

# Betriebssysteme Theorie

WS 2011/12

**Hans-Georg Eßer**  
Dipl.-Math., Dipl.-Inform.

Foliensatz E (10.12.2011)  
Deadlocks



10.12.2011

Betriebssysteme-Theorie, WS 2011/12, Hans-Georg Eßer

Folie E-1

## Was ist ein Deadlock?

- Eine Menge von Prozessen befindet sich in einer **Deadlock-Situation**, wenn:
  - jeder Prozess auf eine Ressource wartet, die von einem anderen Prozess blockiert wird
  - keine der Ressourcen freigegeben werden kann, weil der haltende Prozess (indem er selbst wartet) blockiert ist
- In einer Deadlock-Situation werden also die Prozesse dauerhaft verharren
- Deadlocks sind unbedingt zu vermeiden

10.12.2011

Betriebssysteme-Theorie, WS 2011/12, Hans-Georg Eßer

Folie E-3

## Deadlocks – Gliederung

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

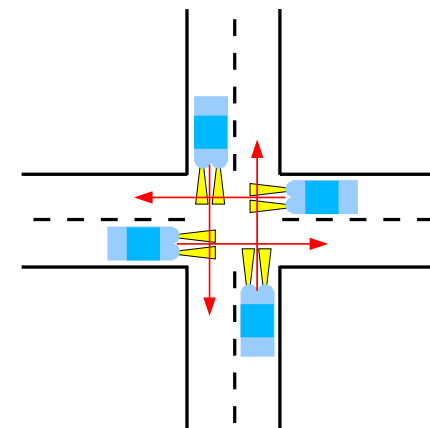
10.12.2011

Betriebssysteme-Theorie, WS 2011/12, Hans-Georg Eßer

Folie E-2

## Deadlock: Rechts vor Links (1)

- Der Klassiker: Rechts-vor-Links-Kreuzung



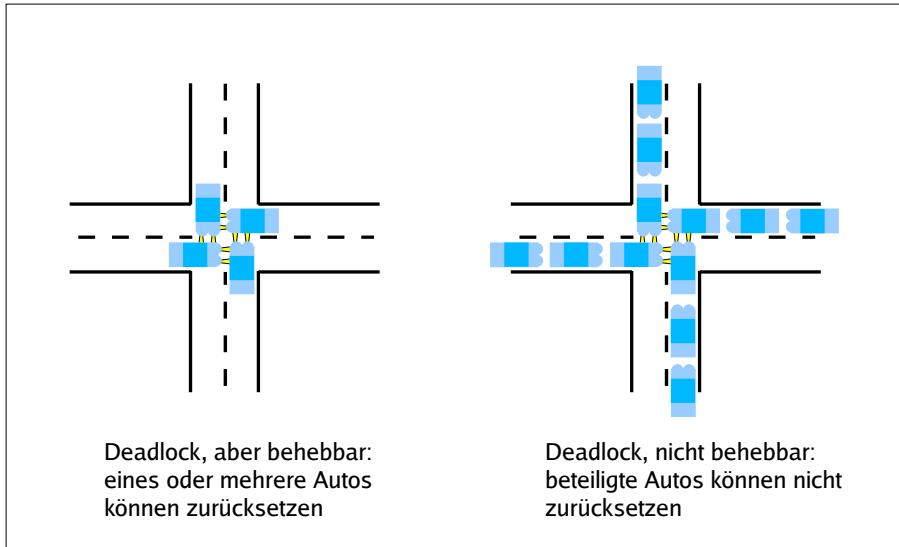
Wer darf fahren?  
Potenzieller Deadlock

10.12.2011

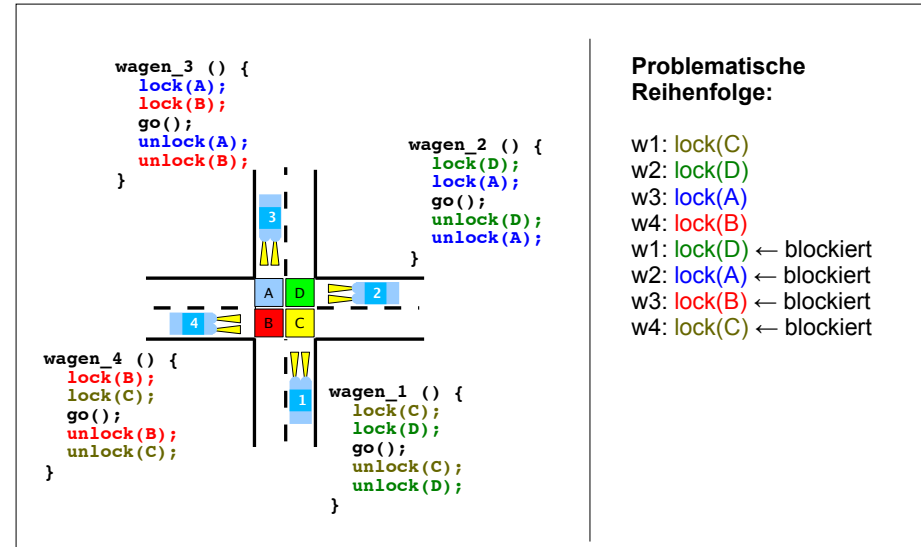
Betriebssysteme-Theorie, WS 2011/12, Hans-Georg Eßer

Folie E-4

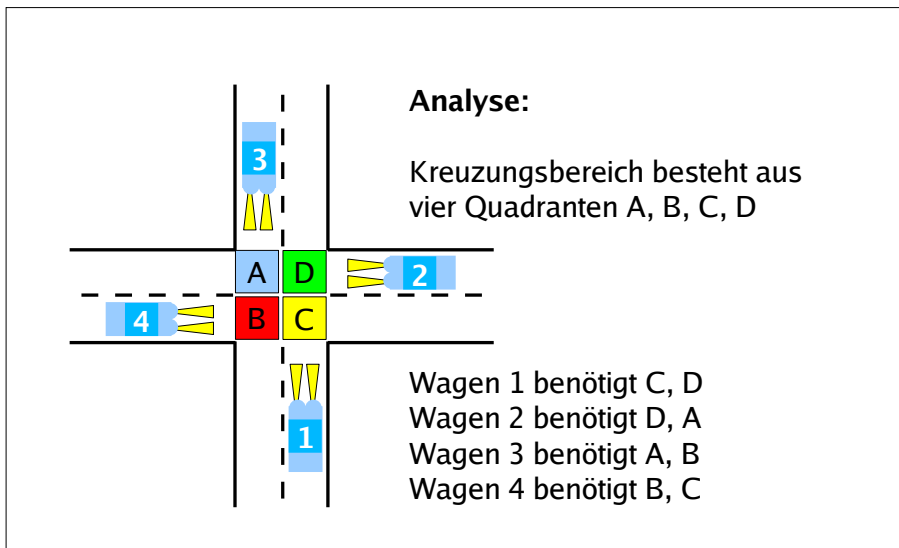
## Deadlock: Rechts vor Links (2)



## Deadlock: Rechts vor Links (4)



## Deadlock: Rechts vor Links (3)

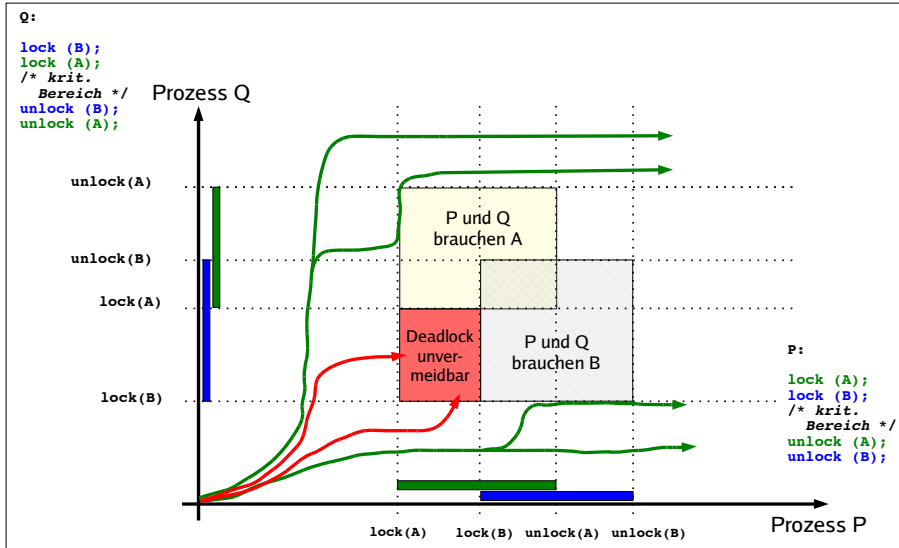


## Deadlock: kleinstes Beispiel (1)

- Zwei Locks A und B
  - z. B. A = Scanner, B = Drucker,
  - Prozesse P, Q wollen beide eine Kopie erstellen
- Locking in verschiedenen Reihenfolgen

Prozess P	Prozess Q	Problematische Reihenfolge:
lock(A);	lock(B);	P: lock(A)
lock(B);	lock(A);	Q: lock(B)
<i>/* krit. Bereich */</i>	<i>/* krit. Bereich */</i>	P: lock(B) ← blockiert
unlock(A);	unlock(B);	Q: lock(A) ← blockiert
unlock(B);	unlock(A);	

## Deadlock: kleinstes Beispiel (2)



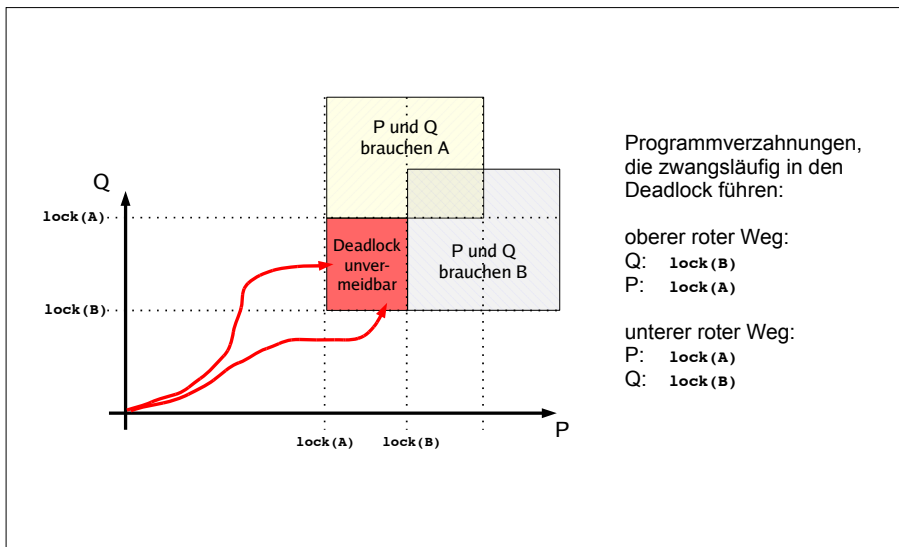
## Deadlock: kleinstes Beispiel (4)

- Problem beheben:  
P benötigt die Locks nicht gleichzeitig

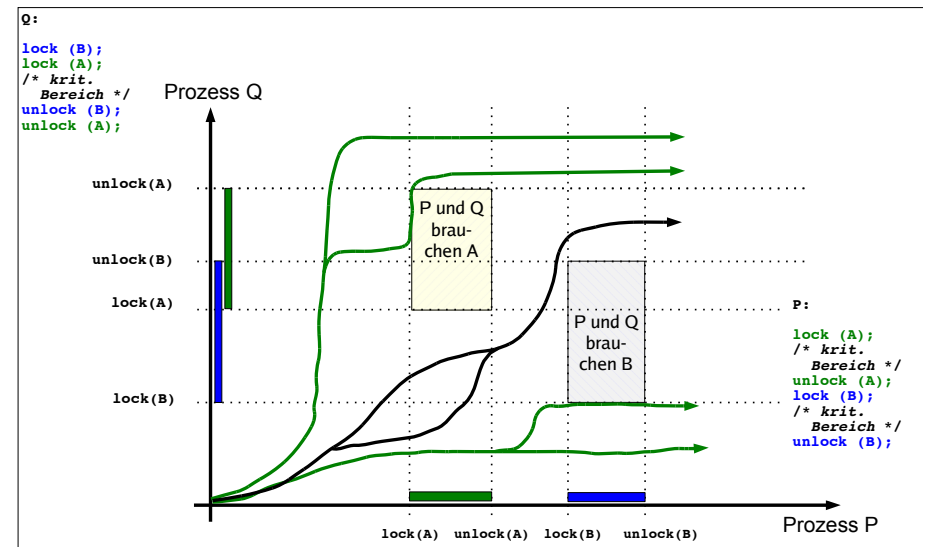
Prozess P	Prozess Q
<pre>lock (A); /* krit. Bereich */ unlock (A);</pre>	<pre>lock (B); lock (A); /* krit. Bereich */</pre>
<pre>lock (B); /* krit. Bereich */ unlock (B);</pre>	<pre>unlock (B); unlock (A);</pre>

- Jetzt kann kein Deadlock mehr auftreten
- Andere Lösung: P und Q fordern A, B in gleicher Reihenfolge an

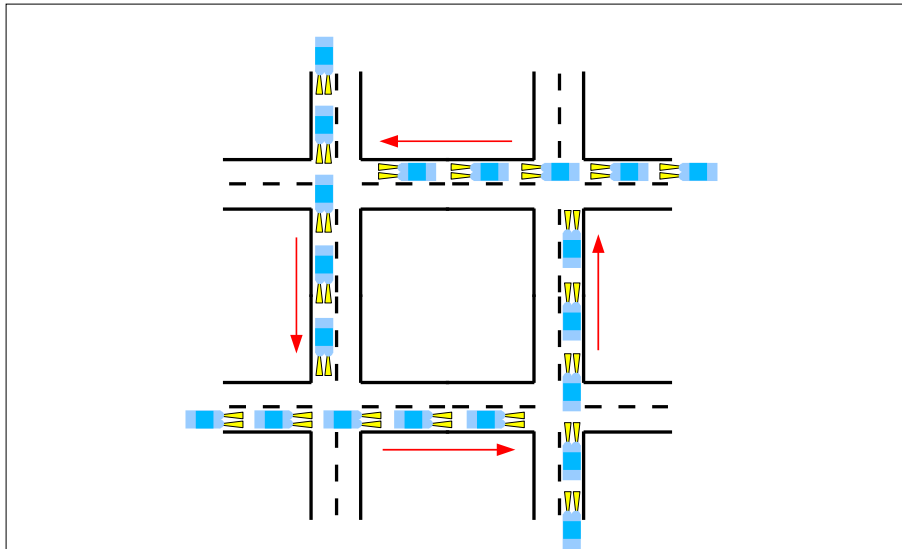
## Deadlock: kleinstes Beispiel (3)



## Deadlock: kleinstes Beispiel (5)



## Grid Lock

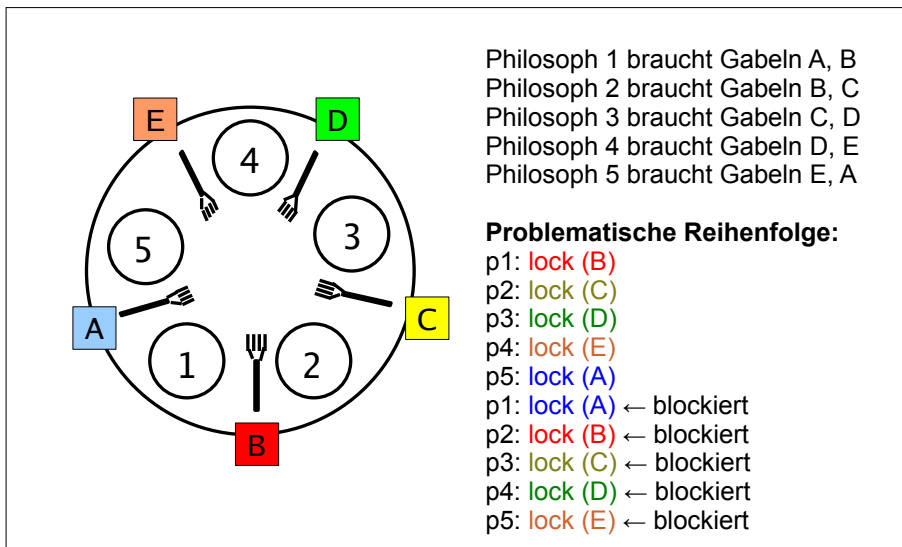


## Ressourcen-Typen (1)

### Zwei Kategorien von Ressourcen: unterbrechbar / nicht unterbrechbar

- unterbrechbare Ressourcen
  - Betriebssystem kann einem Prozess solche Ressourcen wieder entziehen
  - Beispiele:
    - CPU (Scheduler)
    - Hauptspeicher (Speicherverwaltung)
  - das kann Deadlocks vermeiden

## Fünf-Philosophen-Problem



## Ressourcen-Typen (2)

- nicht unterbrechbare Ressourcen
  - Betriebssystem kann Ressource nicht (ohne fehlerhaften Abbruch) entziehen – Prozess muss diese freiwillig zurückgeben
  - Beispiele:
    - DVD-Brenner (Entzug → zerstörter Rohling)
    - Tape-Streamer (Entzug → sinnlose Daten auf Band oder Abbruch der Bandsicherung wegen Timeout)
  - Nur die *nicht* unterbrechbaren sind interessant, weil sie Deadlocks verursachen können

## Ressourcen-Typen (3)

- wiederverwendbare vs. konsumierbare Ressourcen
  - **wiederverwendbar:** Zugriff auf Ressource zwar exklusiv, aber nach Freigabe wieder durch anderen Prozess nutzbar (Platte, RAM, CPU, ...)
  - **konsumierbar:** von einem Prozess erzeugt und von einem anderen Prozess konsumiert (Nachrichten, Interrupts, Signale, ...)

## Deadlock-Bedingungen (2)

- (1) bis (3) sind **notwendige** Bedingungen für einen Deadlock
- (1) bis (3) sind aber auch „wünschenswerte“ Eigenschaften eines Betriebssystems, denn:
  - gegenseitiger Ausschluss ist nötig für korrekte Synchronisation
  - Hold & Wait ist nötig, wenn Prozesse exklusiven Zugriff auf mehrere Ressourcen benötigen
  - Bei manchen Betriebsmitteln ist eine Präemption prinzipiell nicht sinnvoll (z. B. DVD-Brenner, Streamer)

## Deadlock-Bedingungen (1)

### 1. Gegenseitiger Ausschluss (mutual exclusion)

- Ressource ist exklusiv: Es kann stets nur ein Prozess darauf zugreifen

### 2. Hold and Wait (besitzen und warten)

- Ein Prozess ist bereits im Besitz einer oder mehrerer Ressourcen, und
- er kann noch weitere anfordern

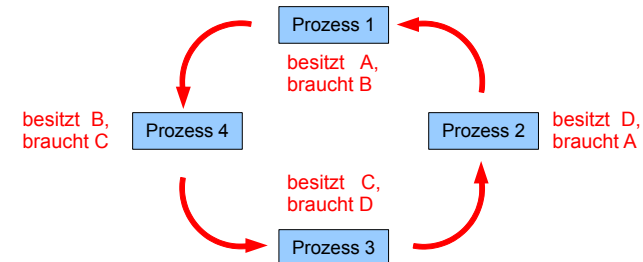
### 3. Ununterbrechbarkeit der Ressourcen

- Die Ressource kann nicht durch das Betriebssystem entzogen werden

## Deadlock-Bedingungen (3)

### 4. Zyklisches Warten

- Man kann die Prozesse in einem Kreis anordnen, in dem jeder Prozess eine Ressource benötigt, die der folgende Prozess im Kreis belegt hat



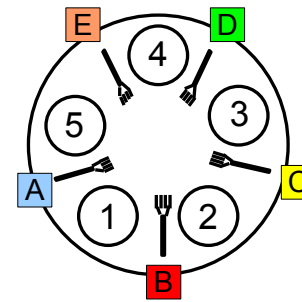
## Deadlock-Bedingungen (4)

1. Gegenseitiger Ausschluss
2. Hold and Wait
3. Ununterbrechbarkeit der Ressourcen
4. Zyklisches Warten

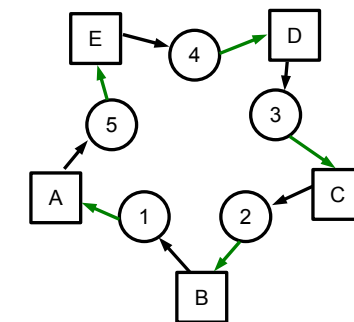
- (1) bis (4) sind **notwendige und hinreichende** Bedingungen für einen Deadlock
- Das zyklische Warten (4) (und dessen Unauflösbarkeit) sind Konsequenzen aus (1) bis (3)
- (4) ist der erfolgsversprechendste Ansatzpunkt, um Deadlocks aus dem Weg zu gehen

## Ressourcen-Zuordnungs-Graph (2)

Philosophen-Beispiel

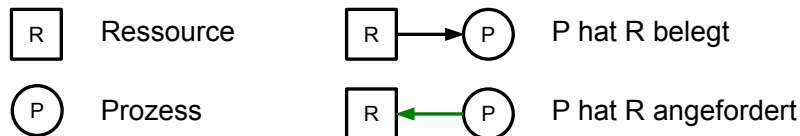


Situation, nachdem alle Philosophen ihre rechte Gabel aufgenommen haben

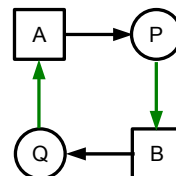


## Ressourcen-Zuordnungs-Graph (1)

- Belegung und (noch unerfüllte) Anforderung grafisch darstellen:

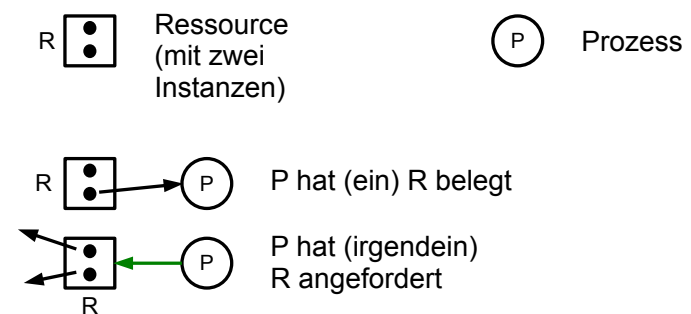


- P, Q aus Minimalbeispiel:
- Deadlock = Kreis im Graph



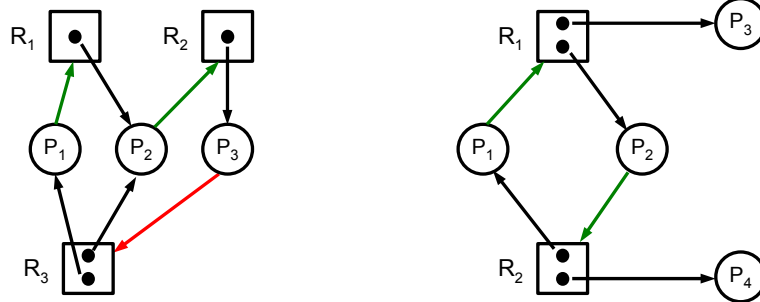
## Ressourcen-Zuordnungs-Graph (3)

- Variante für Ressourcen, die mehrfach vorkommen können



## Ressourcen-Zuordnungs-Graph (4)

- Beispiele mit mehreren Instanzen



Mit roter Kante ( $P_3 \rightarrow R_3$ ) gibt es einen Deadlock (ohne nicht)

Kreis, aber kein Deadlock – Bedingung ist nur **notwendig**, nicht hinreichend!

## 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 ( $i=1, \dots, m$ )  
→ **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) (1)

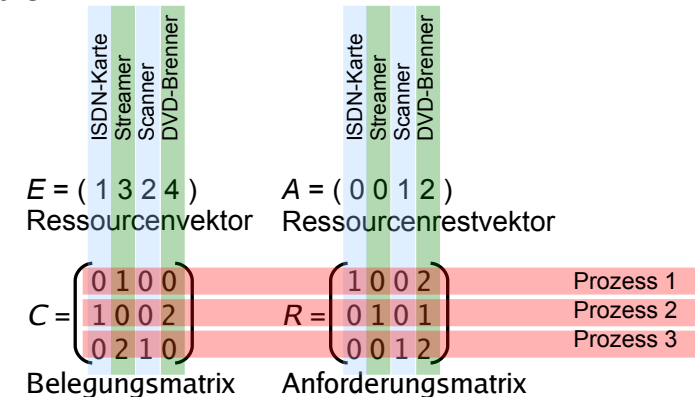
-Vermeidung (avoidance)  
-Verhinderung (prevention)

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

## Deadlock-Erkennung (detection) (3)

-Vermeidung (avoidance)  
-Verhinderung (prevention)

- Beispiel:



## Deadlock-Erkennung (detection) (4)

-Vermeidung (avoidance)  
-Verhinderung (prevention)

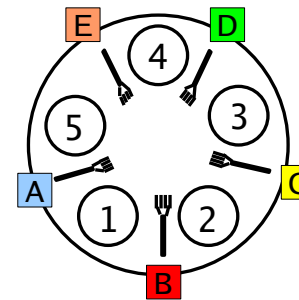
### Algorithmus

- Suche einen unmarkierten Prozess  $P_i$ , dessen verbleibende Anforderungen vollständig erfüllbar sind, also  $R_{ij} \leq A_j$  für alle  $j$
- Gibt es keinen solchen Prozess, beende Algorithmus
- 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
- Weiter mit Schritt 1

## Deadlock-Erkennung (detection) (6)

-Vermeidung (avoidance)  
-Verhinderung (prevention)

### Beispiel (5 Philosophen)



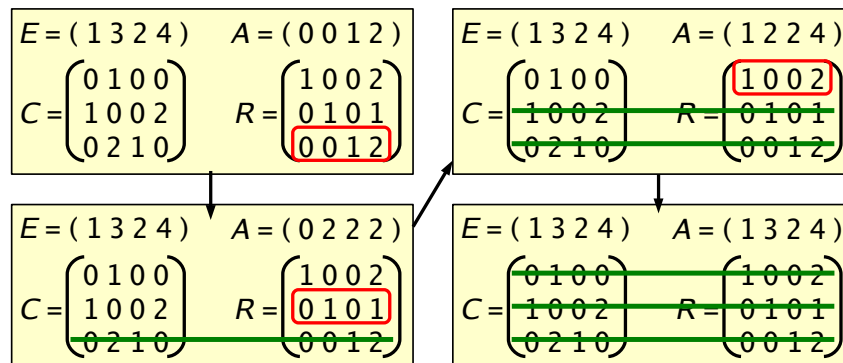
	A	B	C	D	E	A	B	C	D	E
$E =$	1	1	1	1	1	0	0	0	0	0
$A =$	0	0	0	0	0	0	0	0	0	0
$C =$	1	0	0	0	0	0	0	0	0	1
	0	1	0	0	0	0	1	0	0	0
	0	0	1	0	0	0	0	1	0	0
	0	0	0	1	0	0	0	0	1	0
	0	0	0	0	1	0	0	0	0	1
$R =$	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	1	0	0	0	0
	0	1	0	0	0	0	1	0	0	0
	0	0	1	0	0	0	0	1	0	0
	0	0	0	1	0	0	0	0	1	0

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

## 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) (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



## 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

## 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

## 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 Ressourcenforderungen stellen.
- Ein Zustand heißt **unsicher**, wenn er nicht sicher ist.
- Unsicher bedeutet nicht zwangsläufig Deadlock!

## Banker-Algorithmus (2) – Beispiel

Bank: 1200 €, 900 € verliehen, 300 € Cash	<b>sicher</b> , 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 €)
--	---	--

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

Bank: 1200 €, 1000 € verliehen, 200 € Cash	<b>unsicher</b> , weil es keine mögliche Auszahlungsreihenfolge gibt, die die Bank bedienen kann:	(Bank) K2: leiht 200 € ( 0 €) K2: rückz. 400 € ( 400 €) <del>K1: leiht 500 € (-100 €)</del> <del>K3: leiht 600 € (-200 €)</del> (letzte zwei unmöglich)
---	---	--

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

### 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

Übergang sicher → unsicher  
nicht zulassen!

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 €



### 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 E-29)

### Banker-Algorithmus (4)

- Datenstrukturen wie bei Deadlock-Erkennung:
  - $n$  Prozesse  $P_1 \dots P_n$ ,  $m$  Ressourcentypen  $R_1 \dots R_m$  mit je  $E_i$  Ressourcen-Instanzen ( $i=1, \dots, m$ )  
→ **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 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

### Banker-Algorithmus (6) – Beispiel

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

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

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

$$C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \rightarrow A' = (2 \ 3 \ 0) \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix} \quad \text{Jetzt Deadlock-Erkennung durchführen}$$

## Deadlock-Vermeidung (avoidance) (9)

<p>①</p> $E = (1057) \quad A' = (230)$ $C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$	<p>④</p> $E = (1057) \quad A' = (753)$ $C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$
<p>②</p> $E = (1057) \quad A' = (532)$ $C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$	<p>⑤</p> $E = (1057) \quad A' = (1055)$ $C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$
<p>③</p> $E = (1057) \quad A' = (743)$ $C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$	<p>⑥</p> $E = (1057) \quad A' = (1057)$ $C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$ <p style="text-align: right;">OK!</p>

## 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, einen 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?)

## 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)

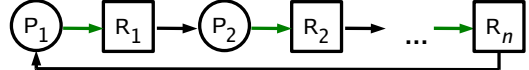
## 2. Hold and Wait

- 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)

### 3. Ununterbrechbarkeit der Ressourcen

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

### 4. Zyklisches Warten (2)

- Annahme: Es gibt einen Zykel 

Für jedes  $i$  gilt:  $ord(R_i) < ord(R_{i+1})$  und wegen des Zyklus 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)

### 4. Zyklisches Warten (1)

- Ressourcen durchnummerieren
  - $ord: R = \{R_1, \dots, R_n\} \rightarrow \mathbb{N}$ ,  $ord(R_i) \neq ord(R_j)$  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