Das Mutual-Exclusion-Problem bei n Prozessen

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 2. November 2020

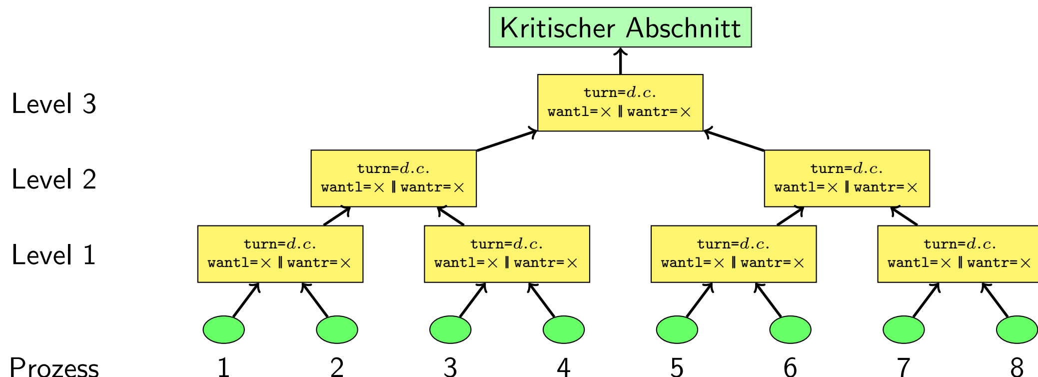
Übersicht: Algorithmen für n Prozesse

1. Tournament-Algorithmen
2. Lamports Algorithmus
3. Bakery-Algorithmus

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"

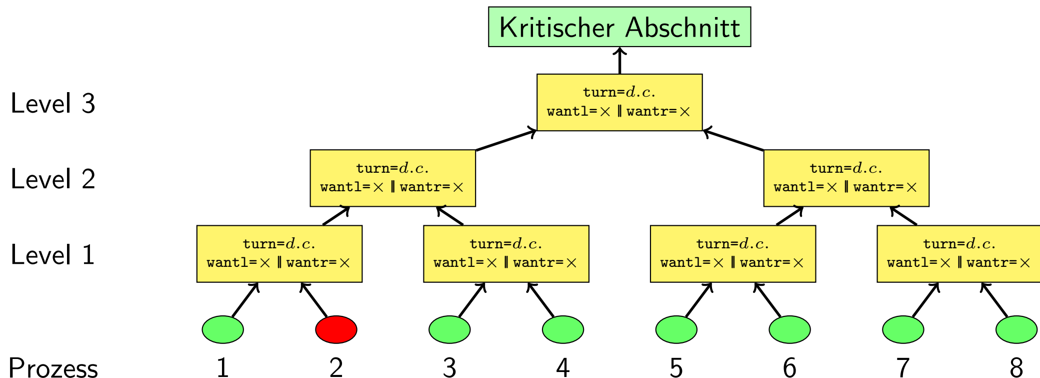
Idee des Tournament-Algorithmus (basierend auf Peterson-Alg.)



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

wantl = want des linken Kindes, wantr = want des rechten Kindes

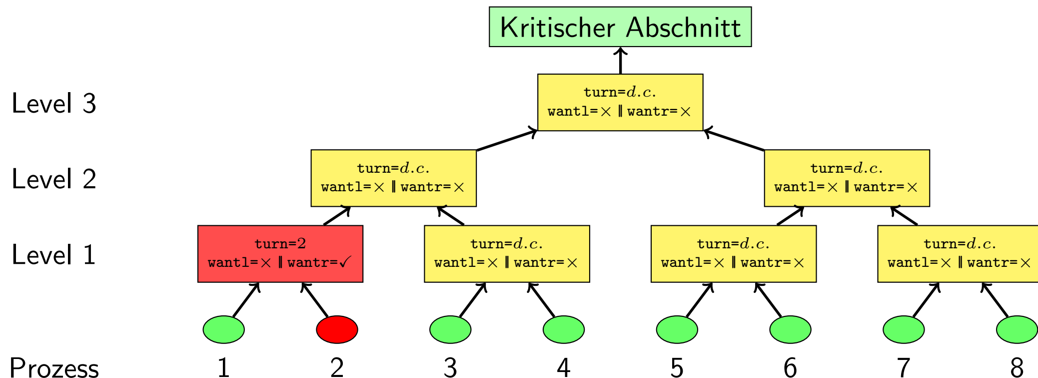
Idee des Tournament-Algorithmus (basierend auf Peterson-Alg.)



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

wantl = want des linken Kindes, wantr = want des rechten Kindes

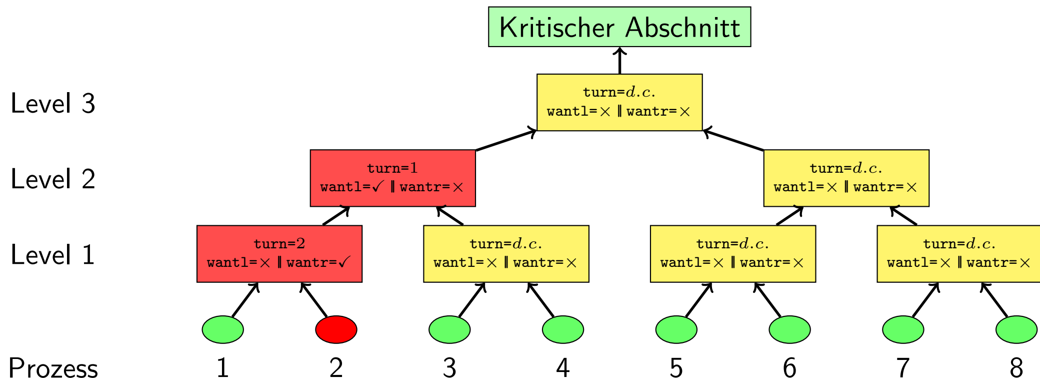
Idee des Tournament-Algorithmus (basierend auf Peterson-Alg.)



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

wantl = want des linken Kindes, wantr = want des rechten Kindes

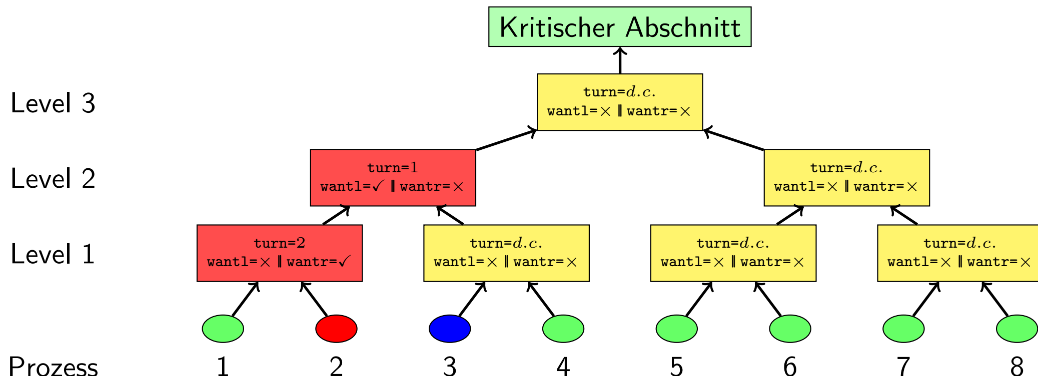
Idee des Tournament-Algorithmus (basierend auf Peterson-Alg.)



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

wantl = want des linken Kindes, wantr = want des rechten Kindes

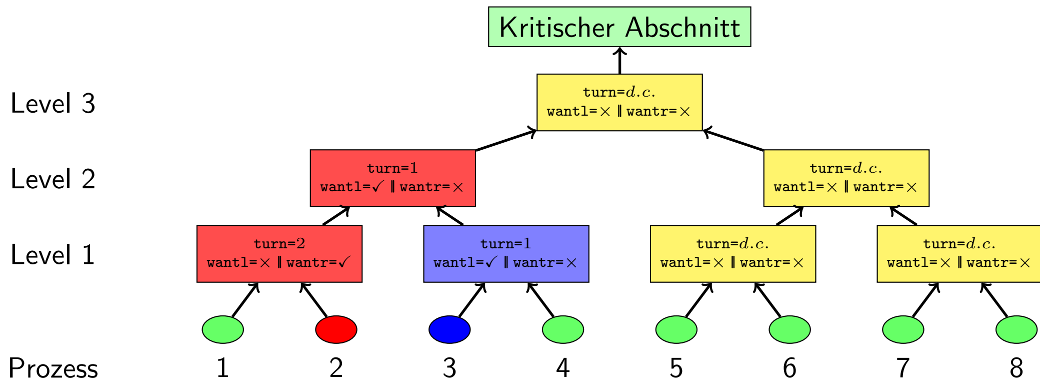
Idee des Tournament-Algorithmus (basierend auf Peterson-Alg.)



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

wantl = want des linken Kindes, wantr = want des rechten Kindes

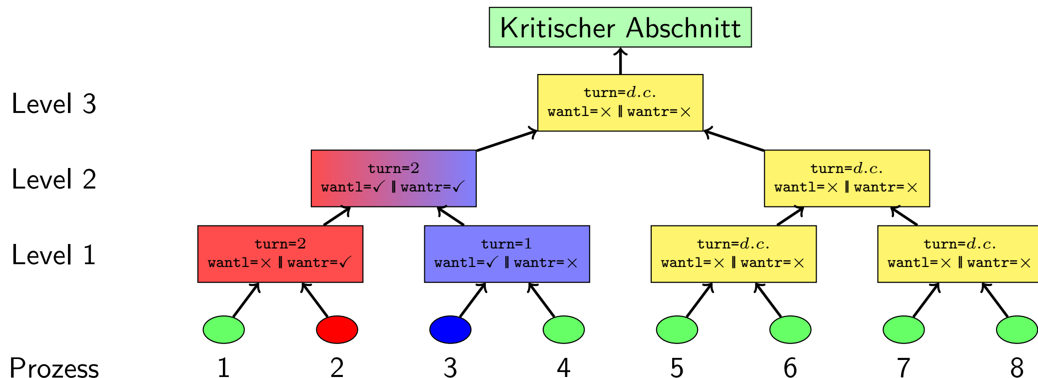
Idee des Tournament-Algorithmus (basierend auf Peterson-Alg.)



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

wantl = want des linken Kindes, wantr = want des rechten Kindes

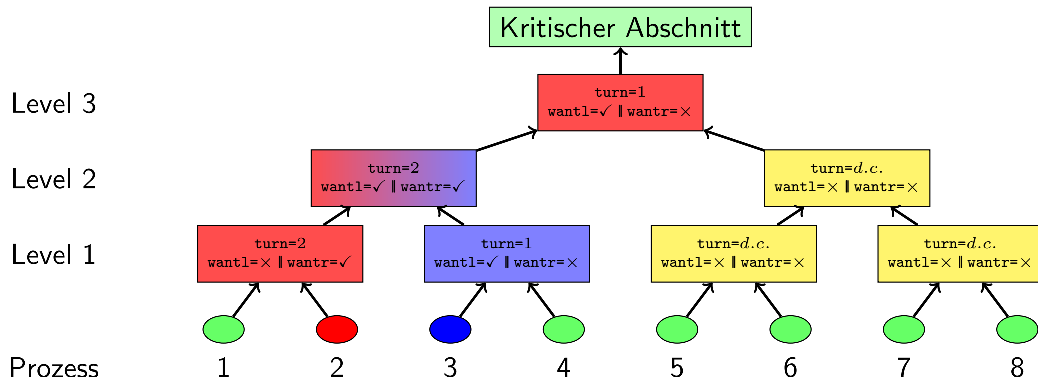
Idee des Tournament-Algorithmus (basierend auf Peterson-Alg.)



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

wantl = want des linken Kindes, wantr = want des rechten Kindes

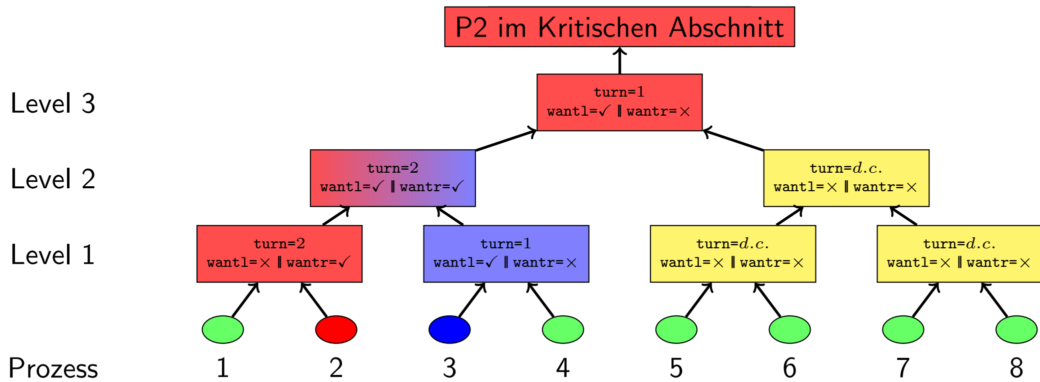
Idee des Tournament-Algorithmus (basierend auf Peterson-Alg.)



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

wantl = want des linken Kindes, wantr = want des rechten Kindes

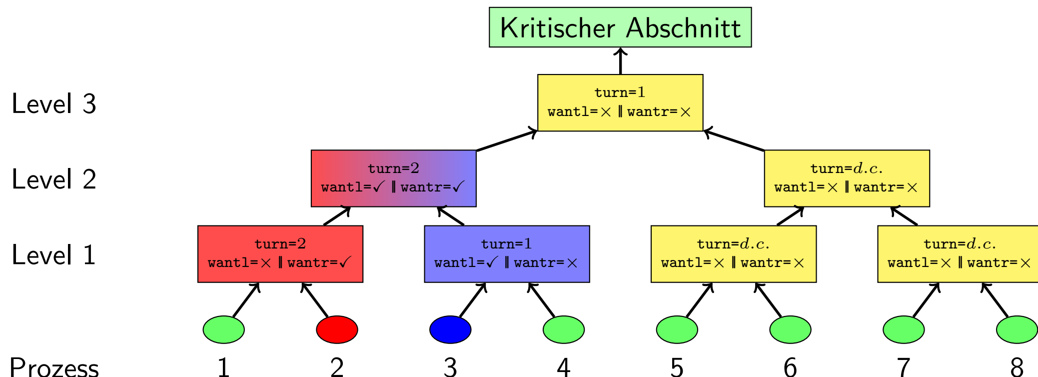
Idee des Tournament-Algorithmus (basierend auf Peterson-Alg.)



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

wantl = want des linken Kindes, wantr = want des rechten Kindes

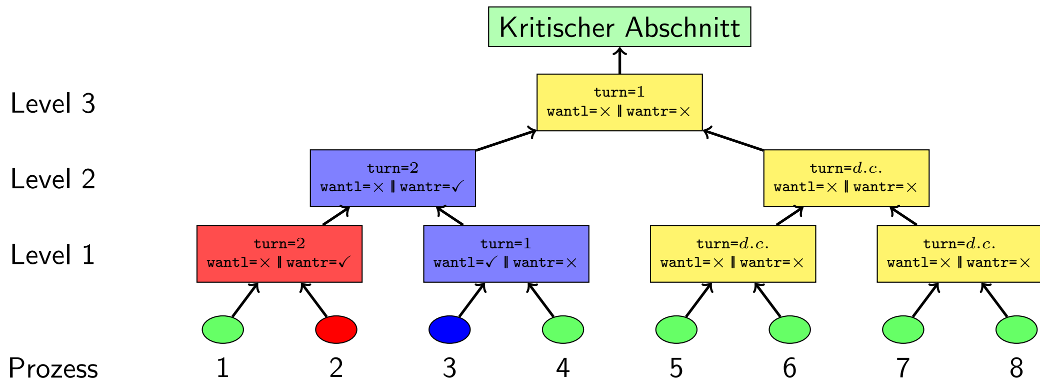
Idee des Tournament-Algorithmus (basierend auf Peterson-Alg.)



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

wantl = `want` des linken Kindes, wantr = `want` des rechten Kindes

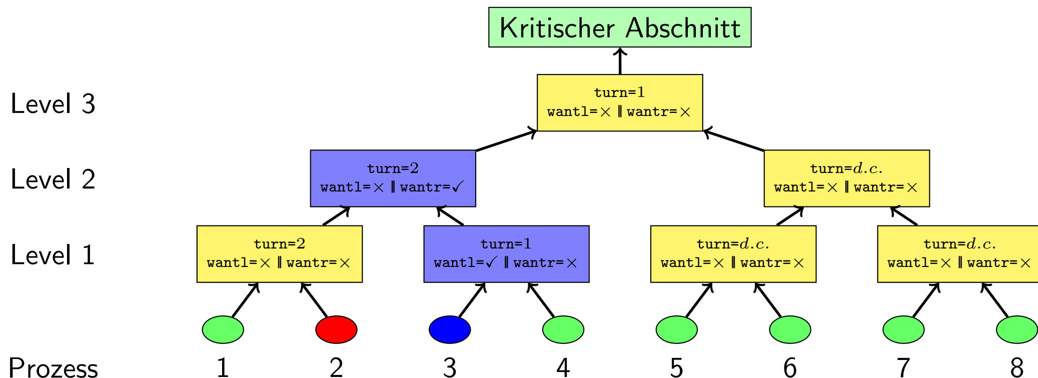
Idee des Tournament-Algorithmus (basierend auf Peterson-Alg.)



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

wantl = want des linken Kindes, wantr = want des rechten Kindes

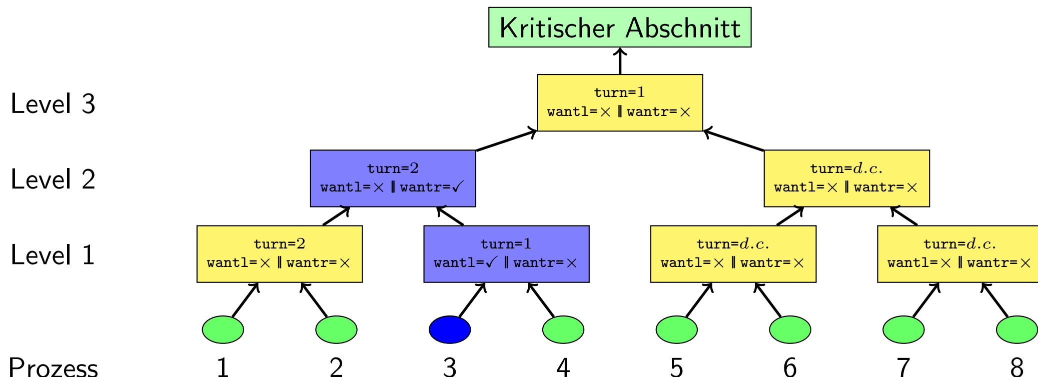
Idee des Tournament-Algorithmus (basierend auf Peterson-Alg.)



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

wantl = want des linken Kindes, wantr = want des rechten Kindes

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



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

wantl = want des linken Kindes, wantr = want des rechten Kindes

Eigenschaften der Tournament-Algorithmen

- Einfach zu implementieren
- Wechselseitiger Ausschluss, Deadlock-Freiheit 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 (US-amerikanischer Informatiker, Turing Award 2013)
- Algorithmus veröffentlicht 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

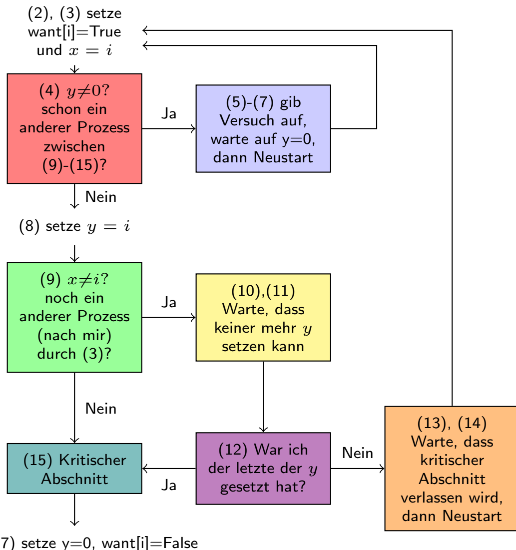
Initial: y=0, für $i = 1, \ldots, n$: want[i]=False, Wert von x egal

```
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
```

Lamports Algorithmus: Programm des i. Prozesses

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

```
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
```



Lamports Algorithmus: Programm des i. Prozesses

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

```
loop forever
(1)    restlicher Code
(2)    want[i] := True;
```

Zwei Möglichkeiten, den Kritischen Abschnitt zu erreichen!

```
(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
```

(2), (3) setze want[i]=True und $x = i$

(4) $y \neq 0$? schon ein anderer Prozess zwischen (9)-(15)?

Ja → (5)-(7) gib Versuch auf, warte auf y=0, dann Neustart

Nein → (8) setze $y = i$

α β

(9) $x \neq i$? noch ein anderer Prozess nach mir durch (3)?

Ja → (10),(11) Warte, dass keiner mehr y setzen kann

Nein → (15) Kritischer Abschnitt

(12) War ich der letzte der y gesetzt hat?

Ja → (15) Kritischer Abschnitt

Nein → (13), (14) Warte, dass kritischer Abschnitt verlassen wird, dann Neustart

(16),(17) setze y=0, want[i]=False

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.

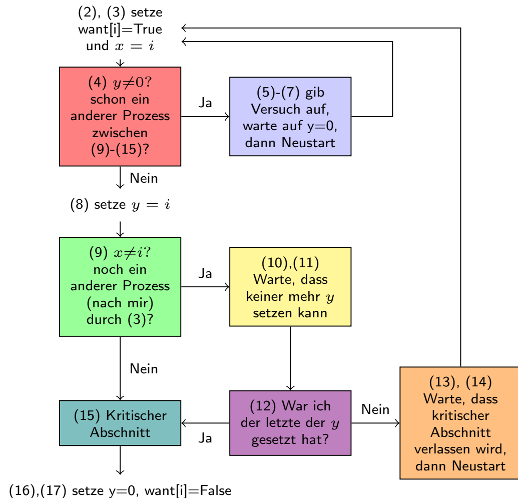
Fallunterscheidung:

- 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:

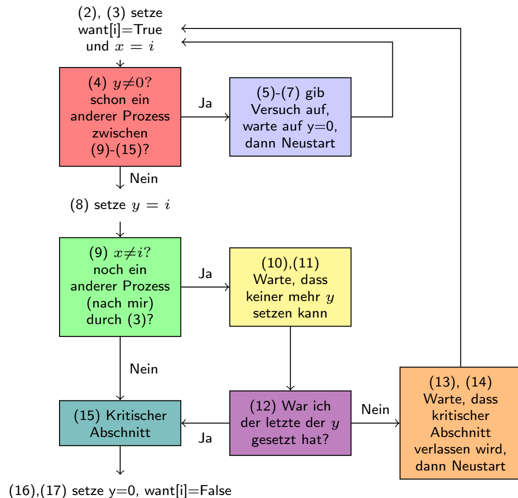
- i: erster Prozess (entlang α)
- j: zweiter Prozess (entlang α oder β)



Lamports Algorithmus garantiert Mutual-Exclusion (2)

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

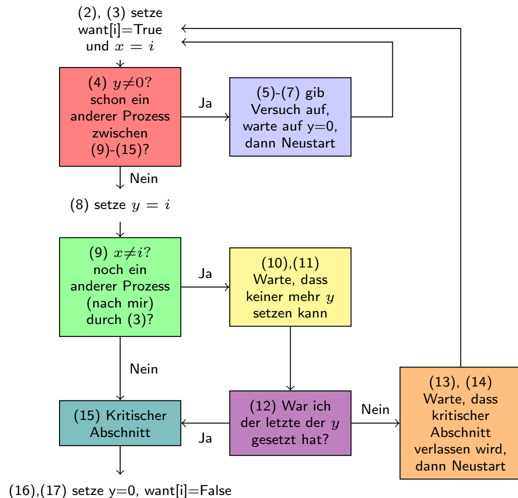
- i: erster Prozess (entlang α)
- j: zweiter Prozess (entlang α oder β)
- Wenn i durch (9) läuft gilt: $x = i$ und $y \neq 0$.



Lamports Algorithmus garantiert Mutual-Exclusion (2)

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

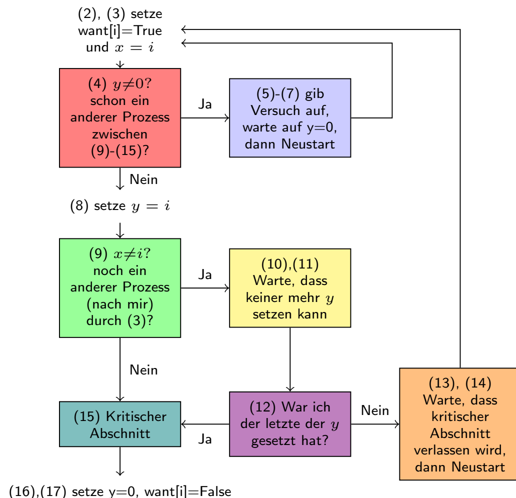
- i: erster Prozess (entlang α)
- j: zweiter Prozess (entlang α oder β)
- Wenn i durch (9) läuft gilt:
  $x = i$ und $y \neq 0$.
- Impliziert:
  (1) j noch nicht durch (3), oder
  (2) j durch (3) bevor i durch (3)



Lamports Algorithmus garantiert Mutual-Exclusion (2)

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

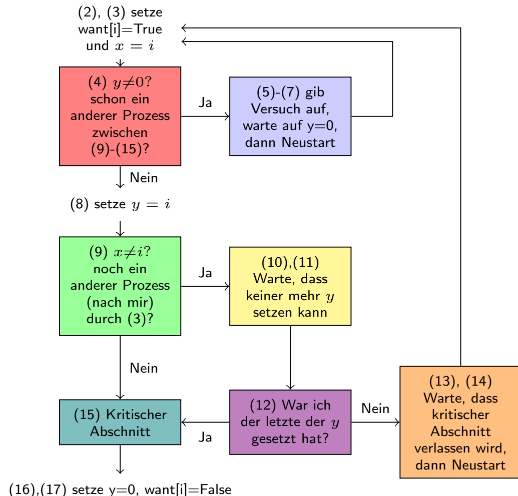
- i: erster Prozess (entlang α)
- j: zweiter Prozess (entlang α oder β)
- Wenn i durch (9) läuft gilt:
  $x = i$ und $y \neq 0$.
- Impliziert:
  (1) j noch nicht durch (3), oder
  (2) j durch (3) bevor i durch (3)
- Fall (1): Da j vor (4):
  – j wird in (5)-(7) geschickt
  – und wartet bis $y = 0$,
  D.h. j kann nicht mehr in KA



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)

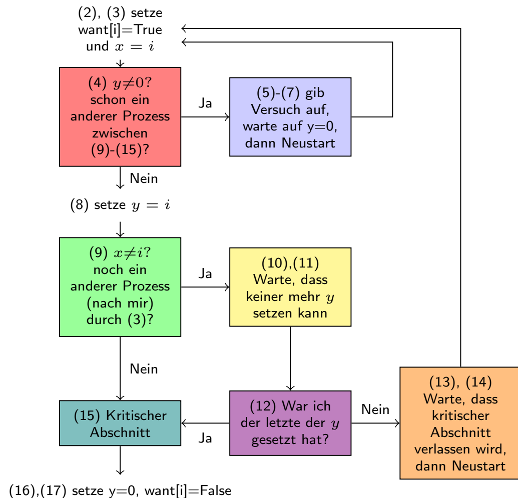


Lamports Algorithmus garantiert Mutual-Exclusion (2)

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)

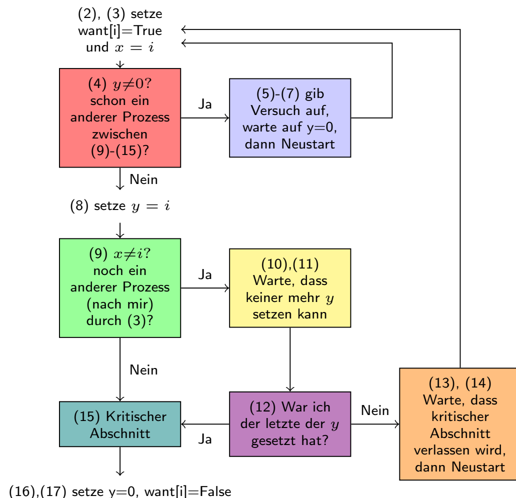
Unmöglich!



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)

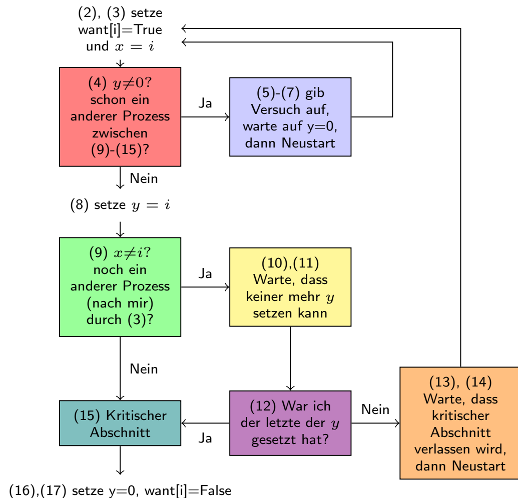


Lamports Algorithmus garantiert Mutual-Exclusion (3)

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:

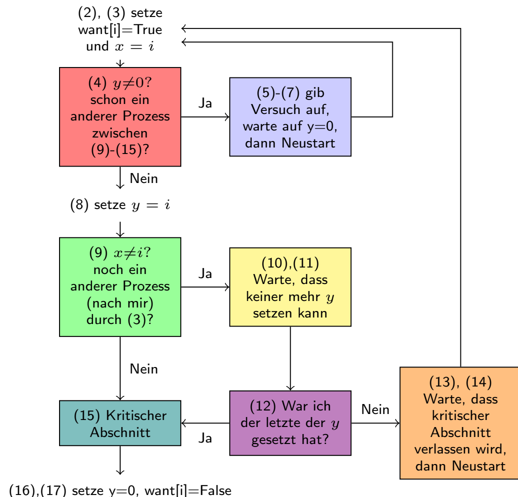
  i hatte darauf gewartet, dass alle want-Einträge falsch sind: Jeder andere Prozess liest entweder $y \neq 0$ in (4) und bleibt am await in (6) hängen, oder (wenn er y=0 gelesen hatte) setzt er seinen want-Eintrag erst auf False, nachdem er y beschrieben hat.



Lamports Algorithmus garantiert Mutual-Exclusion (3)

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

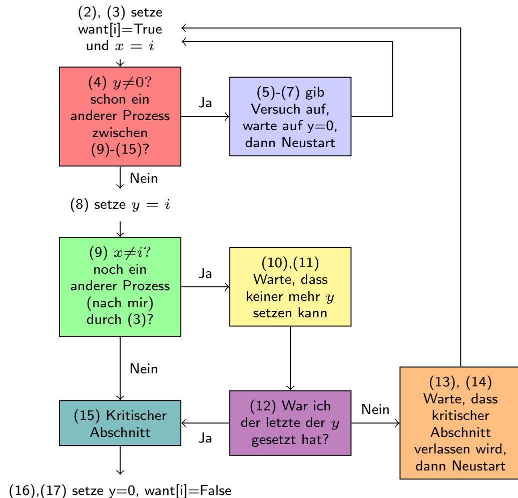
- i: erster Prozess (entlang β)
- j: zweiter Prozess (entlang α oder β)
- Wenn i die for-Schleife durchlaufen hat:
  y kann erst verändert werden nachdem i KA verlässt:
  i hatte darauf gewartet, dass alle want-Einträge falsch sind: Jeder andere Prozess liest entweder $y \neq 0$ in (4) und bleibt am await in (6) hängen, oder (wenn er y=0 gelesen hatte) setzt er seinen want-Eintrag erst auf False, 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.



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 erst auf False, 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.



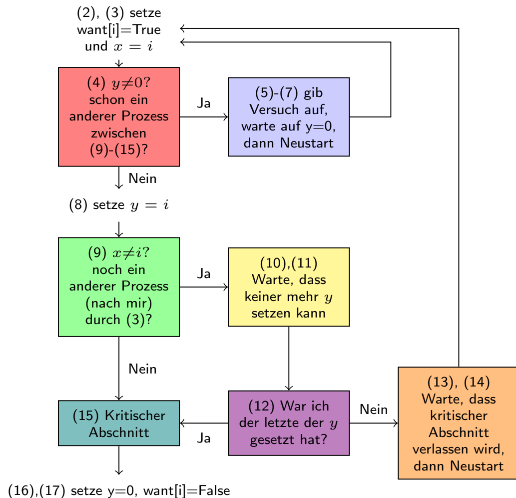
Lamports Algorithmus garantiert Mutual-Exclusion (3)

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 erst auf False, nachdem er y beschrieben hat.

Unmöglich!

- 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.



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 $j_l \in \{2, \ldots, 14\}$
- 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

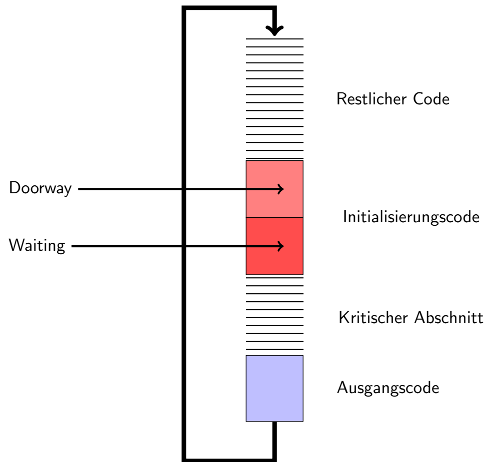
Zusammenfassend:

- 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: Prozess wartet aktiv solange, bis ein anderer Prozess das Warten beendet
- Beispiel: Warten an einem `await`, bis anderer Prozess Wert ändert, sodass Bedingung war wird.
- 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 $i = 1, \dots n$ 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
```

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
```

- Doorway, Waiting

Bakery-Algorithmus: Einfache Variante

Initial: für $i = 1, \dots n$ 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
```

- Doorway, Waiting
- $\geq_{lex}$ = lexikographische Ordnung

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
```

- Doorway, Waiting
- $\geq_{lex}$ = lexikographische Ordnung
- Annahme: Maximum-Berechnung und Zuweisung atomar

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[j]=0 or
          (number[j],j) ≥_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[j]=0 or
          (number[j],j) ≥_lex (number[2],2)
(5) Kritischer Abschnitt
(6) number[i]=0
end loop
```

number[1]	number[2]
0	0

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[j]=0 or
          (number[j],j) ≥_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[j]=0 or
          (number[j],j) ≥_lex (number[2],2)
(5) Kritischer Abschnitt
(6) number[i]=0
end loop
```

number[1]	number[2]
1	0

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[j]=0 or
          (number[j],j) ≥_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[j]=0 or
          (number[j],j) ≥_lex (number[2],2)
(5) Kritischer Abschnitt
(6) number[i]=0
end loop
```

number[1]	number[2]
1	2

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[j]=0 or
          (number[j],j) ≥_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[j]=0 or
          (number[j],j) ≥_lex (number[2],2)
(5) Kritischer Abschnitt
(6) number[i]=0
end loop
```

number[1]	number[2]
1	2

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[j]=0 or
          (number[j],j) ≥_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[j]=0 or
          (number[j],j) ≥_lex (number[2],2)
(5) Kritischer Abschnitt
(6) number[i]=0
end loop
```

number[1]	number[2]
1	2

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[j]=0 or
          (number[j],j) ≥_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[j]=0 or
          (number[j],j) ≥_lex (number[2],2)
(5) Kritischer Abschnitt
(6) number[i]=0
end loop
```

number[1]	number[2]
1	2

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[1]=0 or
          (number[1],1) ≥_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[j]=0 or
          (number[j],j) ≥_lex (number[2],2)
(5) Kritischer Abschnitt
(6) number[i]=0
end loop
```

number[1]	number[2]
1	2

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[2]=0 or
          (number[2],1) ≥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[j]=0 or
          (number[j],j) ≥lex (number[2],2)
(5) Kritischer Abschnitt
(6) number[i]=0
end loop
```

number[1]	number[2]
1	2

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[j]=0 or
          (number[j],j) ≥_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[j]=0 or
          (number[j],j) ≥_lex (number[2],2)
(5) Kritischer Abschnitt
(6) number[i]=0
end loop
```

number[1]	number[2]
1	2

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[j]=0 or
          (number[j],j) ≥_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[1]=0 or
          (number[1],1) ≥_lex (number[2],2)
(5) Kritischer Abschnitt
(6) number[i]=0
end loop
```

number[1]	number[2]
1	2

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[j]=0 or
          (number[j],j) ≥_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[1]=0 or
          (number[1],1) ≥_lex (number[2],2)
(5) Kritischer Abschnitt
(6) number[i]=0
end loop
```

number[1]	number[2]
0	2

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[j]=0 or
          (number[j],j) ≥_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[j]=0 or
          (number[j],j) ≥_lex (number[2],2)
(5) Kritischer Abschnitt
(6) number[i]=0
end loop
```

number[1]	number[2]
0	2

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[j]=0 or
          (number[j],j) ≥_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[j]=0 or
          (number[j],j) ≥_lex (number[2],2)
(5) Kritischer Abschnitt
(6) number[i]=0
end loop
```

number[1]	number[2]
0	2

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[j]=0 or
          (number[j],j) ≥_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[j]=0 or
          (number[j],j) ≥_lex (number[2],2)
(5) Kritischer Abschnitt
(6) number[i]=0
end loop
```

number[1]	number[2]
0	0

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) ≥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) ≥_lex (number[i],i);
    (8)  Kritischer Abschnitt
    (9)  number[i]=0
end loop
```

Korrektheit des Bakery-Algorithmus

```
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 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].

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)

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)

```
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
```

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)

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)

```
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
```

- 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)

Satz

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

```
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
```

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)

Satz

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

```
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
```

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.

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.

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

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

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!)

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] $= 0$.

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.

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.

Zusammenfassung

- Mutual-Exclusion-Problem bei n Prozessen
- Tournament-Algorithmen (einfach aber nicht schnell)
- Lamport-Algorithmus (schnell, aber kein bounded waiting)
- Bakery-Algorithmus (FIFO-Eigenschaft $=$ 0-bounded waiting, nicht schnell)
- Ausblick: Geht es besser?
  Wieviele gemeinsame Variablen benötigt man?
  Wie lange muss ein Prozess warten, bis er den kritischen Abschnitt betreten darf?