5. Synchronisation (4)

5. Synchronisation
5.1 Einführung
5.2 Kritische Abschnitte
5.3 Synchr.-Methoden
5.4 Deadlocks

Gliederung

- Ressourcen-Typen
- Hinreichende und notwendige Deadlock-Bedingungen
- Deadlock-**Erkennung** und -**Behebung**
- Deadlock-**Vermeidung** (**avoidance**): Banker-Algorithmus
- Deadlock-**Verhinderung** (**prevention**)

Deadlock-Erkennung (detection) (1)
-Vermeidung (avoidance)
-Verhinderung (prevention)

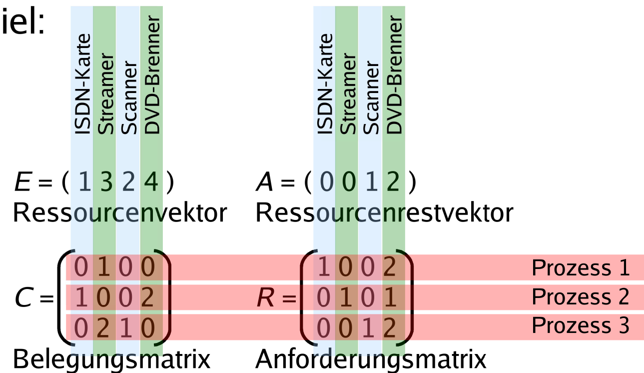
- Idee: Deadlocks zunächst zulassen
- System regelmäßig auf Vorhandensein von Deadlocks überprüfen und diese dann abstellen
- Nutzt drei Datenstrukturen:
  - Belegungsmatrix
  - Ressourcenrestvektor
  - Anforderungsmatrix

Deadlock-Erkennung (detection) (2)
-Vermeidung (avoidance)
-Verhinderung (prevention)

- n Prozesse $P_1, \dots, P_n$
- m Ressourcentypen $R_1, \dots, R_m$
  Vom Typ R_i gibt es E_i Ressourcen-Instanzen (i=1,...,m)
  -> **Ressourcenvektor** $E = (E_1\ E_2 \dots E_m)$
- **Ressourcenrestvektor** A (wie viele sind noch frei?)
- **Belegungsmatrix** C
  C_{ij} = Anzahl Ressourcen vom Typ j, die von Prozess i belegt sind
- **Anforderungsmatrix** R
  R_{ij} = Anzahl Ressourcen vom Typ j, die Prozess i noch benötigt

Deadlock-Erkennung (detection) (3)
-Vermeidung (avoidance)
-Verhinderung (prevention)

- Beispiel:



Deadlock-Erkennung (detection) (4)
-Vermeidung (avoidance)
-Verhinderung (prevention)

Algorithmus

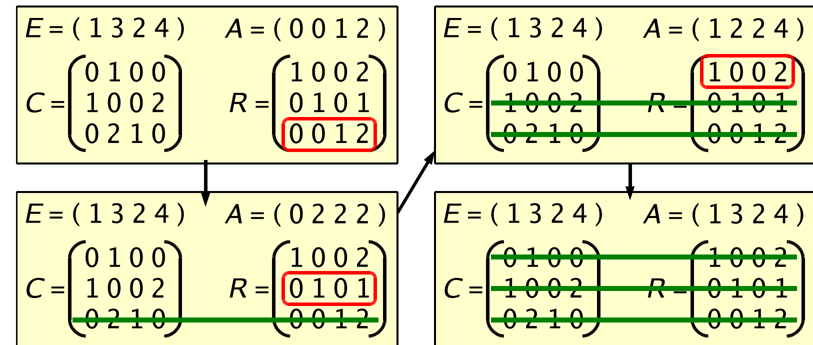
1. Suche einen unmarkierten Prozess P_i, dessen verbleibende Anforderungen vollständig erfüllbar sind, also $R_{ij} \leq A_j$ für alle j
2. Gibt es keinen solchen Prozess, beende Algorithmus
3. Ein solcher Prozess könnte erfolgreich abgearbeitet werden. Simuliere die Rückgabe aller belegten Ressourcen:

   $A := A+C_i$ (i-te Zeile von C)

   Markiere den Prozess – er ist nicht Teil eines Deadlocks
4. Weiter mit Schritt 1

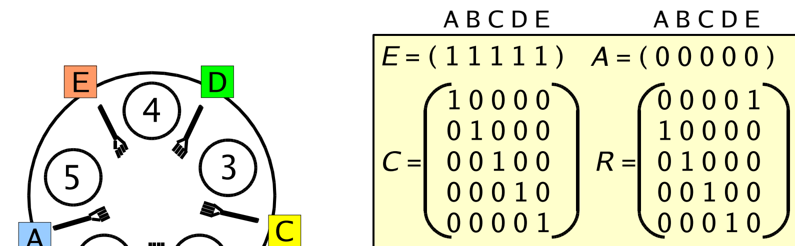
Deadlock-Erkennung (detection) (5)
-Vermeidung (avoidance)
-Verhinderung (prevention)

- Alle Prozesse, die nach diesem Algorithmus nicht markiert sind, sind an einem Deadlock beteiligt
- Beispiel



Deadlock-Erkennung (detection) (6)
-Vermeidung (avoidance)
-Verhinderung (prevention)

Beispiel (5 Philosophen)



- Algorithmus bricht direkt ab
- alle Prozesse sind Teil eines Deadlocks

Deadlock-Erkennung (detection) (7)
-Vermeidung (avoidance)
-Verhinderung (prevention)

Deadlock-Behebung: Was tun, wenn ein Deadlock erkannt wurde?

- **Entziehen** einer Ressource?
  In den Fällen, die wir betrachten, unmöglich (ununterbrechbare Ressourcen)
- **Abbruch** eines Prozesses, der am Deadlock beteiligt ist
- **Rücksetzen** eines Prozesses in einen früheren Prozesszustand, zu dem die Ressource noch nicht gehalten wurde
  - erfordert regelmäßiges Sichern der Prozesszustände

-Erkennung (detection)
Deadlock-Vermeidung (avoidance) (1)
-Verhinderung (prevention)

Deadlock Avoidance (Vermeidung)

- **Idee:** BS erfüllt Ressourcenanforderung nur dann, wenn dadurch auf keinen Fall ein Deadlock entstehen kann
- Das funktioniert nur, wenn man die **Maximalforderungen aller Prozesse** kennt
  - Prozesse registrieren **beim Start** für alle denkbaren Ressourcen ihren Maximalbedarf
  - für die Praxis i. d. R. irrelevant
  - nur in wenigen Spezialfällen nützlich

-Erkennung (detection)
Deadlock-Vermeidung (avoidance) (2)
-Verhinderung (prevention)

Sichere vs. unsichere Zustände

- Ein Zustand heißt **sicher**, wenn es eine Ausführreihenfolge der Prozesse gibt, die auch dann keinen Deadlock verursacht, wenn alle Prozesse sofort ihre maximalen Ressourcen-forderungen stellen.
- Ein Zustand heißt **unsicher**, wenn er nicht sicher ist.
- Unsicher bedeutet nicht zwangsläufig Deadlock!

-Erkennung (detection)
Deadlock-Vermeidung (avoidance) (3)
-Verhinderung (prevention)

Banker-Algorithmus (1)

- Idee: Liquidität im Kreditgeschäft
  - Kunden haben eine Kreditlinie (maximaler Kreditbetrag)
  - Kunden können ihren Kredit in Teilbeträgen in Anspruch nehmen, bis die Kreditlinie ausgeschöpft ist – dann zahlen sie den kompletten Kreditbetrag zurück
  - Prüfe bei Kreditanforderung, ob diese die Bank in einem **sicheren** Zustand lässt, was die Liquidität angeht – wird der Zustand unsicher, lehnt die Bank die Auszahlung ab

-Erkennung (detection)
Deadlock-Vermeidung (avoidance) (4)
-Verhinderung (prevention)

Banker-Algorithmus (2) – Beispiel

Bank: 1200 €,
900 € verliehen, 300 € Cash

☺	Max.	Aktueller Kredit
Kunde 1	1000 €	500 €
Kunde 2	400 €	200 €
Kunde 3	900 €	200 €

sicher, denn es gibt folgende Auszahlungs-/Rückzahlungsreihenfolge:

		(Bank)
K2: leiht	200 €	(100 €)
K2: rückz.	400 €	(500 €)
K1: leiht	500 €	(0 €)
K1: rückz.	1000 €	(1000 €)
K3: leiht	700 €	(300 €)
K3: rückz.	900 €	(1200 €)

Bank: 1200 €,
1000 € verliehen, 200 € Cash

☹	Max.	Aktueller Kredit
Kunde 1	1000 €	500 €
Kunde 2	400 €	200 €
Kunde 3	900 €	300 €

unsicher, weil es keine mögliche Auszahlungsreihenfolge gibt, die die Bank bedienen kann:

		(Bank)
K2: leiht	200 €	(0 €)
K2: rückz.	400 €	(400 €)
~~K1: leiht~~	~~500 €~~	~~(-100 €)~~
~~K3: leiht~~	~~600 €~~	~~(-200 €)~~

(letzte zwei unmöglich)

-Erkennung (detection)
Deadlock-Vermeidung (avoidance) (5)
-Verhinderung (prevention)

Banker-Algorithmus (3) – Beispiel

Bank: 1200 €,
900 € verliehen, 300 € Cash

☺	Max.	Aktueller Kredit
Kunde 1	1000 €	500 €
Kunde 2	400 €	200 €
Kunde 3	900 €	200 €

→ *Kunde 3 fordert 100 € an*

Bank: 1200 €,
1000 € verliehen, 200 € Cash

☹	Max.	Aktueller Kredit
Kunde 1	1000 €	500 €
Kunde 2	400 €	200 €
Kunde 3	900 €	300 €

↓ *Kunde 2 fordert 200 € an und zahlt alles zurück*

Bank: 400 € Cash		
Kunde 1	1000 €	500 €
Kunde 2	400 €	0 €
Kunde 3	900 €	300 €

↯

Übergang sicher → unsicher nicht zulassen!

-Erkennung (detection)
Deadlock-Vermeidung (avoidance) (6)
-Verhinderung (prevention)

Banker-Algorithmus (4)

- Datenstrukturen wie bei Deadlock-Erkennung:
  - n Prozesse $P_1 \ldots P_n$, m Ressourcentypen $R_1 \ldots R_m$ mit je E_i Ressourcen-Instanzen (i=1,...,m) -> **Ressourcenvektor** $E = (E_1\ E_2 \ldots E_m)$
  - **Ressourcenrestvektor** A (wie viele sind noch frei?)
  - **Belegungsmatrix** C
    C_{ij} = Anzahl Ressourcen vom Typ j, die Prozess i belegt
  - **Maximalbelegung** ***Max***:
    Max_{ij} = max. Bedarf, den Prozess i an Ressource j hat
  - **Maximale zukünftige Anforderungen:** $R = Max - C$,
    R_{ij} = Anzahl Ressourcen vom Typ j, die Prozess i noch maximal anfordern kann

-Erkennung (detection)
Deadlock-Vermeidung (avoidance) (7)
-Verhinderung (prevention)

Banker-Algorithmus (5)

Anforderung zulassen, falls

- Anforderung bleibt im Limit des Prozesses
- Zustand nach Gewähren der Anforderung ist sicher

Feststellen, ob ein Zustand sicher ist = Annehmen, dass alle Prozesse sofort ihre Maximalforderungen stellen, und dies auf Deadlocks überprüfen (siehe Algorithmus auf Folie 6)

-Erkennung (detection)
Deadlock-Vermeidung (avoidance) (8)
-Verhinderung (prevention)

Banker-Algorithmus (6) – Beispiel

$$\mathbf{C} = \begin{pmatrix} 0\ 1\ 0 \\ 2\ 0\ 0 \\ 3\ 0\ 2 \\ 2\ 1\ 1 \\ 0\ 0\ 2 \end{pmatrix} \quad \begin{matrix} E = (\ 10\ 5\ 7\ ) \\ \Rightarrow A = (\ \ 3\ 3\ 2\ ) \end{matrix} \quad \mathbf{Max} = \begin{pmatrix} 7\ 5\ 3 \\ 3\ 2\ 2 \\ 9\ 0\ 2 \\ 2\ 2\ 2 \\ 4\ 3\ 3 \end{pmatrix} \quad \mathbf{R{=}Max{-}C} = \begin{pmatrix} 7\ 4\ 3 \\ \mathbf{1\ 2\ 2} \\ 6\ 0\ 0 \\ 0\ 1\ 1 \\ 4\ 3\ 1 \end{pmatrix}$$

Anforderung (1 0 2) durch Prozess P2 – ok?

1. (1 0 2) < (1 2 2), also erste Bedingung erfüllt
2. Auszahlung simulieren

$$C' = \begin{pmatrix} 0\ 1\ 0 \\ \mathbf{3\ 0\ 2} \\ 3\ 0\ 2 \\ 2\ 1\ 1 \\ 0\ 0\ 2 \end{pmatrix} \quad \begin{matrix} E = (\ 10\ 5\ 7\ ) \\ \Rightarrow A' = (\ \ \mathbf{2\ 3\ 0}\ ) \end{matrix} \quad R' = \begin{pmatrix} 7\ 4\ 3 \\ \mathbf{0\ 2\ 0} \\ 6\ 0\ 0 \\ 0\ 1\ 1 \\ 4\ 3\ 1 \end{pmatrix}$$

Jetzt Deadlock-Erkennung durchführen

Deadlock-Vermeidung (avoidance) (9)

1 E = (10 5 7) A' = (2 3 0)
C' = (0 1 0 / 3 0 2 / 3 0 2 / 2 1 1 / 0 0 2), R' = (7 4 3 / [0 2 0] / 6 0 0 / 0 1 1 / 4 3 1)

2 E = (10 5 7) A' = (5 3 2)
C' = (0 1 0 / ~~3 0 2~~ / 3 0 2 / 2 1 1 / 0 0 2), R' = (7 4 3 / ~~0 2 0~~ / 6 0 0 / [0 1 1] / 4 3 1)

3 E = (10 5 7) A' = (7 4 3)
C' = (0 1 0 / ~~3 0 2~~ / 3 0 2 / ~~2 1 1~~ / 0 0 2), R' = ([7 4 3] / ~~0 2 0~~ / 6 0 0 / ~~0 1 1~~ / 4 3 1)

4 E = (10 5 7) A' = (7 5 3)
C' = (~~0 1 0~~ / ~~3 0 2~~ / 3 0 2 / ~~2 1 1~~ / 0 0 2), R' = (~~7 4 3~~ / ~~0 2 0~~ / [6 0 0] / ~~0 1 1~~ / 4 3 1)

5 E = (10 5 7) A' = (10 5 5)
C' = (~~0 1 0~~ / ~~3 0 2~~ / ~~3 0 2~~ / ~~2 1 1~~ / 0 0 2), R' = (~~7 4 3~~ / ~~0 2 0~~ / ~~6 0 0~~ / ~~0 1 1~~ / [4 3 1])

6 E = (10 5 7) A' = (10 5 7)
C' = (~~0 1 0~~ / ~~3 0 2~~ / ~~3 0 2~~ / ~~2 1 1~~ / ~~0 0 2~~), R' = (~~7 4 3~~ / ~~0 2 0~~ / ~~6 0 0~~ / ~~0 1 1~~ / ~~4 3 1~~)

OK!

-Erkennung (detection)
-Vermeidung (avoidance)
Deadlock-Verhinderung (prevention) (1)

Deadlock-Verhinderung (prevention):
Vorbeugendes Verhindern

- mache mindestens eine der vier Deadlock-Bedingungen unerfüllbar
  1. gegenseitiger Ausschluss
  2. Hold and Wait
  3. Ununterbrechbarkeit der Ressourcen
  4. Zyklisches Warten
- dann sind keine Deadlocks mehr möglich (denn die vier Bedingungen sind notwendig)

-Erkennung (detection)
-Vermeidung (avoidance)
Deadlock-Verhinderung (prevention) (2)

1. Gegenseitiger Ausschluss

- Ressourcen nur dann exklusiv Prozessen zuteilen, wenn es keine Alternative dazu gibt
- Beispiel: Statt mehrerer konkurrierender Prozesse, die einen gemeinsamen Drucker verwenden wollen, eine Drucker-Spooler einführen
  - keine Konflikte mehr bei Zugriff auf Drucker (Spooler-Prozess ist der einzige, der direkten Zugriff erhalten kann)
  - aber: Problem evtl. nur verschoben (Größe des Spool-Bereichs bei vielen Druckjobs begrenzt?)

-Erkennung (detection)
-Vermeidung (avoidance)
Deadlock-Verhinderung (prevention) (3)

2. Hold and Wait

- Alle Prozesse müssen die benötigten Ressourcen gleich beim Prozessstart anfordern (und blockieren)
- hat verschiedene Nachteile:
  - Ressourcen-Bedarf entsteht oft dynamisch (ist also beim Start des Prozesses nicht bekannt)
  - verschlechtert Parallelität (Prozess hält Ressourcen über einen längeren Zeitraum)
- Datenbanksysteme: **Two Phase Locking**
  - Sperrphase: Alle Ressourcen erwerben (wenn das nicht klappt -> alle sofort wieder freigeben)
  - Zugriffsphase (anschließend Freigabe)

-Erkennung (detection)
-Vermeidung (avoidance)
Deadlock-Verhinderung (prevention) (4)

3. Ununterbrechbarkeit der Ressourcen

- Ressourcen entziehen?
- siehe Deadlock-Behebung (Abbruch / Rücksetzen)

-Erkennung (detection)
-Vermeidung (avoidance)
Deadlock-Verhinderung (prevention) (5)

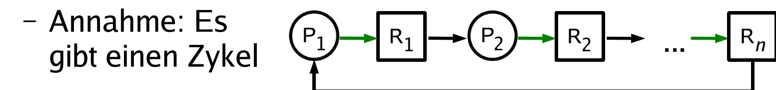
4. Zyklisches Warten (1)

- Ressourcen durchnumerieren
  - *ord*: $\mathbf{R} = \{R_1, ..., R_n\} \rightarrow \mathbb{N}$, $ord\,(R_i) \neq ord\,(R_j)$ für $i \neq j$
- Prozess darf Ressourcen nur in der durch ord vorgegebenen Reihenfolge anfordern
  - Wenn *ord* (R) < *ord* (S), dann ist die Sequenz
    ```
    lock (S);
    lock (R);
    ```
    ungültig
- Das macht Deadlocks unmöglich

-Erkennung (detection)
-Vermeidung (avoidance)
Deadlock-Verhinderung (prevention) (6)

4. Zyklisches Warten (2)

- Annahme: Es gibt einen Zykel



Für jedes *i* gilt: $ord\,(R_i) < ord\,(R_{i+1})$ und wegen des Zykels auch $ord\,(R_n) < ord\,(R_1)$, daraus folgt $ord\,(R_1) < ord\,(R_1)$: Widerspruch

- Problem: Gibt es eine feste Reihenfolge der Ressourcenbelegung, die für alle Prozesse geeignet ist?
- reduziert Parallelität (Ressourcen zu früh belegt)