 7 */ int SN = 0;
/* 8 */ } P[MAX_TEILNEHMERZAHL + 1];
/* 9 */ } Token;
```

Anfangsbesetzung der lokalen Zustände

n	N	N	N	N		N	N	N
n-1	R	N	N	N			N	N
n-2	R	R	N	N			N	N
n-3	R	R	R	N				
2	R	R	R	R		R		
1	R	R	R	R		R	R	H
	P_n	P_{n-1}				P_3	P_2	P_1

BP 1 Gegenseitiger Ausschuß: Algorithmus von Singhal



Anforderung des Tokens durch Prozeß P_i

```
/*10*/ P[i].SN++;
/*11*/ if (P[i].ST != H) {
/*12*/ P[i].ST = R;
/*13*/ for (j = 1; j <= teilnehmerzahl; j++) {
/*14*/ if (P[j].ST == R && j != i) {
/*15*/ send(j, // Empfaenger
/*16*/ REQUEST, // Nachrichtentyp
/*17*/ i, // Sender
/*18*/ P[i].SN) // Daten
/*19*/ };
/*20*/ }
/*21*/ }
/*22*/ wait_for_token();
/*23*/ }
```



Empfang einer REQUEST-Nachricht (i, REQUEST, j, sn)

i Adressat

REQUEST Nachrichtentyp

j Absender

sn Sequenznummer

BP 1	Gegenseitiger Ausschuß: Algorithmus von Singhal
<pre> /*24*/ if (P[j].SN < sn){ /*25*/ switch (P[i].ST) { /*26*/ /*(8)*/ case N: P[j].ST = R; /*27*/ P[j].SN = sn; /*28*/ break; /*29*/ /*(9)*/ case R: if (P[j].ST != R) { /*30*/ P[j].ST = R; /*31*/ send(j, REQUEST, i, P[i].SN); /*32*/ } /*33*/ P[j].SN = sn; /*34*/ break; /*35*/ /*(10)*/ case E: P[j].ST = R; /*36*/ P[j].SN = sn; /*37*/ break; /*38*/ /*(11)*/ case H: P[j].ST = R; /*39*/ P[j].SN = sn; /*40*/ T.P[j].ST = R; /*41*/ T.P[j].SN = sn; /*42*/ P[i].ST = N; /*43*/ send(j, TOKEN, i, token); /*44*/ break; /*45*/ } /*46*/ } </pre>	
02.11.01	Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig 4.2-65

BP 1	Gegenseitiger Ausschuß: Algorithmus von Singhal
<div> <div></div> <div>Empfang des Tokens (i, TOKEN, j, token)</div> </div> <pre> /*47*/ /*(2)*/ P[i].ST = E; /*48*/ execute_critical_section(); </pre> <div> <div></div> <div>Verlassen eines kritischen Abschnitts</div> </div> <pre> /*49*/ /*(3)*/ int tmp; /*50*/ P[i].ST = N; // wird fuer update benötigt /*51*/ T.P[i].ST = N; /*52*/ T.P[i].SN = P[i].SN; /*53*/ update_ST_vectors(); // Abgleich lokaler Zustandsvektor - Token /*54*/ P[i].ST = H; // tatsächlicher Zustand /*55*/ /*56*/ // Arbitrierung nach round robin, /*57*/ // koennte auch anders erfolgen. /*58*/ /*59*/ for (tmp = 1; tmp <= teilnehmerzahl; tmp++) { /*60*/ int j = ((i+tmp-1) % teilnehmerzahl) + 1; /*61*/ if (P[j].ST == R) { /*62*/ P[i].ST = N; /*63*/ send(j, TOKEN, i, token); /*64*/ break; /*65*/ } /*66*/ } </pre>	
02.11.01	Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig 4.2-66

BP 1	Gegenseitiger Ausschuß: Algorithmus von Singhal
<pre> /*67*/ update_ST_vectors() { /*68*/ for (tmp = 1; tmp <= teilnehmerzahl; tmp++) { /*69*/ /*70*/ /*(4)*/ if (P[tmp].ST == R && T.P[tmp].ST == N) { /*71*/ /*72*/ if (P[tmp].SN == T.P[tmp].SN) /*73*/ // Token hat die neuere Information /*74*/ P[tmp].ST = N; /*75*/ /*76*/ else if (P[tmp].SN > T.P[tmp].SN) { /*77*/ // Prozess hat die neuere Information /*78*/ T.P[tmp].ST = R; /*79*/ T.P[tmp].SN = P[tmp].SN; /*80*/ } else /* if (P[tmp].SN < T.P[tmp].SN) */ { /*81*/ // Token hat die neuere Information /*82*/ P[tmp].ST = N; /*83*/ P[tmp].SN = T.P[tmp].SN; /*84*/ } </pre>	
02.11.01	Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig 4.2-67

BP 1	Gegenseitiger Ausschuß: Algorithmus von Singhal
<pre> /*85*/ } else if (P[tmp].ST == N && T.P[tmp].ST == R) { /*86*/ /*(5)*/ if (P[tmp].SN == T.P[tmp].SN) /*87*/ cout << "\n***** error 1\n" << flush; /*88*/ else if (P[tmp].SN > T.P[tmp].SN) /*89*/ cout << "\n***** error 2\n" << flush; /*90*/ else /* if (P[tmp].SN < T.P[tmp].SN) */{ /*91*/ // Token hat die neuere Information /*92*/ P[tmp].ST = R; /*93*/ P[tmp].SN = T.P[tmp].SN; /*94*/ } /*95*/ } else if (P[tmp].ST == N && T.P[tmp].ST == N){ /*96*/ /*(6)*/ if (P[tmp].SN == T.P[tmp].SN) ; /*97*/ // Token und Prozess haben gleiche Information /*98*/ else if (P[tmp].SN > T.P[tmp].SN) /*99*/ cout << "\n***** error 3\n"; /*100*/ else /* if (P[tmp].SN < T.P[tmp].SN) */ /*101*/ // Token hat die neuere Information P[tmp].SN = T.P[tmp].SN; </pre>	
02.11.01	Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig 4.2-68

BP 1 Gegenseitiger Ausschuß: Algorithmus von Singhal

```

/*102*/      } else /* if (P[tmp].ST == R && T.P[tmp].ST == R) */ {
/*103*/      /*(7)*/      if (P[tmp].SN == T.P[tmp].SN) ;
/*104*/      else if (P[tmp].SN > T.P[tmp].SN)
/*105*/      // Prozess hat die neuere Information
/*106*/      T.P[tmp].SN = P[tmp].SN;
/*107*/      else /* if (P[tmp].SN < T.P[tmp].SN) */
/*108*/      // Token hat die neuere Information
/*109*/      P[tmp].SN = T.P[tmp].SN;
/*110*/      }
/*111*/  }
/*112*/  }

```

BP 1 Gegenseitiger Ausschuß: Algorithmus von Singhal

◆ Komprimierbar zu

```

/*113*/ update_ST_vectors() {
/*114*/     for (tmp = 1; tmp <= teilnehmerzahl; tmp++) {
/*115*/         if (P[tmp].SN > T.P[tmp].SN) {
/*116*/             // Prozess hat neuere Information
/*117*/             T.P[tmp].ST = P[tmp].ST;
/*118*/             T.P[tmp].SN = P[tmp].SN;
/*119*/         } else {
/*120*/             // Token hat die neuere Information
/*121*/             P[tmp].ST = T.P[tmp].ST;
/*122*/             P[tmp].SN = T.P[tmp].SN;
/*123*/         }
/*124*/     }
/*125*/ }

```

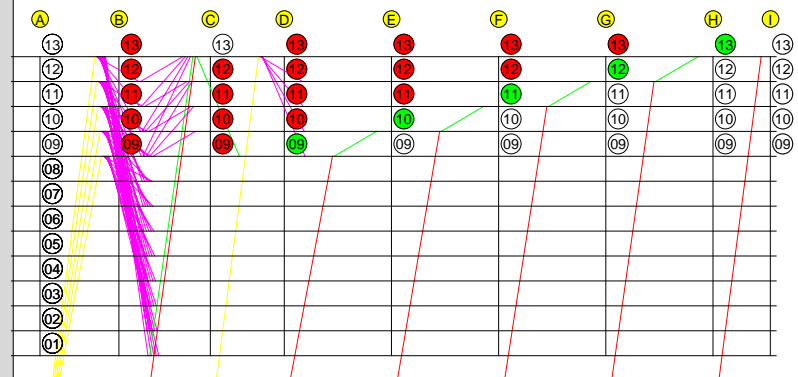
BP 1 Gegenseitiger Ausschuß: Algorithmus von Singhal



Arbitrierung zur Erzielung von Fairneß

- Auswahl nach kleinster Sequenznummer
(zählt, wie oft der anfordernde Prozeß bislang den kritischen Abschnitt ausführte)
oder
- Auswahl der Anforderung des nächstgelegenen Prozesses

BP 1 Gegenseitiger Ausschuß: Algorithmus von Singhal



BP 1	Gegenseitiger Ausschluß: Algorithmus von Singhal
	<p>□ Verifikation</p> <p>◆ Gegenseitiger Ausschluß</p> <p>Trivial, da ein Prozeß seinen kritischen Abschnitt nur dann betritt, wenn er über das Token verfügt und es erst nach Abschluß seines kritischen Abschnitts weitergibt.</p>
02.11.01	<small>Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig</small>

4.2-73

BP 1	Gegenseitiger Ausschluß: Algorithmus von Singhal
	<p>◆ Verklemmungsfreiheit</p> <p>(Partielle) Verklemmung kann nur vorliegen, wenn die Prozesse in zwei Klassen zerfallen, so daß Prozesse einer Klasse keine Anforderungen an die der anderen richten und in beiden Anforderungen vorliegen.</p> <p>Definitionen</p> <p>P_i bezeichne (auch) den Zustandsvektor des i-ten Prozesses</p> <p>$R_i^j = \text{TRUE}$, genau dann wenn $P_i[j].ST = R$ (d. h. P_i muß Anforderung an P_j senden)</p> <p>$N_i^j = \text{TRUE}$, genau dann wenn $P_i[j].ST = N$ (d. h. P_i sendet Anforderung nicht an P_j)</p> <p>P_i ist die Zahl der N-Einträge im Zustandsvektor des i-ten Prozesses</p> <p>NP ist die Menge der Prozesse, die seit Systembeginn noch keine Anforderung gestellt haben</p> <p>EP ist die Menge der Prozesse, die seit Systembeginn wenigstens einmal ihren kritischen Abschnitt durchlaufen haben</p>
02.11.01	<small>Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig</small>

4.2-74

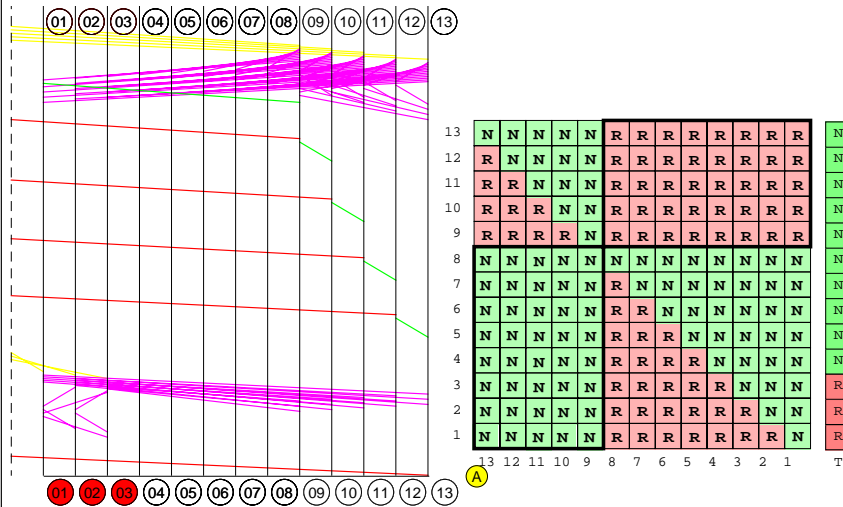
BP 1	Gegenseitiger Ausschluß: Algorithmus von Singhal
	<p>Im Anfangszustand gilt:</p> <p>R_i^j ist erfüllt, für $1 \leq i \leq n$ und $1 \leq j < i$</p> <p>N_i^j ist erfüllt für $1 < i \leq n$ und $i \leq j \leq n$</p> <p>N_i^j ist erfüllt für $1 < j \leq n$</p> <p>Eine Sequenz P_n, P_{n-1}, \dots, P_1 ist "vollständig geordnet" wenn gilt:</p> <p>$P_n < P_{n-1} < \dots < P_1$</p> <p>L4.1.1 Lemma</p> <p>Das Token hat niemals falsche R-Einträge, d. h. für alle j gilt: $T.P[j].SN == 'R' \Rightarrow P[j].SN == 'R'$.</p> <p>Beweis: Induktion über die Zustandsänderungen.</p>
02.11.01	<small>Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig</small>

4.2-75

BP 1	Gegenseitiger Ausschluß: Algorithmus von Singhal
	<p>L4.1.2 Lemma</p> <p>Wenn</p> <ul style="list-style-type: none"> das System im Anfangszustand startet, alle Prozesse in EP genau einmal einen kritischen Abschnitt durchlaufen haben und alle REQUEST-Nachrichten angekommen und bearbeitet sind, <p>dann gilt:</p> <p>$\forall P_i, P_j (P_i \in EP \wedge P_j \in NP \Rightarrow R_i^j)$</p> <p>und</p> <p>$\forall P_i, P_j (P_i \in EP \wedge P_j \in NP \Rightarrow N_i^j)$</p>
02.11.01	<small>Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig</small>

4.2-76

BP 1 Gegenseitiger Ausschuß: Algorithmus von Singhal



BP 1 Gegenseitiger Ausschuß: Algorithmus von Singhal

L4.1.3 Lemma

Wenn

- das System im Anfangszustand startet,
- $P_{i1}, P_{i2}, \dots, P_{im}$ das Token anfordern,
- P_{ie} im kritischen Abschnitt ist und
- alle gesandten REQUEST-Nachrichten angekommen und bearbeitet sind,

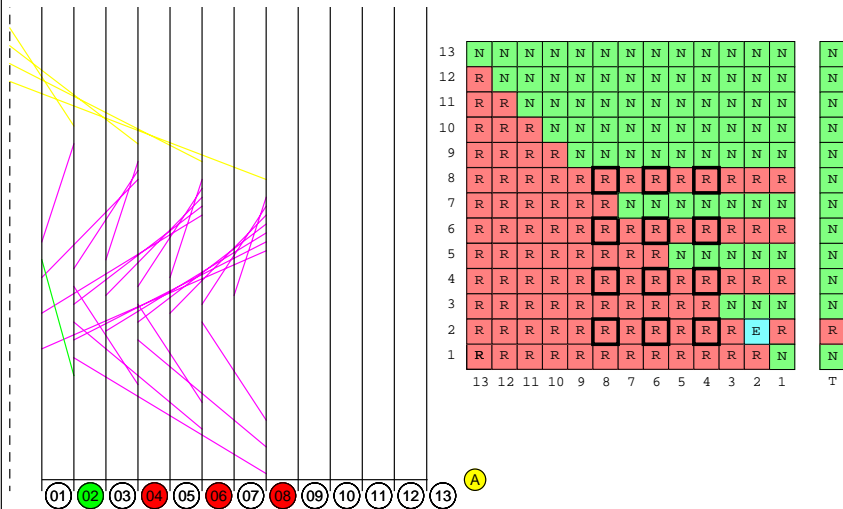
dann gilt:

$$\forall i, j (i \in \{i1, i2, \dots, im\} \wedge j \in \{i1, i2, \dots, im, ie\} \Rightarrow P_i[j] = R)$$

und

$$\forall j ((j \in \{i1, i2, \dots, im\} \Rightarrow P_{ie}[j] = R) \wedge (P_{ie}[ie] = E))$$

BP 1 Gegenseitiger Ausschuß: Algorithmus von Singhal



BP 1 Gegenseitiger Ausschuß: Algorithmus von Singhal

L4.1.4 Lemma

Wenn

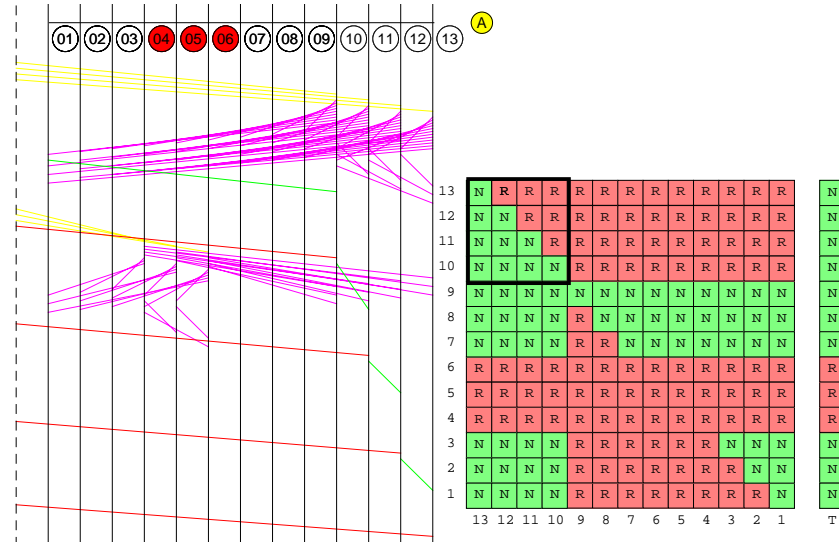
- Prozesse $P_{im}, P_{im-1}, \dots, P_{i1}$ eben ihre kritischen Abschnitte in dieser Reihenfolge ausgeführt haben und
- alle von ihnen gesandten REQUEST-Nachrichten empfangen und verarbeitet wurden,

dann gilt:

$$\forall j, k (1 \leq k < j \leq m \Rightarrow R_{ij}^{ik})$$

$$\forall j, k (1 \leq j < k \leq m \Rightarrow N_{ij}^{ik})$$

BP 1 Gegenseitiger Ausschluß: Algorithmus von Singhal



BP 1 Gegenseitiger Ausschluß: Algorithmus von Singhal

L4.1.5 Lemma

Wenn

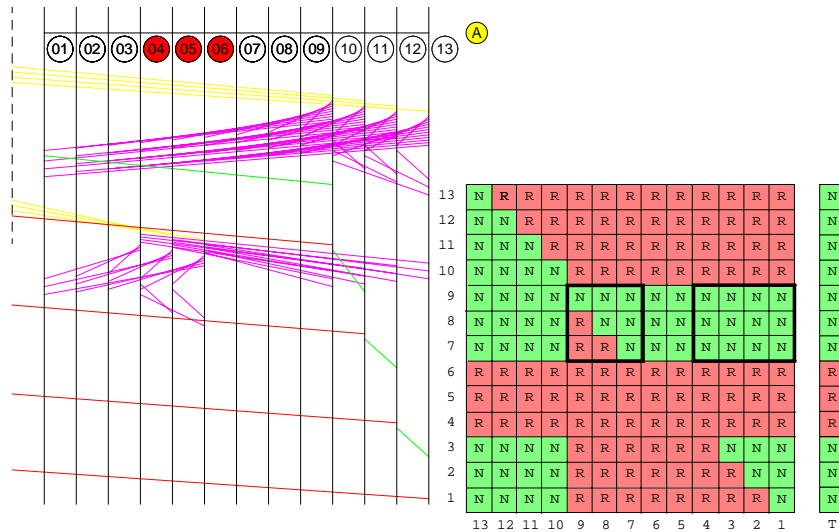
- das System im Anfangszustand startet,
- alle Prozesse in $NP = \{P_{im}, P_{im-1}, \dots, P_{i1}\}$ anfänglich $|P_{im}| < |P_{im-1}| < \dots < |P_{i1}|$ erfüllen und
- alle REQUEST-Nachrichten angekommen und bearbeitet sind,

dann gilt:

$$\forall j, k (1 \leq k < j \leq m \Rightarrow R_{ij}^{ik})$$

$$\forall j, k (1 < j < k \leq m \Rightarrow N_{ij}^{ik})$$

BP 1 Gegenseitiger Ausschluß: Algorithmus von Singhal



BP 1 Gegenseitiger Ausschluß: Algorithmus von Singhal

S4.2 Satz

Wenn das System im Anfangszustand startet und zu jedem Betrachtungszeitpunkt folgende Eigenschaften besitzt:

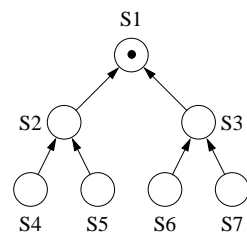
- Alle Prozesse $P_{im}, P_{im-1}, \dots, P_{i1}$ haben ihre kritischen Abschnitte genau einmal und in dieser Reihenfolge ausgeführt.
- Alle REQUEST-Nachrichten sind angekommen und bearbeitet.
- Prozesse $P_{jr}, P_{jr-1}, \dots, P_{j1}$ haben nie Anforderungen gestellt und $|P_{jr}| < |P_{jr-1}| < \dots < |P_{j1}|$.
- Kein Prozeß fordert das Token an.

Dann bilden $P_{jr}, P_{jr-1}, \dots, P_{j1}, P_{im}, P_{im-1}, \dots, P_{i1}$ eine vollständig geordnete Sequenz.

Beweis: Folgt unmittelbar aus Lemmata 2, 4 und 5.

BP 1	Gegenseitiger Ausschluß: Algorithmus von Singhal
S4.3	<p>Satz</p> <p>Wenn das System im Anfangszustand startet und zu jedem Zeitpunkt folgende Eigenschaften besitzt:</p> <ul style="list-style-type: none"> • Einige oder alle Prozesse haben ihren kritischen Abschnitt beliebige Male durchlaufen. • Alle REQUEST-Nachrichten wurden empfangen und bearbeitet. • Kein Prozeß hat eine Anforderung des Tokens ausstehen oder befindet sich im kritischen Abschnitt. <p>Dann bildet die Gesamtheit der Prozesse eine vollständig geordnete Sequenz.</p>
02.11.01	<small>Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig</small>

4.2-85

BP 1	Gegenseitiger Ausschluß: Algorithmus von Raymond
4.2.3	<p>Algorithmus von Raymond</p> <p>Die Prozesse sind als logischer Baum angeordnet</p>  <pre> graph BT S1((S1)) --> S2((S2)) S1 --> S3((S3)) S2 --> S4((S4)) S2 --> S5((S5)) S3 --> S6((S6)) S3 --> S7((S7)) </pre> <p>Datenstrukturen</p> <ul style="list-style-type: none"> • Lokale Datenstruktur des Prozesses P_i <ul style="list-style-type: none"> int holder; // Nummer des Vorgängers! // Bei der Wurzel: eigene Nummer int_fifo queue; // Warteschlange von // Anforderern
02.11.01	<small>Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig</small>

4.2-86

BP 1	Gegenseitiger Ausschluß: Algorithmus von Raymond
<div data-bbox="145 901 168 917" style="border: 1px solid black; width: 10px; height: 10px; display: inline-block;"></div>	<p>Anforderung des Tokens</p> <ul style="list-style-type: none"> • Wenn Prozeß P_i das Token benötigt und seine Warteschlange leer ist, sendet er eine REQUEST-Nachricht an seinen Vorgänger. Er reiht sich in seine eigene Warteschlange ein. • Wenn Prozeß P_i von P_j eine REQUEST-Nachricht erhält, nimmt er P_j in seine Warteschlange auf. Hat er selbst keine REQUEST-Nachricht ausstehen, so sendet er eine REQUEST-Nachricht an seinen Vorgänger. • Wenn die Wurzel eine REQUEST-Nachricht erhält sendet sie das Token zu dem Anforderer und vermerkt ihn in der Variablen holder. • Wenn ein Prozeß das Token erhält, tilgt er den ersten Eintrag seiner Warteschlange, sendet das Token dem entsprechenden Prozeß und vermerkt den Empfänger in seiner Variablen holder. Wenn seine eigene Warteschlange dann noch nicht leer ist, sendet der Prozeß eine REQUEST-Nachricht an den in holder verzeichneten Prozeß.
02.11.01	<small>Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig</small>

4.2-87

BP 1	Gegenseitiger Ausschluß: Algorithmus von Raymond
<div data-bbox="1198 901 1220 917" style="border: 1px solid black; width: 10px; height: 10px; display: inline-block;"></div>	<p>Ausführung eines kritischen Abschnitts</p> <ul style="list-style-type: none"> • Prozeß P_i führt den kritischen Abschnitt aus, sobald er über das Token verfügt und als erster in seiner Warteschlange steht. Sein Eintrag wird in diesem Fall aus der Warteschlange entfernt. <p>Verlassen eines kritischen Abschnitts</p> <ul style="list-style-type: none"> • Falls die eigene Warteschlange nicht leer ist, wird das Token an den an erster Stelle stehenden Prozeß gesendet, der Empfänger in holder vermerkt und der Eintrag aus der Warteschlange entfernt. • Wenn dann die eigene Warteschlange noch nicht leer ist, wird eine REQUEST-Nachricht an den in holder vermerkten Prozeß gesandt.
02.11.01	<small>Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme), F. Hofmann Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig</small>

4.2-88