Das Mutual-Exclusion-Problem

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 2. November 2020

Übersicht

Einleitendes Beispiel: Too much milk

- Alice und Bob teilen sich einen Kühlschrank
- Sie wollen, dass stets genug Milch im Kühlschrank ist
- 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 einer von beiden geht Milch kaufen, wenn keine Milch im Kühlschrank ist.
- Einer von beiden 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";
```

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.

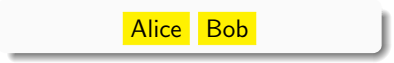
Als Prozesse:

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	Programm für Bob
(A1) schreibe a1;	(B1) schreibe b1;
(A2) if b2	(B2) if ¬a2
(A3) then schreibe a2;	(B3) then schreibe b2;
(A4) else entferne a2;	(B4) else entferne b2
(A5) while b1 ∧ ((a2 ∧ b2) ∨ (¬a2 ∧ ¬b2))	(B5) while a1 ∧ ((a2 ∧ ¬b2) ∨ (¬a2 ∧ b2))
(A6) do skip;	(B6) do skip;
(A7) if keine Milch	(B7) if keine Milch
(A8) then kaufe Milch;	(B8) then kaufe Milch;
(A9) entferne a1;	(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 informell:
  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 Initialisierungscodes, 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 machen die Beweise direkt (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. für gemeinsame Variablen

Abkürzung:

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

Dekkers Algorithmus

- Historisch die erste korrekte Lösung
- Theodorus Jozef Dekker = Niederländischer 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:

1. 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:

2. 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.
3. 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. 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, gilt: 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 ist 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

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 (amerik. Informatiker) 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 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. $\Longrightarrow$ Ein Prozess bleibt am `await` hängen. Anderer Prozess:

1. Er verbleibt für immer in seinem restlichen Code. $\Longrightarrow$ 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 von Joep L. W. Kessels (niederländischer Informatiker)
- 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:

$$\begin{aligned} \text{turn} &= 1, \quad \text{wenn turnp} = \text{turnq} \\ \text{turn} &= 2, \quad \text{wenn turnp} \neq \text{turnq} \end{aligned}$$

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)

Zusammenfassung

- Mutual-Exclusion-Problem: Programmiere Initalisierungscode + Abschlusscode, sodass wechselseitiger Ausschluss und Deadlockfreiheit gilt
- Unterschied: Deadlockfreiheit und Starvationfreiheit
- Drei Algorithmen für zwei Prozesse: Dekker, Peterson, Kessels
- Modell: Nur Lesen und Schreiben auf gemeinsamen Speicher ist atomar
- Ausblick: N statt zwei Prozesse und wieviel Speicher und Laufzeit wird benötigt?