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

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.

Verteilte Darstellung des Systemzustandes

4.1.4 Der Algorithmus von Sanders

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.

Universeller erlaubnisbasierter Algorithmus

4.2 Token-basierte Algorithmen

4.2.1 Der Algorithmus von Suzuki und Kasami

4.2.2 Der Algorithmus von Singhal

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.

4.2.3 Der Algorithmus von Raymond

❑ Systemmodell

- **Prozesse (Knoten, ’sites’) wickeln zeitlich sequentiell eine Folge von Aktionen ab**

- **Es gibt drei Arten von Aktionen:**
  - lokale, die keinerlei Auswirkungen auf andere Prozesse haben,
  - Aktionen zum Senden von Nachrichten,
  - Aktionen zum Empfangen von Nachrichten.

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

BP 1 **Gegenseitiger Ausschluß: Systemmodell, logische Uhren**

❑ Logische Uhren nach Lamport

- **Jeder Prozeß verfügt über eine Uhr C_i , die in folgender Weise zählt:**
  - Bei Ausführung einer lokalen Aktion und bei Ausführung einer Sende-Aktion durch Prozeß **i**, wird seine Uhr $\mathbf{C_i}$ um 1 erhöht.
  - Jede Nachricht **m** trägt als Zeitstempel $\mathbf{t_m}$ den Stand der Uhr des Senders mit sich, den sie unmittelbar vor der Sendeaktion hatte.
  - Nach Ausführung einer Empfangsaktion durch Prozeß **i** bei Uhrenstand $\mathbf{C_i}$ und empfangener Nachricht **m** wird die lokale Uhr auf $\mathbf{max}(\mathbf{C_i}, \mathbf{t_m}) + \mathbf{1}$ gestellt.

- **Ergänzung zu vollständiger Ordnung**

  Zeitstempel bestehend aus Prozessornummer i und lokaler Zeit C_i.

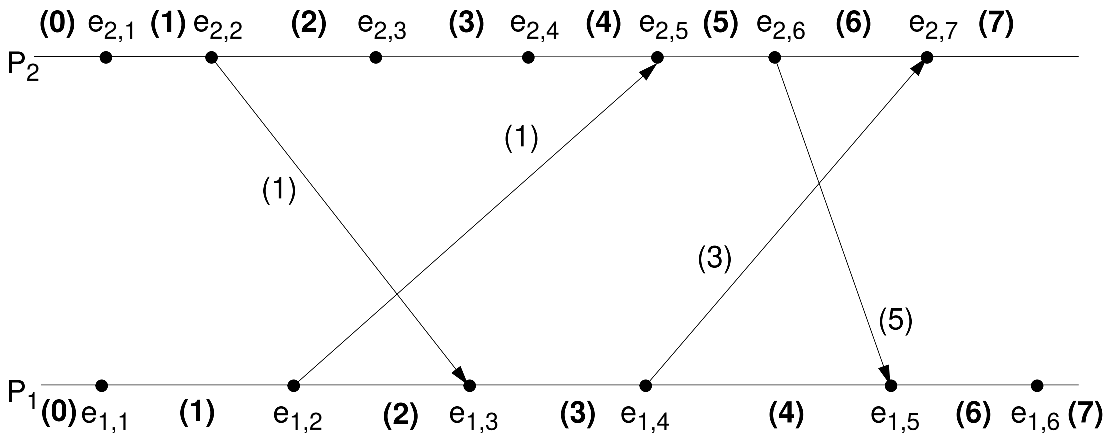
  Anordnung: $(C_i, i) < (C_k, k)$ **genau dann, wenn** $C_i < C_k$ **oder** $C_i = C_k$ **und** $i < k$.

13.12.99
Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig

BP 1

Gegenseitiger Ausschluß: Systemmodell, logische Uhren

- Beispiel



BP 1

Gegenseitiger Ausschluß: Systemmodell, logische Uhren

❑ **Wesentliche Gesichtspunkte**

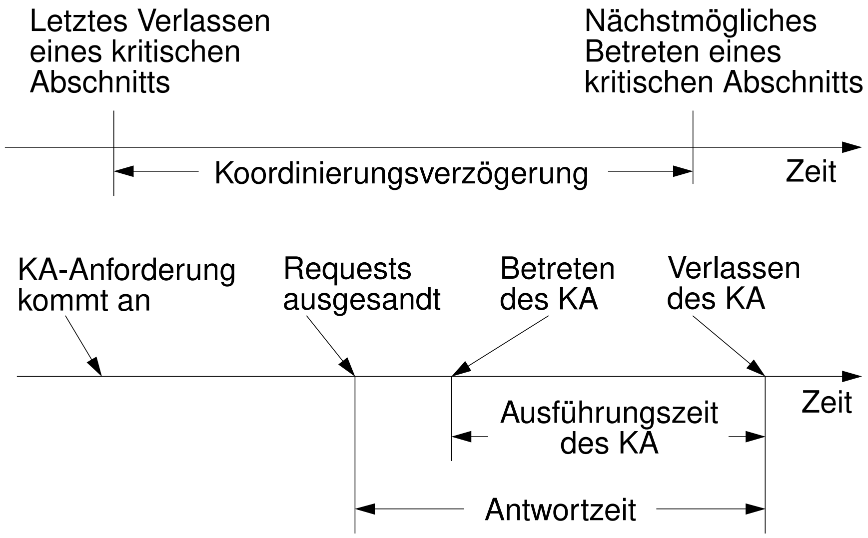
- **Gewährleistung des gegenseitigen Ausschlusses**
- **Verklemmungsfrei**
- **Aushungerungsfrei**

  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**

Gegenseitiger Ausschluß: Systemmodell, logische Uhren

- Erzielte Performanz



Gegenseitiger Ausschluß: 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.**

Gegenseitiger Ausschluß: 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

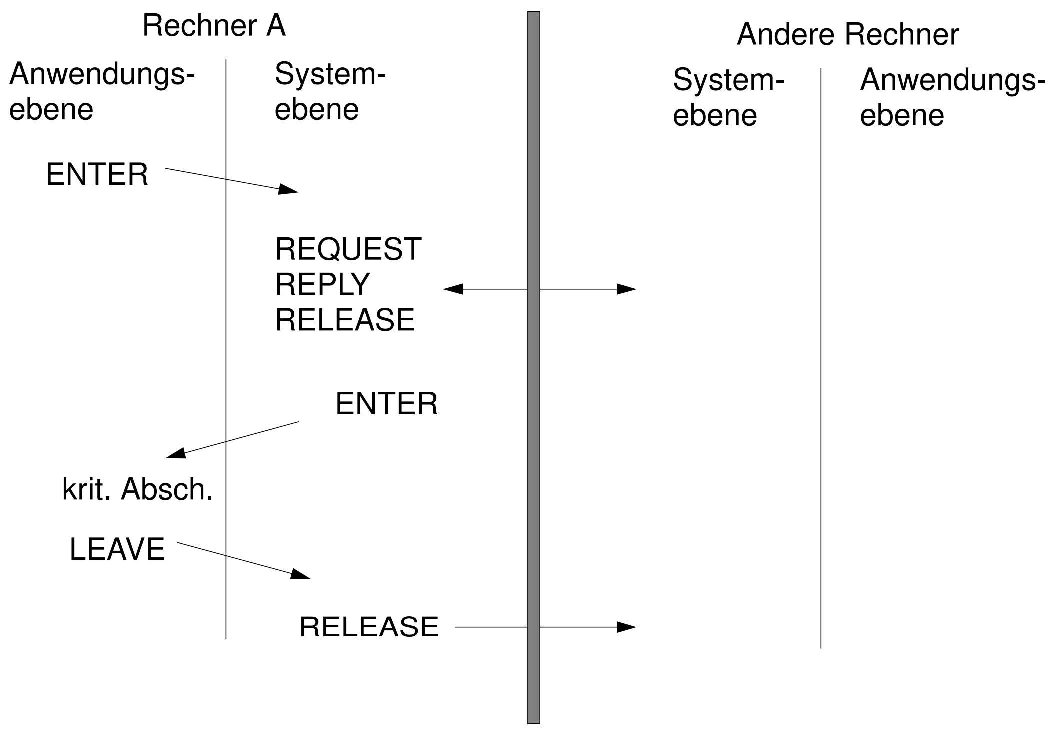
- **Verhalten**

  T mittlere Nachrichtenlaufzeit; E mittlere Ausführungszeit für k. A.

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

Gegenseitiger Ausschluß: Algorithmus von Lamport

- **Implementationsarchitektur**



Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig

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) { ... }
}
```


Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann


```
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;
  ...
 }
```


Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig

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; }
}
```


Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

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) { ... }

}
```


Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig

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;
  }
}
```



```
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);
  }
}
```

Gegenseitiger Ausschluß: 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;
  }
}
```


Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Gegenseitiger Ausschluß: Der Algorithmus von Lamport

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

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


Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig

Gegenseitiger Ausschluß: Der Algorithmus von Lamport

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

  scheduler.sleep(2000);

  // Process S1 wants to enter critical section
  if (Thread.currentThread().getName().equals("S1")) {
    ((Connection)(outgoingConnections.elementAt(nodeNumber)))
      .send("ENTER");

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


Gegenseitiger Ausschluß: Der Algorithmus von Lamport

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

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

```
} else if (message.type.equals("REQUEST")) {
  queue[sender] = new QueueEntry(message);
  for (i = 0; i < Lamport.NUMBEROFPROCESSES; i++) {
    ((Connection)(outgoingConnections.elementAt(i)))
      .send("REPLY");
  }

} else if (message.type.equals("REPLY")) {
  if (check()) {
  // Process enters critical section
    scheduler.sleep(10000);
    ((Connection)(outgoingConnections.elementAt(i)))
      .send("LEAVE");
  }
```



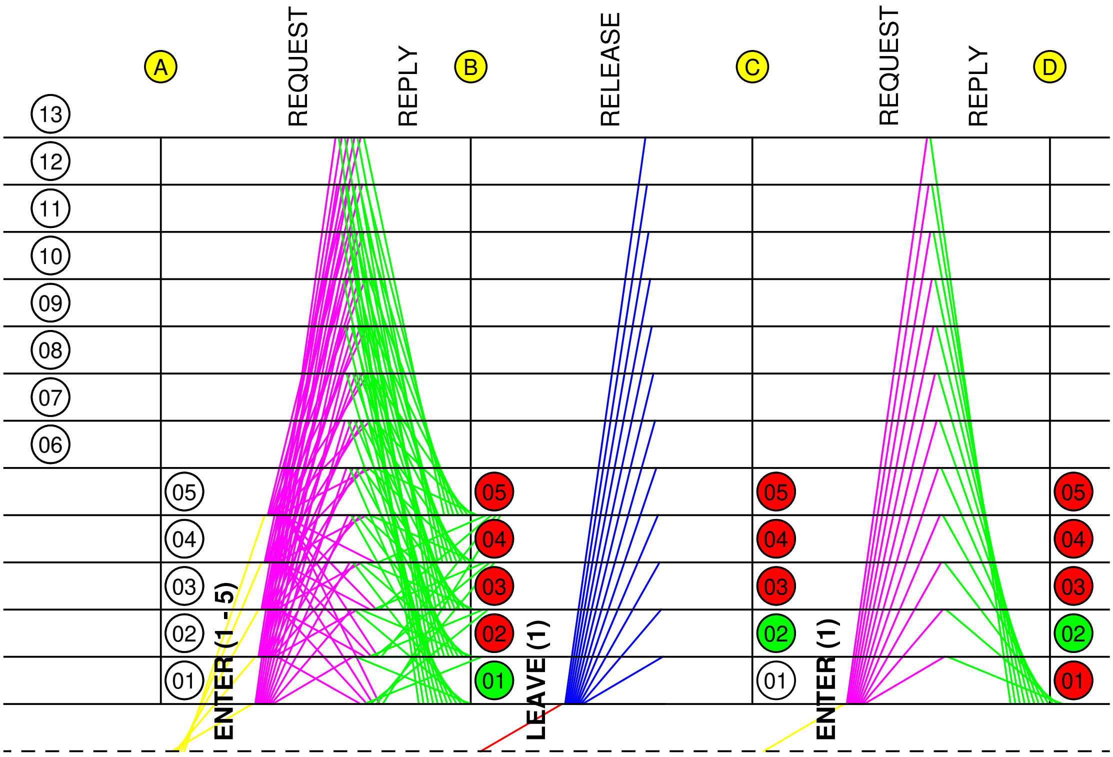
```
} else if (message.type.equals("RELEASE")) {
  queue[sender].type = "OUTSIDE";
  if (check()) {
  // Process enters critical section
    scheduler.sleep(10000);
    ((Connection)(outgoingConnections.elementAt(i)))
      .send("LEAVE");
  }

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

BP 1 Gegenseitiger Ausschluß: 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.NUMBEROFPROCESSES; 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 Ausschluß: Der Algorithmus von Lamport



4.1.2 Der Algorithmus von Ricart und Agrawala

- **Nachrichtenkanal: zuverlässig**
  (FIFO-Eigenschaft nicht notwendig!)
- **Eintrittswunsch des Knotens S_i (site i)**
  REQUEST(ts, i) an alle
- **Bearbeitung von REQUESTs:**
  - **Wenn S_j eine REQUEST-Nachricht von S_i erhält, sendet er eine REPLY-Nachricht an S_i, falls er**
    - 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.**

- **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**
  - **S_i versendet zu allen aufgeschobenen REQUESTs eine REPLY-Nachricht.**

❑ **Verhalten**

T mittlere Nachrichtenlaufzeit;
E mittlere Ausführungszeit für k. A.

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

Der Algorithmus von Ricart und Agrawala

```
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;
  }
}
```



```
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);
  }
}
```

Gegenseitiger Ausschluß: 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";
  }
```


Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Gegenseitiger Ausschluß: Algorithmus von Ricart und Agrawala

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

    scheduler.sleep(2000);

    // Process S1 wants to enter critical section
    if (Thread.currentThread().getName().equals("S0")
        || Thread.currentThread().getName().equals("S1")) {
      ((Connection)(outgoingConnections.elementAt(nodeNumber)))
        .send("ENTER");
    }

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


Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig

Gegenseitiger Ausschluß: 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");
  }
```


Gegenseitiger Ausschluß: 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);
}
}
}
```

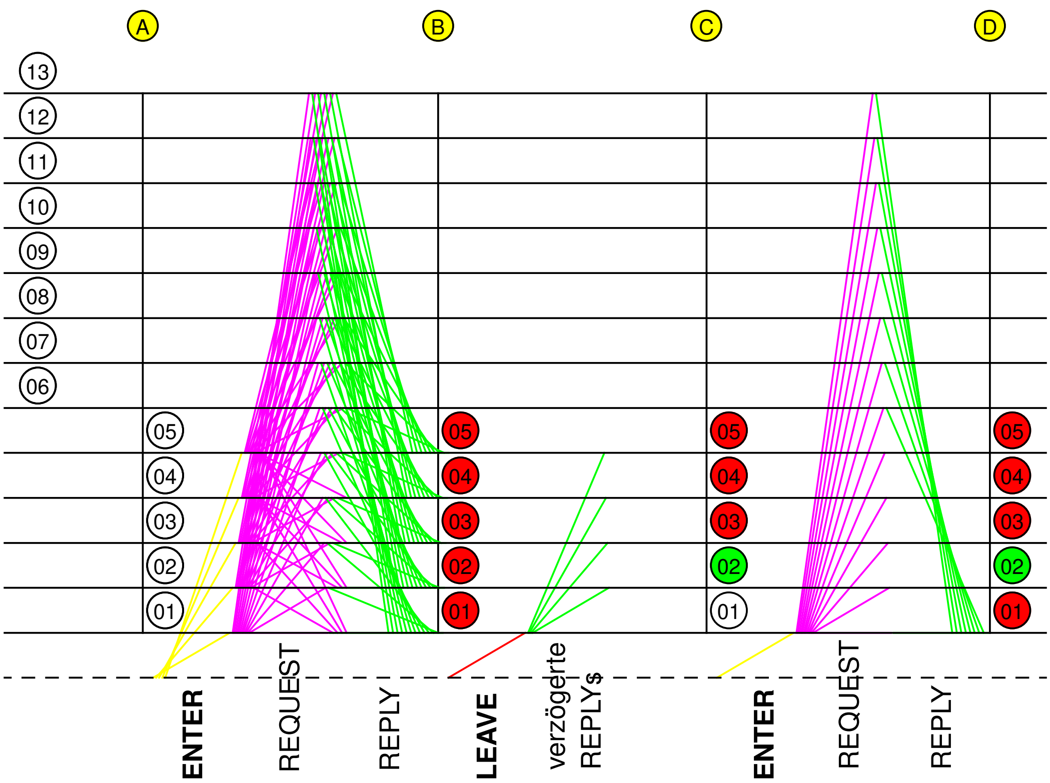
BP 1

Gegenseitiger Ausschluß: 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 Ausschluß: Algorithmus von Ricart und Agrawala



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 $(1 \le i \le n)$ zugeordnet derart, daß folgende Eigenschaften erfüllt sind:

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

❑ **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.

Gegenseitiger Ausschluß: Algorithmus von Maekawa

- Nachrichtenformat: (source, target, timestamp, message)
- Übergangsfunktion

1. Zugang anfordern durch Knoten i:

   send(i, s, T, REQUEST), an alle $s \in S_i$

2. Bei Empfang von Anforderung r = (i, s, T', REQUEST) durch s:

   frei: r in Anforderungsmenge aufnehmen
   belegt für r notieren;
   send(s, i, T, LOCKED);

   belegt für r': r in Anforderungsmenge aufnehmen;
   r nicht ältestes Element der Anf.-Menge:
   send(s, i, T, FAILED);
   r ältestes Element der Anf.-Menge:
   send(s, r'.source, T, INQUIRE),
   falls noch nicht geschehen;

Gegenseitiger Ausschluß: Algorithmus von Maekawa

3. Bei Empfang von (s, i, T', INQUIRE) durch i:

   Eine der letzten Anforderungen von i wurde von einem Quorums-Mitglied mit FAILED beantwortet:
   send(i, s, T, RELINQUISH);
   LOCKED(s) aus der Erlaubnismenge entfernen;

   sonst: ignorieren, aber vormerken!!!!;

4. Bei Empfang von RELINQUISH:

   Bestehende Belegung (*belegt für r*) aufheben;
   Für älteste Anforderung r' aus der Anforderungsmenge
   belegt für r' notieren;
   send(s, r'.source, T, LOCKED);

Gegenseitiger Ausschluß: Algorithmus von Maekawa

5. Bei Empfang von (s, i, T', LOCKED)

LOCKED(s) in die Erlaubnismenge aufnehmen

Von allen Mitgliedern des Quorums Erlaubnis (LOCKED) erhalten:

Kritischen Abschnitt bearbeiten;

6. Beenden des kritischen Abschnitts

Erlaubnismenge leeren

send(i, s, T, RELEASE) für alle s $\in$ S_i

Gegenseitiger Ausschluß: Algorithmus von Maekawa

7. Bei Empfang von RELEASE:

belegt für r:

r aus Anforderungsmenge entfernen

Anforderungsmenge enthält neben r weitere Anforderungen:

Für älteste Anforderung r' $\neq$ r aus der Anforderungsmenge

***belegt für r'* notieren;**

send(s, r'.source, T, LOCKED);

8. Bei Empfang von (s, i, T', FAILED) durch i

Eine der letzten Anforderungen von i wurde vom Quorums-Mitglied t mit INQUIRE beantwortet:

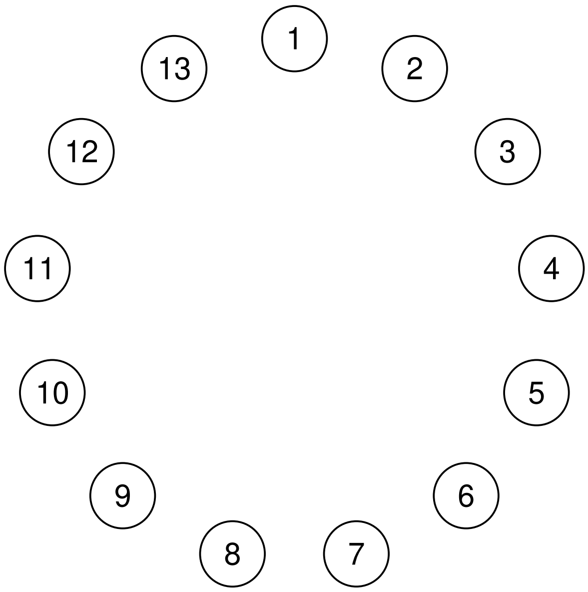
send(i, t, T, RELINQUISH);

LOCKED(t) aus der Erlaubnismenge entfernen;

sonst: ignorieren, aber vormerken!!!!

Gegenseitiger Ausschluß: Algorithmus von Maekawa

(A)

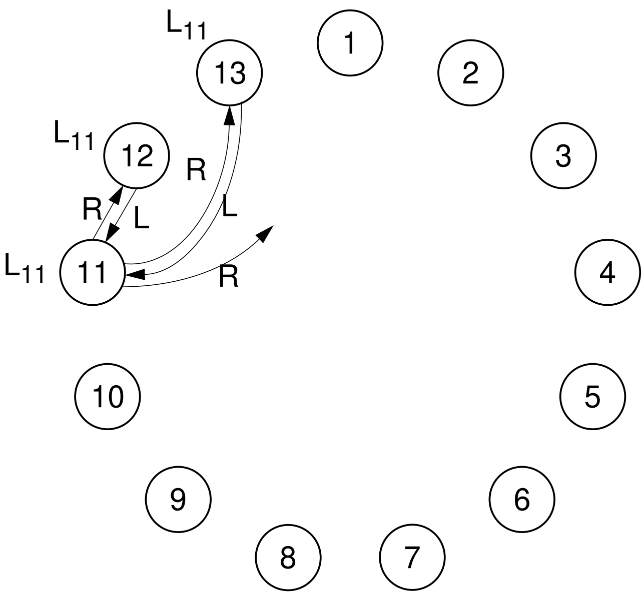


$S_1 = \{ 1, 2, 3, 4 \}$
$S_2 = \{ 2, 5, 8, 11 \}$
$S_3 = \{ 3, 6, 8, 13 \}$
$S_4 = \{ 4, 6, 10, 11 \}$
$S_5 = \{ 1, 5, 6, 7 \}$
$S_6 = \{ 2, 6, 9, 12 \}$
$S_7 = \{ 2, 7, 10, 13 \}$
$S_8 = \{ 1, 8, 9, 10 \}$
$S_9 = \{ 3, 7, 9, 11 \}$
$S_{10} = \{ 3, 5, 10, 12 \}$
$S_{11} = \{ 1, 11, 12, 13 \}$
$S_{12}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

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Gegenseitiger Ausschluß: Algorithmus von Maekawa

(B)



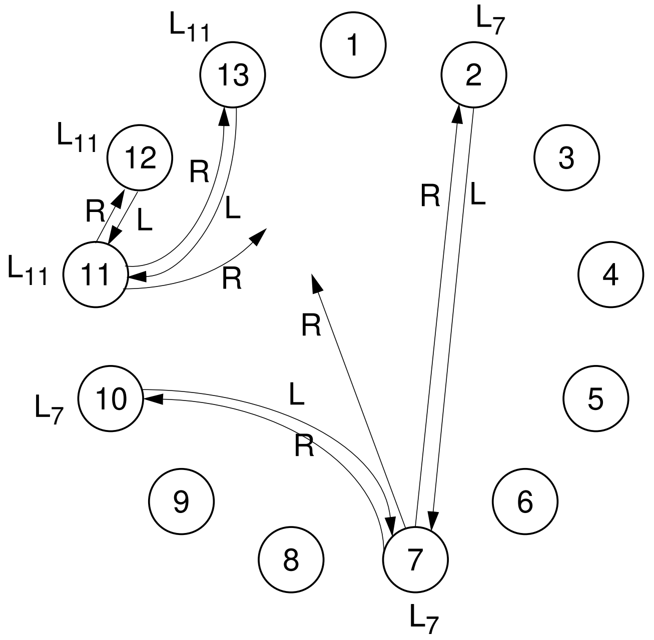
$S_1 = \{ 1, 2, 3, 4 \}$
$S_2 = \{ 2, 5, 8, 11 \}$
$S_3 = \{ 3, 6, 8, 13 \}$
$S_4 = \{ 4, 6, 10, 11 \}$
$S_5 = \{ 1, 5, 6, 7 \}$
$S_6 = \{ 2, 6, 9, 12 \}$
$S_7 = \{ 2, 7, 10, 13 \}$
$S_8 = \{ 1, 8, 9, 10 \}$
$S_9 = \{ 3, 7, 9, 11 \}$
$S_{10} = \{ 3, 5, 10, 12 \}$
$S_{11} = \{ 1, 11, 12, 13 \}$
$S_{12}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

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig

BP 1

Gegenseitiger Ausschluß: Algorithmus von Maekawa

(C)

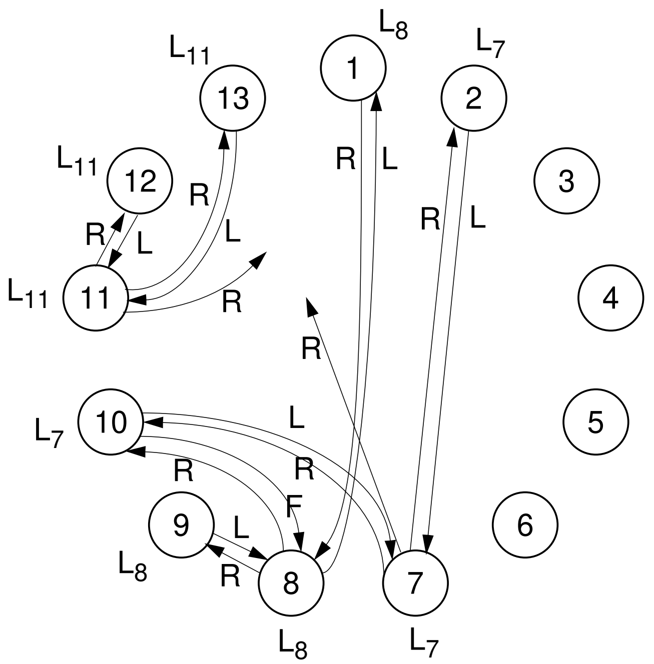


$S_1 = \{ 1, 2, 3, 4 \}$
$S_2 = \{ 2, 5, 8, 11 \}$
$S_3 = \{ 3, 6, 8, 13 \}$
$S_4 = \{ 4, 6, 10, 11 \}$
$S_5 = \{ 1, 5, 6, 7 \}$
$S_6 = \{ 2, 6, 9, 12 \}$
$S_7 = \{ 2, 7, 10, 13 \}$
$S_8 = \{ 1, 8, 9, 10 \}$
$S_9 = \{ 3, 7, 9, 11 \}$
$S_{10} = \{ 3, 5, 10, 12 \}$
$S_{11} = \{ 1, 11, 12, 13 \}$
$S_{12}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

BP 1

Gegenseitiger Ausschluß: Algorithmus von Maekawa

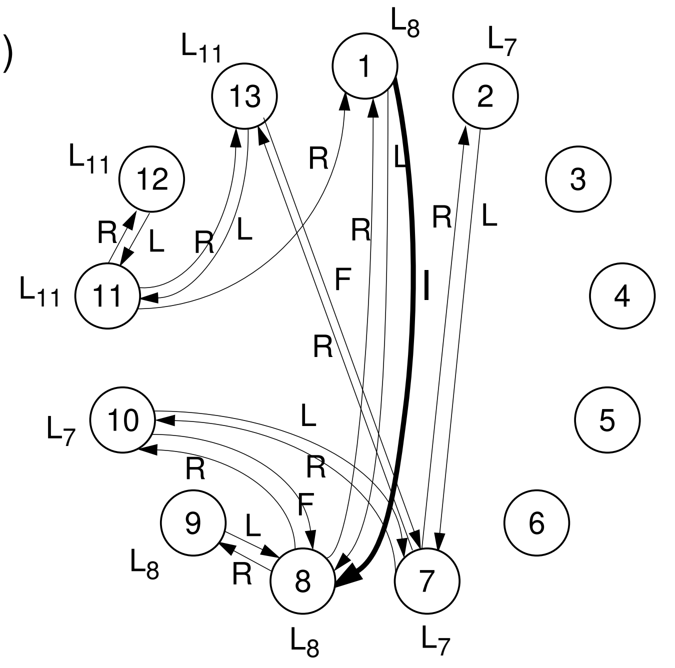
(D)



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

Gegenseitiger Ausschluß: Algorithmus von Maekawa

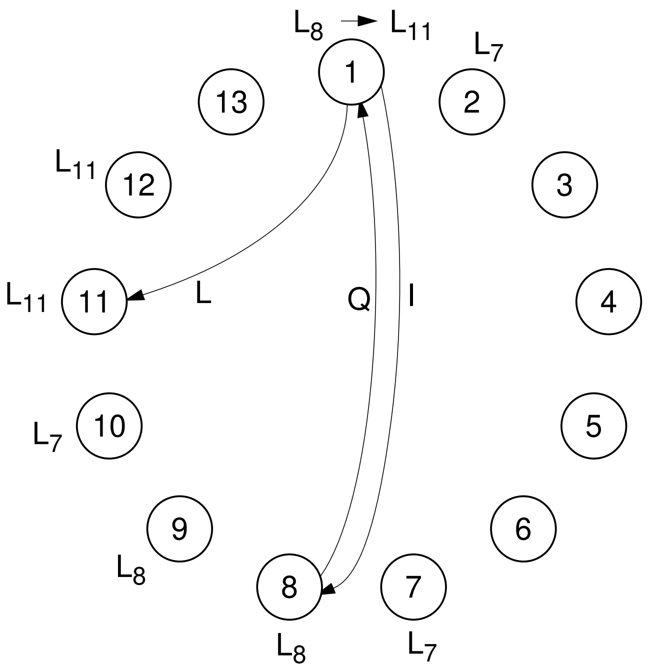
(E$_1$)



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

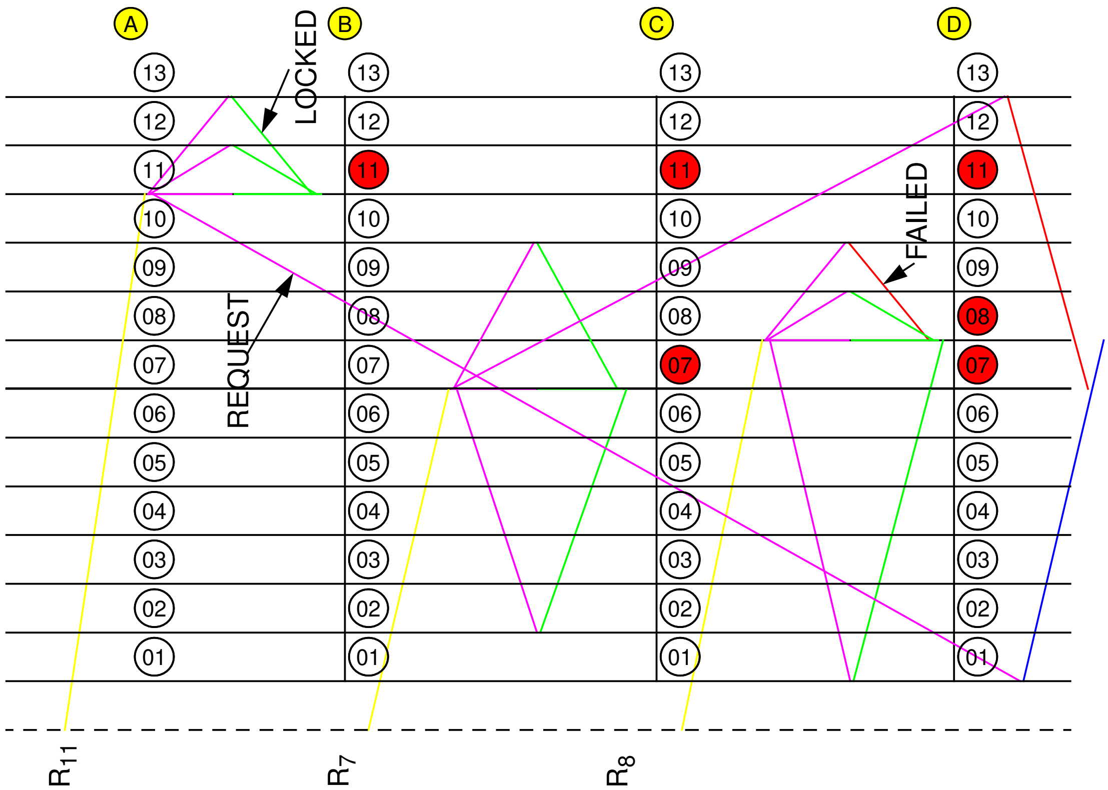
Gegenseitiger Ausschluß: Algorithmus von Maekawa

(E$_2$)



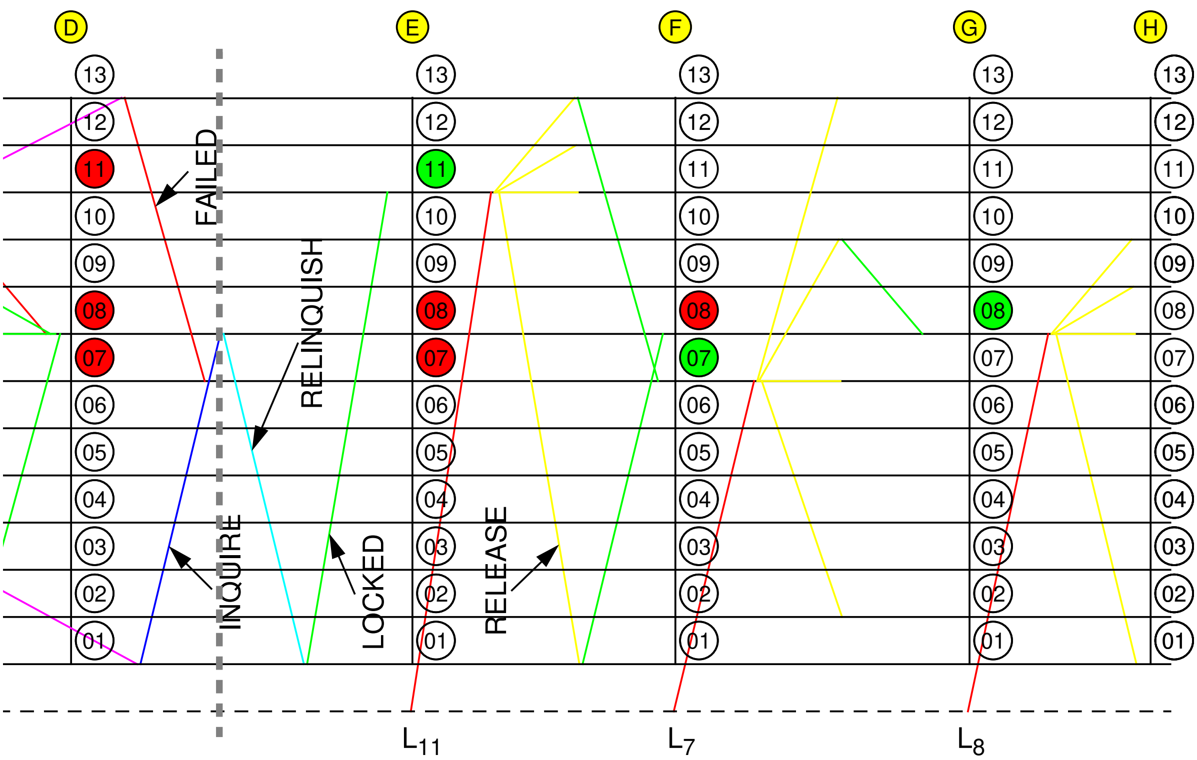
$S_1 = \{ 1, 2, 3, 4 \}$
$S_2 = \{ 2, 5, 8, 11 \}$
$S_3 = \{ 3, 6, 8, 13 \}$
$S_4 = \{ 4, 6, 10, 11 \}$
$S_5 = \{ 1, 5, 6, 7 \}$
$S_6 = \{ 2, 6, 9, 12 \}$
$S_7 = \{ 2, 7, 10, 13 \}$
$S_8 = \{ 1, 8, 9, 10 \}$
$S_9 = \{ 3, 7, 9, 11 \}$
$S_{10} = \{ 3, 5, 10, 12 \}$
$S_{11} = \{ 1, 11, 12, 13 \}$
$S_{12}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

Gegenseitiger Ausschluß: Algorithmus von Maekawa



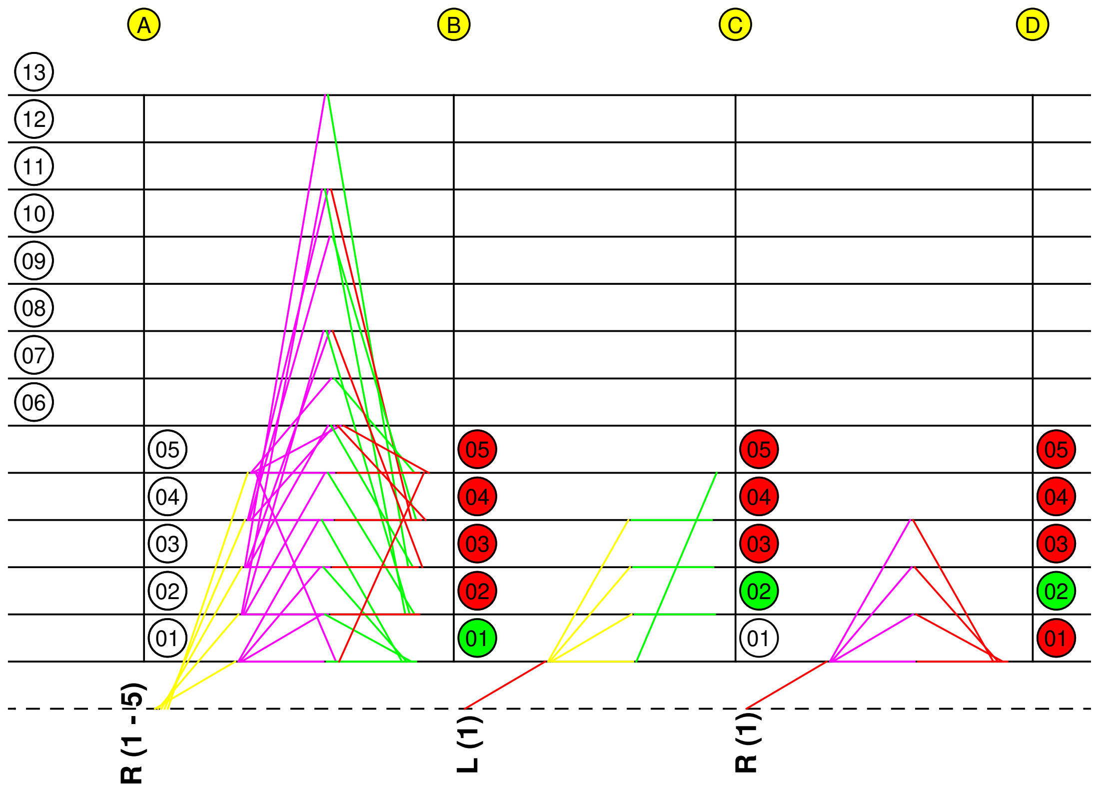
Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Gegenseitiger Ausschluß: Algorithmus von Maekawa



Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Gegenseitiger Ausschluß: Algorithmus von Maekawa



Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

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

1. $\forall i(1 \le i \le N \Rightarrow I_i \subseteq R_i)$
2. $\forall i, j(1 \le i, j \le N \Rightarrow (I_i \cap I_j \ne \emptyset) \vee (i \in R_j \wedge j \in R_i))$

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Gegenseitiger Ausschluß: 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

Gegenseitiger Ausschluß: Algorithmus von Sanders

- **Empfang von r = (r, s, -, RELEASE)**

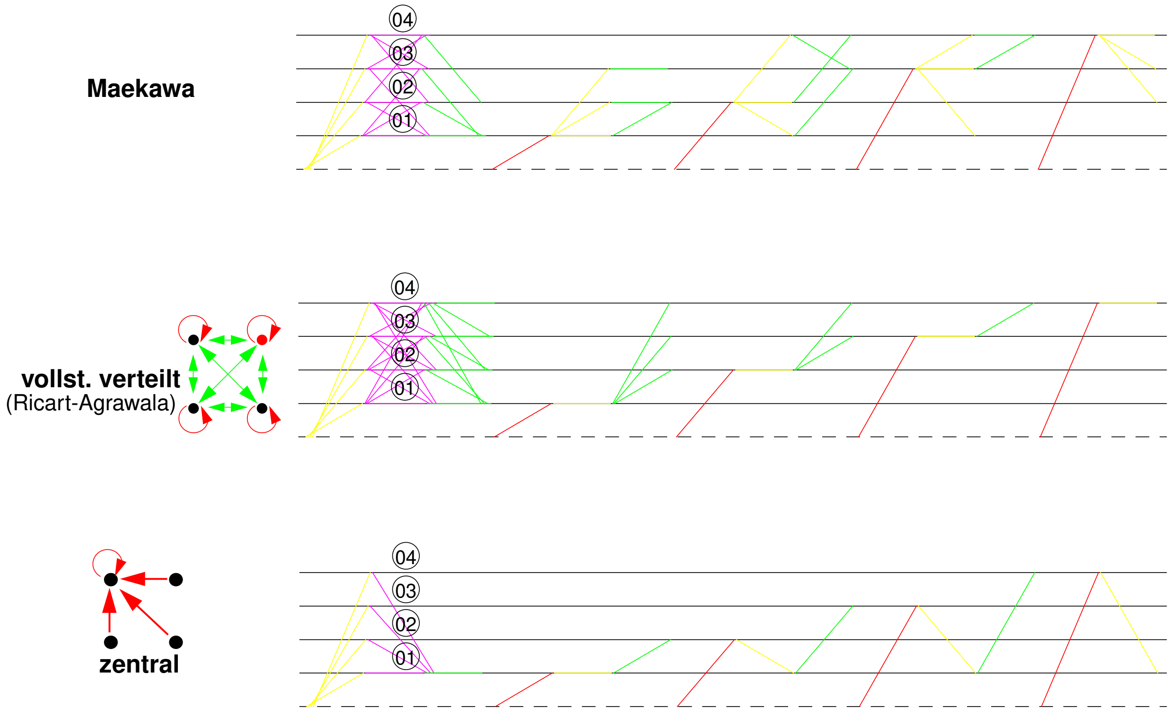
  status = frei
  solange status == frei und Anforderungsmenge != $\varnothing$
  send(s, r'.source,-,LOCKED)
  für älteste bestehende Anforderung r'
  status = belegt(r'), falls r'.source $\in St_s$
  r' aus der Anforderungsmenge entfernen

Nicht verklemmungsfrei! Lösung analog Maekawa

- **Graphische Darstellung**

  $i \rightarrow j$ $j \in I_i$
  $i \rightarrow j$ $j \in R_l, j \notin I_i$

Gegenseitiger Ausschluß: Algorithmus von Sanders



Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Gegenseitiger Ausschluß: 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

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

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

Weiter ist

$$\mathbf{NM_i} \le |\mathbf{I_i} - \{\mathbf{i}\}| + 2(|\mathbf{R_i} - \{\mathbf{i}\}|) + \mathbf{DM} \quad \textbf{mit}$$

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

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Gegenseitiger Ausschluß: Algorithmus von Sanders

❑ Spezialisierungen

- **Lamport**

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

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

- **Ricart-Agrawala**

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

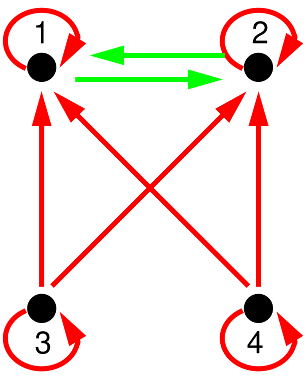
  $2(N-1) \le NM_i \le 2(N-1) + N$

Gegenseitiger Ausschluß: Algorithmus von Sanders

❑ Ein neuer Algorithmus

$I_1 = \{1\}$, $R_1 = \{1, 2\}$, $St_1 = \{1, 3, 4\}$

$I_2 = \{2\}$, $R_2 = \{1, 2\}$, $St_2 = \{2, 3, 4\}$



$I_3 = \{1, 2, 3\}$, $R_3 = \{1, 2, 3\}$, $St_3 = \{3\}$

$I_4 = \{1, 2, 4\}$, $R_4 = \{1, 2, 4\}$, $St_4 = \{4\}$

x → y (grün): x muß y fragen, aber nicht informieren

x → y (rot): x muß y fragen und informieren

Gegenseitiger Ausschluß: Algorithmus von Sanders

- **Zahl auszutauschender Nachrichten**

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

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

- **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

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Gegenseitiger Ausschluß: 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.

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig

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

❑ **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.

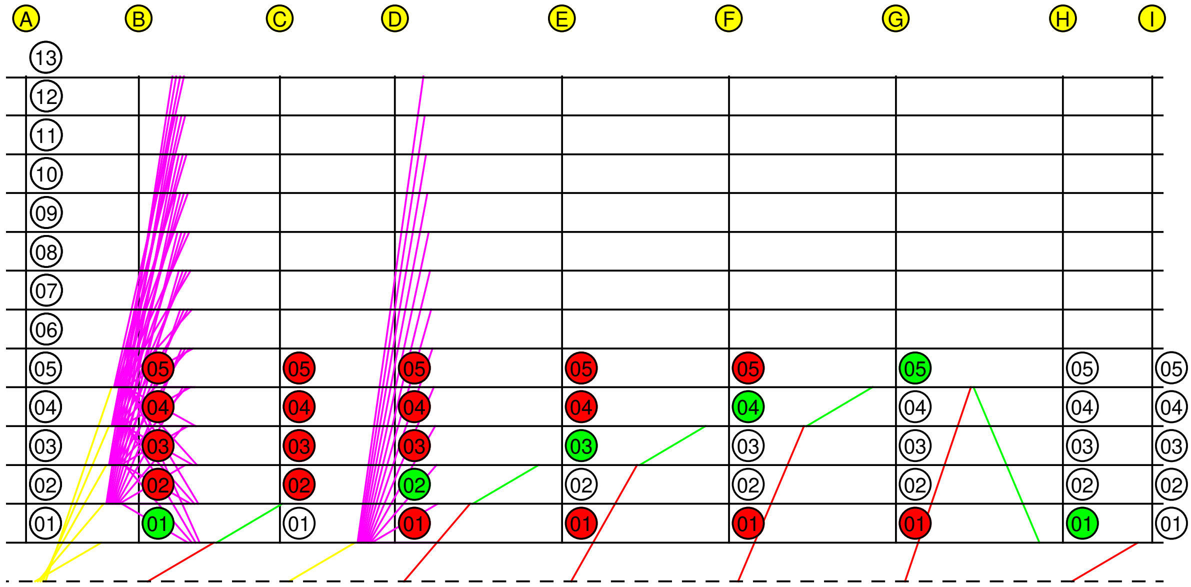
❑ **Ausführung eines kritischen Abschnitts**

- **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.**

Gegenseitiger Ausschluß: Algorithmus von Suzuki-Kasami



Gegenseitiger Ausschluß: 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)enum t_state {R, E, H, N};

```
struct { t_state ST; // state
         int     SN; // sequence number
       } P[MAX_TEILNEHMERZAHL + 1];
```

❑ Datenstruktur des Tokens

```
struct {
  struct {
    t_state ST = N;
    int     SN = 0;
  } P[MAX_TEILNEHMERZAHL + 1];
} Token;
```

Anfangsbesetzung der lokalen Zustände

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

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

❑ Anforderung des Tokens durch Prozeß P_i

```
/*(1)*/  P[i].SN++;
         if (P[i].ST != H) {
           P[i].ST = R;
           for (j = 1; j <= teilnehmerzahl; j++) {
             if (P[j].ST == R && j != i) {
                send(j,        // Empfaenger
                     REQUEST,  // Nachrichtentyp
                     i,        // Sender
                     P[i].SN)  // Daten
                 );
             }
           }
           wait_for_token();
         }
```

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

i	**Adressat**
REQUEST	**Nachrichtentyp**
j	**Absender**
sn	**Sequenznummer**

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

```
          if (P[j].SN < sn){
            switch (P[i].ST) {
/*(8)*/       case N: P[j].ST = R;
                      P[j].SN = sn;
                      break;
/*(9)*/       case R: if (P[j].ST != R) {
                        P[j].ST = R;
                        send(j, REQUEST, i, P[i].SN);
                      }
                      P[j].SN = sn;
                      break;
/*(10)*/      case E: P [j].ST = R;
                      P[j].SN = sn;
                      break;
/*(11)*/      case H: P[j].ST = R;
                      P[j].SN = sn;
                      T.P[j].ST = R;
                      T.P[j].SN = sn;
                      P[i].ST = N;
                      send(j, TOKEN, i, token);
                      break;
            }
          }
```


Empfang des Tokens (i, TOKEN, j, token)

```
/*(2)*/   P[i].ST = E;
          execute_critical_section();
```

Verlassen eines kritischen Abschnitts

```
/*(3)*/   int tmp;
          P[i].ST = N;           // wird fuer update benötigt
          T.P[i].ST = N;
          T.P[i].SN = P[i].SN;
          update_ST_vectors();
          P[i].ST = H;           // tatsaechlicher Zustand

                                 // Arbitrierung nach round robin,
                                 // koennte auch anders erfolgen.

          for (tmp = 1; tmp <= teilnehmerzahl; tmp++) {
            int j = ((i+tmp-1) % teilnehmerzahl) + 1;
            if (P[j].ST == R) {
              P[i].ST = N;
              send(j, TOKEN, i, token);
              break;
            }
          }
```

Gegenseitiger Ausschluß: Algorithmus von Singhal

```
update_ST_vectors() {
  for (tmp = 1; tmp <= teilnehmerzahl; tmp++) {

/*(4)*/   if (P[tmp].ST == R && T.P[tmp].ST == N) {

            if (P[tmp].SN == T.P[tmp].SN)
            // Token hat die neuere Information
              P[tmp].ST = N;

            else if (P[tmp].SN > T.P[tmp].SN) {
            // Prozess hat die neuere Information
              T.P[tmp].ST = R;
              T.P[tmp].SN = P[tmp].SN;
            } else /* if (P[tmp].SN < T.P[tmp].SN) */ {
            // Token hat die neuere Information
              P[tmp].ST = N;
              P[tmp].SN = T.P[tmp].SN;
            }
```


Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Gegenseitiger Ausschluß: Algorithmus von Singhal

```
          } else if (P[tmp].ST == N && T.P[tmp].ST == R) {
/*(5)*/     if (P[tmp].SN == T.P[tmp].SN)
              cout << "\n***** error 1\n" << flush;
            else if (P[tmp].SN > T.P[tmp].SN)
              cout << "\n***** error 2\n" << flush;
            else /* if (P[tmp].SN < T.P[tmp].SN) */{
            // Token hat die neuere Information
              P[tmp].ST = R;
              P[tmp].SN = T.P[tmp].SN;
            }
          } else if (P[tmp].ST == N && T.P[tmp].ST == N){
/*(6)*/     if (P[tmp].SN == T.P[tmp].SN) ;
            // Token und Prozess haben gleiche Information
           else if (P[tmp].SN > T.P[tmp].SN)
               cout << "\n***** error 3\n";
            else /* if (P[tmp].SN < T.P[tmp].SN) */
            // Token hat die neuere Information
             P[tmp].SN = T.P[tmp].SN;
```


Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Gegenseitiger Ausschluß: Algorithmus von Singhal

```
            } else /* if (P[tmp].ST == R && T.P[tmp].ST == R) */ {
/*(7)*/       if (P[tmp].SN == T.P[tmp].SN) ;
              else if (P[tmp].SN > T.P[tmp].SN)
              // Prozess hat die neuere Information
               T.P[tmp].SN = P[tmp].SN;
              else /* if (P[tmp].SN < T.P[tmp].SN) */
              // Token hat die neuere Information
                P[tmp].SN = T.P[tmp].SN;
            }
    }
  }
```


Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Gegenseitiger Ausschluß: Algorithmus von Singhal

- **Komprimierbar zu**

```
update_ST_vectors() {
  for (tmp = 1; tmp <= teilnehmerzahl; tmp++) {
    if (P[tmp].SN > T.P[tmp].SN) {
    // Prozess hat neuere Information
      T.P[tmp].ST = P[tmp].ST;
      T.P[tmp].SN = P[tmp].SN;
    } else {
    // Token hat die neuere Information
      P[tmp].ST = T.P[tmp].ST;
      P[tmp].SN = T.P[tmp].SN;
    }
  }
}
```


Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

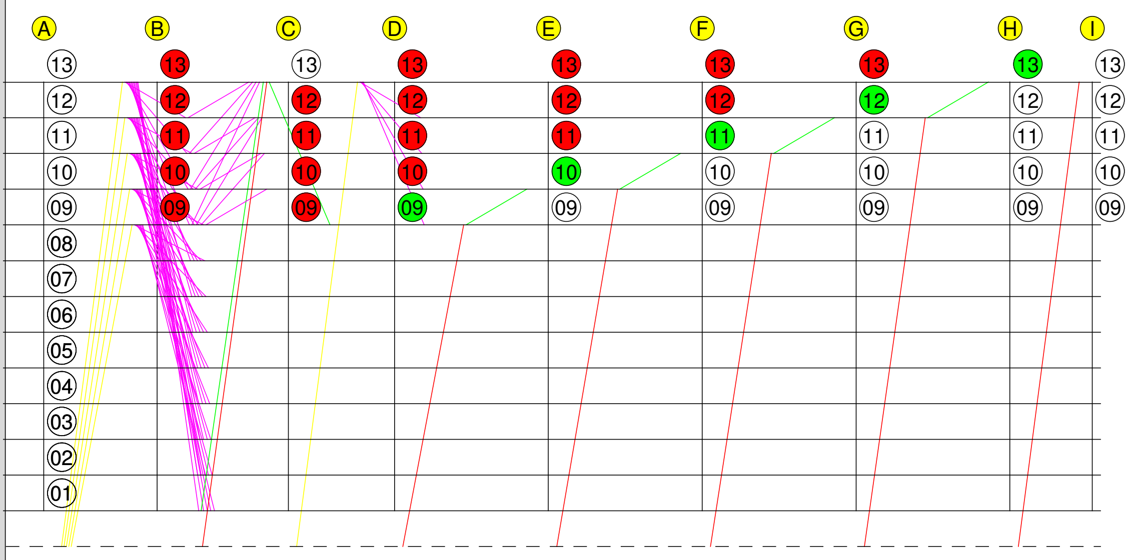
Gegenseitiger Ausschluß: Algorithmus von Singhal

❑ **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**

Gegenseitiger Ausschluß: Algorithmus von Singhal



Verifikation

Gegenseitiger Ausschluß

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.

Verklemmungsfreiheit

(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.

Definitionen

P_i bezeichne (auch) den Zustandsvektor des i-ten Prozesses

$\mathbf{R_i^j} = \mathbf{TRUE}$, genau dann wenn $P_i[j].ST = R$

$\mathbf{N_i^j} = \mathbf{TRUE}$, genau dann wenn $P_i[j].ST = N$

$|\mathbf{P_i}|$ ist die Zahl der N-Einträge im Zustandsvektor des i-ten Prozesses

NP **ist die Menge der Prozesse, die seit Systembeginn noch keine Anforderung gestellt haben**

EP ist die Menge der Prozesse, die seit Systembeginn wenigstens einmal ihren kritischen Abschnitt durchlaufen haben

Gegenseitiger Ausschluß: Algorithmus von Singhal

Im Anfangszustand gilt:

R_i^j ist erfüllt, für $1 \le i \le n$ und $1 \le j < i$

N_i^j ist erfüllt für $1 < i \le n$ und $i \le j \le n$

N_1^j ist erfüllt für $1 < j \le n$

Eine Sequenz $P_n, P_{n-1}, ..., P_1$ ist "vollständig geordnet" wenn gilt:

$$|P_n| < |P_{n-1}| < \dots < |P_1|$$

L4.1.1 Lemma

Das Token hat niemals falsche R-Einträge.

Gegenseitiger Ausschluß: Algorithmus von Singhal

L4.1.2 Lemma

Wenn

- das System im Anfangszustand startet,
- alle Prozesse in **EP** ihren kritischen Abschnitt genau einmal durchlaufen haben und
- alle REQUEST-Nachrichten angekommen und bearbeitet sind,

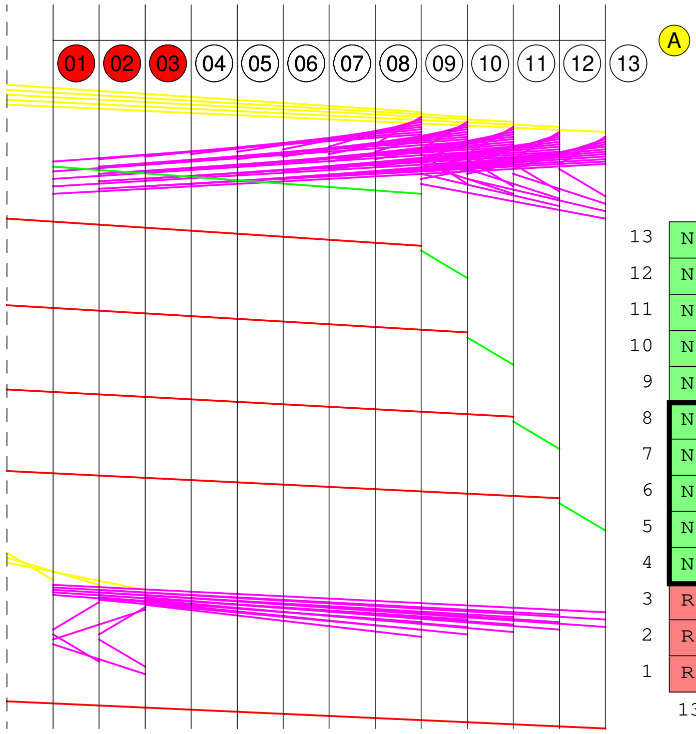
dann gilt:

$$\forall P_i, P_j (P_i \in EP \wedge P_j \in NP \Rightarrow R_j^i)$$

und

$$\forall P_i, P_j (P_i \in EP \wedge P_j \in NP \Rightarrow N_i^j)$$

Gegenseitiger Ausschluß: Algorithmus von Singhal



	13	12	11	10	9	8	7	6	5	4	3	2	1	T
13	N	R	R	R	R	R	R	R	R	R	R	R	R	N
12	N	N	R	R	R	R	R	R	R	R	R	R	R	N
11	N	N	N	R	R	R	R	R	R	R	R	R	R	N
10	N	N	N	N	R	R	R	R	R	R	R	R	R	N
9	N	N	N	N	N	R	R	R	R	R	R	R	R	N
8	N	N	N	N	N	N	N	N	N	N	N	N	N	N
7	N	N	N	N	N	R	N	N	N	N	N	N	N	N
6	N	N	N	N	N	R	R	N	N	N	N	N	N	N
5	N	N	N	N	N	R	R	R	N	N	N	N	N	N
4	N	N	N	N	N	R	R	R	R	N	N	N	N	N
3	R	R	R	R	R	R	R	R	R	R	R	R	R	R
2	R	R	R	R	R	R	R	R	R	R	R	R	R	R
1	R	R	R	R	R	R	R	R	R	R	R	R	R	R

Gegenseitiger Ausschluß: Algorithmus von Singhal

L4.1.3 Lemma

Wenn

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

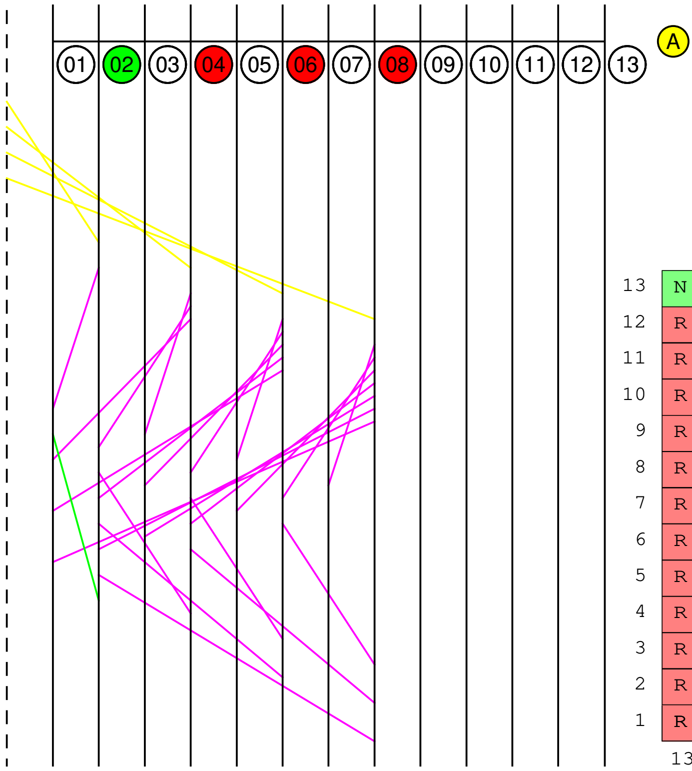
dann gilt:

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

und

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

Gegenseitiger Ausschluß: Algorithmus von Singhal



	13	12	11	10	9	8	7	6	5	4	3	2	1	T
13	N	N	N	N	N	N	N	N	N	N	N	N	N	N
12	R	N	N	N	N	N	N	N	N	N	N	N	N	N
11	R	R	N	N	N	N	N	N	N	N	N	N	N	N
10	R	R	R	N	N	N	N	N	N	N	N	N	N	N
9	R	R	R	R	N	N	N	N	N	N	N	N	N	N
8	R	R	R	R	R	R	R	R	R	R	R	R	R	N
7	R	R	R	R	R	R	N	N	N	N	N	N	N	N
6	R	R	R	R	R	R	R	R	R	R	R	R	R	N
5	R	R	R	R	R	R	R	R	N	N	N	N	N	N
4	R	R	R	R	R	R	R	R	R	R	R	R	R	N
3	R	R	R	R	R	R	R	R	R	R	N	N	N	N
2	R	R	R	R	R	R	R	R	R	R	R	E	R	R
1	R	R	R	R	R	R	R	R	R	R	R	R	N	N

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Gegenseitiger Ausschluß: Algorithmus von Singhal

L4.1.4 Lemma

Wenn

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

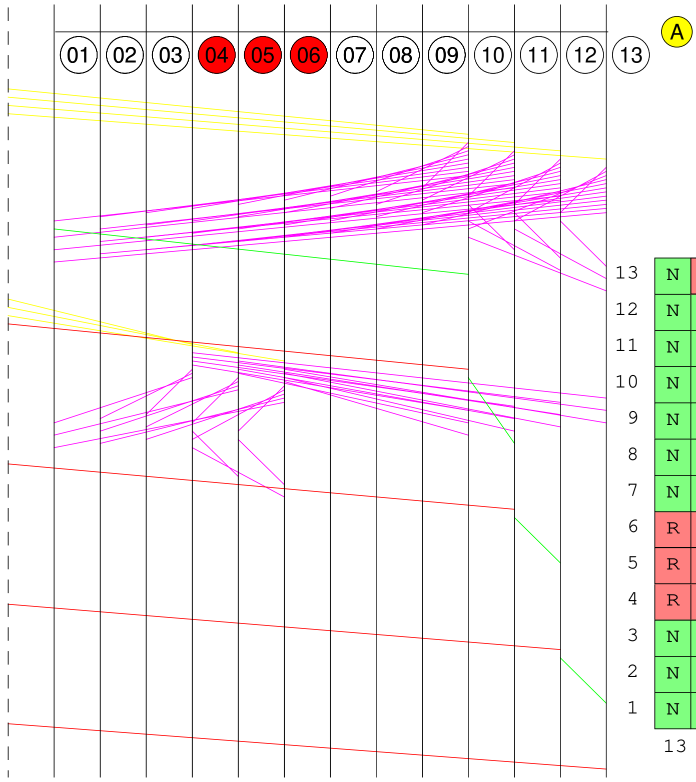
dann gilt:

$$\forall \mathbf{j}, \mathbf{k}(\mathbf{1} < \mathbf{j} \leq \mathbf{m} \wedge \mathbf{1} \leq \mathbf{k} < \mathbf{j} \Rightarrow \mathbf{R}_{\mathbf{ij}}^{\mathbf{ik}})$$

$$\forall \mathbf{j}, \mathbf{k}(\mathbf{1} \leq \mathbf{j} < \mathbf{m} \wedge \mathbf{j} < \mathbf{k} \leq \mathbf{m} \Rightarrow \mathbf{N}_{\mathbf{ij}}^{\mathbf{ik}})$$

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Gegenseitiger Ausschluß: Algorithmus von Singhal



	13	12	11	10	9	8	7	6	5	4	3	2	1	T
13	N	R	R	R	R	R	R	R	R	R	R	R	R	N
12	N	N	R	R	R	R	R	R	R	R	R	R	R	N
11	N	N	N	R	R	R	R	R	R	R	R	R	R	N
10	N	N	N	N	R	R	R	R	R	R	R	R	R	N
9	N	N	N	N	N	N	N	N	N	N	N	N	N	N
8	N	N	N	N	R	N	N	N	N	N	N	N	N	N
7	N	N	N	N	R	R	N	N	N	N	N	N	N	N
6	R	R	R	R	R	R	R	R	R	R	R	R	R	R
5	R	R	R	R	R	R	R	R	R	R	R	R	R	R
4	R	R	R	R	R	R	R	R	R	R	R	R	R	R
3	N	N	N	N	R	R	R	R	R	R	N	N	N	N
2	N	N	N	N	R	R	R	R	R	R	R	N	N	N
1	N	N	N	N	R	R	R	R	R	R	R	R	N	N

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Gegenseitiger Ausschluß: Algorithmus von Singhal

L4.1.5 Lemma

Wenn

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

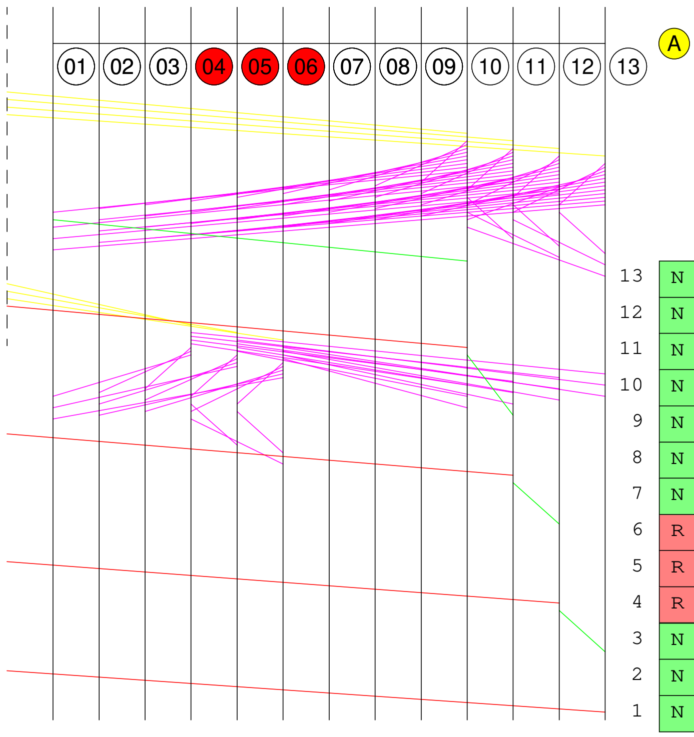
dann gilt:

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

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

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Gegenseitiger Ausschluß: Algorithmus von Singhal



	13	12	11	10	9	8	7	6	5	4	3	2	1	T
13	N	R	R	R	R	R	R	R	R	R	R	R	R	N
12	N	N	R	R	R	R	R	R	R	R	R	R	R	N
11	N	N	N	R	R	R	R	R	R	R	R	R	R	N
10	N	N	N	N	R	R	R	R	R	R	R	R	R	N
9	N	N	N	N	N	N	N	N	N	N	N	N	N	N
8	N	N	N	N	R	N	N	N	N	N	N	N	N	N
7	N	N	N	N	R	R	N	N	N	N	N	N	N	N
6	R	R	R	R	R	R	R	R	R	R	R	R	R	R
5	R	R	R	R	R	R	R	R	R	R	R	R	R	R
4	R	R	R	R	R	R	R	R	R	R	R	R	R	R
3	N	N	N	N	R	R	R	R	R	R	N	N	N	N
2	N	N	N	N	R	R	R	R	R	R	R	N	N	N
1	N	N	N	N	R	R	R	R	R	R	R	R	N	N

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

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}, ..., 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}, ..., P_{j1} haben nie Anforderungen gestellt und $|\mathbf{P_{jr}}| < |\mathbf{P_{jr-1}}| < \ldots < |\mathbf{P_{j1}}|$.
- Kein Prozeß fordert das Token an.

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

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

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

S4.3 **Satz**

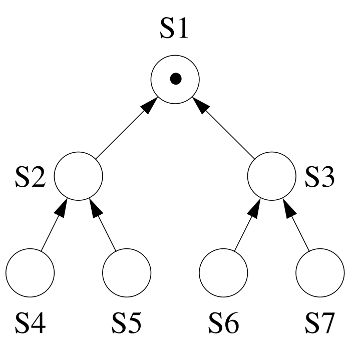
Wenn das System im Anfangszustand startet und zu jedem Zeitpunkt folgende Eigenschaften besitzt:

- 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.

Dann bildet die Gesamtheit der Prozesse eine vollständig geordnete Sequenz.

4.2.3 **Algorithmus von Raymond**

Die Prozesse sind als logischer Baum angeordnet



❑ **Datenstrukturen**

- Lokale Datenstruktur des Prozesses P_i

  int holder; // Nummer des Vorgängers!
  // Bei der Wurzel: eigene Nummer

  int_fifo queue; // Warteschlange von
  // Anforderern

Gegenseitiger Ausschluß: Algorithmus von Raymond

- **Anforderung des Tokens**
  - 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 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 der 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.

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Gegenseitiger Ausschluß: Algorithmus von Raymond

- **Ausführung eines kritischen Abschnitts**
  - 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.

- **Verlassen eines kritischen Abschnitts**
  - 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 seine eigene Warteschlange noch nicht leer ist, wird eine REQUEST-Nachricht an den in holder vermerkten Prozeß gesandt.

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig