# Zugriff auf mehrere Ressourcen

Prof. Dr. David Sabel

LFE Theoretische Informatik


Letzte Änderung der Folien: 4. Dezember 2020

## Übersicht





## Deadlocks bei mehreren Ressourcen

**Deadlock beim Mutual-Exclusion Problem**:

- Deadlock: Kein Prozess kommt in den kritischen Abschnitt, obwohl mindestens ein Prozess in den kritischen Abschnitt möchte
- Dies kommt dem Belegen **einer** Ressource gleich

**Jetzt:**

- Wir betrachten Prozesse, die **mehrere** solcher Ressourcen belegen möchten, wobei die einzelnen Ressourcen jeweils durch Sperren geschützt sind.
- Wir nehmen an, dass es nur Operationen zum Anfordern (request) und Freigeben (release) der Ressourcen gibt.
- Die genaue Implementierung der Sperren lassen wir dabei außer Acht (diese können z.B. durch Semaphore oder Monitore erfolgen).



## Veranschaulichung zum Belegen von Ressourcen

Prozess $P_1$ will Ressource $R_1$ belegen (hat sie aber nicht):


Prozess $P_1$ hat Ressource $R_1$ belegt:

## Deadlocks bei mehreren Ressourcen (2)

**Deadlocks allgemeiner (bei mehreren Ressourcen)**

**Definition**

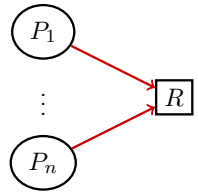
Eine Menge von Prozessen ist deadlocked (verklemmt), wenn jeder (nicht beendete) Prozess aus der Menge auf ein Ereignis wartet, das nur ein anderer Prozess aus der Menge herbeiführen kann.

Das Ereignis entspricht meist dem Freigeben einer Ressource.



## Beispiele

- Mutual-Exclusion Problem: Deadlock, wenn es nicht mehr möglich ist, dass irgendein Prozess den kritischen Abschnitt betritt, obwohl Prozesse in den kritischen Abschnitt möchten

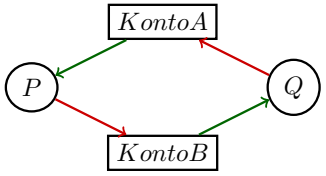





## Beispiele (2)

- Zwei Prozesse machen Umbuchungen zwischen Konto A und Konto B.
- Vorgehensweise:
  - Sperren des ersten Kontos
  - Sperren des zweiten Kontos
  - Überweisung von erstem Konto auf zweites Konto durchführen
  - Entsperren der Konten
- 

| Prozess P | Prozess Q |
|---|---|
| wait(KontoA); | wait(KontoB); |
| wait(KontoB); | wait(KontoA); |
| buche von A nach B | buche von B nach A |
| signal(KontoA); | signal(KontoB); |
| signal(KontoB); | signal(KontoA); |



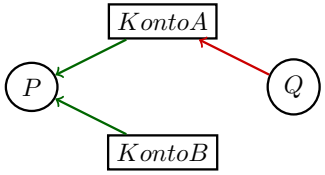

Deadlock!



## Beispiele (3)

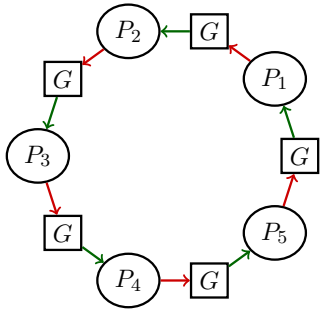
- Funktionierende Lösung (Deadlock-frei)
- 

| Prozess P | Prozess Q |
|---|---|
| wait(KontoA); | wait(KontoA); |
| wait(KontoB); | wait(KontoB); |
| buche von A nach B | buche von B nach A |
| signal(KontoA); | signal(KontoA); |
| signal(KontoB); | signal(KontoB); |



## Beispiele (4)

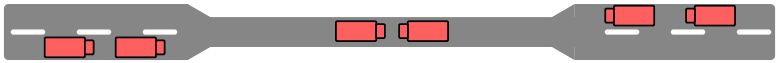
- Ähnliches Problem bei: Speisende Philosophen
- Deadlock möglich, wenn alle Philosophen unsychronisiert
- Alle haben die linke Gabel keiner die rechte.





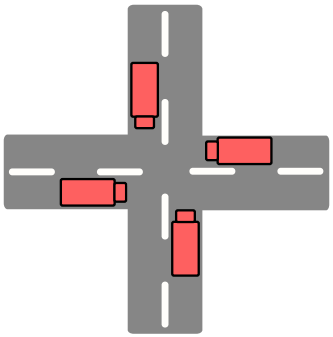

## Beispiele (5)

- Engstelle
- Nur ein Fahrzeug kann passieren





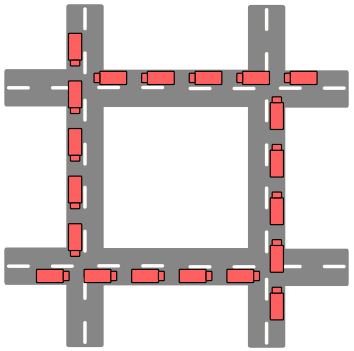
## Beispiele (6)



- Alle warten, dass “rechts frei” ist



## Beispiele (7)



- Gleiches Problem, aber etwas komplizierter

## Deadlock-Behandlung

**Vier Ansätze**

1. Ignorieren: Keine Vorkehrungen, Hoffnung, dass Deadlocks nur selten auftreten.
2. Deadlock-Erkennung und -Beseitigung: Laufzeitsystem erkennt Deadlocks und beseitigt sie. Problem: Finde Algorithmus der Deadlocks erkennt.
3. Deadlock-Vermeidung: Algorithmus verwaltet Ressourcen und lässt Situation nicht zu, die zu einem Deadlock führen können.
4. Deadlock-Verhinderung: Der Programmierer entwirft die Programme so, dass Deadlocks nicht auftreten können.



## Deadlock-Behandlung (2)

- Offensichtlich: Beste Methode: Deadlock-Verhinderung
- Dafür muss man wissen:
  Unter welchen Umständen kann ein Deadlock auftreten?
- Wir betrachten nun Bedingungen für Deadlock
  und Deadlock-Verhinderung



## Wann tritt Deadlock auf?

**Vier notwendige Bedingungen** (alle gleichzeitig erfüllt):

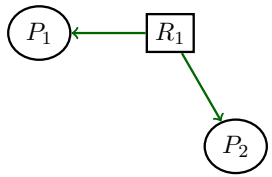
1. Wechselseitiger Ausschluss (Mutual-Exclusion): Nur ein Prozess kann gleichzeitig auf eine Ressource zugreifen,
2. Halten und Warten (Hold and Wait): Ein Prozess kann eine Ressource anfordern (auf eine Ressource warten), während er eine andere Ressource bereits belegt hat.
3. Keine Bevorzugung/Unterbrechung (No Preemption): Jede Ressource kann nur durch den Prozess freigegeben (entsperrt) werden, der sie belegt hat.
4. Zirkuläres Warten: Es gibt zyklische Abhängigkeit zwischen wartenden Prozessen: Jeder wartende Prozess möchte Zugriff auf die Ressource, die der nächste Prozesse im Zyklus belegt hat.



## Veranschaulichung der 4 Bedingungen

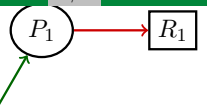
1. Wechselseitiger Ausschluss (Mutual-Exclusion):
   Nur ein ausgehender (grüner) Pfeil pro Ressource
2. Halten und Warten (Hold and Wait):
   Rote und grüne Pfeile für einen Prozess möglich
3. Keine Bevorzugung/Unterbrechung (No Preemption):
   Grüne Pfeile können nur verändert werden vom Prozess, der Pfeil hat
4. Zirkuläres Warten: Zyklus im Graphen

**Verboten**:



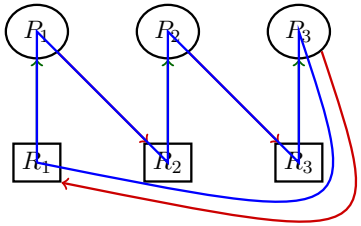

**Erlaubt**:





## Beispiel

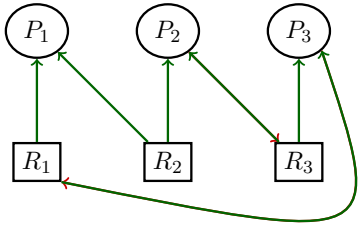
| Prozess | Prozess 2: | Prozess 3: |
|---|---|---|
| Request $R_1$ | Request $R_2$ | Request $R_3$ |
| Request $R_2$ | Request $R_3$ | Request $R_1$ |
| Release $R_1$ | Release $R_2$ | Release $R_3$ |
| Release $R_2$ | Release $R_3$ | Release $R_1$ |






## Beispiel: Anderes Scheduling

| Prozess | Prozess 2: | Prozess 3: |
|---|---|---|
| Request $R_1$ | Request $R_2$ | Request $R_3$ |
| Request $R_2$ | Request $R_3$ | Request $R_1$ |
| Release $R_1$ | Release $R_2$ | Release $R_3$ |
| Release $R_2$ | Release $R_3$ | Release $R_1$ |




**Deadlock-Verhinderung**: Programm so erstellt, dass unabhängig vom Scheduling kein Deadlock auftritt



**Deadlock-Vermeidung**: Scheduler erkennt Deadlock-Gefahr und schließt diese Schedulings aus

## Deadlock-Verhinderung

**Ansatz:** Greife eine der vier Bedingungen an, so dass sie nie wahr wird.

- Wechselseitiger Ausschluss: Im allgemeinen schwer möglich, manchmal aber schon: Z.B. Druckerzugriff wird durch Spooler geregelt.
- No Preemption: Schwierig, man kann z.B. nicht den Zugriff auf den Drucker entziehen, wenn Prozess noch nicht fertig gedruckt hat usw.



## Verhindern von Hold and Wait

- Möglichkeit: Prozess fordert zu Beginn alle Ressourcen an, die er benötigt.
- Philosophen: Exklusiver Zugriff auf alle Gabeln
- 1. Problem: Evtl. zu sequentiell
- 2. Problem: Oft nicht klar, welche Ressourcen jeder Prozess braucht
- Variation dieser Lösung: 2-Phasen Sperrprotokoll

## 2-Phasen Sperrprotokoll

Die Prozesse arbeiten in zwei Phasen

Jeder Prozess führt dabei aus:

- 1. Phase: Der Prozess versucht alle benötigten Ressourcen zu belegen.
  Ist **eine** benötigte Ressource **nicht frei**, so gibt der Prozess **alle** belegten Ressourcen zurück und der Prozess startet von neuem mit Phase 1.
- 2. Phase: Der Prozess hat alle benötigten Ressourcen
  Nachdem er fertig mit seiner Berechnung ist, gibt er alle Ressourcen wieder frei.



## Beispiel: Speisende Philosophen

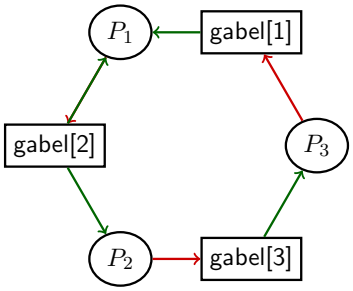
- Initial alle Gabeln (Semaphore) mit 1 initialisiert
- `tryWait`: Nicht-blockierende Implementierung von `wait`:

$$\texttt{tryWait}(S) = \begin{cases} \texttt{True}, & \text{wenn Semaphor belegt werden konnte} \\ \texttt{False}, & \text{sonst} \end{cases}$$

- Phase 1, Phase 2

Philosoph $i$

```
loop forever
(1)  l := tryWait(gabel[i]);
(2)  if l then
(3)     r:=tryWait(gabel[i+1]);
(4)     if r then
(5)        Philosoph isst
(6)        signal(gabel[i + 1]);
(7)        signal(gabel[i]);
(8)     else signal(gabel[i]);
(9)  Philosoph denkt;
end loop
```




- Deadlock nicht möglich!
- Aber Livelock!



## Timestamping-Ordering

- Prozesse erhalten eindeutigen Zeitstempel, wenn sie beginnen.

**Zwei-Phasen Sperrprotokoll mit Timestamping**

Jeder Prozess geht dabei so vor:

- 1. Phase: Der Prozess versucht alle benötigten Ressourcen auf einmal zu sperren.
  Ist eine benötigte Ressource belegt mit **kleinerem Zeitstempel**, dann gibt Prozess **alle** Ressourcen frei und startet von neuem.
  Ist **eigener** Zeitstempel **kleiner**, dann wartet der Prozess auf die restlichen Ressourcen.
- 2. Phase: Wenn der Prozess erfolgreich in diese Phase gekommen ist, hat er alle benötigten Ressourcen. Er benutzt sie und gibt sie anschließend wieder frei.

Beachte: Neue Zeitstempel werden nur vergeben, nach erfolgreichem Durchlauf durch beide Phasen.



## Timestamping-Ordering (2)

- Deadlock-frei
- Livelock nicht möglich
- Starvation-frei.

**Definition**

Starvation ist eine Situation, in der ein Prozess niemals (nach beliebig vielen Berechnungsschritten) in der Lage ist, alle benötigten Ressourcen zu belegen.

## Zirkuläres Warten verhindern

- Versuche das zirkuläre Warten zu verhindern.
- Philosophen-Problem: $N$. Prozess hebt zuerst rechte Gabel
- Dann kann kein zirkuläres Warten enstehen
- Denn die Gabeln wurden **total geordnet**: $1 < 2 \ldots N$.
- Und die Philosophen haben die Ressourcen entsprechend dieser Ordnung belegt.



## Total-Order Theorem

**Allgemein gilt:**

**Total-Order Theorem**

Sind alle gemeinsamen Ressourcen durch eine totale Ordnung geordnet und jeder Prozess belegt seine benötigten Ressourcen in aufsteigender Reihenfolge bezüglich der totalen Ordnung, dann ist ein Deadlock unmöglich.

Beweis: durch Widerspruch. Annahme: Es gibt einen Deadlock.
D.h. es gibt Ressourcen $R_1, \ldots, R_n$ und Prozesse $P_1, \ldots, P_n$ mit

- Prozess $P_i$ hat Ressource $R_{i-1}$ belegt
- Prozess $P_i$ wartet auf Ressource $R_i$

Sei $R_j$ die kleinste Ressource aus $\{R_1, \ldots, R_n\}$ bezüglich der totalen Ordnung. Dann wartet Prozess $P_j$ auf Ressource $R_j$, wobei er Ressource $R_{j-1}$ schon belegt hat. Widerspruch.



## Total Order Theorem (2)

Man kann nachweisen:

**Lemma**

Ein Deadlock-freies System, indem die Belegung von (allen) *einzelnen* Ressourcen Starvation-frei ist, ist auch insgesamt Starvation-frei

Mit dem Total Order Theorem lässt sich folgern:

Wenn es eine totale Ordnung der Ressourcen gibt, die Ressourcen entsprechend dieser Ordnung belegt werden und die Belegung einzelner Ressourcen Starvation-frei ist, dann ist das Gesamtsystem Starvation-frei.



## Deadlock-Vermeidung

Erinnerung:

- Deadlock-Vermeidung: Algorithmus verwaltet Ressourcen und lässt Situation nicht zu, die zu einem Deadlock führen können.

Im folgenden:

- Beispiel von Dijkstra zur Deadlock-Vermeidung:
- Problem des Deadly Embrace
- Lösung: Bankier-Algorithmus

## Deadly Embrace

Annahme:

- Es gibt $m$ gleiche Ressourcen
- Jeder Prozesse $P_i$ benötigt eine gewisse Zahl $m_i \leq m$ dieser Ressource
- Prozesse fordern Ressourcen nach und nach an
- Die maximal benötigte Anzahl $m_i$ ist beim Start bekannt
- Hat ein Prozess seine maximale Anzahl, terminiert er und gibt die Ressourcen zurück.
- Problem: Implementiere Ressourcenverwalter



## Dijkstra's Veranschaulichung

- Ressource: Geld
- Prozesse sind Bankkunden
- Ressourcenverwalter: Bankier



## Deadlock-Vermeidung: Beispiel

- Vorhandene Ressourcen am Anfang: 98 EUR
- 2 Kunden, beide benötigen 50 EUR
- Beide Kunden haben bereits 48 EUR erhalten
- Beide Kunden fordern 1 EUR an.
- Soll der Bankier beide Anforderungen zulassen?
- Nein: Dann haben beide Kunden 49 EUR, die Bank 0 EUR
- Ein Deadlock ist eingetreten.
- Ein Kunde fordert 1 EUR an
- Soll er das Geld bekommen?
- Ja, denn danach ist der Zustand immer noch **sicher**
- **sicher** = Deadlock noch vermeidbar (muss nicht eintreten)



## Lösungsversuch

Naive Lösung:

- Kunden erhalten Reihenfolge
- Bankier bedient immer einen Kunden, bis er seinen maximalen Betrag erhalten hat
- Alle andere Kunden müssen warten
- Schlecht, da sequentieller Algorithmus

Deshalb:

- Zusätzliche Anforderung: Erlaube soviel Nebenläufigkeit wie möglich
- Lehne nur dann Anfrage ab, wenn der Zustand dann unsicher würde, d.h. ein Deadlock eintreten muss

## Bankier-Algorithmus: Datenstrukturen

- Annahme: Verschiedene Ressourcen, z.B. mehrere Währungen
- Vektor $\overrightarrow{A}$: Aktueller Vorrat in der Bank. Jede Komponente von $\overrightarrow{A}$ ist nicht-negative Ganzzahl und stellt Vorrat einer Währung dar

$\mathcal{P}$ Menge der Kunden (Prozesse). Für $P \in \mathcal{P}$ sei:

- $\overrightarrow{M_P}$ der Vektor der maximal durch Prozess $P$ anzufordernden Ressourcen
- $\overrightarrow{C_P}$ der Vektor der bereits an Prozess $P$ vergebenen Ressourcen



## Bankier-Algorithmus: Sicherer Zustand

sicher = Deadlock in der Zukunft vermeidbar

Ein Zustand (mit all seinen Vektoren) ist sicher,
wenn es eine Permutation $\pi$ der Prozesse $P_1, \ldots, P_n \in \mathcal{P}$ gibt,
sodass es für jeden Prozesse $P_i$ entsprechend der Reihenfolge der Permutation genügend Ressourcen gibt, wenn er dran ist.

Genügend Ressourcen bedeutet hierbei:

$$\overrightarrow{A} + \left( \sum_{\pi(j)<\pi(i)} \overrightarrow{C_{P_{\pi(j)}}} \right) - \overrightarrow{M_{P_i}} + \overrightarrow{C_{P_i}} \geq \overrightarrow{0}$$

- Zu den aktuell verfügbaren Ressourcen $\overrightarrow{A}$
  dürfen momentan vergebenen Ressourcen hinzuaddiert werden, deren zugehörige Prozesse entsprechend der Permutation vor $P_i$ vollständig bedient werden.



## Bankier-Algorithmus: Grundgerüst

- Bankier erhält eine Ressourcenanfrage $\overrightarrow{L_P}$ eines Prozesses $P \in \mathcal{P}$.
- Konsistenzbedingung: $\overrightarrow{L_P} + \overrightarrow{C_P} \leq \overrightarrow{M_P}$
- Bankier berechnet den Folgezustand, alsob $P$ die Ressourcen erhält, d.h.

$$\begin{array}{rcl} \overrightarrow{A} & := & \overrightarrow{A} - \overrightarrow{L_P} \\ \overrightarrow{C_P} & := & \overrightarrow{C_P} + \overrightarrow{L_P} \end{array}$$

- Anschließend: Teste ob Zustand noch sicher
- Wenn unsicher, dann stelle Ursprungszustand her und lasse $P$ warten

Wenn Kunde maximale Ressourcen erhält wird $\overrightarrow{A}$ automatisch angepasst,



## Bankier-Algorithmus: Test auf Sicherheit

```
function testeZustand(P, A):
  if P = ∅ then
      return "sicher"
  else
      if ∃P ∈ P mit M_P − C_P ≤ A then
          A := A + C_P;
          P := P \ {P};
          testeZustand(P, A)
      else
        return "unsicher"
```

## Eigenschaften

- Algorithmus berechnet eine der gesuchten Permutationen
- Laufzeit $O(|\mathcal{P}|^2)$!
- Kriterium ist ausschließlich „Kein Deadlock“ sonst keine „Optimierung“
- kleine Verbesserung:
  Wenn $\overrightarrow{A} \geq \overrightarrow{M_P} - \overrightarrow{C_P}$ (genügend Ressourcen vorhanden um $P$ komplett zu bedienen) dann ist nach Anfrage $\overrightarrow{L_P}$ der Zustand immer sicher (Test muss nicht ausgeführt werden)



## Beispiel

4 Ressourcen (EUR, USD, JYN, SFR)
4 Prozesse A, B, C, D
Aktueller Zustand

| Maximal-Werte | Erhaltene Werte | Verfügbare Ressourcen |
|---|---|---|
| $\overrightarrow{M_A} = (4,7,1,1)$ | $\overrightarrow{C_A} = (1,1,0,0)$ | $\overrightarrow{A} = (2,2,3,3)$ |
| $\overrightarrow{M_B} = (0,8,1,5)$ | $\overrightarrow{C_B} = (0,5,0,3)$ | |
| $\overrightarrow{M_C} = (2,2,4,2)$ | $\overrightarrow{C_C} = (0,2,1,0)$ | |
| $\overrightarrow{M_D} = (2,0,0,2)$ | $\overrightarrow{C_D} = (1,0,0,1)$ | |

Ist der Zustand sicher?



## Beispiel (2)

Wir betrachten nun die Anfrage $L_A = (2,2,0,0)$, d.h. Prozess A möchte zwei weitere EUR und zwei weitere USD belegen.
Nach Aktualisierung ( $\overrightarrow{A} := \overrightarrow{A} - \overrightarrow{L_A}$ und $\overrightarrow{C_A} := \overrightarrow{C_A} + \overrightarrow{L_A}$) erhalten wir den Zustand:

| Maximal-Werte | Erhaltene Werte | Verfügbare Ressourcen |
|---|---|---|
| $\overrightarrow{M_A} = (4,7,1,1)$ | $\overrightarrow{C_A} = (3,3,0,0)$ | $\overrightarrow{A} = (0,0,3,3)$ |
| $\overrightarrow{M_B} = (0,8,1,5)$ | $\overrightarrow{C_B} = (0,5,0,3)$ | |
| $\overrightarrow{M_C} = (2,2,4,2)$ | $\overrightarrow{C_C} = (0,2,1,0)$ | |
| $\overrightarrow{M_D} = (2,0,0,2)$ | $\overrightarrow{C_D} = (1,0,0,1)$ | |

Bleibt der Zustand sicher?