

```

Sep 19 14:20:18 amd64 sshd(20494): Accepted rsa for esser from ::ffff:87.234.201.207 port 61557
Sep 19 14:20:44 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 01:00:01 amd64 (user/sbin/cron[2978]): (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 20 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 02:00:01 amd64 (user/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(6516): 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 12:48:44 amd64 sshd(6609): Accepted rsa for esser from ::ffff:87.234.201.207 port 62105
Sep 20 12:54:44 amd64 sshd(6694): 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 15:27:15 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(10241): Accepted rsa for esser from ::ffff:87.234.201.207 port 63546
Sep 20 01:00:01 amd64 (user/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 (user/sbin/cron[17878]): (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 19:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 01:00:01 amd64 (user/sbin/cron[14674]): (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 (user/sbin/cron[16991]): (root) CMD (/sbin/evlogmgr -c "age > *30d*")
Sep 22 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 20:23:21 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 01:00:01 amd64 (user/sbin/cron[124739]): (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 23 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 02:00:01 amd64 (user/sbin/cron[124739]): (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 23 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
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 ssh2
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 (user/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 (user/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 syslog-ng[7653]: STATS: dropped 0
Sep 24 11:15:48 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 13:49:08 amd64 sshd(23197): 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_midi_event: unsupported module, tainting kernel.
Sep 24 15:42:07 amd64 syslog-ng[7653]: STATS: dropped 0
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 (user/sbin/cron[8621]): (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 (user/sbin/cron[1484]): (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(11586): 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 14:08:33 amd64 sshd(11630): 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

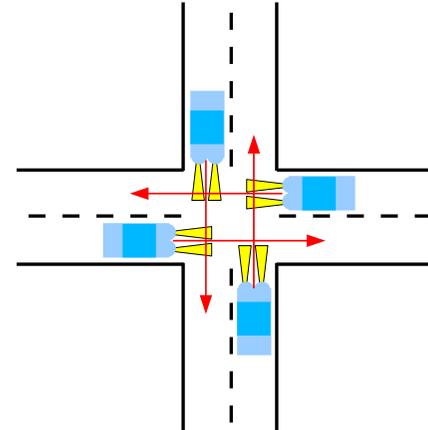
```

7 Deadlocks (1)

/home/esser/Daten/Dozent/Folien/bs-esser-21.odp

Deadlock: Rechts vor Links (1)

- Der Klassiker: Rechts-vor-Links-Kreuzung



Wer darf fahren?
Potenzieller Deadlock

Hans-Georg Eßer, FH München

Betriebssysteme I, WS 2006/07 – 2007-01-15

7. Deadlocks (1) – Folie 3

Was ist ein Deadlock?

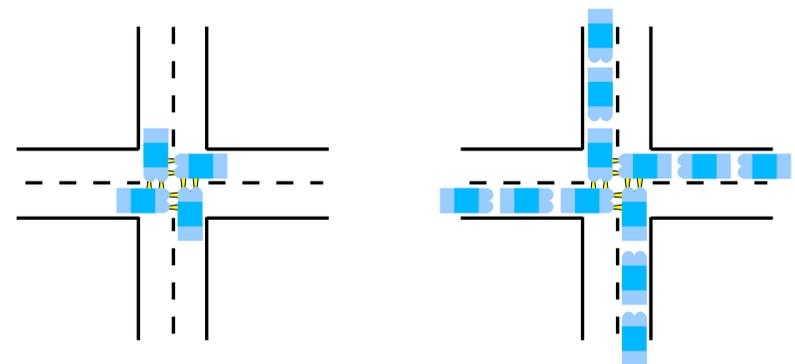
- Eine Menge von Prozessen befindet sich in einer **Deadlock-Situation**, wenn:
 - jeder Prozess auf eine Ressource wartet, die von einem anderen Prozess blockiert wird
 - keine der Ressourcen kann freigegeben werden, weil der haltende Prozess (indem er selbst wartet) blockiert ist
- In einer Deadlock-Situation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

Hans-Georg Eßer, FH München

Betriebssysteme I, WS 2006/07 – 2007-01-15

7. Deadlocks (1) – Folie 2

Deadlock: Rechts vor Links (2)



Deadlock, aber behebbar:
eines oder mehrere Autos
können zurücksetzen

Deadlock, nicht behebbar:
beteiligte Autos können
nicht zurücksetzen

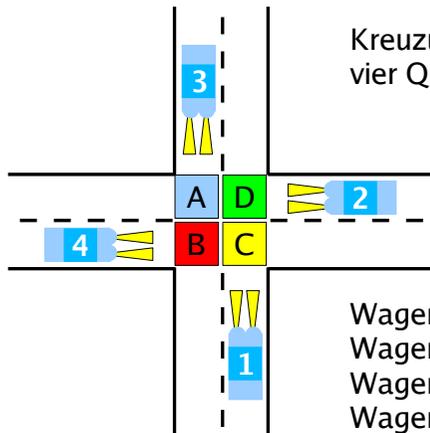
Hans-Georg Eßer, FH München

Betriebssysteme I, WS 2006/07 – 2007-01-15

7. Deadlocks (1) – Folie 4

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: 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

```
lock (A);
lock (B);

/* krit. Bereich */

unlock (B);
unlock (A);
```

Prozess Q

```
lock (B);
lock (A);

/* krit. Bereich */

unlock (A);
unlock (B);
```

Problematische Reihenfolge:

- P: lock(A)
- Q: lock(B)
- P: lock(B) <- blockiert
- Q: lock(A) <- blockiert

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 (2)

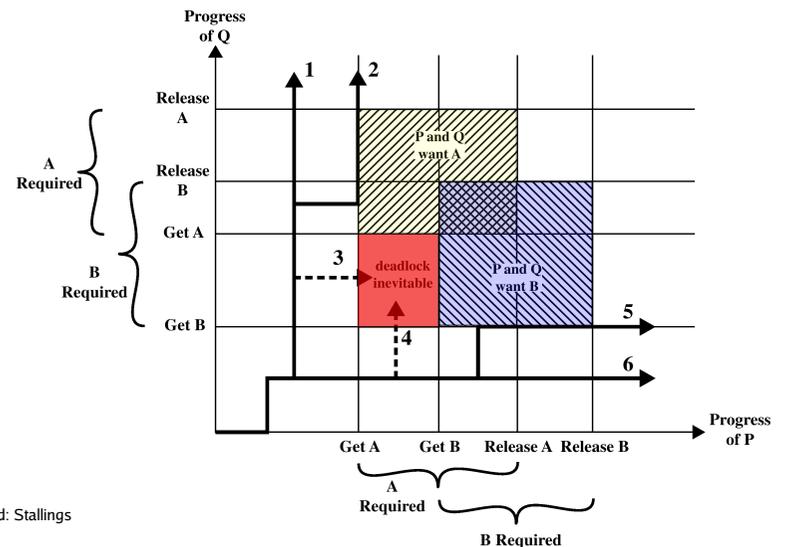


Bild: Stallings

Deadlock: kleinstes Beispiel (3)

- Problem beheben:
P benötigt die Locks nicht gleichzeitig

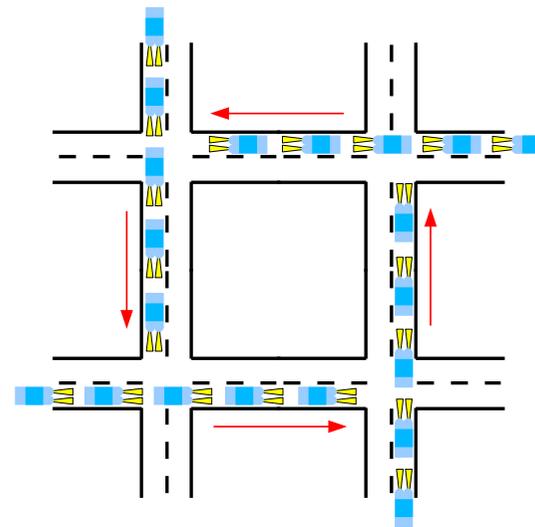
```

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

lock (B);           unlock (A);
/* krit. Bereich */ unlock (B);
unlock (B);         unlock (B);
    
```

- Jetzt kann kein Deadlock mehr auftreten

Deadlock: Grid Lock



- Der Klassiker:
Rechts-vor-Links-Kreuzung

Deadlock: kleinstes Beispiel (4)

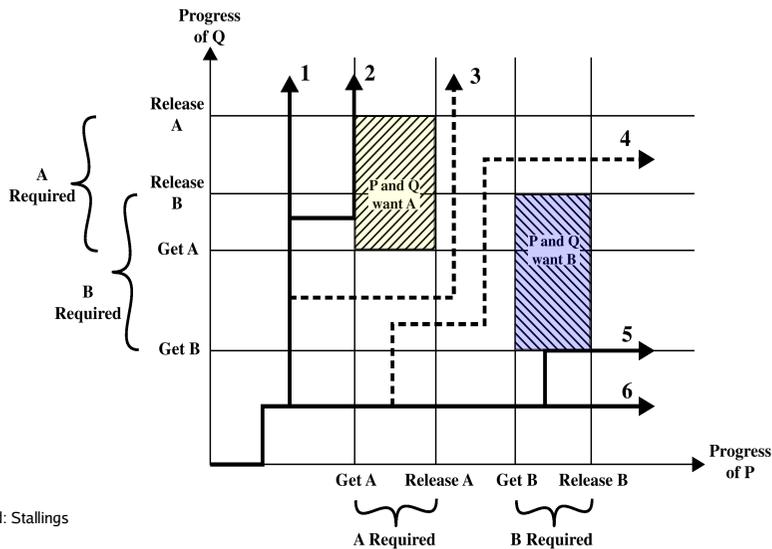
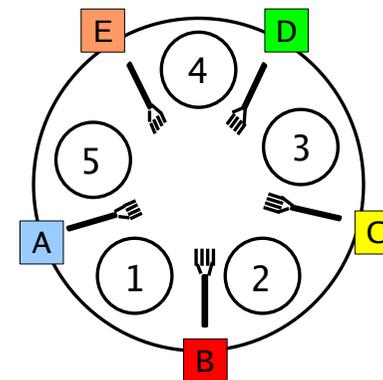


Bild: Stallings

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

Gliederung

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

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 wg. Timeout)
- Nur die *nicht* unterbrechbaren sind interessant, weil sie Deadlocks verursachen können

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

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 (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

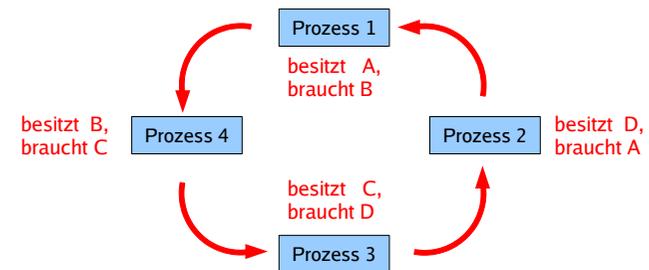
3. Ununterbrechbarkeit der Ressourcen

- 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 (2)

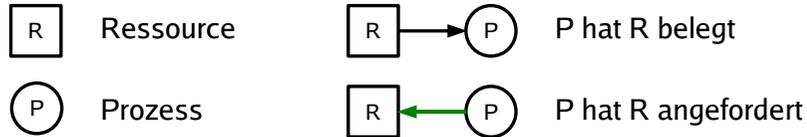
- (1) bis (3) sind **notwendige** Bedingungen für einen Deadlock
- (1) bis (3) sind aber auch „wünschenswerte“ Eigenschaften eines Betriebssystems, denn:
 - gegenseitiger Ausschluss ist nötig für korrekte Synchronisation
 - Hold & Wait ist nötig, wenn Prozesse exklusiven Zugriff auf mehrere Ressourcen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

Deadlock-Bedingungen (4)

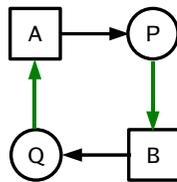
- (1) bis (4) sind **notwendige und hinreichende** Bedingungen für einen Deadlock
- Das zyklische Warten (4) (und dessen Unauflösbarkeit) sind Konsequenzen aus (1) bis (3)
- (4) ist der erfolgversprechendste Ansatzpunkt, um Deadlocks aus dem Weg zu gehen

Ressourcen-Zuordnungs-Graph (1)

- Belegung und (noch unerfüllte) Anforderung grafisch darstellen:

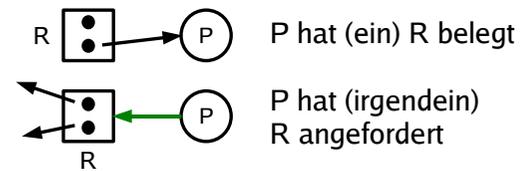


- P, Q aus Minimalbeispiel:
- Deadlock = Kreis im Graph



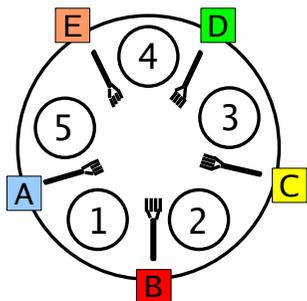
Ressourcen-Zuordnungs-Graph (3)

- Variante für Ressourcen, die mehrfach vorkommen können

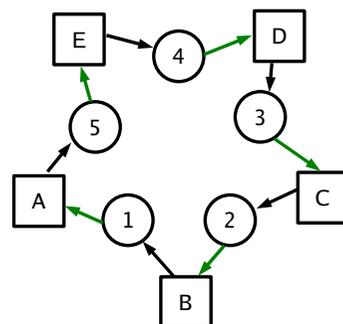


Ressourcen-Zuordnungs-Graph (2)

Philosophen-Beispiel

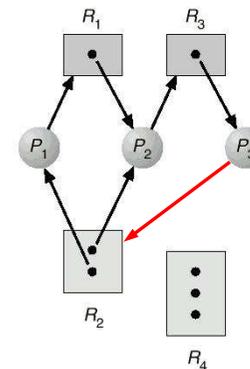


Situation, nachdem alle Philosophen ihre rechte Gabel aufgenommen haben

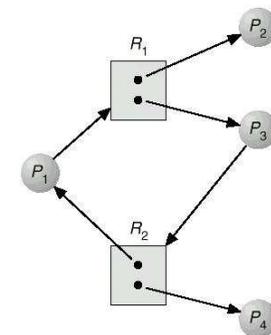


Ressourcen-Zuordnungs-Graph (4)

- Beispiele mit mehreren Instanzen



Mit roter Kante ($P_3 \rightarrow R_2$) gibt es einen Deadlock (ohne nicht)



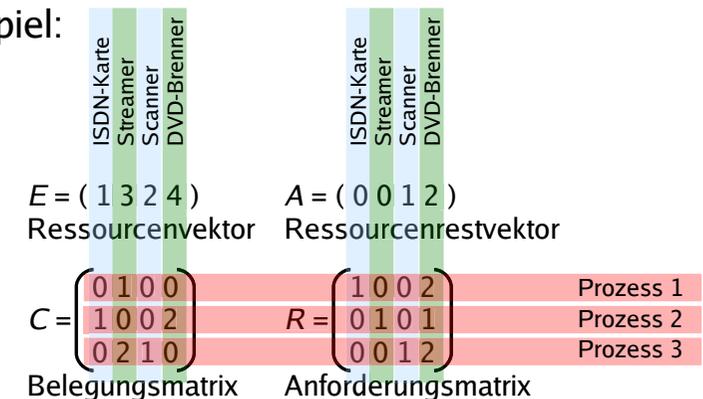
Kreis, aber kein Deadlock – Bedingung ist nur **notwendig**, nicht hinreichend!

Deadlock-Erkennung (1)

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

Deadlock-Erkennung (3)

- Beispiel:



Deadlock-Erkennung (2)

- 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

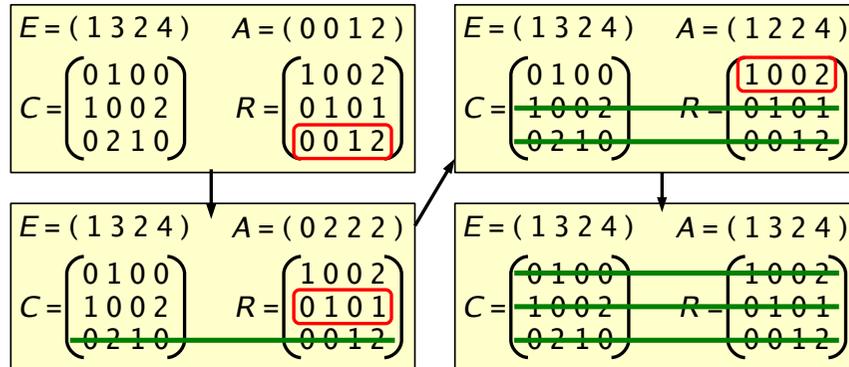
Deadlock-Erkennung (4)

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 den 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 (5)

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



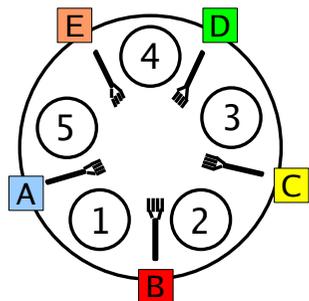
Deadlock-Erkennung (7): Behebung

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-Erkennung (6)

- Beispiel (5 Philosophen)



A B C D E					A B C D E				
$E = (1\ 1\ 1\ 1\ 1)$					$A = (0\ 0\ 0\ 0\ 0)$				
$C = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$					$R = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$				

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

Deadlock-Vermeidung (1)

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

Deadlock-Vermeidung (2)

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-Vermeidung (4)

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 €



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

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



→ Kunde 3 fordert 100 € an

↓ 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!



Deadlock-Vermeidung (3)

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

Deadlock-Vermeidung (5)

Banker-Algorithmus (3)

- 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

Deadlock-Vermeidung (6)

Banker-Algorithmus (4)

Feststellen, ob ein Zustand sicher ist = Annehmen, dass alle Prozesse sofort ihre Maximalforderungen stellen, und dies auf Deadlocks überprüfen

Vorschau

Nächste Vorlesung: 17.01.2007

**Deadlock-Verhinderung
(Deadlock Prevention)**

Probeklausur