

```

Sep 19 14:20:18 amd64 sshd[20494]: Accepted rsa for esser from ::ffff:87.234.201.207 port 61557
Sep 19 14:27:41 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 01:00:01 amd64 /usr/sbin/cron[29278]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 20 02:00:01 amd64 /usr/sbin/cron[30103]: (root) CMD (/sbin/evlogmgr -c "age > *30d**")
Sep 20 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 12:46:44 amd64 sshd[6036]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62004
Sep 20 12:46:44 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 12:48:41 amd64 sshd[6039]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62105
Sep 20 12:56:44 amd64 sshd[6094]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62514
Sep 20 15:27:35 amd64 sshd[9077]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64242
Sep 20 15:27:35 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 16:37:11 amd64 sshd[10102]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63375
Sep 20 16:37:11 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 16:38:10 amd64 sshd[10140]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62546
Sep 21 01:00:01 amd64 /usr/sbin/cron[17055]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 21 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 02:00:01 amd64 /usr/sbin/cron[17978]: (root) CMD (/sbin/evlogmgr -c "age > *30d**")
Sep 21 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 17:43:26 amd64 sshd[31088]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63397
Sep 21 17:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 17:53:39 amd64 sshd[31269]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64391
Sep 21 18:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 18:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 01:00:01 amd64 /usr/sbin/cron[4674]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 22 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 02:00:01 amd64 /usr/sbin/cron[5498]: (root) CMD (/sbin/evlogmgr -c "age > *30d**")
Sep 22 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 20:23:23 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 01:00:01 amd64 /usr/sbin/cron[10311]: (root) CMD (/sbin/evlogmgr -c "age > *30d**")
Sep 23 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 02:00:01 amd64 /usr/sbin/cron[10311]: (root) CMD (/sbin/evlogmgr -c "age > *30d**")
Sep 23 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 18:04:05 amd64 sshd[6554]: Accepted publickey for esser from ::ffff:192.168.1.5 port 59771 sshd
Sep 23 18:04:05 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 18:04:34 amd64 sshd[6606]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62093
Sep 24 01:00:01 amd64 /usr/sbin/cron[12436]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 24 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 02:00:01 amd64 /usr/sbin/cron[13253]: (root) CMD (/sbin/evlogmgr -c "age > *30d**")
Sep 24 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 11:15:48 amd64 sshd[20988]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64456
Sep 24 11:15:48 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 13:49:08 amd64 sshd[12177]: Accepted rsa for esser from ::ffff:87.234.201.207 port 61330
Sep 24 13:49:08 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 15:42:07 amd64 kernel: snd_seq_mid_event: unsupported module, tainting kernel.
Sep 24 15:42:07 amd64 kernel: snd_seq_oss: unsupported module, tainting kernel.
Sep 24 20:25:31 amd64 sshd[29399]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62566
Sep 24 20:25:31 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 01:00:02 amd64 /usr/sbin/cron[662]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 25 01:00:02 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 02:00:01 amd64 /usr/sbin/cron[1404]: (root) CMD (/sbin/evlogmgr -c "age > *30d**")
Sep 25 02:00:02 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 10:59:25 amd64 sshd[8889]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64183
Sep 25 10:59:25 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 10:59:47 amd64 sshd[8921]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64253
Sep 25 11:30:02 amd64 sshd[9372]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62029
Sep 25 11:59:25 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 14:05:37 amd64 sshd[11554]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62822
Sep 25 14:05:37 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 14:06:10 amd64 sshd[11561]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62951
Sep 25 14:07:17 amd64 sshd[11608]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63392
Sep 25 14:08:23 amd64 sshd[11609]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63709
Sep 25 15:25:33 amd64 sshd[12930]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62778

```

# 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)

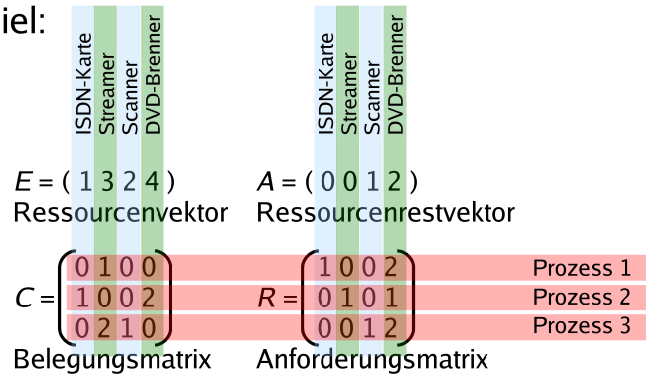
-Vermeidung (avoidance)  
-Verhinderung (prevention)

- $n$  Prozesse  $P_1, \dots, P_n$
- $m$  Ressourcentypen  $R_1, \dots, R_m$   
Vom Typ  $R_i$  gibt es  $E_i$  Ressourcen-Instanzen ( $i=1, \dots, m$ )  
-> **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) (3)

- Vermeidung (avoidance)
- Verhinderung (prevention)

• Beispiel:

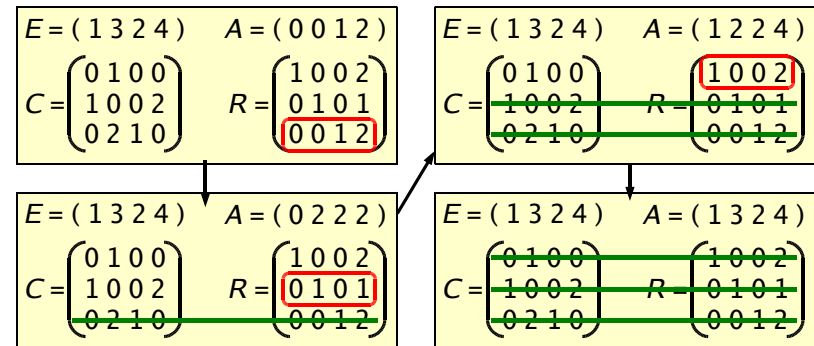


### Deadlock-Erkennung (detection) (5)

- Vermeidung (avoidance)
- Verhinderung (prevention)

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

• Beispiel



### Deadlock-Erkennung (detection) (4)

- Vermeidung (avoidance)
- Verhinderung (prevention)

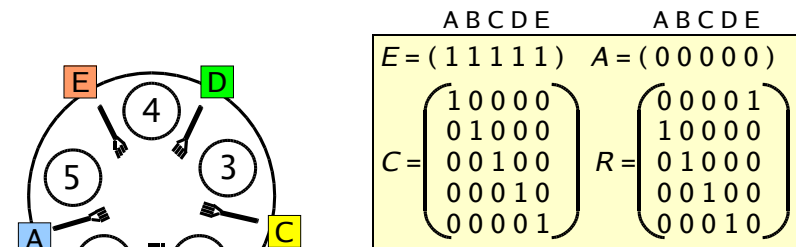
#### Algorithmus

- 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)



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

**-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) (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) (6)**  
 -Verhinderung (prevention)

**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

-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) (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**

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

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

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

$$C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \Rightarrow A' = \begin{pmatrix} 10 & 5 & 7 \\ 2 & 3 & 0 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix} \quad \text{Jetzt Deadlock-Erkennung durchführen}$$

-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)

**Deadlock-Vermeidung (avoidance) (9)**

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

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

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

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

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

6.  $E = (10\ 5\ 7) \quad A' = (10\ 5\ 7)$   
 $C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$  **OK!**

-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) (5)

#### 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

- Erkennung (detection)
- Vermeidung (avoidance)

### Deadlock-Verhinderung (prevention) (4)

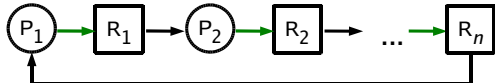
#### 3. Ununterbrechbarkeit der Ressourcen

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

- Erkennung (detection)
- Vermeidung (avoidance)

### Deadlock-Verhinderung (prevention) (6)

#### 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)