LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Prinzipien, Modelle und Algorithmen der Nebenläufigen Programmierung
Wintersemester 2019/20

Synchronisation Teil I: Modellannahmen und das Mutual-Exclusion-Problem

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 27. November 2019

Übersicht

1. Modell
   - Nebenläufiges Programm
   - Interleaving
   - Fairness
   - Weitere
2. Das Mutual-Exclusion-Problem für 2 Prozesse
   - Einleitendes Beispiel: Too much milk
   - Formalisierung des Mutual-Exclusion-Problems
   - Algorithmen für 2 Prozesse mit atomarem Read & Write
3. Das Mutual-Exclusion-Problem für n Prozesse
   - Tournament-Algorithmen
   - Lamports Algorithmus
   - Bakery-Algorithmus
4. Drei Komplexitätsresultate
   - Eine genaue Schranke für den Platzbedarf
   - Ein Resultat zur Laufzeit

Modellannahmen: Nebenläufiges Programm

Nebenläufiges Programm

- Nebenläufiges Programm: *Endliche Menge von Prozessen*
- Prozess: *Sequentielles Programm aus atomaren Berechnungsschritten*

- Annahme für unsere sämtlichen Modelle
- Definition von Berechnungsschritt modellabhängig!

Modellannahmen: Interleaving-Annahme

Interleaving-Annahme

Ausführung eines nebenläufigen Programms:

Sequenz der atomaren Berechnungsschritte der Prozesse, die beliebig durchmischt sein können.

- Nur ein Berechnungsschritt gleichzeitig, d.h. zwei Schritte überlappen nie
- (Fast) beliebiges Durchmischen (Interleaving) von Berechnungsschritten der einzelnen Prozesse
- (Fast) alle Ausführungsreihenfolgen erlaubt
- „Fast“ wird später erklärt

Warum ist die Interleaving-Annahme sinnvoll?

Granularität

Berechnungsschritte können beliebig definiert werden (d.h. beliebig klein)

Multitasking-Systeme

- Einprozessor-Systeme mit Multitasking machen Interleaving
- Allerdings: Reale Implementierungen benutzen meist Verfahren (z.B. Zeitscheiben), die nicht jede Reihenfolge zulassen.
- "Prozess führt eine Zeit lang Schritte aus, dann wird gewechselt"
- Aber: "eine Zeit lang" problematisch:
  - Formal schwer zu fassen, kompliziert
  - Neue (schnellere) Hardware erfordert Modellanpassung
  - Berücksichtigung von Umwelteinflüssen (z.B. Phasenschwankungen)
- Alle Reihenfolgen umfassen auch die kleinere Menge der realen Reihenfolgen (Korrektheit aller impliziert Korrektheit eines Teils)

Warum ist die Interleaving-Annahme sinnvoll? (2)

Multiprozessor-Systeme

- Annahme scheint unrealistisch, parallele und überlappende Berechnungsschritte möglich
- Aber: Gemeinsame Ressourcen (z.B. Speicher), sind auf Hardwareebene vor parallelen Zugriffen geschützt und erzwingen Sequentialisierung
- Ressourcen-autonome Schritte sind immer noch parallel
- Das ist aber nicht sichtbar!

Modellannahmen: Die Fairness-Annahme

- vorhin: . . . Fast beliebiges Interleaving . . .
- Die Einschränkung kommt durch die Fairness-Annahme
- Verschieden starke Annahme von Fairness
- Unser Begriff relativ schwach

Fairness-Annahme

Jeder Prozess für den ein Berechnungsschritt möglich ist, führt in der Gesamt-Auswertungssequenz diesen Schritt nach endlich vielen Berechnungsschritten durch.

Beispiel zur Fairness

(X am Anfang 0)

Prozess P:	Prozess Q:
(P1) `if` $X \neq 10$ `then goto` (P1);	(Q1) $X{:=}10$;

Auswertungssequenz die keine Fairness beachtet

(P1), (P1), (P1), (P1), (P1), (P1), . . . unendlich so weiter

Beachte: Die Fairness-Annahme macht nur eine Aussage über unendliche Auswertungsreihenfolgen

Auswertungssequenzen unter Beachtung von Fairness

$\underbrace{(P1),(P1),\ldots,(P1)}_{n\text{-mal}}(Q1)(P1)$ für beliebiges $n \in \mathbb{N}_0$

Beispiel zur Fairness (2)

(X am Anfang 0)

Prozess P:	Prozess Q:
(P1) `if` $X \neq 10$ `then goto` (P1);	(Q1) X:=10;

Folgerung

Unter Einhaltung der Fairness-Annahme:
Obiges Programm terminiert immer
(Da (Q1) irgendwann ausgeführt werden muss)

Noch ein Beispiel zur Fairness

(Der Wert von X vor der Ausführung sei 0)

Prozess P:	Prozess Q:
(P1) `if X` $\neq$ 10 `then goto` (P1); (P2) `goto` (P1)	(Q1) X:=10;

Erlaubte Sequenzen

$\underbrace{(P1),(P1),\ldots,(P1)}_{n\text{-mal}}(Q1)\underbrace{(P1),(P2),(P1),(P2),\ldots}_{\text{unendlich oft}}$ mit $n \in \mathbb{N}_0$

Verbotene Sequenzen

$(P1),(P1),(P1),\ldots$ unendlich lange

Weitere Annahmen

Bekannte Prozesse

- Ein Programm besteht aus einer festen Anzahl von Prozessen
- Es werden keine Prozesse vom Programm neu erzeugt
- Prozesse sind identifizierbar

Achtung!

Diese Annahme werden wir nicht für die ganze Vorlesung beibehalten!

Weitere Annahmen (2)

Programmiersprache

- Zunächst gehen wir von einer kleinen Pseudo-Programmiersprache aus, die imperativ ist.
- Befehle: `if-then`, `goto`, Zuweisungen $X := 10$, `while`-Schleifen usw.
- Jede nummerierte Zeile wird atomar ausgeführt.
- erlaubte Operationen auf dem Speicher werden wir festlegen

Das Problem vom wechselseitigen Ausschluss

Das Mutual-Exclusion Problem

Einleitendes Beispiel: Too much milk

- Alice und Bob teilen sich einen Kühlschrank
- Sie wollen, dass stets genug Milch im Kühlschrank
- Weder zuviel noch zu wenig Milch.

Naive Lösung

Wenn jemand sieht, dass keine Milch im Kühlschrank ist, geht er Milch kaufen.

Naive Lösung funktioniert nicht

	Alice	Bob
17:00	kommt nach hause	
17:05	bemerkt: keine Milch im Kühlschrank	
17:10	geht zum Supermarkt	
17:15		kommt nach hause
17:20		bemerkt: keine Milch im Kühlschrank
17:25		geht zum Supermarkt
17:30	kommt vom Supermarkt zurück, packt Milch in den Kühlschrank	
17:40		kommt vom Supermarkt zurück, packt Milch in den Kühlschrank

$\Longrightarrow$ **Zuviel Milch im Kühlschrank**

Too much milk: Spezifikation

Alice und Bob vereinbaren, Notizen am Kühlschrank zu hinterlassen, die beide lesen können.

Dadurch soll sichergestellt werden, dass:

- Höchstens eine der beiden Personen geht Milch kaufen, wenn keine Milch im Kühlschrank ist.
- Eine der beiden Personen geht Milch kaufen, wenn keine Milch im Kühlschrank ist.

Nicht erlaubt: Nur Bob kauft Milch. Weil: Alice dann u.U. unendlich lange auf Milch wartet. (Verletzt 2. Bedingung)

Annahme: Alice und Bob können sich nicht sehen, d.h. sie kommunizieren nur über die Notizen

1. Lösungsversuch

Wenn keine Notiz am Kühlschrank und keine Milch im Kühlschrank ist, dann schreibe Notiz an den Kühlschrank, gehe Milch kaufen, stelle die Milch in den Kühlschrank und entferne danach die Notiz am Kühlschrank.

Als Prozesse:

Programm für Alice:

```
(A1) if keine Notiz then
(A2)   if keine Milch then
(A3)     schreibe Notiz;
(A4)     kaufe Milch;
(A5)     entferne Notiz;
```

Programm für Bob:

```
(B1) if keine Notiz then
(B2)   if keine Milch then
(B3)     schreibe Notiz;
(B4)     kaufe Milch;
(B5)     entferne Notiz;
```

2. Lösungsversuch

Hinterlasse als erstes eine Notiz am Kühlschrank, dann prüfe ob eine Notiz des anderen vorhanden ist. Nur wenn keine weitere Notiz vorhanden ist, prüfe ob Milch vorhanden ist und gehe Milch kaufen, wenn keine Milch vorhanden ist. Danach entferne die Notiz.

Als Prozesse:

Programm für Alice:

```
(A1) schreibe Notiz "Alice";
(A2) if keine Notiz von Bob then
(A3)   if keine Milch then
(A4)     kaufe Milch;
(A5) entferne Notiz "Alice";
```

Programm für Bob:

```
(B1) schreibe Notiz "Bob";
(B2) if keine Notiz von Alice then
(B3)   if keine Milch then
(B4)     kaufe Milch;
(B5) entferne Notiz "Bob";
```

Alice Bob Keine Milch!

3. Lösungsversuch

Alice

Wie vorher: Schreibe Notiz, wenn keine Notiz von Bob, prüfe ob Milch vorhanden und gehe Milch kaufen. Entferne Notiz.

Bob

Schreibe Notiz, warte bis keine Notiz von Alice am Kühlschrank hängt, dann prüfe, ob Milch leer ist, gehe Milch kaufen, entferne eigene Notiz.

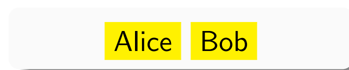
Als Prozesse:

Programm für Alice:

```
(A1) schreibe Notiz "Alice";
(A2) if keine Notiz von Bob then
(A3)   if keine Milch then
(A4)     kaufe Milch;
(A5) entferne Notiz "Alice";
```

Programm für Bob:

```
(B1) schreibe Notiz "Bob";
(B2) while Notiz von Alice do skip;
(B3) if keine Milch then
(B4)   kaufe Milch;
(B5) entferne Notiz "Bob";
```



Eine korrekte Lösung

Programm für Alice

```
(A1) schreibe a1;
(A2) if b2
(A3)   then schreibe a2;
(A4)   else entferne a2;
(A5) while
       b1 ∧ ((a2 ∧ b2) ∨ (¬a2 ∧ ¬b2))
(A6)   do skip;
(A7) if keine Milch
(A8)   then kaufe Milch;
(A9) entferne a1;
```

Programm für Bob

```
(B1) schreibe b1;
(B2) if ¬a2
(B3)   then schreibe b2;
(B4)   else entferne b2
(B5) while
       a1 ∧ ((a2 ∧ ¬b2) ∨ (¬a2 ∧ b2))
(B6)   do skip;
(B7) if keine Milch
(B8)   then kaufe Milch;
(B9) entferne b1;
```

a1	a2	b2	b1	prüft Milch
✓	dc	dc	×	Alice
×	dc	dc	✓	Bob

dc = don't care

a1	a2	b2	b1	prüft Milch
✓	×	✓	✓	Alice
✓	✓	×	✓	Alice
✓	✓	✓	✓	Bob
✓	×	×	✓	Bob

Im Folgenden:

Mutual-Exclusion Problem: Formalisierung

Definition des Problems und einer Lösung
Annahmen über das Modell
Deadlock- bzw. Starvation-Freiheit

Das Mutual-Exclusion-Problem

- Mutual-Exclusion-Problem =
  Garantie, dass der exklusive Zugriff auf eine gemeinsam genutzte Ressource sichergestellt wird
- Zugriffscode wird als kritischer Abschnitt bezeichnet.
- Race Condition: Situationen in denen mehrere Prozesse auf eine gemeinsame Ressource zugreifen und der Wert der Ressource hängt von Ausführungsreihenfolge ab.
- Ziel des Mutual-Exclusion-Problems:
  Vermeide Race Conditions.

Das Mutual-Exclusion-Problem, formales Modell

Code-Struktur jedes Prozesses

```
loop forever
    restlicher Code
    Initialisierungscode
    Kritischer Abschnitt
    Abschlusscode
end loop
```


Das Mutual-Exclusion-Problem, formales Modell (2)

Annahmen

- Im restlichen Code kann ein Prozess keinen Einfluss auf andere Prozesse nehmen. Ansonsten ist dort alles erlaubt (Endlosschleifen usw.)
- Ressourcen (Programmvariablen), die im Initialisierungscode oder Abschlusscode benutzt werden, werden durch den Code im Kritischen Abschnitt und den restlichen Code nicht verändert.

```
loop forever
    restlicher Code
    Initialisierungscode
    Kritischer Abschnitt
    Abschlusscode
end loop
```

Das Mutual-Exclusion-Problem, formales Modell (3)

Annahmen (Fortsetzung)

- Keine Fehler bei Ausführung des Initialisierungscode, des Codes im kritischen Abschnitt und im Abschlusscode
- Code im kritschen Abschnitt und im Abschlusscode: Nur endlich viele Ausführungsschritte (keine Schleifen!). Impliziert (wegen Fairness): Prozess verlässt nach Betreten des kritischen Abschnitt diesen und führt Abschlusscode durch.

```
loop forever
    restlicher Code
    Initialisierungscode
    Kritischer Abschnitt
    Abschlusscode
end loop
```


Korrektheitskriterien

Lösung des Mutual-Exclusion-Problems

Fülle Initialisierungs- und Abschlusscode, so dass die folgenden Anforderungen erfüllt sind:

- Wechselseitiger Ausschluss: Es sind niemals zwei oder mehr Prozesse zugleich in ihrem kritischen Abschnitt.
- Deadlock-Freiheit: Wenn ein Prozess seinen kritischen Abschnitt betreten möchte, dann betritt irgendein Prozess schließlich den kritischen Abschnitt.

Korrektheitskriterien (2)

Stärkere Forderung gegenüber Deadlock-Freiheit:

Starvation-Freiheit

- Starvation = Verhungern
- Starvation-Freiheit: Wenn ein Prozess seinen kritischen Abschnitt betreten möchte, dann muss er ihn nach endlich vielen Berechnungsschritten betreten.
- Ein Starvation-freier Algorithmus ist auch Deadlock-frei.

Formale Beweistechniken

- Wechselseitiger Ausschluss ist eine sog. Sicherheitseigenschaft (Safety Property), eine Eigenschaft die immer erfüllt sein muss.
- Deadlock- bzw. Starvation-Freiheit sind sog. Lebendigkeitseigenschaften (Liveness Property), Eigenschaften die irgendwann erfüllt sein müssen.
- Mittels Temporallogik lässt sich formal nachweisen, dass diese Eigenschaften erfüllt sind.
- Z.B. kann Model-Checking für den Beweis verwendet werden
- Wir: Korrektheitsbeweis eher informell.

Wir betrachten:

Lösungen zum Mutual-Exclusion Problem

bei 2 Prozessen

und atomarem Lesen und Schreiben

Algorithmen für 2 Prozesse mit atomarem Read+Write

Im Folgenden:

- Einige Algorithmen zum Mutual-Exclusion Problem
- Nebenläufiges Programm mit genau zwei Prozessen

Annahmen:

- Als atomare Speicheroperationen des gemeinsamen Speichers nur Lesen und Schreiben
- Z.B. verboten swap(X,Y), X := Y , etc.

Abkürzung:

- `await` *Bedingung*;
  anstelle von
  `while` ¬*Bedingung* `do skip;`

Dekkers Algorithmus

- Historisch die erste korrekte Lösung
- Theodorus Jozef Dekker = Holländischer Mathematiker
- Erwähnung des Dekker-Algorithmus: Dijkstra 1965
- Nicht ganz einfache Lösung

Algorithmus von Dekker

Initial: wantp = False, wantq = False, turn =1

Prozess P:

```
loop forever
(P1)    restlicher Code
(P2)    wantp := True;
(P3)    while wantq do
(P4)        if turn=2 then
(P5)            wantp := False;
(P6)            await turn=1;
(P7)            wantp := True;
(P8)    Kritischer Abschnitt
(P9)    turn := 2;
(P10)   wantp := False;
end loop
```

Prozess Q:

```
loop forever
(Q1)    restlicher Code
(Q2)    wantq := True;
(Q3)    while wantp do
(Q4)        if turn=1 then
(Q5)            wantq := False
(Q6)            await turn=2
(Q7)            wantq := True;
(Q8)    Kritischer Abschnitt
(Q9)    turn := 1;
(Q10)   wantq :=False;
end loop
```

Der Dekker-Algorithmus ist korrekt (1)

Lemma

Der Algorithmus von Dekker erfüllt den wechselseitigen Ausschluss.

Beweis:

Annahme: Aussage ist falsch.
$\Longrightarrow$ $\exists$ Zustand mit P und Q im kritischen Abschnitt
Drei Fälle:

- P und Q sind nie durch den Schleifenkörper gelaufen. D.h.
  - ohne Q-Befehle …(P3), (P8) …, und
  - ohne P-Befehle …(Q3), (Q8) ….

  Unmöglich, da dann wantp und wantq falsch sein müssen.

```
Initial: wantp = False,
         wantq = False,
         turn =1
Prozess P:
 loop forever
 (P1)     restlicher Code
 (P2)     wantp := True;
 (P3)     while wantq do
 (P4)         if turn=2 then
 (P5)             wantp := False;
 (P6)             await turn=1;
 (P7)             wantp := True;
 (P8)     Kritischer Abschnitt
 (P9)     turn := 2;
 (P10)    wantp := False
 end loop
Prozess Q:
 loop forever
 (Q1)     restlicher Code
 (Q2)     wantq := True;
 (Q3)     while wantp do
 (Q4)         if turn=1 then
 (Q5)             wantq := False;
 (Q6)             await turn=2;
 (Q7)             wantq := True;
 (Q8)     Kritischer Abschnitt
 (Q9)     turn := 1;
 (Q10)    wantq :=False
 end loop
```


Der Dekker-Algorithmus ist korrekt (2)

Lemma

Der Algorithmus von Dekker erfüllt den wechselseitigen Ausschluss.

Beweis:

- Ein Prozess hat den Schleifenkörper durchlaufen, während der andere im kritischen Abschnitt ist.
  Der durch die Schleife laufende Prozess wird nie die while-Bedingung zu False auswerten, solange der andere Prozess im KA ist. Also: Unmöglich.
- Beide Prozesse durchlaufen den Schleifenkörper.
  $\Longrightarrow$ Ein Prozess bleibt am `await` hängen (turn wird erst nach dem kritischen Abschnitt verändert). Unmöglich.

D.h. Annahme falsch, wechselseitiger Ausschluss erfüllt

```
Initial: wantp = False,
         wantq = False,
         turn =1
Prozess P:
 loop forever
 (P1)     restlicher Code
 (P2)     wantp := True;
 (P3)     while wantq do
 (P4)         if turn=2 then
 (P5)             wantp := False;
 (P6)             await turn=1;
 (P7)             wantp := True;
 (P8)     Kritischer Abschnitt
 (P9)     turn := 2;
 (P10)    wantp := False;
 end loop
Prozess Q:
 loop forever
 (Q1)     restlicher Code
 (Q2)     wantq := True;
 (Q3)     while wantp do
 (Q4)         if turn=1 then
 (Q5)             wantq := False;
 (Q6)             await turn=2;
 (Q7)             wantq := True;
 (Q8)     Kritischer Abschnitt
 (Q9)     turn := 1;
 (Q10)    wantq :=False
 end loop
```


Der Dekker-Algorithmus ist korrekt (3)

Lemma

Der Algorithmus von Dekker ist Starvation-frei.

Beweis:

Annahme: Aussage ist falsch. O.B.d.A. Prozess P verhungert im Initialisierungscode. Mögliche Gründe:
(1) Hängenbleiben am `await`. Wo befindet sich Q?

- Q ist im restlichen Code. Unmöglich, da wantq mal True gewesen sein muss, impliziert dass: Q hat KA durchlaufen und Q hätte dann im Ausgangscode turn auf 1 gesetzt.
- Q ist im kritischen Abschnitt oder Abschlusscode $\Rightarrow$ turn auf 1 und turn kann nicht mehr auf 2 gesetzt werden
- Q is im Initialisierungscode. wantp muss falsch sein, da P in (P6) ist.
  $\Rightarrow$ Q betritt kritischen Abschnitt, setzt danach turn = 1.

```
Initial: wantp = False,
         wantq = False,
         turn =1
Prozess P:
 loop forever
 (P1)     restlicher Code
 (P2)     wantp := True;
 (P3)     while wantq do
 (P4)         if turn=2 then
 (P5)             wantp := False
 (P6)             await turn=1
 (P7)             wantp := True;
 (P8)     Kritischer Abschnitt
 (P9)     turn := 2;
 (P10)    wantp := False;
 end loop
Prozess Q:
 loop forever
 (Q1)     restlicher Code
 (Q2)     wantq := True;
 (Q3)     while wantp do
 (Q4)         if turn=1 then
 (Q5)             wantq := False
 (Q6)             await turn=2
 (Q7)             wantq := True;
 (Q8)     Kritischer Abschnitt
 (Q9)     turn := 1;
 (Q10)    wantq :=False;
 end loop
```


Der Dekker-Algorithmus ist korrekt (4)

Lemma

Der Algorithmus von Dekker ist Starvation-frei.

Beweis (Forts.):

(2) `while`-Schleife wird unendlich oft durchlaufen:

- Kein Hängenbleiben am await, daher turn = 1
- Wert von turn kann nur durch P im Abschlusscode verändert werden
- Q ist im restlichen Code. Dann ist wantq = False, while wird nicht unendlich oft durchlaufen
- Q ist im kritischen Abschnitt oder im Abschlusscode. Dann wird irgendwann wantq = False gesetzt und daher while nicht unendlich oft durchlaufen.
- Q im Initialisierungscode: Dann muss Q am await hängen bleiben (da turn = 1). Dann ist wantq = False , while wird nicht unendlich oft durchlaufen.

```
Initial: wantp = False,
         wantq = False,
         turn =1
Prozess P:
 loop forever
 (P1)     restlicher Code
 (P2)     wantp := True;
 (P3)     while wantq do
 (P4)         if turn=2 then
 (P5)             wantp := False
 (P6)             await turn=1
 (P7)             wantp := True;
 (P8)     Kritischer Abschnitt
 (P9)     turn := 2;
 (P10)    wantp := False;
 end loop
Prozess Q:
 loop forever
 (Q1)     restlicher Code
 (Q2)     wantq := True;
 (Q3)     while wantp do
 (Q4)         if turn=1 then
 (Q5)             wantq := False
 (Q6)             await turn=2
 (Q7)             wantq := True;
 (Q8)     Kritischer Abschnitt
 (Q9)     turn := 1;
 (Q10)    wantq :=False;
 end loop
```

Algorithmus von Peterson

- Einfachere Variante von Dekkers Algorithmus
- Artikel von Gary L. Peterson 1981
- Algorithmus und Korrektheitsbeweis in 2 Seiten

Algorithmus von Peterson

Initial: wantp = False, wantq = False, turn egal

Prozess P:

```
 loop forever
(P1)  restlicher Code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False or turn = 2
(P5)  Kritischer Abschnitt
(P6)  wantp := False;
 end loop
```

Prozess Q:

```
 loop forever
(Q1)  restlicher Code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False or turn = 1
(Q5)  Kritischer Abschnitt
(Q6)  wantq := False;
 end loop
```


Der Peterson-Algorithmus ist korrekt (1)

Lemma

Der Algorithmus von Peterson garantiert wechselseitigen Ausschluss

Beweis:

Annahme: Aussage ist falsch.

- Es gibt einen Zustand, so dass P und Q im kritischen Abschnitt sind.
- Die Tests in (P4) und (Q4) können nicht direkt nacheinander wahr geworden sein, d.h. weder (P4),(Q4),(?5),(?5) noch (P4),(P5),(Q4),(Q5) (analog mit P,Q umgekehrt)
  Denn: turn hätte dann nur einen der beiden Prozesse durchgelassen (wantp und wantq müssen beide wahr sein)

Initial: wantp = False,wantq = False,turn =egal

Prozess P:

```
 loop forever
(P1)  restlicher Code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False
        or turn = 2
(P5)  Kritischer Abschnitt
(P6)  wantp := False;
 end loop
```

Prozess Q:

```
 loop forever
(Q1)  restlicher Code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False
        or turn = 1
(Q5)  Kritischer Abschnitt
(Q6)  wantq := False;
 end loop
```


Der Peterson-Algorithmus ist korrekt (1)

Lemma

Der Algorithmus von Peterson garantiert wechselseitigen Ausschluss

Beweis:

- Also: Zweiter Prozess der kritischen Abschnitt betrat muss mind. 1 Befehl durchgeführt haben nachdem der erste Prozess den kritischen Abschnitt betreten hat. (P5),. . . ,(Q3),(Q4),(Q5) bzw. analog wenn Q zuerst im KA
- (Q3) (bzw. (P3) setzt aber den Wert von turn so, dass der setzende Prozess Q (bzw. P) nicht über die await-Bedingung hinweg kommt.

Initial: wantp = False,wantq = False,turn =egal

Prozess P:

```
 loop forever
(P1)  restlicher Code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False
        or turn = 2
(P5)  Kritischer Abschnitt
(P6)  wantp := False;
 end loop
```

Prozess Q:

```
 loop forever
(Q1)  restlicher Code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False
        or turn = 1
(Q5)  Kritischer Abschnitt
(Q6)  wantq := False;
 end loop
```

Der Peterson-Algorithmus ist korrekt (3)

Lemma

Der Algorithmus von Peterson ist Starvation-frei.

Beweis:

Nehme an, das sei falsch. $\implies$ Ein Prozess bleibt am `await` hängen. Anderer Prozess:

1. Er verbleibt für immer in seinem restlichen Code. $\implies$ Seine want-Variable falsch. Unmöglich.
2. Er verbleibt auch für immer in seinem Initalisierungscode. Unmöglich, da `await`-Bedingung für einen stets wahr
3. Er durchläuft wiederholend seinen kritischen Abschnitt. Unmöglich, da er nach dem ersten Durchlaufen, beim zweiten Ausführen des Eingangscode die turn-Variable umsetzt.

Initial: wantp = False,wantq = False,turn =egal

Prozess P:

```
loop forever
(P1)  restlicher Code
(P2)  wantp := True;
(P3)  turn := 1;
(P4)  await wantq = False
        or turn = 2
(P5)  Kritischer Abschnitt
(P6)  wantp := False;
end loop
```

Prozess Q:

```
loop forever
(Q1)  restlicher Code
(Q2)  wantq := True;
(Q3)  turn := 2;
(Q4)  await wantp = False
        or turn = 1
(Q5)  Kritischer Abschnitt
(Q6)  wantq := False;
end loop
```


Kessels' Algorithmus

- Variante von Petersons Algorithmus
- Veröffentlicht 1982
- Vorteil gegenüber Dekkers/Petersons Algorithmus:
  - Single-Writer, Multiple-Reader Algorithmus, d.h.
  - Gemeinsame Speicherplätze werden nur von einem Prozess beschrieben
  - (aber von mehreren gelesen)
  - Ermöglicht z.B. einfache Implementierung in Message-Passing Modellen

Idee der Abänderung von Petersons Algorithmus

- Peterson benutzt gemeinsame Variable `turn`, die beide Prozesse lesen und schreiben
- Idee: Benutze zwei Variablen `turnp` und `turnq`, so dass
  - Nur Prozess P beschreibt `turnp`
  - Nur Prozess Q beschreibt `turnq`
  - Der eigentliche Wert von turn wird berechnet als:

$$\text{turn} = 1, \quad \text{wenn turnp} = \text{turnq}$$
$$\text{turn} = 2, \quad \text{wenn turnp} \neq \text{turnq}$$

Algorithmus von Kessels

Initial: wantp = False, wantq = False, turnp, turnq egal

Prozess P:

```
loop forever
(P1)  restlicher Code
(P2)  wantp := True;
(P3)  localp := turnq;
(P4)  turnp := localp;
(P5)  await
        wantq = False
        or localp ≠ turnq
(P6)  Kritischer Abschnitt
(P7)  wantp := False;
end loop
```

Prozess Q:

```
loop forever
(Q1)  restlicher Code
(Q2)  wantq := True;
(Q3)  localq := if turnp = 1
                  then 2
                  else 1;
(Q4)  turnq := localq;
(Q5)  await
        wantp = False
        or localq = turnp
(Q6)  Kritischer Abschnitt
(Q7)  wantq := False;
end loop
```

Eigenschaften von Kessels' Algorithmus

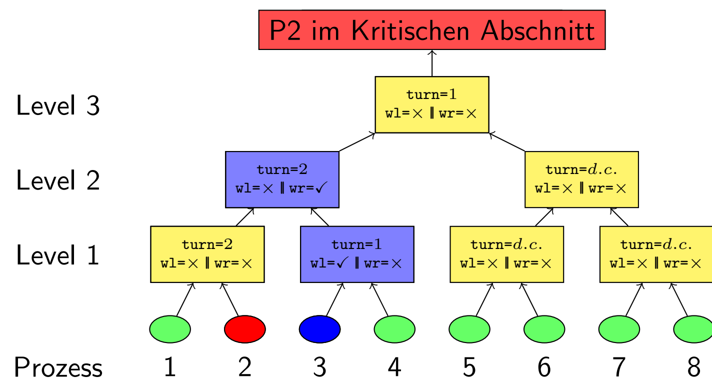
Satz
Der Algorithmus von Kessels garantiert wechselseitigen Ausschluss und ist Starvation-frei.

Beweis: Analog zum Peterson-Algorithmus, wobei mehr Fallunterscheidungen nötig sind (da mehr Schritte)

Mutual-Exclusion für n Prozesse

- Gesucht: Algorithmen die für n Prozesse funktionieren.
- Annahme: n ist bekannt und bleibt konstant.
- Einfache Idee:
  Benutze die Algorithmen für 2 Prozesse "Baum-artig"

Idee des Tournament-Algorithmus (basierend auf Peterson-Alg.)



turn: 2 = linkes Kind darf durch, 1 = rechtes Kind darf durch

wl = want des linken Kindes, wr = want des rechten Kindes

Eigenschaften der Tournament-Algorithmen

- Einfach zu implementieren
- Wechselseitiger Ausschluss und Starvation-Freiheit, wenn Algorithmus für 2 Prozesse diese Eigenschaften hat.
- Nachteil: $\lceil \log_2 n \rceil$-maliges Ausführen des Initialisierungs- und Ausgangscodes
- Auch dann, wenn nur ein einzelner Prozess in den kritischen Abschnitt will!

Lamports Algorithmus

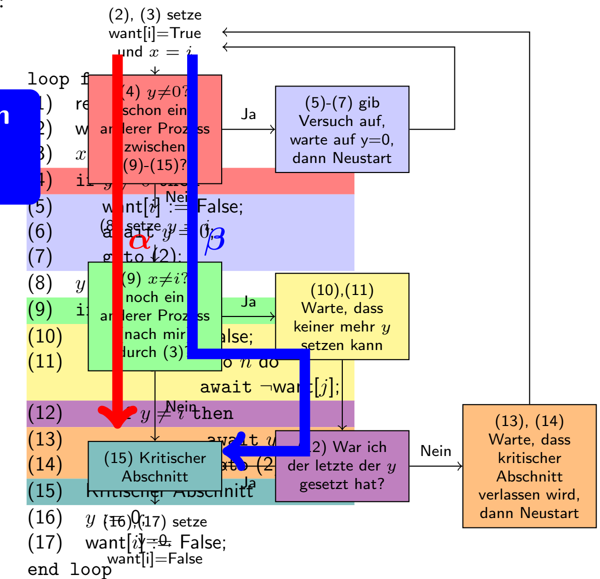
- Leslie Lamport 1987
- Schneller Algorithmus für N Prozesse
- schnell = Wenn nur ein Prozess in den kritischen Abschnitt will, dann nur konstante Laufzeit

Lamports Algorithmus: Programm des i. Prozesses

Initial: y=0, für alle $i \in \{1,\ldots,n\}$:
want[i]=False, Wert von x egal

Zwei Möglichkeiten, den Kritischen Abschnitt zu erreichen!

```
loop forever
(5)      want[i] := False;
(6)      await y = 0;
(7)      goto (2);
(8)   y := i;
(9)   if x ≠ i then
(10)     want[i] := False;
(11)     for j := 1 to n do
                 await ¬want[j];
(12)     if y ≠ i then
(13)             await y = 0;
(14)             goto (2);
(15)  Kritischer Abschnitt
(16)  y := 0;
(17)  want[i] := False;
end loop
```



Lamports Algorithmus: Einfache Eigenschaften

Satz

Für Lamports Algorithmus gilt:

- Wenn ein einzelner Prozess in den Kritischen Abschnitt möchte, dann führt er nur konstant viele Operationen durch.
- Lamports Algorithmus ist nicht Starvation frei.

Beweis: Folgt direkt aus dem Algorithmus / Flussdiagramm.

Lamports Algorithmus garantiert Mutual-Exclusion (1)

Satz

Lamports Algorithmus garantiert wechselseitigen Ausschluss.

Beweis durch Widerspruch:

Annahme: Prozess i und Prozess j sind gleichzeitig im kritischen Abschnitt, wobei i zuerst im kritischen Abschnitt war.

Fallunterscheidung:

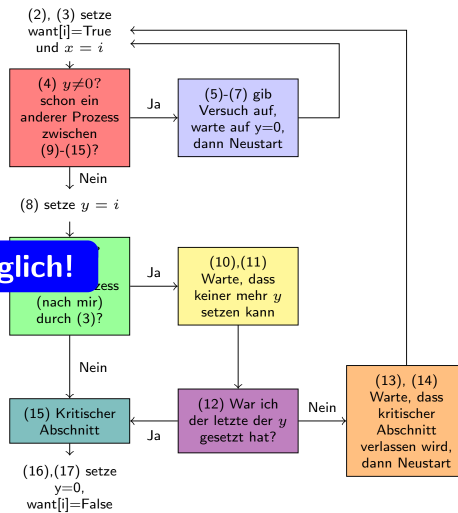
- Prozess i erreicht den kritischen Abschnitt entlang Pfad α.
- Prozess i erreicht den kritischen Abschnitt entlang Pfad β.

Lamports Algorithmus garantiert Mutual-Exclusion (2)

Fall 1: 2 Prozesse betreten den kritischen Abschnitt, erster entlang α:

- i: erster Prozess (entlang α)
- j: zweiter Prozess (entlang α oder β)
- Wenn i durch (9) läuft gilt:
  $x = i$ und $y \neq 0$.
- Impliziert:
  (1) j noch nicht durch (3), oder
  (2) j durch (3) bevor i durch (3)
- Fall (1): Da j vor (4):
  – j wird in (5)-(7) geschickt
  – und wartet bis $y = 0$,
  D.h. j kann nicht mehr in KA
- Fall (2): Wenn j durch (9), gilt $x \neq j$.
  D.h. kein Eingang entlang α.
  Entlang β: an der `for`-Schleife wird j warten bis want[i]=False gilt (geht erst nachdem i KA verlässt)

Unmöglich!

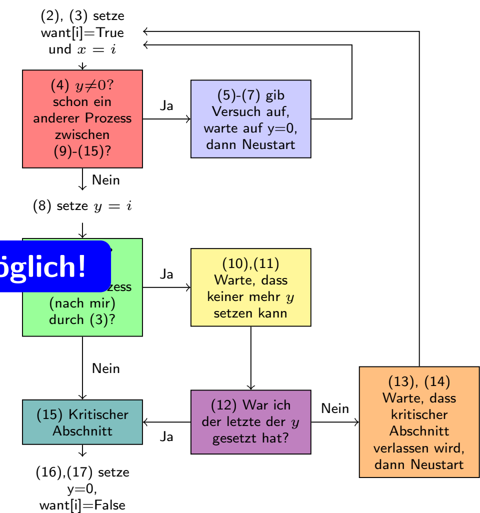


Lamports Algorithmus garantiert Mutual-Exclusion (3)

Fall 2: 2 Prozesse betreten den kritischen Abschnitt, erster entlang β:

- i: erster Prozess (entlang β)
- j: zweiter Prozess (entlang α oder β)
- Wenn i die for-Schleife durchlaufen hat: y kann erst verändert werden nachdem i KA verlässt:
  i hatte darauf gewartet, dass alle want-Einträge falsch sind: Jeder andere Prozess liest entweder $y \neq 0$ in (4) und bleibt am await in (6) hängen, oder (wenn er y=0 gelesen hatte) setzt er seinen want-Eintrag nachdem er y beschrieben hat.
- Deshalb: j kann y nicht verändern, und $y = i$ gilt, bis i KA verlässt
  D.h. j nicht entlang β.
- j nicht entlang α: Da $y = i$ und y kann nicht geändert werden: Alle andere Prozesse werden bei Zeile (4) zum Warten gelenkt.

Unmöglich!



Lamports Algorithmus ist Deadlock-frei (1)

Satz
Lamports Algorithmus ist Deadlock-frei.

Beweis: Nächste Folien.

Lamports Algorithmus ist Deadlock-frei (2)

```
loop forever
(1)   restlicher Code
(2)   want[i] := True;
(3)   x := i;
(4)   if y ≠ 0 then
(5)       want[i] := False;
(6)       await y = 0;
(7)       goto (2);
(8)   y := i;
(9)   if x ≠ i then
(10)      want[i] := False;
(11)      for j := 1 to n do
                  await ¬want[j];
(12)      if y ≠ i then
(13)                  await y = 0;
(14)                  goto (2);
(15)   Kritischer Abschnitt
(16)   y := 0;
(17)   want[i] := False;
end loop
```

Sei $\mathcal{P} = (i_1, j_1), (i_2, j_2), \ldots$ eine Berechnungsequenz, wobei

- i_k: Nummer des Prozesses, der einen Schritt macht
- j_k: Programmzeile, die Prozess i_k ausführt
- Zu jedem (i_k, j_k) gibt es einen festen Zustand (Variablenbelegung + Codezeiger der Prozesse)

Lamports Algorithmus ist Deadlock-frei (3)

```
loop forever
(1)   restlicher Code
(2)   want[i] := True;
(3)   x := i;
(4)   if y ≠ 0 then
(5)      want[i] := False;
(6)      await y = 0;
(7)      goto (2);
(8)   y := i;
(9)   if x ≠ i then
(10)       want[i] := False;
(11)       for j := 1 to n do
                 await ¬want[j];
(12)       if y ≠ i then
(13)             await y = 0;
(14)             goto (2);
(15)  Kritischer Abschnitt
(16)  y := 0;
(17)  want[i] := False;
end loop
```

Nehme an: $\mathcal{P}$ widerlegt die Deadlock-Freiheit

- D.h. es gibt einen Schritt nach dem kein Prozess mehr den kritischen Abschnitt erreicht: $\Rightarrow$ Es gibt k sodass für alle $k' \geq k$: $j_{k'} \neq 15$.
- Aber mindestens einer will in den kritischen Abschnitt (ist im Initialisierungscode): $\Rightarrow$ Es gibt $l \geq k'$ so dass $j_l \in \{2, \dots, 14\}$
- Nach weiteren Schritten kann kein anderer Prozess mehr im Abschlusscode sein: $\Rightarrow$ Es gibt $k_2 \geq l$, so dass für alle $k'_2 \geq k$ gilt: $j_{k'_2} \in \{1 - 14\}$
- Nach weiteren Schritten muss irgendein Prozess y setzen und danach kann keiner y auf 0 setzen: $\Rightarrow$ Es gibt $k_3 \geq k_2$, so dass $y \neq 0$ für alle Zustände ab k_3

Lamports Algorithmus ist Deadlock-frei (4)

```
loop forever
(1)   restlicher Code
(2)   want[i] := True;
(3)   x := i;
(4)   if y ≠ 0 then
(5)      want[i] := False;
(6)      await y = 0;
(7)      goto (2);
(8)   y := i;
(9)   if x ≠ i then
(10)       want[i] := False;
(11)       for j := 1 to n do
                 await ¬want[j];
(12)       if y ≠ i then
(13)             await y = 0;
(14)             goto (2);
(15)  Kritischer Abschnitt
(16)  y := 0;
(17)  want[i] := False;
end loop
```

- Für alle Schritte nach k_3 gilt:
  Wenn Prozess i nicht Zeile 8 als nächstes ausführen will,
  dann wird Prozess i nie Zeile 8 ausführen.
  (Da $y \neq 0$ wird er am await hängen bleiben).
- Nachdem alle Prozesse, die noch Zeile 8 ausführen können, dies getan haben,
  wird also kein weiterer mehr y umsetzen können.
  Sei $m \geq k_3$, wobei Schritt $(i_m, 8)$ das letzte Mal ist, dass Zeile 8 ausgeführt wird.
- Dann gilt danach: Alle Prozesse ungleich i_m werden ihren want-Eintrag nach endlichen vielen Schritten auf False setzen und y verbleibt auf i_m.
- D.h. Prozess i_m kann an keinem await hängen bleiben und muss somit den kritischen Abschnitt erreichen.

Eigenschaften von Lamports Algorithmus

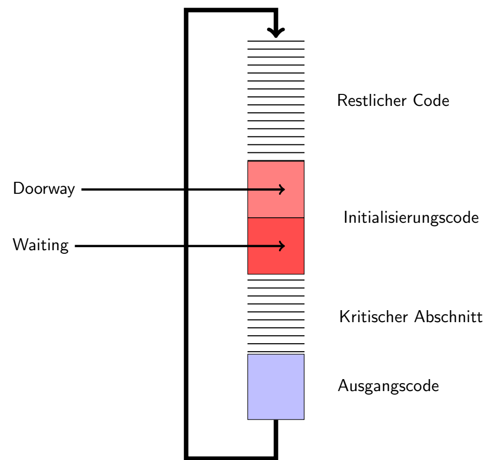
Zusammenfassend:

- Erfüllt wechselseitigen Ausschluss
- Erfüllt Deadlock-Freiheit
- Konstante Anzahl an Operationen, wenn nur ein Prozess den kritischen Abschnitt betreten möchte
- Erfüllt keine Starvation-Freiheit

Bounded Waiting

- Rückblick:
  Starvation-Freiheit: Wenn ein Prozess seinen kritischen Abschnitt betreten möchte, dann muss er ihn nach endlich vielen Berechnungsschritten betreten.
- Sagt nichts darüber aus, wie lange eine Prozess warten muss
- Wartender Prozess: Aktives Warten solange, bis ein anderer Prozess das Warten beendet
- Beispiel: Warten an einem `await`, bis anderer Prozess Wert ändert.
- Aufteilung des Initialisierungscode in: Doorway und Waiting

Bounded Waiting (2)



Bounded Waiting (3)

Definition

Ein Mutual-Exclusion Algorithmus erfüllt

r-bounded waiting, wenn für jeden Prozess i gilt: Wenn Prozess i den Doorway verlässt, bevor Prozess j (mit $j \neq i$) den Doorway betritt, dann betritt Prozess j den kritischen Abschnitt höchstens r Mal, bevor Prozess i den kritischen Abschnitt betritt.

bounded waiting, wenn es ein $r \in \mathbb{N}$ gibt, so dass der Algorithmus r-bounded waiting erfüllt.

die FIFO-Eigenschaft, wenn er 0-bounded waiting erfüllt. (FIFO = first-in-first-out)

Bemerkungen

- Bounded Waiting impliziert nicht Deadlock-Freiheit!
- Lamports Algorithmus erfüllt kein Bounded-Waiting!
- FIFO-Eigenschaft informell: Erster Prozess, der den Doorway überschritten hat ist als erster im kritischen Abschnitt

Bakery-Algorithmus

- Erfüllt die FIFO-Eigenschaft (0-bounded waiting)
- Idee des Algorithmus:
- Im Doorway „zieht" Prozess eine Nummer, die größer ist als jede andere vergebene Nummer
- Prozess mit kleinster Nummer darf in den KA.
- Verfahren wurde wohl in Bäckereien benutzt (daher der Name)

Bakery-Algorithmus: Einfache Variante

Initial: für $i = 1, \ldots n$ number[i]=0

Programm des i. Prozesses

```
loop forever
(1)   restlicher Code
(2)   number[i] := 1+max(number);
(3)   for j:=1 to n do
(4)         await number[j]=0 or (number[j],j) ≥lex (number[i],i)
(5)   Kritischer Abschnitt
(6)   number[i]=0
end loop
```


Beispiel

Programm des 1. Prozesses

```
loop forever
(1) restlicher Code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)     await
           number[j12]=0 or
           (number[j12],j11) ≥lex (number[1],1)
(5) Kritischer Abschnitt
(6) number[i]=0
end loop
```

Programm des 2. Prozesses

```
loop forever
(1) restlicher Code
(2) number[i] := 1+max(number);
(3) for j:=1 to 2 do
(4)     await
           number[j1]=0 or
           (number[j1],j1) ≥lex (number[2],2)
(5) Kritischer Abschnitt
(6) number[i]=0
end loop
```

number[1] number[2]

Bakery-Algorithmus: Einfache Variante, Problem bei (2)

Initial: für $i = 1, \ldots n$ number[i]=0

Programm des i. Prozesses

```
loop forever
(1)   restlicher Code
(2)   number[i] := 1+max(number);
(3)   for j:=1 to n do
(4)         await number[j]=0 or (number[j],j) ≥lex (number[i],i)
(5)   Kritischer Abschnitt
(6)   number[i]=0
end loop
```

Wenn $\max$(number) atomar berechnet wird, aber die Zuweisung $number[i] := 1 + \max(number)$ erst im nächsten Schritt erfolgt:

- Prozesse i und j mit $i < j$ können number[i] und number[j] auf denselben Wert setzen
- Wechselseitiger Ausschluss gilt nicht: j führt (4) aus während number[i] $= 0$, und i führt (4) danach aus

Bakery-Algorithmus

- Einfacher Bakery Algorithmus erfüllt Mutual-Exclusion, Starvation-Freiheit und FIFO-Eigenschaft.
- Aber: Annahme über atomares Maximum zu stark!
- Deswegen: Erweiterter Bakery-Algorithmus

Erweiterter Bakery-Algorithmus

Initial: für $i = 1, \ldots n$ number[i]=0, choosing[i]=False

Programm des i. Prozesses

loop forever
(1) restlicher Code
(2) choosing[i] := True;
(3) number[i] := 1+maximum(number);
(4) choosing[i] := False;
(5) for j:=1 to n do
(6) await choosing[j]=False;
(7) await number[j]=0 or (number[j],j) $\geq_{lex}$ (number[i],i);
(8) Kritischer Abschnitt
(9) number[i]=0
end loop

- choosing markiert Ein- und Austritt in den Doorway
- await-Abfrage sichert zu, dass Nummern erst verglichen werden, wenn number "richtig" gesetzt.

Berechnung des Maximums:

(1) max := 0
(2) for j:=1 to n do
(3) current := number[j];
(4) if max < current then max := current
(5) number[i] := 1+max:

Lemma I

Wenn der Wert von number[k] nicht geändert wird, während Prozess i das Maximum berechnet, dann ist der Wert number[i] anschließend größer als der Wert von number[k].

Korrektheit des Bakery-Algorithmus

Sprechweisen:

loop forever
(1) restlicher Code
(2) choosing[i] := True;
(3) number[i] := 1+maximum(number);
(4) choosing[i] := False;
(5) for j:=1 to n do
(6) await choosing[j]=False;
(7) await number[j]=0 or
(number[j],j) $\geq_{lex}$ (number[i],i);
(8) Kritischer Abschnitt
(9) number[i]=0
end loop

Doorway (2)–(4)

Bäckerei (5)–(8)

Korrektheit des Bakery-Algorithmus

Lemma II

Wenn Prozess i und Prozess k beide in der Bäckerei sind und i in die Bäckerei eintritt, bevor k in den Doorway eintritt, dann gilt number[i] < number[k].

loop forever
(1) restlicher Code
(2) choosing[i] := True;
(3) number[i] := 1+maximum(number);
(4) choosing[i] := False;
(5) for j:=1 to n do
(6) await choosing[j]=False;
(7) await number[j]=0 or
(number[j],j) $\geq_{lex}$ (number[i],i);
(8) Kritischer Abschnitt
(9) number[i]=0
end loop

Beweis: Offensichtlich aus Lemma I. Denn Prozess i berechnet seinen Wert, während Prozess k seinen number-Wert nicht ändern kann (außer auf 0 setzen).

Korrektheit des Bakery-Algorithmus (2)

```
loop forever
(1)  restlicher Code
(2)  choosing[i] := True;
(3)  number[i] := 1+maximum(number);
(4)  choosing[i] := False;
(5)  for j:=1 to n do
(6)      await choosing[j]=False;
(7)      await number[j]=0 or
             (number[j],j) ≥lex (number[i],i);
(8)  Kritischer Abschnitt
(9)  number[i]=0
end loop
```

Lemma III

Wenn Prozess i im kritischen Abschnitt ist und Prozess k in der Bäckerei ist, dann gilt (number[i],i) $<_{lex}$ (number[k],k)

Beweis:

- Sei T_6^i der Schritt, indem Prozess i in Zeile (6) durch das `await`-Statement für $j = k$ hindurchkommt (d.h. Prozess i liest zum letzten Mal choosing[k]).
- Analog sei T_7^i der Schritt, indem Prozess i das letzte Mal number[k] gelesen hat.
- Es muss gelten T_6^i liegt vor T_7^i (notiert als $T_6^i < T_7^i$).
- Für Prozess k seien T_2^k, T_3^k, T_4^k jeweils die Schritte in denen er die Programmzeilen (2), (3), (4) zum letzten Mal abgearbeitet hat. Es muss gelten $T_2^k < T_3^k < T_4^k$.

Korrektheit des Bakery-Algorithmus (3)

```
loop forever
(1)  restlicher Code
(2)  choosing[i] := True;
(3)  number[i] := 1+maximum(number);
(4)  choosing[i] := False;
(5)  for j:=1 to n do
(6)      await choosing[j]=False;
(7)      await number[j]=0 or
             (number[j],j) ≥lex (number[i],i);
(8)  Kritischer Abschnitt
(9)  number[i]=0
end loop
```


Lemma III

Wenn Prozess i im kritischen Abschnitt ist und Prozess k in der Bäckerei ist, dann gilt (number[i],i) $<_{lex}$ (number[k],k)

- Im Schritt T_6^i hatte choosing[k] den Wert False und zu den Schritten T_2^k, T_3^k und T_4^k hatte choosing[k] den Wert True.
- Das ergibt zwei Fälle
  1. $T_6^i < T_2^k$: Aus Lemma II folgt number[i] $<$ number[k]
  2. $T_4^k < T_6^i$: Dann gilt $T_3^k < T_4^k < T_6^i < T_7^i$, d.h. zum Zeitpunkt als Prozess i Zeile (7) ausführt und number[k] liest, gilt number[k] $\neq 0$ (er wurde zum Zeitpunkt T_3^k gesetzt!). Da die `await`-Bedingung in Zeile (7) wahr ist muss (number[i],i) $<_{lex}$ (number[k],k) gegolten haben.

Korrektheit des Bakery-Algorithmus (4)

```
loop forever
(1)  restlicher Code
(2)  choosing[i] := True;
(3)  number[i] := 1+maximum(number);
(4)  choosing[i] := False;
(5)  for j:=1 to n do
(6)      await choosing[j]=False;
(7)      await number[j]=0 or
             (number[j],j) ≥lex (number[i],i);
(8)  Kritischer Abschnitt
(9)  number[i]=0
end loop
```

Satz

Der erweiterte Bakery-Algorithmus garantiert wechselseitigen Ausschluss, ist Starvation-frei, und erfüllt die FIFO-Eigenschaft.

Beweis:

- Mutual-Exclusion folgt direkt aus Lemma III, da sonst ein Widerspruch hergeleitet werden kann.
- Die FIFO-Eigenschaft folgt aus den Lemmas II und III.

Korrektheit des Bakery-Algorithmus (5)

```
loop forever
(1)  restlicher Code
(2)  choosing[i] := True;
(3)  number[i] := 1+maximum(number);
(4)  choosing[i] := False;
(5)  for j:=1 to n do
(6)      await choosing[j]=False;
(7)      await number[j]=0 or
             (number[j],j) ≥lex (number[i],i);
(8)  Kritischer Abschnitt
(9)  number[i]=0
end loop
```


Satz

Der erweiterte Bakery-Algorithmus garantiert wechselseitigen Ausschluss, ist Starvation-frei, und erfüllt die FIFO-Eigenschaft.

Beweis:

- Deadlock-Freiheit: Widerspruchsbeweis
  - unendlich lange Berechungsfolge, so dass kein Prozess mehr in den KA
  - aber mindestens ein Prozess in der Bäckerei
  - dann: Es gibt Zeitpunkt zudem kein Prozess mehr in Doorway und die Bäckerei eintritt.
  - Ab diesem Zeitpunkt muss ein Prozess die `for`-Schleife durchlaufen können.
- Starvation-Freiheit folgt aus Deadlock-Freiheit und FIFO-Eigenschaft.

Nachteile des Bakery-Algorithmus

- Algorithmus nicht schnell: Wenn nur ein Prozess in den kritischen Abschnitt will, muss er $O(n)$ Schritte im Initialisierungscode ausführen

Nachteile des Bakery-Algorithmus (2)

Prozess 1	$\xrightarrow{(1)-(8)}$		$\xrightarrow{(9)}$		$\xrightarrow{(1)-(4)}$		$\xrightarrow{(5)-(8)}$		$\xrightarrow{(9)}$	
Prozess 2		$\xrightarrow{(1)-(4)}$		$\xrightarrow{(5)-(8)}$		$\xrightarrow{(9)}$		$\xrightarrow{(1)-(4)}$		
number[1]	0	1	1	0	0	3	3	3	3	0
number[2]	0	0	2	2	2	2	0	0	4	4

- Maximum wird immer berechnet, bevor number[i] wieder auf 0 gesetzt ist.
- D.h. der Algorithmus benötigt unbegrenzt große Zahlen im number-Feld.
- Es gibt Varianten (Black-White-Bakery-Alg.), die mit begrenzten Zahlen auskommen.

Falsche Berechnung des Maximums:

```
(1)  maxpos := i
(2)  for j:=1 to n do
(3)      if number[maxpos] < number[j] then maxpos := j
(4)  number[i] := 1+number[maxpos]:
```

- Mit dieser Maximum-Berechnung ist **kein** wechselseitiger Ausschluss garantiert.
- Prozesse 1,2,3.
- Erst berechnen Proz. 2 und 3 jeweils number[2] = 1 und number[3] = 1
- Dann Prozess 2 in den kritischen Abschnitt und Prozess 3 wartet
- Dann Prozess 1 bis vor (4) der Maximumberechnung (maxpos = 2!)
- Dann Prozess 2 im Abschlusscode, setzt number[2] = 0.
- Dann Prozess 3 in den Kritischen Abschnitt.
- Dann Prozess 1 auch in den Kritischen Abschnitt.

Komplexitätsresultate

- Platzbedarf
- Zeitbedarf
- Beachte: Modell ist immer noch: Nur atomare Lese- und Schreibbefehle für den gemeinsamen Speicher.

Eine untere Schranke für den Platzbedarf

Theorem

Jeder Deadlock-freie Mutual-Exclusion Algorithmus für n Prozesse benötigt mindestens n gemeinsam genutzte Speicherplätze.

Bemerkungen zum Beweis (1980 von J.Burns und N.A. Lynch):

- Bei Single-Writer, Multiple-Reader Algorithmen einfach, da jeder Prozess, den Speicher verändern muss, bevor er den kritischen Abschnitt erkennt, sonst wäre wechselseitiger Ausschluss unmöglich (Prozesse müssen erkennen, ob ein anderer Prozess im kritischen Abschnitt ist.)
- Der allgemeine Beweis ist relativ kompliziert. Teile der Argumentation sind, dass mit weniger Speicherplätzen, nicht unterschieden werden kann, ob ein Prozess im restlichen Code oder im Kritischen Abschnitt ist, woraus sich wechselseitiger Ausschluss widerlegen lässt.
- Wir lassen den Beweis weg.

Eine obere Schranke für den Platzbedarf

Theorem

Es gibt einen Deadlock-freien Mutual-Exclusion Algorithmus für n Prozesse der n gemeinsame Bits verwendet.

Beweis:

- Ein-Bit-Algorithmus
- sowohl J.E.Burns im Jahr 1981 als auch von L.Lamport im Jahr 1986

Idee des Ein-Bit-Algorithmus

Initial: für $i = 1, \dots n$ want[i]= False,

Programm des i. Prozesses

```
loop forever
(1)    restlicher Code
(2)    want[i]:= True;
(3)    for local:= 1 to n do
(4)      if local≠ i then await want[local] = False;
(5)    Kritischer Abschnitt
(6)    want[i] = False
end loop
```

- Erfüllt wechselseitigen Ausschluss:
  Der erste Prozess im Kritischen Abschnitt hat want auf True, jeder andere wird dies lesen und stecken bleiben
- Erfüllt nicht die Deadlock-Freiheit
- Algorithmus so falsch!

Der Ein-Bit-Algorithmus

Initial: für $i = 1, \dots n$ want[i]= False,

Programm des i. Prozesses

```
loop forever
(1)  restlicher Code
(2)  repeat
(3)    want[i]:= True;
(4)    local := 1:
(5)    while (want[i] = True) and (local < i) do
(6)      if want[local] = True then
(7)        want[i] := False;
(8)        await want[local] = False;
(9)      local := local + 1
(10) until want[i] = True;
(11) for local:= i+1 to n do
(12)   await want[local] = False;
(13) Kritischer Abschnitt
(14) want[i] = False
end loop
```

Tests für $j = 1 \dots i - 1$

Tests für $j = i + 1 \dots n$

- Idee im Groben wie vorher: Teste alle anderen want-Wert auf False, bevor in den KA eingetreten wird.
- Daher: wechselseitiger Ausschluss ist erfüllt.

Der Ein-Bit-Algorithmus (2)

Initial: für $i = 1, \ldots n$ want[i]= False,

Programm des i. Prozesses

```
loop forever
(1)  restlicher Code
(2)  repeat
(3)     want[i]:= True;
(4)     local := 1:
(5)     while (want[i] = True) and (local < i) do
(6)        if want[local] = True then
(7)           want[i] := False;
(8)           await want[local] = False;
(9)        local := local + 1
(10) until want[i] = True;
(11) for local:= i+1 to n do
(12)    await want[local] = False;
(13) Kritischer Abschnitt
(14) want[i] = False
end loop
```

Deadlock-Freiheit:

- Beweis-Idee: Bei Deadlock-Berechnungsfolge kann man schließen:
- Irgendwann alle Prozesse:
  - await in Zeile (8)
  - in der for-Schleife in Zeilen (11)-(12)
  - oder für immer im restlichen Code
- und: mind. ein Prozess in der for-Schleife
- Prozess mit größter Nummer wird for-Schleife durchlaufen

Eigenschaften des Ein-Bit-Algorithmus

- Garantiert wechselseitigen Ausschluss und Deadlock-Freiheit
- Starvation ist möglich
- Nicht symmetrisch: Z.B. Prozess mit Nummer 1 durchläuft die `repeat`-Schleife sofort
- Nicht schnell: Wenn nur ein Prozess in den KA will, muss er alle n-Bits testen
- Aber: Platz-optimal, da nur n-Bits gemeinsamer Speicher

Ein Resultat zur Laufzeit

Theorem (R. Alur und G.Taubenfeld, 1992)

Es gibt keinen (Deadlock-freien) Mutual-Exclusion Algorithmus für 2 (oder auch n) Prozesse, der eine obere Schranke hat für die Anzahl an Speicherzugriffen (des gemeinsamen Speichers), die ein Prozess ausführen muss, bevor er den kritischen Abschnitt betreten darf.

- D.h. Prozesse müssen beliebig lang "warten" bis sie in den kritischen Abschnitt dürfen
- Es gibt keinen Algorithmus der das verhindern kann
- Achtung: Für dieses Modell (Lese- und Schreiboperatiomen atomar)!
- Resultat meint alle Fälle, es gibt Unterfälle in denen man eine Schranke angeben kann
- z.B.: Fall, in dem nur ein Prozess in den kritischen Abschnitt will

Beweis

Sei M ein Deadlock-freier Mutual-Exclusion Algorithmus für 2 Prozesse P_1 und P_2

Berechnungsbaum T_M für M:

- binärer Baum
- Jeder Knoten entspricht Zustand der Ausführung (alle Belegungen)
- Wurzel: Erster interessanter Zustand: P_1 und P_2 direkt vor dem Eintritt in Initialsierungscode
- linkes Kind eines Knotens: Nachfolgezustand nach einem Schritt von P_1
- rechtes Kind eines Knotens: Nachfolgezustand nach einem Schritt von P_2
- Blatt: P_1 oder P_2 hat kritischen Abschnitt betreten (dann stoppe)

Beweis(2)

Markierung der Knoten von T_M

- Blatt ist genau mit 1 oder genau mit 2 markiert, jenachdem welches P_i im KA ist
- innerer Knoten ist mit 1, 2 oder (1 und 2) markiert, je nachdem wie seine Kinder markiert sind.

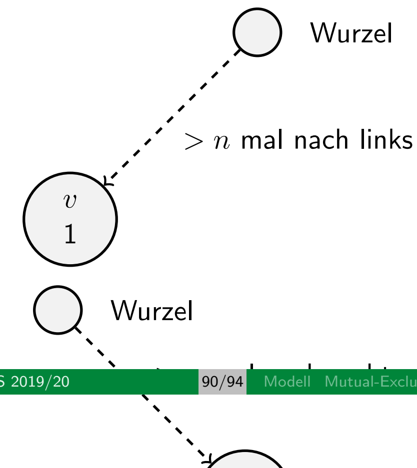
Ähnlichkeit

- Zwei Knoten v, w sind ähnlich bzgl. P_i (geschrieben $v\langle P_i\rangle w$), gdw.
  - Schritte die P_i von der Wurzel zu v macht = Schritte die P_i von der Wurzel zu w macht
  - Gemeinsame Variablen und lokalen Variablen von P_i sind identisch für v und w

Beweis (3)

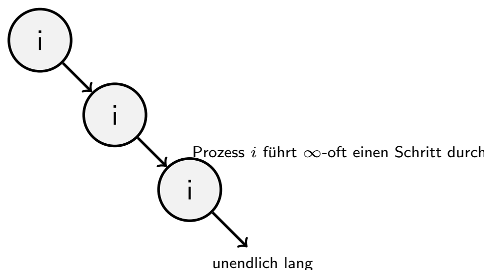
Theorem ist bewiesen wenn:

Für jedes $n > 0$ und $i \in \{1, 2\}$:
Es gibt ein Blatt v mit Markierung i, sodass
auf dem Pfad von der Wurzel bis zu v
werden mehr als n Schritte für Prozess P_i ausgeführt



Beweis (4)

Fall: Es gibt unendlichen langen Pfad in T_M, der unendlich viele Knoten enthält, die alle mit i markiert sind und Prozess P_i führt unendlich viele Schritte auf diesem Pfad aus.



Dann: Für jedes n kann der gesuchte Pfad konstruiert werden.
Deshalb: Annahme A: T_M hat keinen solchen unendlichen Pfad

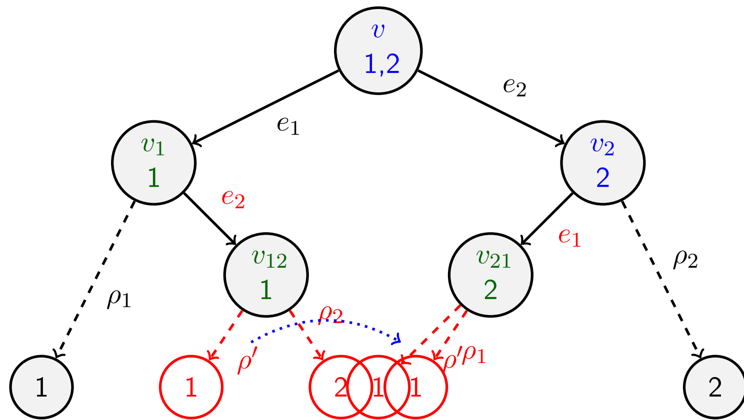
Beweis (5)

Wir zeigen nun: Annahme A führt zum Widerspruch.

Da Algorithmus Deadlock-frei muss gelten (w.g. Annahme A):
Es gibt Knoten v, v_1, v_2 mit

- v ist mit $1, 2$ markiert
- Die beiden Knoten v_1 und v_2 sind jeweils mit genau einer Zahl markiert.

Beweis (6): Fall 1: v_1 mit 1, v_2 mit 2 markiert



Beweis (7): Fall 2: v_1 mit 2, v_2 mit 1 markiert