

# Betriebssysteme

WS 2015/16

Hans-Georg Eßer

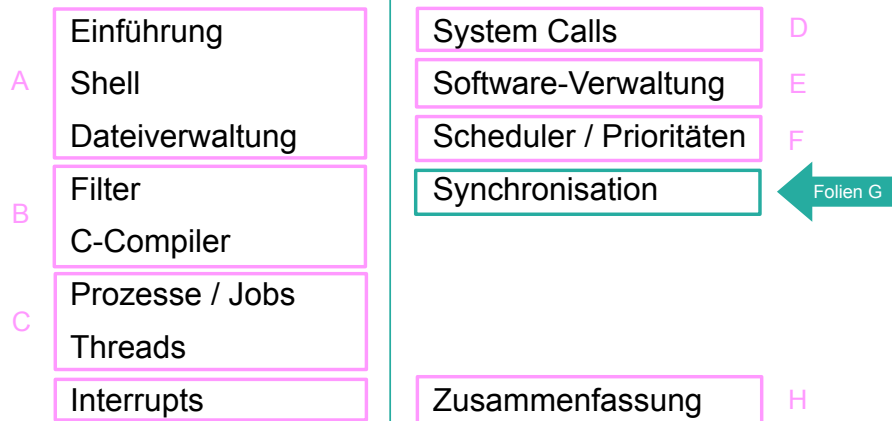
**Foliensatz G:**

v1.2, 2015/08/20

- Synchronisation
- Deadlocks

# Synchronisation

# Übersicht



# Einführung (1)

- Es gibt Prozesse (oder Threads oder Kernel-Funktionen) mit gemeinsamem Zugriff auf bestimmte Daten, z. B.
  - Threads des gleichen Prozesses: gemeinsamer Speicher
  - Prozesse / Threads öffnen die gleiche Datei zum Lesen / Schreiben
  - Mehr-Prozessor-System: Scheduler (je einer pro CPU) greifen auf gleiche Prozesslisten / Warteschlangen zu

- Synchronisation: Probleme mit „gleichzeitigem“ Zugriff auf Datenstrukturen
- Beispiel: Zwei Threads erhöhen einen Zähler

```
erhoehe_zaeher() {
  w=read(Adresse);
  w=w+1;
  write(Adresse,w);
}
```

Ausgangssituation: w=10

<b>T1:</b> w=read(Adresse); // 10 w=w+1; // 11 ----- write(Adresse,w); // 11 !!	<b>T2:</b> w=read(Adresse); // 10 w=w+1; // 11 ----- write(Adresse,w); // 11
---	--

Ergebnis nach T1, T2: w=11 – nicht 12!

- Ursache: erhoehe\_zaeher() arbeitet nicht **atomar**:
  - Scheduler kann die Funktion unterbrechen
  - Funktion kann auf mehreren CPUs gleichzeitig laufen
- Lösung: Finde alle Code-Teile, die auf gemeinsame Daten zugreifen, und stelle sicher, dass immer nur ein Prozess auf diese Daten zugreift (gegenseitiger Ausschluss, mutual exclusion)

- Gewünscht wäre eine der folgenden Reihenfolgen:

Ausgangssituation: w=10

<b>P1:</b> w=read(Adr); // 10 w=w+1; // 11 write(Adr,w); // 11 ----- w=read(Adr); // 11 w=w+1; // 12 write(Adr,w); // 12	<b>P2:</b> w=read(Adr); // 10 w=w+1; // 11 write(Adr,w); // 11 ----- w=read(Adr); // 11 w=w+1; // 12 write(Adr,w); // 12
---	---

Ergebnis nach P1, P2: w=12

Ausgangssituation: w=10

<b>P1:</b> w=read(Adr); // 10 w=w+1; // 11 write(Adr,w); // 11 ----- w=read(Adr); // 11 w=w+1; // 12 write(Adr,w); // 12	<b>P2:</b> w=read(Adr); // 10 w=w+1; // 11 write(Adr,w); // 11 ----- w=read(Adr); // 11 w=w+1; // 12 write(Adr,w); // 12
---	---

Ergebnis nach P1, P2: w=12

- Analoges Problem bei Datenbanken:

```
exec sql CONNECT ...
exec sql SELECT kontostand INTO $var FROM Konto
      WHERE kontonummer = $knr
$var = $var - abhebung
exec sql UPDATE Konto SET kontostand = $var
      WHERE kontonummer = $knr
exec sql DISCONNECT
```

Bei parallelem Zugriff auf gleichen Datensatz kann es zu Fehlern kommen

→ Lösung über (Datenbank-) **Transaktionen**, die u. a. **atomar und isoliert (ACID-Prinzip)** erfolgen müssen

### Race Condition:

- Mehrere parallele Threads / Prozesse nutzen eine gemeinsame Ressource
- Zustand hängt von Reihenfolge der Ausführung ab
- Race: die Threads liefern sich „ein Rennen“ um den ersten / schnellsten Zugriff

### Warum Race Conditions vermeiden?

- Ergebnisse von parallelen Berechnungen sind nicht eindeutig (d. h. potenziell falsch)
- Bei Programmtests könnte (durch Zufall) immer eine „korrekte“ Ausführreihenfolge auftreten; später beim Praxiseinsatz dann aber gelegentlich eine „falsche“.
- Race Conditions sind auch Sicherheitslücken

- Idee: Zugriff via Lock auf einen Prozess (Thread, ...) beschränken:

```
int Lock = UNLOCKED;           // global
erhoehe_zaebler( ) {
    while (Lock == LOCKED) { } ; // warten
    Lock = LOCKED;
    // Anfang Datennutzung
    w=read(Adresse);
    w=w+1;
    write(Adresse,w);
    // Ende Datennutzung
    Lock = UNLOCKED;
}
```

- Problem: Lock-Variable nicht geschützt

- Nicht alle Zugriffe sind problematisch:
  - Gleichzeitiges Lesen von Daten stört nicht
  - Prozesse, die „disjunkt“ sind (d. h.: die keine gemeinsamen Daten haben) können ohne Schutz zugreifen
- Sobald mehrere Prozesse/Threads/... gemeinsam auf ein Objekt zugreifen – und mindestens einer davon schreibend –, ist das Verhalten des Gesamtsystems **unvorhersehbar** und **nicht reproduzierbar**.

- Einführung, Race Conditions
- Kritische Abschnitte und gegenseitiger Ausschluss
- Synchronisationsmethoden, Standard-Primitive:
  - Mutexe
  - Semaphore

- Bestimmen des kritischen Abschnitts nicht ganz eindeutig:

```
void test () {  
    z = global[i];  
    z = z + 1;  
    global[i] = z;  
    // was anderes tun  
    z = global[j];  
    z = z - 1;  
    global[j] = z;  
}
```

- zwei kritische Abschnitte oder nur einer?

- Programmteil, der auf gemeinsame Daten zugreift
  - Müssen nicht verschiedene Funktionen sein: auch mehrere Instanzen (z. B. Threads), welche dieselbe Funktion ausführen
- Block zwischen erstem und letztem Zugriff
- Nicht „den Code schützen“, sondern die Daten
- Formulierung: kritischen Abschnitt „betreten“ und „verlassen“ (enter / leave critical section)

- Anforderung an parallele Threads:
  - Es darf maximal ein Thread gleichzeitig im kritischen Abschnitt sein
  - Kein Thread, der außerhalb kritischer Abschnitte ist, darf einen anderen blockieren
  - Kein Thread soll ewig auf das Betreten eines kritischen Abschnitts warten
  - Deadlocks sollen vermieden werden (z. B.: zwei Prozesse sind in verschiedenen kritischen Abschnitten und blockieren sich gegenseitig)

- Tritt nie mehr als ein Thread gleichzeitig in den kritischen Abschnitt ein, heißt das „**gegenseitiger Ausschluss**“ (englisch: **mutual exclusion**, kurz: mutex)
- Es ist Aufgabe der Programmierer, diese Bedingung zu garantieren
- Das Betriebssystem bietet Hilfsmittel, mit denen gegenseitiger Ausschluss durchgesetzt werden kann, schützt aber nicht vor Programmierfehlern

- **TSL** muss zwei Dinge leisten:
  - atomar laufen, so dass der Test-und-Setzen-Vorgang nicht durch einen anderen Prozess unterbrochen wird
  - Im Falle mehrerer CPUs den Speicherbus sperren, damit kein Prozess auf einer anderen CPU (deren Interrupts nicht gesperrt sind!) auf die gleiche Variable zugreifen kann

- **Maschineninstruktion** (z. B. mit dem Namen **TSL = Test and Set Lock**), die **atomar** eine Lock-Variable liest und setzt, also ohne dazwischen unterbrochen werden zu können.

```
enter:
    tsl register, flag ; Variablenwert in Register kopieren und
                       ; dann Variable auf 1 setzen
    cmp register, 0    ; War die Variable 0?
    jnz enter         ; Nicht 0: Lock war gesetzt, also Schleife
    ret

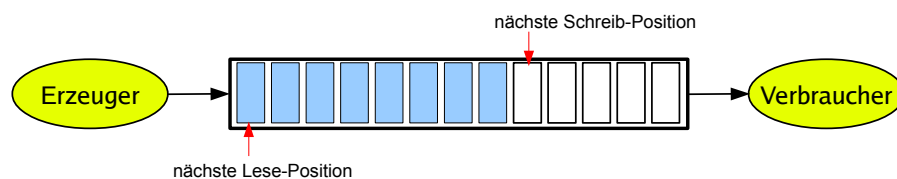
leave:
    mov flag, 0       ; 0 in flag speichern: Lock freigeben
    ret
```

- **Aktives Warten (busy waiting)**:
  - Ausführen einer Schleife, bis eine Variable einen bestimmten Wert annimmt.
  - Der **Thread ist bereit** und **belegt die CPU**.
  - Die Variable muss von einem anderen Thread gesetzt werden.
    - (Großes) Problem, wenn der andere Thread endet.
    - (Großes) Problem, wenn der andere Thread – z. B. wegen niedriger Priorität – nicht dazu kommt, die Variable zu setzen.

- **Passives Warten (sleep and wake):**
  - Ein **Thread blockiert** und wartet auf ein Ereignis, das ihn wieder in den Zustand „bereit“ versetzt.
  - Blockierter Thread **verschwendet keine CPU-Zeit**.
  - Ein anderer Thread muss das Eintreten des Ereignisses bewirken.
    - (Kleines) Problem, wenn der andere Thread endet.
  - Bei Eintreten des Ereignisses muss der blockierte Thread geweckt werden, z. B.
    - explizit durch einen anderen Thread,
    - durch Mechanismen des Betriebssystems.

- **Synchronisation**
  - **Puffer nicht überfüllen:**  
Wenn der Puffer voll ist, muss der Erzeuger warten, bis der Verbraucher eine Information aus dem Puffer abgeholt hat, und erst dann weiter arbeiten.
  - **Nicht aus leerem Puffer lesen:**  
Wenn der Puffer leer ist, muss der Verbraucher warten, bis der Erzeuger eine Information im Puffer abgelegt hat, und erst dann weiter arbeiten.

- Beim **Erzeuger-Verbraucher-Problem** (producer consumer problem, bounded buffer problem) gibt es zwei kooperierende Threads:
  - Der Erzeuger speichert Informationen in einem **beschränkten Puffer**.
  - Der Verbraucher liest Informationen aus diesem Puffer.



- Realisierung mit passivem Warten:
  - Eine gemeinsam benutzte Variable „count“ zählt die belegten Positionen im Puffer.
  - Wenn der Erzeuger eine Information einstellt und der Puffer leer war ( $\text{count} == 0$ ), weckt er den Verbraucher; bei vollem Puffer blockiert er.
  - Wenn der Verbraucher eine Information abholt und der Puffer voll war ( $\text{count} == \text{max}$ ), weckt er den Erzeuger; bei leerem Puffer blockiert er.

```

#define N 100 // Anzahl der Plätze im Puffer
int count = 0; // Anzahl der belegten Plätze im Puffer

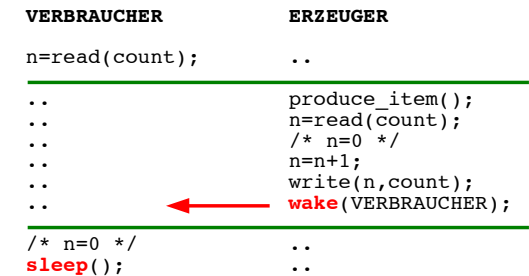
producer () {
    while (TRUE) { // Endlosschleife
        produce_item (item); // Erzeuge etwas für den Puffer
        if (count == N) sleep(); // Wenn Puffer voll: schlafen legen
        enter_item (item); // In den Puffer einstellen
        count = count + 1; // Zahl der belegten Plätze inkrementieren
        if (count == 1) wake(consumer); // war der Puffer vorher leer?
    }
}

consumer () {
    while (TRUE) { // Endlosschleife
        if (count == 0) sleep(); // Wenn Puffer leer: schlafen legen
        remove_item (item); // Etwas aus dem Puffer entnehmen
        count = count - 1; // Zahl der belegten Plätze dekrementieren
        if (count == N-1) wake(producer); // war der Puffer vorher voll?
        consume_item (item); // Verarbeiten
    }
}

```

- Das Programm enthält eine *race condition*, die zu einem *Deadlock* führen kann, z. B. wie folgt:
  - Verbraucher liest Variable count, die den Wert 0 hat.
  - Kontextwechsel zum Erzeuger.
  - Erzeuger stellt etwas in den Puffer ein, erhöht count und weckt den Verbraucher, da count vorher 0 war.
  - Verbraucher legt sich schlafen, da er für count noch den Wert 0 gespeichert hat (der zwischenzeitlich erhöht wurde).
  - Erzeuger schreibt den Puffer voll und legt sich dann auch schlafen.

- **Problemursache:** Wakeup-Signal für einen – noch nicht – schlafenden Prozess wird ignoriert
- Falsche Reihenfolge
- Weckruf „irgendwie“ für spätere Verwendung aufbewahren...



- Lösungsmöglichkeit: Systemaufrufe *sleep* und *wake* verwenden ein „**wakeup pending bit**“:
    - Bei *wake()* für einen nicht schlafenden Thread dessen wakeup pending bit setzen.
    - Bei *sleep()* das wakeup pending bit des Threads überprüfen – wenn es gesetzt ist, den Thread nicht schlafen legen.
- Aber: Lösung lässt sich nicht verallgemeinern (mehrere zu synchronisierende Prozesse benötigen evtl. zusätzliche solche Bits)

Ein **Semaphor** ist eine Integer- (Zähler-) Variable, die man wie folgt verwendet:

- Semaphor hat festgelegten Anfangswert N („Anzahl der verfügbaren Ressourcen“).
- Beim **Anfordern** eines Semaphors (P- oder **Wait-Operation**): P = (niederl.) probeer
  - Semaphor-Wert um 1 erniedrigen, falls er >0 ist,
  - Thread blockieren und in eine Warteschlange einreihen, wenn der Semaphor-Wert 0 ist.

- Pseudo-Code für Semaphor-Operationen

```
wait (sem) {
  if (sem>0)
    sem--;
  else {
    ADD_CALLER_TO (QUEUE(sem));
    SLEEP;
  }
}

signal (sem) {
  if (P in QUEUE(sem)) {
    WAKEUP (P);
    REMOVE (P, QUEUE);
  }
  else
    sem++;
}
```

- Bei **Freigabe** eines Semaphors (V- oder **Signal-Operation**): V = (niederl.) vrijgeven
  - einen Thread aus der Warteschlange wecken, falls diese nicht leer ist,
  - Semaphor-Wert um 1 erhöhen (wenn es keinen auf den Semaphor wartenden Thread gibt)
- Code sieht dann immer so aus:
 

```
wait (&sem);
/* Code, der die Ressource nutzt */
signal (&sem);
```
- in vielen Büchern: **P**(&sem), **V**(&sem)

- **Mutex**: boolesche Variable (true/false), die den Zugriff auf gemeinsam genutzte Daten synchronisiert
  - true: Zugang erlaubt
  - false: Zugang verboten
- **blockierend**: Ein Thread, der sich Zugang verschaffen will, während ein anderer Thread Zugang hat, blockiert → Warteschlange
- Bei Freigabe:
  - Warteschlange enthält Threads → einen wecken
  - Warteschlange leer: Mutex auf true setzen



- **Mutex (mutual exclusion) = binärer Semaphore**, also ein Semaphore, der nur die Werte 0 / 1 annehmen kann. Pseudo-Code:

```

wait (mutex) {
  if (mutex==1)
    mutex=0;
  else {
    ADD_CALLER_TO (QUEUE(mutex));
    SLEEP
  }
}

signal (mutex) {
  if (P in QUEUE(mutex)) {
    WAKEUP (P);
    REMOVE (P, QUEUE);
  }
  else
    mutex=1;
}

```

- Neue Interpretation: wait → lock  
signal → unlock
- Mutexe für exklusiven Zugriff (kritische Abschnitte)

- Bei Mutexen / Semaphore müssen die beiden Operationen wait() und signal() **atomar** implementiert sein:

Während der Ausführung von wait() / signal() darf kein anderer Prozess an die Reihe kommen

- Betriebssysteme können Mutexe und Semaphore **blockierend** oder **nicht-blockierend** implementieren
- blockierend:
  - wenn der Versuch, den Zähler zu erniedrigen, scheitert  
→ warten
- nicht blockierend:
  - wenn der Versuch scheitert  
→ vielleicht etwas anderes tun

- Mutexe / Semaphore verwalten Warteschlangen (der Prozesse, die schlafen gelegt wurden)
- Beim Aufruf von signal() muss evtl. ein Prozess geweckt werden
- Auswahl des zu weckenden Prozesses ist ein ähnliches Problem wie die Prozess-Auswahl im Scheduler
  - FIFO: **starker** Semaphore / Mutex
  - zufällig: **schwacher** Semaphore / Mutex

```

typedef int semaphore;
semaphore mutex = 1; // kontrolliert Zugriff auf Puffer
semaphore empty = N; // zählt freie Plätze im Puffer
semaphore full = 0; // zählt belegte Plätze im Puffer

producer() {
    while (TRUE) { // Endlosschleife
        produce_item(item); // erzeuge etwas für den Puffer
        wait (empty); // leere Plätze dekrementieren bzw. blockieren
        wait (mutex); // Eintritt in den kritischen Abschnitt
        enter_item (item); // in den Puffer einstellen
        signal (mutex); // kritischen Abschnitt verlassen
        signal (full); // belegte Plätze erhöhen, evtl. consumer wecken
    }
}

consumer() {
    while (TRUE) { // Endlosschleife
        wait (full); // belegte Plätze dekrementieren bzw. blockieren
        wait (mutex); // Eintritt in den kritischen Abschnitt
        remove_item(item); // aus dem Puffer entnehmen
        signal (mutex); // kritischen Abschnitt verlassen
        signal (empty); // freie Plätze erhöhen, evtl. producer wecken
        consume_entry (item); // verbrauchen
    }
}

```

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

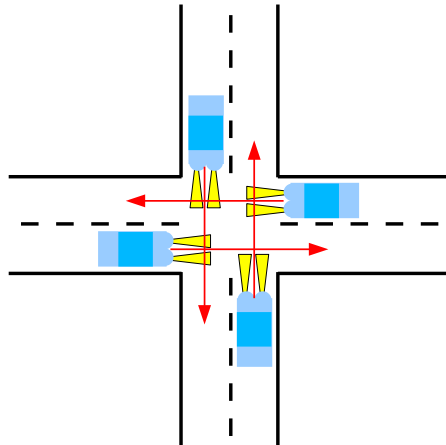
## Deadlocks

## 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

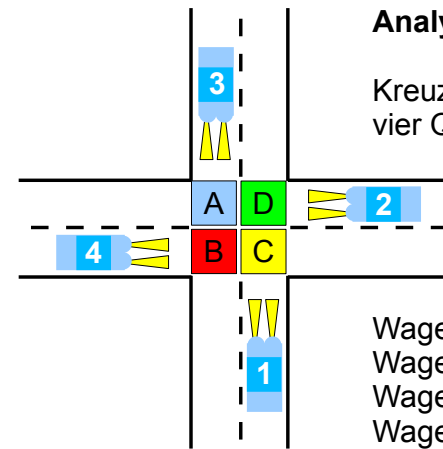
## Deadlock: Rechts vor Links (1)

- Der Klassiker: Rechts-vor-Links-Kreuzung



Wer darf fahren?  
Potenzieller Deadlock

## Deadlock: Rechts vor Links (3)

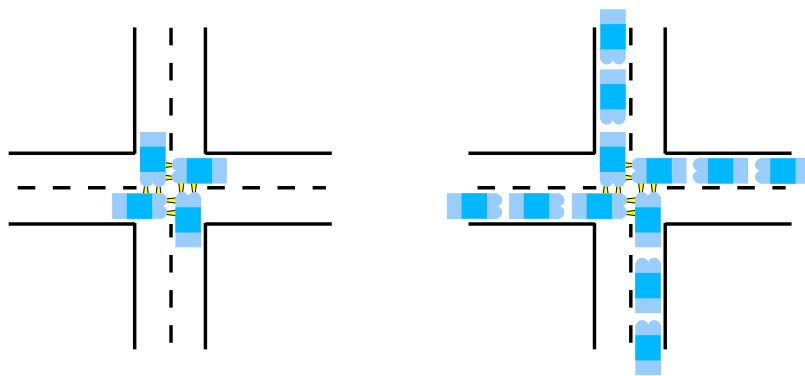


Analyse:

Kreuzungsbereich besteht aus vier Quadranten A, B, C, D

- Wagen 1 benötigt C, D
- Wagen 2 benötigt D, A
- Wagen 3 benötigt A, B
- Wagen 4 benötigt B, C

## Deadlock: Rechts vor Links (2)



Deadlock, aber behebbar:  
eines oder mehrere Autos  
können zurücksetzen

Deadlock, nicht behebbar:  
beteiligte Autos können nicht  
zurücksetzen

## Deadlock: Rechts vor Links (4)

```

wagen_3 () {
  lock(A);
  lock(B);
  go();
  unlock(A);
  unlock(B);
}

wagen_2 () {
  lock(D);
  lock(A);
  go();
  unlock(D);
  unlock(A);
}

wagen_4 () {
  lock(B);
  lock(C);
  go();
  unlock(B);
  unlock(C);
}

wagen_1 () {
  lock(C);
  lock(D);
  go();
  unlock(C);
  unlock(D);
}
    
```

Problematische  
Reihenfolge:

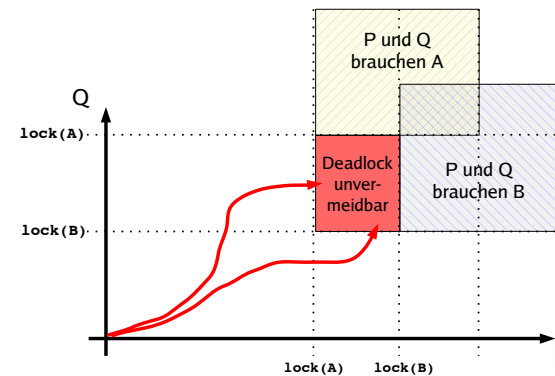
- w1: lock(C)
- w2: lock(D)
- w3: lock(A)
- w4: lock(B)
- w1: lock(D) ← blockiert
- w2: lock(A) ← blockiert
- w3: lock(B) ← blockiert
- w4: lock(C) ← blockiert

## 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

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

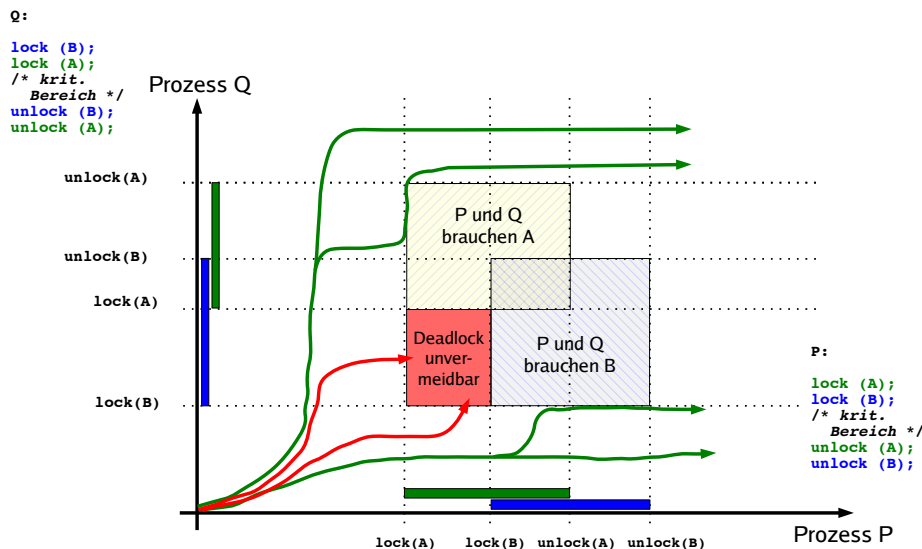
## Deadlock: kleinstes Beispiel (3)



Programmverzahnungen, die zwangsläufig in den Deadlock führen:

- oberer roter Weg:  
 Q: lock (B)  
 P: lock (A)
- unterer roter Weg:  
 P: lock (A)  
 Q: lock (B)

## Deadlock: kleinstes Beispiel (2)



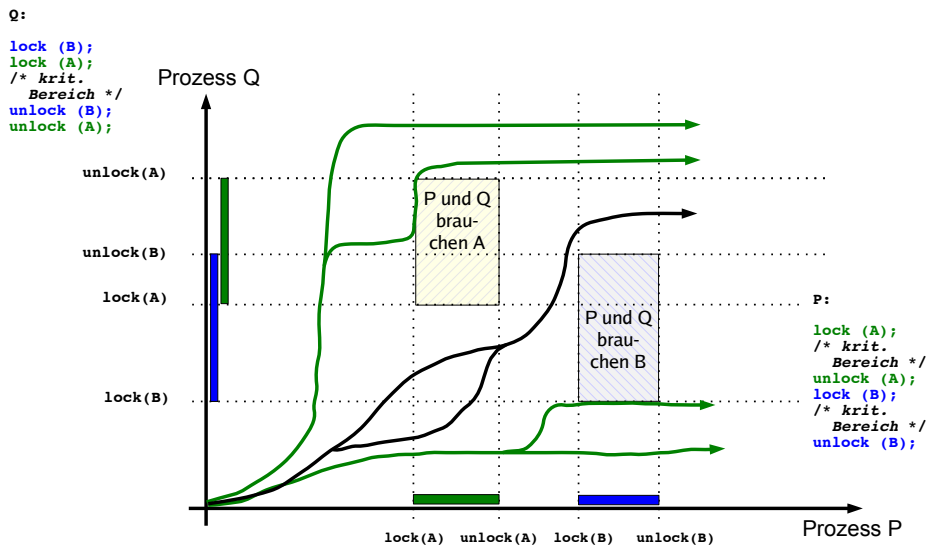
## Deadlock: kleinstes Beispiel (4)

- Problem beheben: P benötigt die Locks nicht gleichzeitig

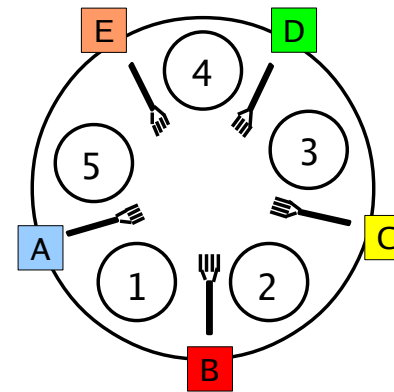
<pre> Prozess P lock (A); /* krit. Bereich */ unlock (A); lock (B); /* krit. Bereich */ unlock (B);                 </pre>	<pre> Prozess Q lock (B); lock (A); /* krit. Bereich */ unlock (B); unlock (A);                 </pre>
--	--

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

# Deadlock: kleinstes Beispiel (5)



# Fünf-Philosophen-Problem

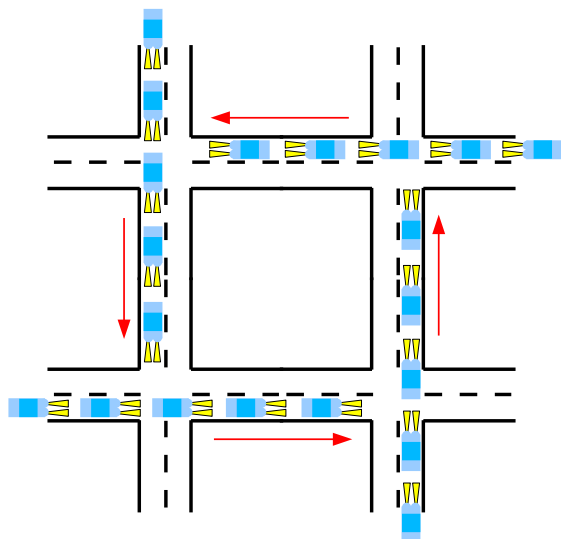


Philosoph 1 braucht Gabeln A, B  
 Philosoph 2 braucht Gabeln B, C  
 Philosoph 3 braucht Gabeln C, D  
 Philosoph 4 braucht Gabeln D, E  
 Philosoph 5 braucht Gabeln E, A

## Problematische Reihenfolge:

- p1: lock (B)
- p2: lock (C)
- p3: lock (D)
- p4: lock (E)
- p5: lock (A)
- p1: lock (A) ← blockiert
- p2: lock (B) ← blockiert
- p3: lock (C) ← blockiert
- p4: lock (D) ← blockiert
- p5: lock (E) ← blockiert

# 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

- 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

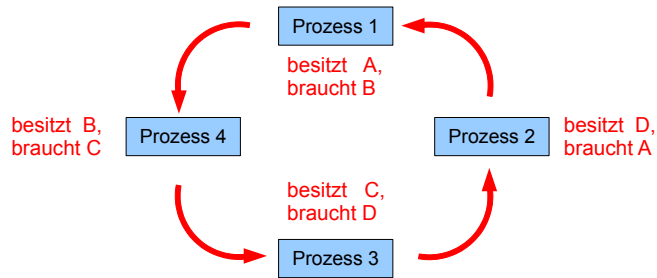
1. **Gegenseitiger Ausschluss (mutual exclusion)**
  - Ressource ist exklusiv: Es kann stets nur ein Prozess darauf zugreifen
2. **Hold and Wait (besitzen und warten)**
  - Ein Prozess ist bereits im Besitz einer oder mehrerer Ressourcen, und
  - er kann noch weitere anfordern
3. **Ununterbrechbarkeit der Ressourcen**
  - Die Ressource kann nicht durch das Betriebssystem entzogen werden

- 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, ...)

- (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)

## 4. Zyklisches Warten

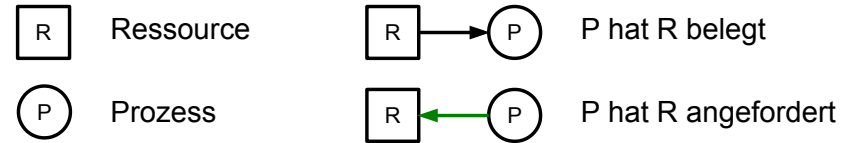
- Man kann die Prozesse in einem Kreis anordnen, in dem jeder Prozess eine Ressource benötigt, die der folgende Prozess im Kreis belegt hat



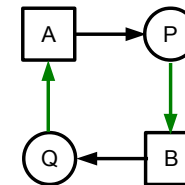
- Gegenseitiger Ausschluss
- Hold and Wait
- Ununterbrechbarkeit der Ressourcen
- Zyklisches Warten

- (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 erfolgversprechendste Ansatzpunkt, um Deadlocks aus dem Weg zu gehen

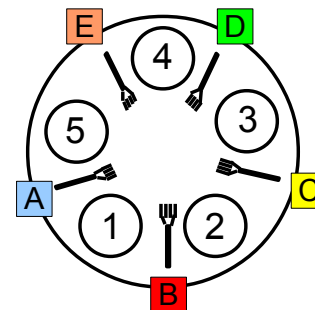
- Belegung und (noch unerfüllte) Anforderung grafisch darstellen:



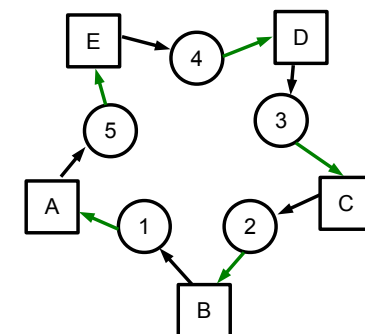
- P, Q aus Minimalbeispiel:
- Deadlock = Kreis im Graph



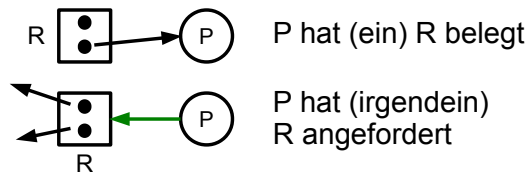
Philosophen-Beispiel



Situation, nachdem alle Philosophen ihre rechte Gabel aufgenommen haben

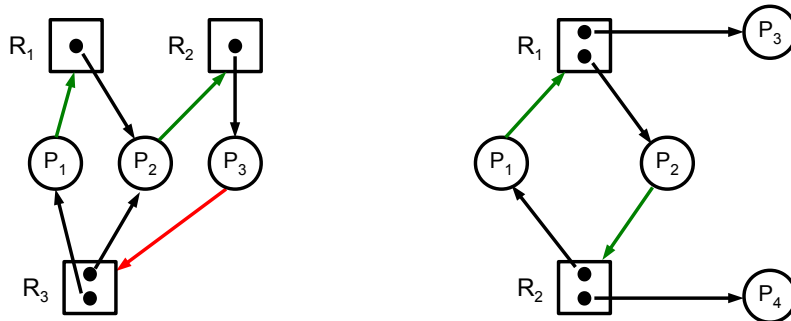


- Variante für Ressourcen, die mehrfach vorkommen können



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

- Beispiele mit mehreren Instanzen



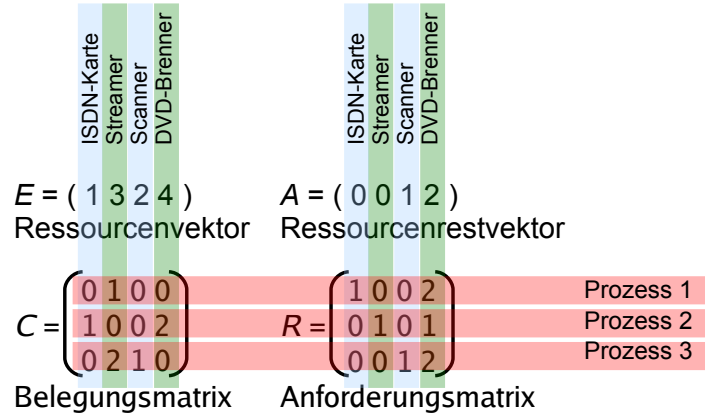
Mit roter Kante ( $P_3 \rightarrow R_3$ ) gibt es einen Deadlock (ohne nicht)

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

- $n$  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

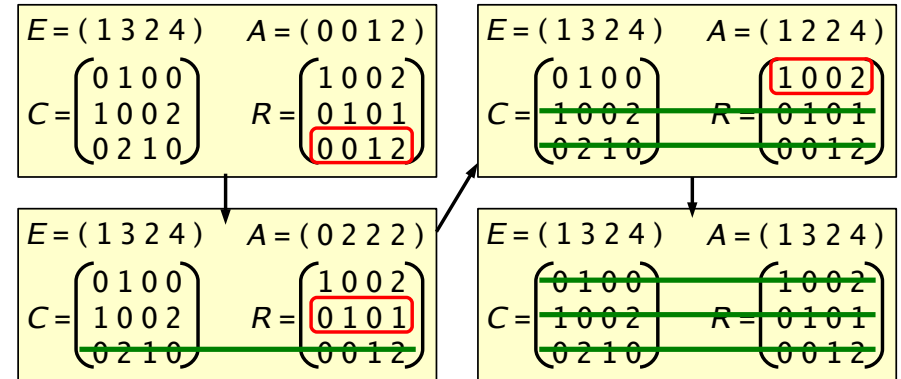


• Beispiel:



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

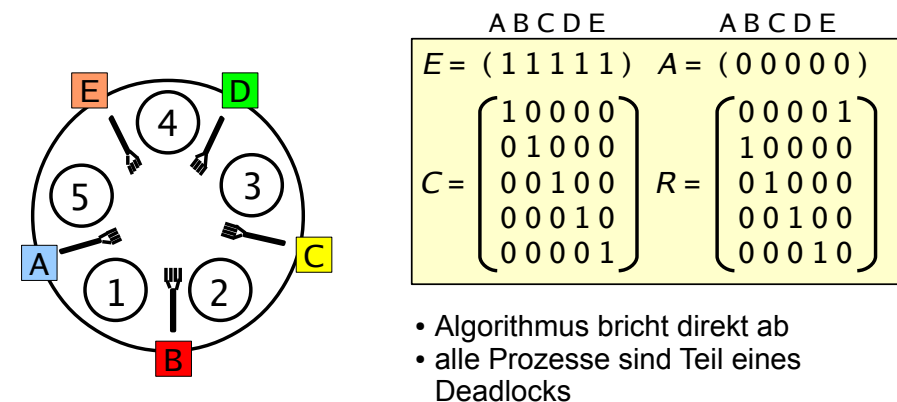
• Beispiel



**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

**Beispiel (5 Philosophen)**



## 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

## 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!

## 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

### Banker-Algorithmus (2) – Beispiel

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

**sicher**, denn es gibt folgende Auszahlungs-/Rückzahlungsreihenfolge:

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

(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

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

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

(Bank)

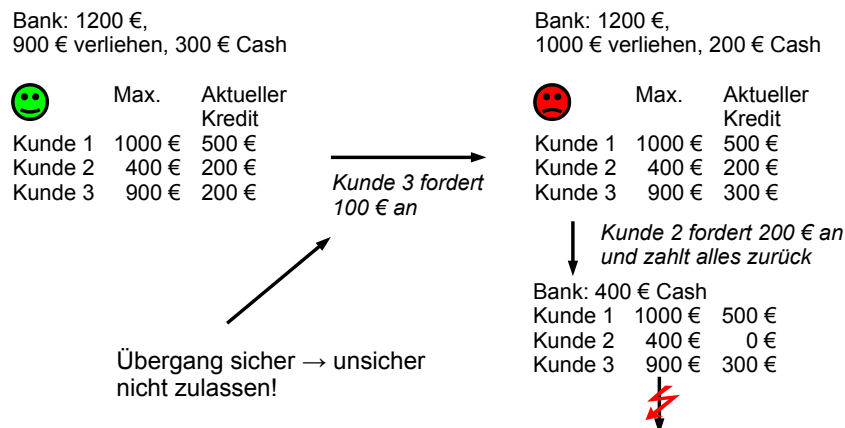
K2: leiht	200 €	( 0 €)
K2: rückz.	400 €	( 400 €)
<del>K1: leiht</del>	<del>500 €</del>	<del>(-100 €)</del>
<del>K3: leiht</del>	<del>600 €</del>	<del>(-200 €)</del>

(letzte zwei unmöglich)

### 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 (3) – Beispiel



### 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 G-67)

**Banker-Algorithmus (6) – Beispiel**

$$C = \begin{pmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \rightarrow E = (1057) \rightarrow A = (332)$$

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

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

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

$$C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \rightarrow E = (1057) \rightarrow A' = (230)$$

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

**Deadlock-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)

**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 Spooler-Bereichs bei vielen Druckjobs begrenzt?)

## 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)

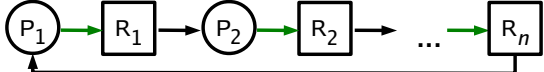
## 4. Zyklisches Warten (1)

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

## 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)