

```
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 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
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[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 12:48:41 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 15:27:35 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 63546
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[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 /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[5491]: (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 /usr/sbin/cron[24739]: (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 /usr/sbin/cron[25555]: (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 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 /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[20998]: 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[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 /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[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
```

5. Synchronisation

5. Synchronisation

5.1 Einführung

5.2 Kritische Abschnitte

5.3 Synchr.-Methoden

5.4 Deadlocks

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
 - SMP-System: Scheduler (je einer pro CPU) greifen auf gleiche Prozesslisten / Warteschlangen zu

Einführung (2)

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

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

Ausgangssituation: w=10

P1:

```
w=read(Adresse); // 10  
w=w+1;           // 11
```

P2:

```
w=read(Adresse); // 10  
w=w+1;           // 11  
write(Adresse,w); // 11
```

```
write(Adresse,w); // 11 !!
```

Ergebnis nach P1, P2: w=11 – nicht 12!

Einführung (3)

- Gewünscht wäre eine der folgenden Reihenfolgen:

Ausgangssituation: w=10

P1:	P2:
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

P1:	P2:
	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

Einführung (4)

- **Ursache:** `erhoehe_zaehler()` 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)

Einführung (5)

- 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

- Definition der (Datenbank-) **Transaktion**, die u.a. **atomar und isoliert** erfolgen muss

Einführung (6)

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

Einführung (7)

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

Einführung (8)

Race Condition als Sicherheitslücke

- Wird von Angreifern genutzt
- Einfaches Beispiel:

```
read(command)
f=open("/tmp/script","w")
write(f,command)
f.close()
chmod("/tmp/script","a+x")
system("/tmp/script")
```

Angreifer ändert Dateiinhalt vor dem chmod;
Programm läuft mit Rechten des Opfers

Einführung (9)

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

```
erhoehe_zaebler( ) {
    flag=read(Lock);
    if (flag == LOCK_UNSET) {
        set(Lock);
        /* Anfang des „kritischen Bereichs“ */
        w=read(Adresse); w=w+1;
        write(Adresse,w);
        /* Ende des „kritischen Bereichs“ */
        release(Lock);
    };
}
```

- Problem: Lock-Variable nicht geschützt

Einführung (10)

- Nicht alle Zugriffe auf Daten 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**.

Inhaltsübersicht: Synchronisation

- 5.1 Einführung, Race Conditions
- 5.2 Kritische Abschnitte und gegenseitiger Ausschluss
- 5.3 Synchronisationsmethoden
 - Programmtechnische Synchronisation
 - Standard-Primitive: Mutexe, Semaphore, Monitore
 - Locking
 - Nachrichten
- 5.4 Deadlocks
 - Definition und Beispiele
 - Deadlocks erkennen

Kritische Bereiche (1)

- Programmteil, der auf gemeinsame Daten zugreift
 - müssen nicht verschiedene Programme sein: auch mehrere Instanzen des gleichen Programms!
- Block zwischen erstem und letztem Zugriff
- Formulierung: kritischen Bereich „betreten“ und „verlassen“ (enter / leave critical section)

Kritische Bereiche (2)

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

Gegenseitiger Ausschluss

- Tritt nie mehr als ein Thread gleichzeitig in den kritischen Bereich 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

Programmtechnische Synchr. (1)

1. Versuch: Lock-Variable (wie in Einführung)

- Lock-Variable auf *false* initialisiert
- Prozess, der kritischen Bereich betreten will, prüft `lock==false` – wenn Bedingung erfüllt ist:
 - `lock=true` setzen,
 - Bereich betreten und wieder verlassen
 - `lock=false` (zurück)setzen
- Verschiebt Problem nur auf die Lock-Variable

```
while ( lock ) {  
    /* warten */  
};  
lock = true;  
kritischer_bereich();  
lock = false;
```


Programmtechnische Synchr. (2)

2. Versuch: Nächsten Prozess speichern

- Lock-Variable *turn* legt fest, welcher Prozess als nächster den kritischen Bereich betreten darf:

```
while (true) {  
    while (turn != 1) {  
        /* warten */  
    };  
    kritischer_bereich();  
    turn=2;  
}
```

```
while (true) {  
    while (turn != 2) {  
        /* warten */  
    };  
    kritischer_bereich();  
    turn=1;  
}
```

- verhindert Race Conditions
- Aber: kritischer Bereich kann nur abwechselnd betreten werden

Programmtechnische Synchr. (3)

3. Versuch: Für jeden Thread separate Variable, die „Thread ist in krit. Bereich“ anzeigt

```
while (true) {
  C1=true;
  while (C2) {
    /* warten */
  };
  kritischer_bereich();
  C1=false;
}
```

```
while (true) {
  C2=true;
  while (C1) {
    /* warten */
  };
  kritischer_bereich();
  C2=false;
}
```

- verhindert Race Conditions
- Deadlock tritt auf, wenn beide gleichzeitig den kritischen Bereich betreten wollen

Programmtechnische Synchr. (4)

4. Versuch (Dekker): Kombination aus Lock-Variablen und wechselnder Reihenfolge

```
while (true) {
  C1=true;
  while (C2) {
    if (turn != 1) {
      C1=false;
      while (turn != 1) {
        /* warten */
      };
      C1=true;
    };
  };
  kritischer_bereich();
  turn=2;
  C1=false;
}
```

```
while (true) {
  C2=true;
  while (C1) {
    if (turn != 2) {
      C2=false;
      while (turn != 2) {
        /* warten */
      };
      C2=true;
    };
  };
  kritischer_bereich();
  turn=1;
  C2=false;
}
```

Programmtechnische Synchr. (5)

Alternative: Petersons Algorithmus

```
C1=true;
turn=2;
while (C2 && turn==2)
    /* warten */;
kritischer_abschnitt();
C1=false;
```

```
C2=true;
turn=1;
while (C1 && turn==1)
    /* warten */;
kritischer_abschnitt();
C2=false;
```

Programmtechnische Synchr. (6)

Petersons Algorithmus – gegenseitiger Ausschluss gewährt:

- Wenn P_1 C_1 auf *true* setzt, kann P_2 seinen kritischen Bereich nicht mehr betreten
- War P_2 schon im kritischen Bereich, dann war C_2 schon *true*, d. h., P_1 durfte nicht in seinen kritischen Bereich

Programmtechnische Synchr. (7)

Petersons Algorithmus – **keine gegenseitige Blockade:**

Angenommen, P1 ist in der While-Schleife blockiert, d. h.:
 $C2=true$ und $turn=2$ (P1 kann den krit. Bereich betreten, wenn eine der Bedingungen nicht mehr gilt, also entweder $C2=false$ oder $turn=1$ wird)

Dann nur 2 Möglichkeiten:

- P2 wartet auf Einlass in den krit. Bereich → das kann nicht sein, denn mit $turn=2$ darf P2 in seinen kritischen Bereich
- P2 nutzt wiederholt den krit. Bereich, monopolisiert Zugang zu ihm → das kann auch nicht sein, weil P2 vor dem Betreten die $turn$ -Variable auf 1 setzt (und damit P1 den Vortritt lassen würde)

Test-and-Set-Lock (TSL) (1)

- **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
```

Test-and-Set-Lock (TSL) (2)

- **TSL** muss im Falle mehrerer CPUs den Speicherbus sperren, damit kein Prozess auf einer anderen CPU auf die gleiche Variable zugreifen kann

Aktives und passives Warten (1)

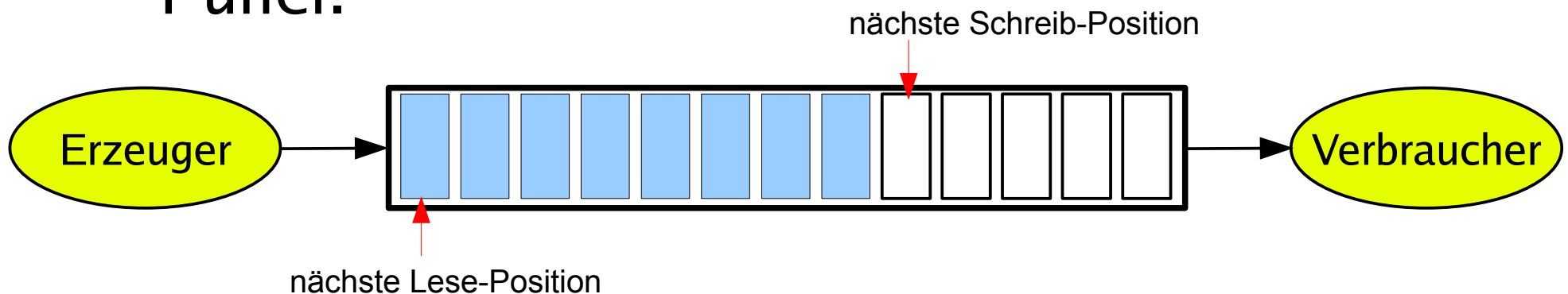
- **Aktives Warten (busy waiting):**
 - Ausführen einer Schleife, bis eine Variable einen bestimmten Wert annimmt.
 - Der **Thread ist bereit** und **belegt die CPU**.
 - Die Variable muss von einem anderen Thread gesetzt werden.
 - (Großes) Problem, wenn der andere Thread endet.
 - (Großes) Problem,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

Aktives und passives Warten (2)

- **Passives Warten (sleep and wake):**
 - Ein **Thread blockiert** und wartet auf ein Ereignis, das ihn wieder in den Zustand „bereit“ versetzt.
 - Der blockierte 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.

Erzeuger-Verbraucher-Problem (1)

- 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.



Erzeuger-Verbraucher-Problem (2)

- **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.

Erzeuger-Verbraucher-Problem (3)

- 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.

Erzeuger-Verbraucher-Problem mit sleep / wake

```
#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
    }
}
```


Deadlock-Problem bei sleep / wake (1)

- 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.

Deadlock-Problem bei sleep / wake (2)

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

VERBRAUCHER	ERZEUGER
n=read(count);	..
<hr/>	
..	produce_item();
..	n=read(count);
..	/* n=0 */
..	n=n+1;
..	write(n,count);
..	wake (VERBRAUCHER);
<hr/>	
/* n=0 */	..
sleep ();	..



Deadlock-Problem bei sleep / wake (3)

- 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)

Semaphore (1)

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):
 - Semaphor-Wert um 1 erniedrigen, falls er >0 ist,
 - Thread blockieren und in eine Warteschlange einreihen, wenn der Semaphor-Wert 0 ist.

Semaphore (2)

- Bei **Freigabe** eines Semaphors (V- oder **Signal-Operation**):
 - 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);
```

Semaphore (3)

- Variante: Negative Semaphor-Werte
 - Semaphor zählt Anzahl der wartenden Threads
 - **Anfordern** (WAIT):
 - Semaphor-Wert um 1 erniedrigen (~~falls er positiv ist~~)
 - Thread blockieren und in eine Warteschlange einreihen, wenn der Semaphor-Wert ≤ 0 ist.
 - **Freigabe** (SIGNAL):
 - Thread aus der Warteschlange wecken (falls nicht leer)
 - Semaphor-Wert um 1 erhöhen (~~wenn es keinen auf den Semaphor wartenden Thread gibt~~)

Semaphore (4)

Standard-Variante:
Semaphor kann nur
Werte ≥ 0
annehmen

```
wait (sem) {  
    if (sem>0)  
        sem--;  
    else BLOCK_CALLER;  
}
```

```
signal (sem) {  
    if (P in QUEUE(sem)) {  
        wakeup (P);  
        remove (P, QUEUE);  
    }  
    else sem++;  
}
```

Variante: Semaphor
auch negativ,
speichert Größe der
Warteschlange

```
wait (sem) {  
    if (sem<1)  
        BLOCK_CALLER;  
    sem--;  
}
```

```
signal (sem) {  
    if (P in QUEUE(sem)) {  
        wakeup (P);  
        remove (P, QUEUE);  
    }  
    sem++;  
}
```

Mutexe (1)

- **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

Mutexe (2)

- **Mutex (mutual exclusion) = binärer Semaphor**, also ein Semaphor, der nur die Werte 0 / 1 annehmen kann

```
wait (mutex) {
    if (mutex==1)
        mutex=0;
    else BLOCK_CALLER;
}

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 Bereiche)

Blockieren?

- 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

Atomare Operationen

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

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

Warteschlangen

- 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** Semaphor / Mutex
 - zufällig: **schwacher** Semaphor / Mutex

Erzeuger-Verbraucher-Problem mit Semaphoren und Mutexen

```
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 Bereich
        enter_item (item);    // In den Puffer einstellen
        signal (mutex);       // Kritischen Bereich 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 Bereich
        remove_item(item);    // Aus dem Puffer entnehmen
        signal (mutex);       // Kritischen Bereich verlassen
        signal (empty);       // Freie Plätze erhöhen, evtl. producer wecken
        consume_entry (item); // Verbrauchen
    }
}
```

Monitore (1)

Motivation

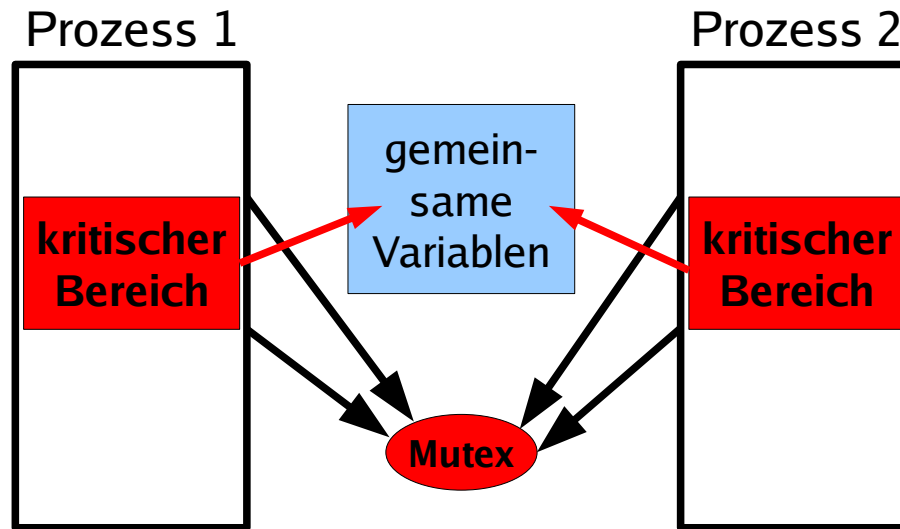
- Arbeit mit Semaphoren und Mutexen zwingt den Programmierer, vor und nach jedem kritischen Bereich `wait()` und `signal()` aufzurufen
- Wird dies ein einziges Mal vergessen, funktioniert die Synchronisation nicht mehr
- **Monitor** kapselt die kritischen Bereiche
- Monitor muss von Programmiersprache unterstützt werden (z.B. Java, Concurrent Pascal)

Monitore (2)

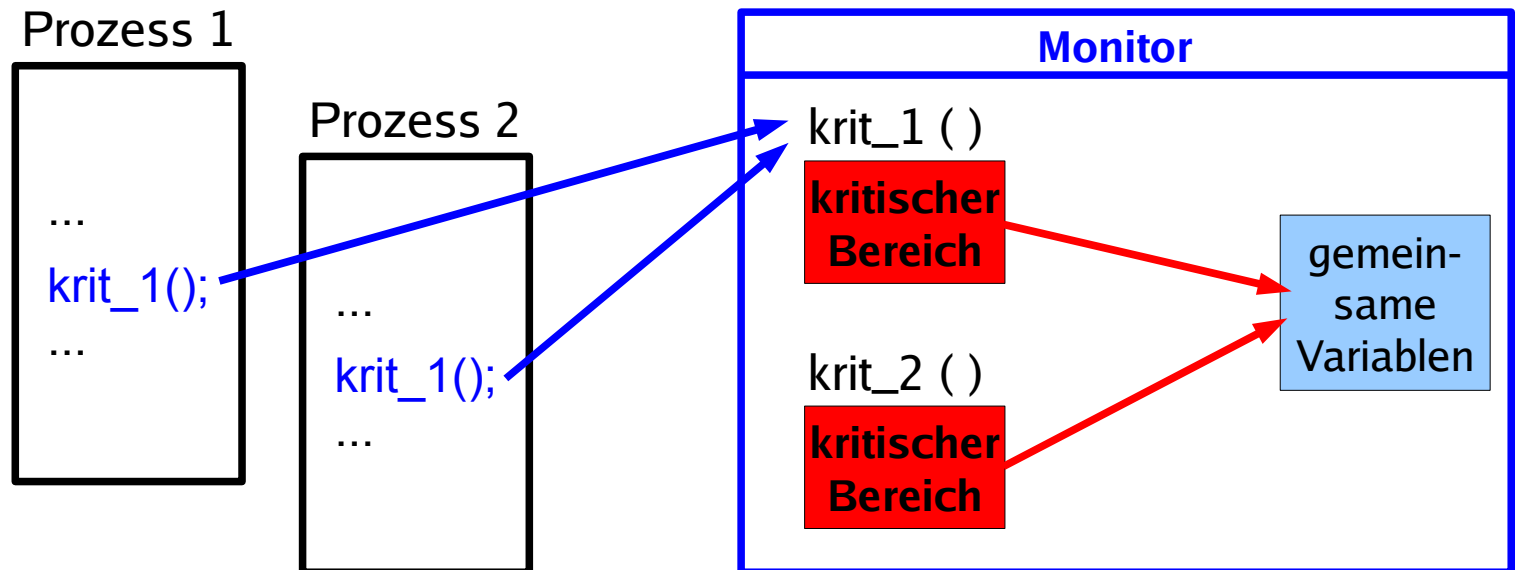
- **Monitor:** Sammlung von Prozeduren, Variablen und speziellen **Bedingungsvariablen:**
 - Prozesse können die Prozeduren des Monitors aufrufen, können aber nicht von außerhalb des Monitors auf dessen Datenstrukturen zugreifen.
 - Zu jedem Zeitpunkt kann **nur ein einziger Prozess aktiv im Monitor** sein (d. h.: eine Monitor-Prozedur ausführen).
- Monitor wird durch Verlassen der Monitorprozedur freigegeben

Monitore (3)

Mutex



Monitor



Monitore (4)

Einfaches Beispiel: Zugriff auf eine Festplatte; mit Mutex

```
mutex disk_access = 1;
```

```
wait (disk_access);  
// Daten von der Platte lesen  
signal (disk_access);  
  
wait (disk_access);  
// Daten auf die Platte schreiben  
signal (disk_access);
```

Gleiches Beispiel, mit Monitor

```
monitor disk {  
  entry read (diskaddr, memaddr) {  
    // Daten von der Platte lesen  
  };  
  entry write (diskaddr, memaddr) {  
    // Daten auf die Platte schreiben  
  };  
  init () {  
    // Gerät initialisieren  
  };  
};
```

```
disk.read (da, ma);  
  
disk.write (da, ma);
```

Monitor (5)

- Monitor ist ein Konstrukt, das Teil einer Programmiersprache ist
- Compiler – und nicht der Programmierer – ist für gegenseitigen Ausschluss zuständig
- Umsetzung (durch den Compiler) z. B. mit Semaphor/Mutex:

```
- monitor disk           → semaphore m_disk = 1;  
  
- entry funktion () {   → void funktion () {  
    /* Code */           wait (m_disk);  
    }                   /* Code */  
                        signal (m_disk);  
                        }  
  
- disk.funktion();      → funktion();
```


Monitor (6)

- Monitor-Konzept erinnert an
 - Klassen (objektorientierte Programmierung)
 - Module (modulare Programmierung)
- Kapselung der Prozeduren und Variablen (außer über als public deklarierte Prozeduren kein Zugriff auf Monitor)
- Einfaches und übersichtliches Verfahren, um kritische Bereiche zu schützen, aber:
- Was tun, wenn ein Prozess im Monitor blockieren muss?

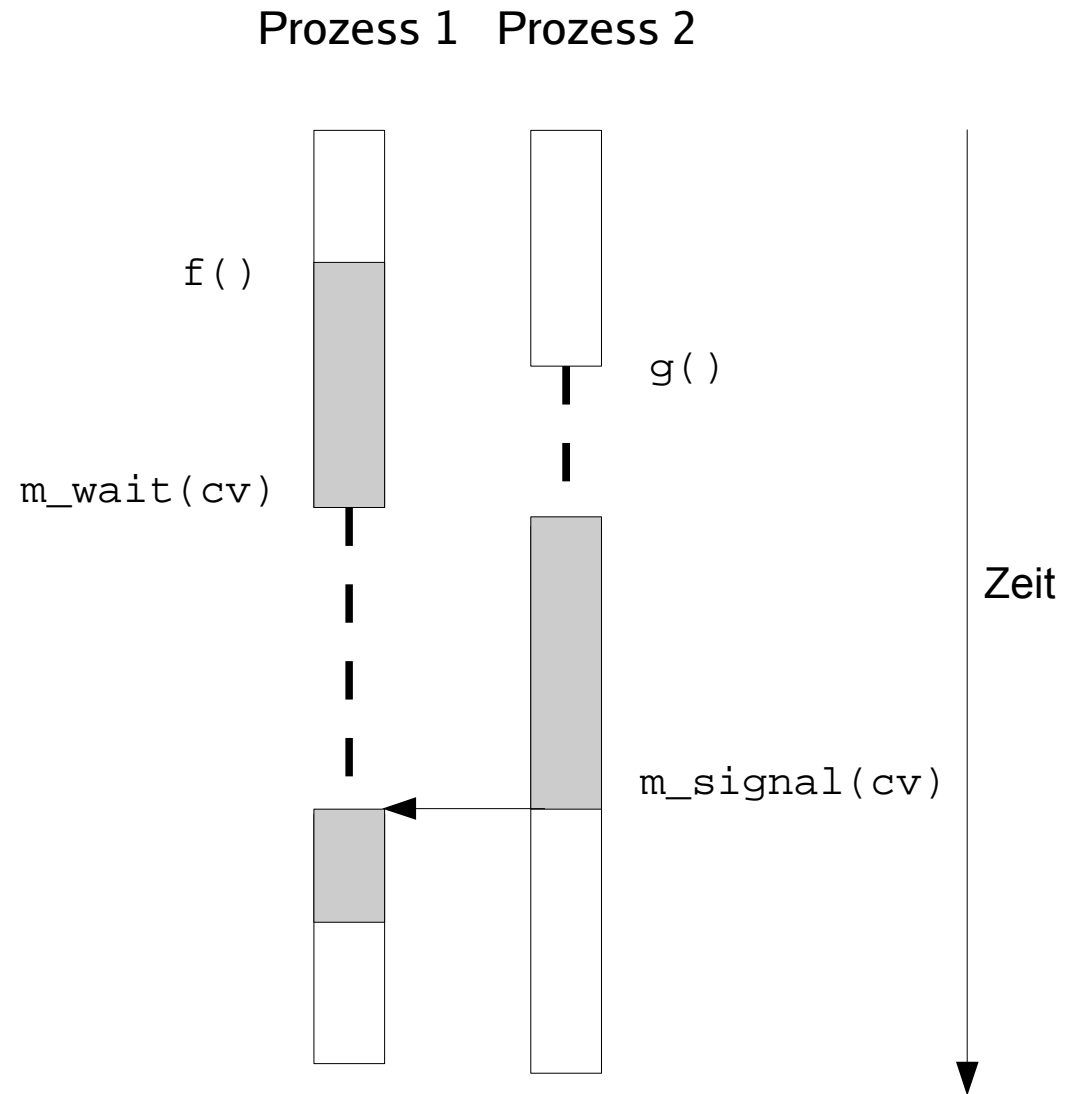
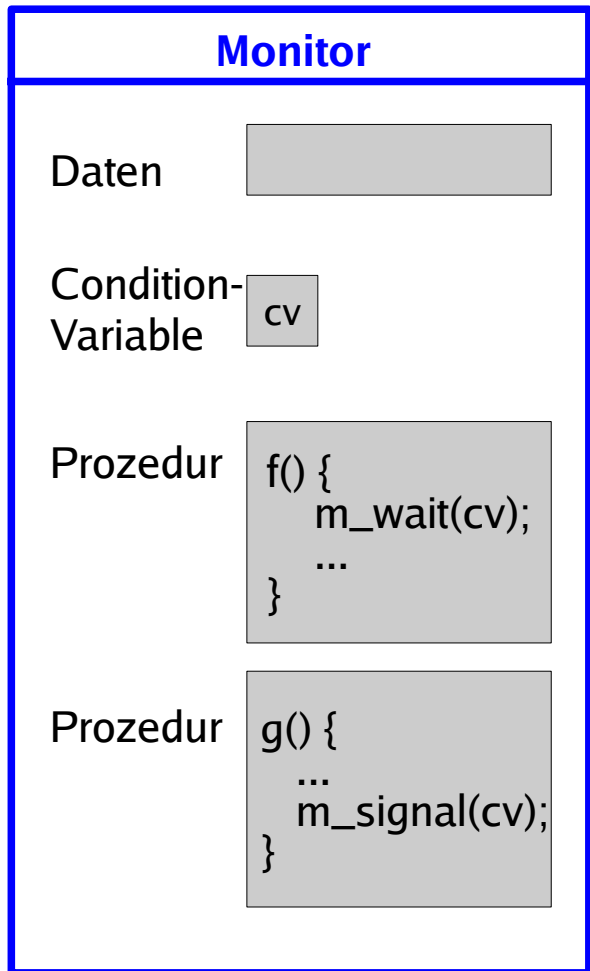
Monitor (7)

Zustandsvariablen (condition variables)

Idee: Prozess in Monitor muss darauf warten, dass eine bestimmte Bedingung (condition) erfüllt ist. Für jede „Zustandsvariable“ Wait- und Signal-Funktionen:

- **m_wait** (var): aufrufenden Prozess sperren (er gibt den Monitor frei)
- **m_signal** (var): gesperrten Prozess entsperren (weckt einen Prozess, der den Monitor mit m_wait() verlassen hat); erfolgt unmittelbar vor Verlassen des Monitors

Monitore (8)



Monitore (9)

- Gesperrte Prozesse landen in einer Warteschlange, die der Zustandsvariable zugeordnet ist
- Interne Warteschlangen haben Vorrang vor Prozessen, die von außen kommen
- Implementation mit Mutex/Semaphor:

```
conditionVariable {
    int queueSize = 0;
    mutex m;
    semaphore waiting;

    wait() {
        m.lock();
        queueSize++;
        m.release();
        waiting.down();
    }
}
```

```
signal() {
    m.lock();
    while (queueSize > 0) {
        // alle wecken
        queueSize--;
        waiting.up();
    }
    m.release();
}
```

Monitore (10)

Erzeuger-
Verbraucher-
Problem
mit Monitor

```
monitor iostream {
    item buffer;
    int count;
    const int bufsize = 64;
    condition nonempty, nonfull;

    entry append(item x) {
        while (count == bufsize) m_wait(nonfull);
        put(buffer, x); // put ist lokale Prozedur
        count = 1;
        m_signal(nonempty);
    }

    entry remove(item x) {
        while (count == 0) m_wait(nonempty);
        get(buffer, x); // get ist lokale Prozedur
        count = 0;
        m_signal(nonfull);
    }

    init() {
        count = 0; // Initialisierung
    }
}
```

Quelle: Prof. Scheidig, Univ. Saarbrücken,
http://hssun5.cs.uni-sb.de/lehrstuhl/WS0607/Vorlesung_Betriebssysteme/
- angepasst an C-artige Syntax

Java und Monitore (1)

- Java verwendet Monitore zur Synchronisation
- Schlüsselwort „synchronized“
- Klasse, in der alle Methoden synchronized sind, ist ein Monitor
- Keine benannten Zustandsvariablen
- Warteschlangen:
 - m_wait: wait
 - m_signal: notify (weckt einen Prozess)
notifyAll (weckt alle Prozesse)

Java und Monitore (2)

```
class BoundedBuffer extends MyObject {
    private int size = 0;
    private double[] buf = null;
    private int front = 0, rear = 0,
        count = 0;

    public BoundedBuffer(int size) {
        this.size = size;
        buf = new double[size];
    }

    public synchronized void
    deposit(double data) {
        while (count == size) wait();
        buf[rear] = data;
        rear = (rear+1) % size;
        count++;
        if (count == 1) notify();
    }

    public synchronized double fetch() {
        double result;
        while (count == 0) wait();
        result = buf[front];
        front = (front+1) % size;
        count--;
        if (count == size-1) notify();
        return result;
    }
}
```

Quelle: <http://www.mcs.drexel.edu/~shartley/ConcProgJava/Monitors/bbse.java>

Locking (1)

Locking erweitert die Funktionalität von Mutexen, indem es verschiedene **Lock-Modi** (Zugriffsarten) unterscheidet, und deren „Verträglichkeit“ miteinander festlegt:

- **Concurrent Read:** Lesezugriff, andere Schreiber erlaubt.
- **Concurrent Write:** Schreibzugriff, andere Schreiber erlaubt.
- **Protected Read:** Lesezugriff, andere Leser erlaubt, aber keine Schreiber (share lock)
- **Protected Write:** Schreibzugriff, andere Leser erlaubt, aber kein weiterer Schreiber (update lock)
- **Exclusive:** Schreibzugriff, keine anderen Zugriffe erlaubt

Locking (2)

	concurrent read	concurrent write	protected read	protected write	exclusive
concurrent read	X	X	X	X	-
concurrent write	X	X	-	-	-
protected read	X	-	X	-	-
protected write	X	-	-	-	-
exclusive	-	-	-	-	-

Locking (3)

- Thread fordert Lock in bestimmtem Modus an.
 - Ist der Lock-Modus mit den vorhandenen Locks anderer Threads verträglich, wird das Lock gewährt.
 - Ist der Lock-Modus zu einem Lock eines anderen Threads unverträglich, **blockiert** der Thread, **bis** das Lock **gewährt** werden kann.
- Locking-Mechanismen werden implementiert
 - vom Betriebssystem
 - von Anwendungsprogrammen (speziell Datenbanken)

Nachrichten (1)

- Nachrichtenaustausch über zwei Systemaufrufe
 - `send (destination, &message);`
 - `receive (source, &message);`
- Synchroner Kommunikation:
Threads blockieren, wenn *send* bzw. *receive* nicht sofort ausgeführt werden können, z. B. weil
 - die Gegenseite keinen entsprechenden Befehl abgesetzt hat,
 - ein Zwischenpuffer für die Nachrichten voll bzw. leer ist.

Nachrichten (2)

- **Vorteil:** funktioniert auch bei Systemen ohne gemeinsamen Hauptspeicher (distributed systems, client-server-computing)
- **Nachteile:**
 - aufwändiger durch Duplizieren der Daten
 - Verwaltung der Namen für Quelle und Ziel nötig
 - Vorkehrungen gegen Verlust der Meldung nötig
- Implementierung z. B. durch Pipes oder Mailslots (Windows) oder RPCs.
- Auch Broadcast an mehrere Prozesse möglich

Nachrichten (3)

- **synchron vs. asynchron**
 - synchron: *send / receive* blockieren, bis zugehörige Operation auf Gegenseite abgeschlossen ist
 - asynchron: *send*-Call kehrt sofort zurück; Erfolg des Versands ist evtl. überprüfbar, z. B.:
 - Gegenseite schickt explizit Antwort
 - Messaging-System sendet Signal bei Zustellung
- **verbindungsorientiert vs. verbindungslos**
 - verbindungsorientiert: „stehende Verbindung“ (TCP)
 - verbindungslos (vgl. UDP)

Beispiel für Nachrichten

Zwei Prozesse wechseln sich im Zugriff ab

```
void funktion (int id) {
    int otherid = 1 - id;
    char message[10] = "";
    // ein Prozess darf zuerst; ID 0
    if (id==0) {
        message = "go";
    }
    // unkritische Befehle
    while message != "go" {
        receive (otherid, &message);
    }
    // kritischer Bereich
    send (otherid, "go");
    message = "";
    // mehr unkritische Befehle
}
```

p0 ruft *funktion(0)* auf,
p1 ruft *funktion(1)* auf.

p0 darf zuerst in
den kritischen Bereich

```
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 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
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[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 12:48:41 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 15:27:35 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 63546
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[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 /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[549]: (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 /usr/sbin/cron[24291]: (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 /usr/sbin/cron[25555]: (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 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 /usr/sbin/cron[1234]: (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[20998]: 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[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 /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[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
```

Linux: Synchronisation in Kernel

Synchronisation im Linux-Kernel

- Atomare Operationen
 - auf Integer-Variablen
(`atomic_set`, `atomic_add`, `atomic_inc`, ...)
 - Bit-Operationen auf Bitvektoren
(`set_bit`, `clear_bit`, `test_and_set`, ...)
- Spin Locks / Reader-Writer Spin Locks
- Semaphore / Reader-Writer-Semaphore

Atomare Integer-Operationen (1)

- Neuer Typ *atomic_t* (24 Bit Integer)
- Initialisierung: *atomic_t var = ATOMIC_INIT(0);*
- Wert setzen: *atomic_set (&var, wert);*
- Addieren: *atomic_add (wert, &var);*
- ++: *atomic_inc (&var);*
- Subtrahieren: *atomic_sub (wert, &var);*
- --: *atomic_dec (&var);*
- Auslesen: *int i = atomic_read (&var);*

Atomare Integer-Operationen (2)

- ***res = atomic_sub_and_test (i, &var);***
zieht atomar *i* von *var* ab.
 - Rückgabewert *true*, falls Ergebnis 0 ist;
 - Rückgabewert *false*, falls Ergebnis nicht 0
- ***res = atomic_dec_and_test (&var);***
res = atomic_inc_and_test (&var);
führt atomar *var--*; bzw. *var++*; aus.
 - Rückgabewert *true*, falls Ergebnis 0 ist;
 - Rückgabewert *false*, falls Ergebnis nicht 0

Atomare Integer-Operationen (3)

- *res = atomic_add_negative (i, &var);*
addiert atomar *i* zu *var*.
 - Rückgabewert true, falls Ergebnis negativ ist;
 - Rückgabewert false, falls Ergebnis ≥ 0 ist

Atomare Bit-Operationen (1)

- Einzelne Bits in Bitvektoren setzen
- Datentyp: beliebig, z. B. *unsigned long bitvektor = 0;*
 - nur über Pointer ansprechen
 - Anzahl der setz-/testbaren Bits hängt von Größe des verwendeten Datentyps ab
- ***set_bit*** (*i*, *&bitvektor*); *i*-tes Bit setzen
- ***clear_bit*** (*i*, *&bitvektor*); *i*-tes Bit löschen
- ***change_bit*** (*i*, *&bitvektor*); *i*-tes Bit kippen

Atomare Bit-Operationen (2)

- Test-and-Set-Operationen geben zusätzlich den vorherigen Wert des jeweiligen Bits zurück
 - $b = \text{test_and_set_bit}(i, \&\text{bitvektor});$
 - $b = \text{test_and_clear_bit}(i, \&\text{bitvektor});$
 - $b = \text{test_and_change_bit}(i, \&\text{bitvektor});$
- Einzelne Bits auslesen
 - $b = \text{test_bit}(i, \&\text{bitvektor});$
- Suchfunktionen
 - $pos = \text{find_first_bit}(\&\text{bitvektor}, \text{laenge});$
 - $pos = \text{find_first_zero_bit}(\&\text{bitvektor}, \text{laenge});$

Spin Locks (1)

- Lock mit Mutex-Funktion:
Gegenseitiger Ausschluss
- Code, der ein Spin Lock anfordert und nicht erhält, läuft in Schleife weiter, bis das Lock verfügbar wird („spinning“)
- Typ: *spinlock_t*

```
spinlock_t xy_lock = SPIN_LOCK_UNLOCKED
```

```
spin_lock (&xy_lock);  
/* kritischer Abschnitt */  
spin_unlock (&xy_lock);
```

Spin Locks (2)

- Da Spin Locks nicht schlafen, kann man sie in Interrupt-Handlern verwenden
- In dem Fall: zusätzlich Interrupts sperren:

```
spinlock_t xy_lock = SPIN_LOCK_UNLOCKED  
unsigned long flags;
```

```
spin_lock_irqsave (&xy_lock, flags);  
/* kritischer Abschnitt */  
spin_unlock_irqrestore (&xy_lock, flags);
```

(aktuelle Interrupts in *flags* sichern, dann sperren bzw. ursprünglichen Zustand wiederherstellen)

Spin Locks (3)

- Wenn zu Beginn alle Interrupts aktiviert sind, geht es auch einfacher:

```
spinlock_t xy_lock = SPIN_LOCK_UNLOCKED
```

```
spin_lock_irq (&xy_lock);  
/* kritischer Abschnitt */  
spin_unlock_irq (&xy_lock);
```

schaltet alle Interrupts aus bzw. wieder an

- Spin Locks sind nicht „rekursiv“, d. h.: es ist nicht möglich, das gleiche Spin Lock zweimal nacheinander anzufordern, etwa beim rekursiven Aufruf einer Funktion

Spin Locks (4)

- Um Blockieren zu vermeiden, ist Lock-Abfrage mit *spin_is_locked (&xy_lock)*; möglich
- Locking-Versuch mit *spin_try_lock*:

```
if ( spin_try_lock (&xy_lock) ) {  
    /* kritischer Abschnitt */  
    spin_unlock (&xy_lock);  
} else {  
    /* durfte nicht in den kritischen Abschnitt */  
}
```

- Beide Funktionen sollte man nicht verwenden:
Entweder braucht man das Lock (und muss dann ggf. warten), oder man braucht es nicht...

Reader Writer Locks (1)

- Alternative zu normalen Locks, die mehrere Lesezugriffe zulässt – bei schreibendem Zugriff aber exklusiv (wie ein normales Lock) ist

```
rwlock_t xy_rwlock = RW_LOCK_UNLOCKED;
```

Lesender Code

```
read_lock (&xy_rwlock) {  
    /* kritischer Abschnitt,  
       read-only */  
read_unlock (&xy_rwlock);
```

Schreibender Code

```
write_lock (&xy_rwlock) {  
    /* kritischer Abschnitt,  
       read & write */  
write_unlock (&xy_rwlock);
```

- Nur bei klarer Trennung zwischen lesenden / schreibenden Programmteilen!

Reader Writer Locks (2)

- | | Es gibt schon einen Leser | Es gibt schon einen Schreiber | Noch keine Sperre |
|-----------------------------------|---------------------------|-------------------------------|-------------------|
| <code>read_lock(&lck)</code> | erfolgreich | schlägt fehl | erfolgreich |
| <code>write_lock(&lck)</code> | schlägt fehl | schlägt fehl | erfolgreich |

- Auch hier Varianten für Interrupt-Behandlung:
 - `read_lock_irq` `read_unlock_irq`
 - `read_lock_irqsave` `read_unlock_irqrestore`
 - `write_lock_irq` `write_unlock_irq`
 - `write_lock_irqsave` `write_unlock_irqrestore`

Semaphore (1)

- Kernel-Semaphore sind „schlafende“ Locks
- Ist ein Semaphor schon gelockt, werden weitere Interessenten in eine Warteschlange eingereiht.
- Bei Freigabe eines Semaphors wird der erste wartende Thread in der Warteschlange geweckt
- Semaphore eignen sich für Sperren, die über einen längeren Zeitraum gehalten werden
 - keine Verschwendung von Rechenzeit

Semaphore (2)

- Semaphore sind nur im Prozess-Kontext einsetzbar, nicht in Interrupt-Handlern (Interrupt-Handler werden nicht vom Scheduler behandelt)
- Code, der einen Semaphor verwenden will, darf nicht bereits ein normales Lock besitzen (Semaphor-Zugriff kann dazu führen, dass der Thread sich schlafen legt.)
- Semaphore können auch mehr als einen Thread auf die Ressource zugreifen lassen

Semaphore (3)

Typ: *semaphore*

Statische Deklaration

```
static DECLARE_SEMAPHORE_GENERIC (name, count);  
static DECLARE_MUTEX (name);           /* count=1 */
```

Dynamische Semaphor-Erzeugung

```
sema_init (&sem, count);  
init_MUTEX (&sem);                     /* count=1 */
```

- **Verwendung mit *up()* und *down()***

```
down (&sem);  
/* kritischer Abschnitt */  
up (&sem);
```

Semaphore (4)

- Varianten von *down()*
 - *down(&sem);*
nicht unterbrechbarer Schlaf, falls Semaphor nicht verfügbar
 - *down_interruptible(&sem);*
unterbrechbarer Schlaf, falls Sem. nicht verfügbar
 - *down_trylock(&sem);*
versucht, den Semaphor zu erhalten – falls das nicht gelingt, kehrt die Funktion sofort mit False-Wert zurück

Semaphore (5)

Beispiel für `down_trylock()`

```
/* Auszug aus /usr/src/linux/kernel/printk.c */

if (!down_trylock(&console_sem)) {
    console_locked = 1;
    /*
     * We own the drivers. We can drop the spinlock and let
     * release_console_sem() print the text
     */
    spin_unlock_irqrestore(&logbuf_lock, flags);
    console_may_schedule = 0;
    release_console_sem();
    /* Funktion release_console_sem() führt up(&console_sem); aus */
} else {
    /*
     * Someone else owns the drivers. We drop the spinlock, which
     * allows the semaphore holder to proceed and to call the
     * console drivers with the output which we just produced.
     */
    spin_unlock_irqrestore(&logbuf_lock, flags);
}
```


Reader-Writer-Semaphore (1)

- Analog zu Reader Writer Locks:
Typ *rw_semaphore*, der spezielle Up- und Down-Operationen für Lese- und Schreibzugriff erlaubt
- Alle Reader-Writer-Semaphore sind Mutexe
(Zähler ist bei Initialisierung immer 1)

Statische Deklaration

```
static DECLARE_RWSEM (name);
```

Dynamische Semaphor-Erzeugung

```
init_rwsem (&sem);
```

Reader-Writer-Semaphore (2)

```
static DECLARE_RWSEM (xy_rwsem);
```

Lesender Code

```
down_read (&xy_rwsem) {  
    /* kritischer Abschnitt,  
       read-only */  
    up_read (&xy_rwsem);
```

Schreibender Code

```
down_write (&xy_rwsem) {  
    /* kritischer Abschnitt,  
       lesen und schreiben */  
    up_write (&xy_rwsem);
```

Genau wie bei Reader Writer Locks:

	Es gibt schon einen Leser	Es gibt schon einen Schreiber	Noch keine Sperre
<code>down_read(&sem)</code>	erfolgreich	schlägt fehl	erfolgreich
<code>down_write(&sem)</code>	schlägt fehl	schlägt fehl	erfolgreich

```
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 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
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[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 12:48:41 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 15:27:35 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 63546
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[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 /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[5499]: (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 /usr/sbin/cron[24739]: (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 /usr/sbin/cron[25555]: (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 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 /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[20998]: 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[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 /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[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
```

Deadlocks

Deadlocks – Gliederung

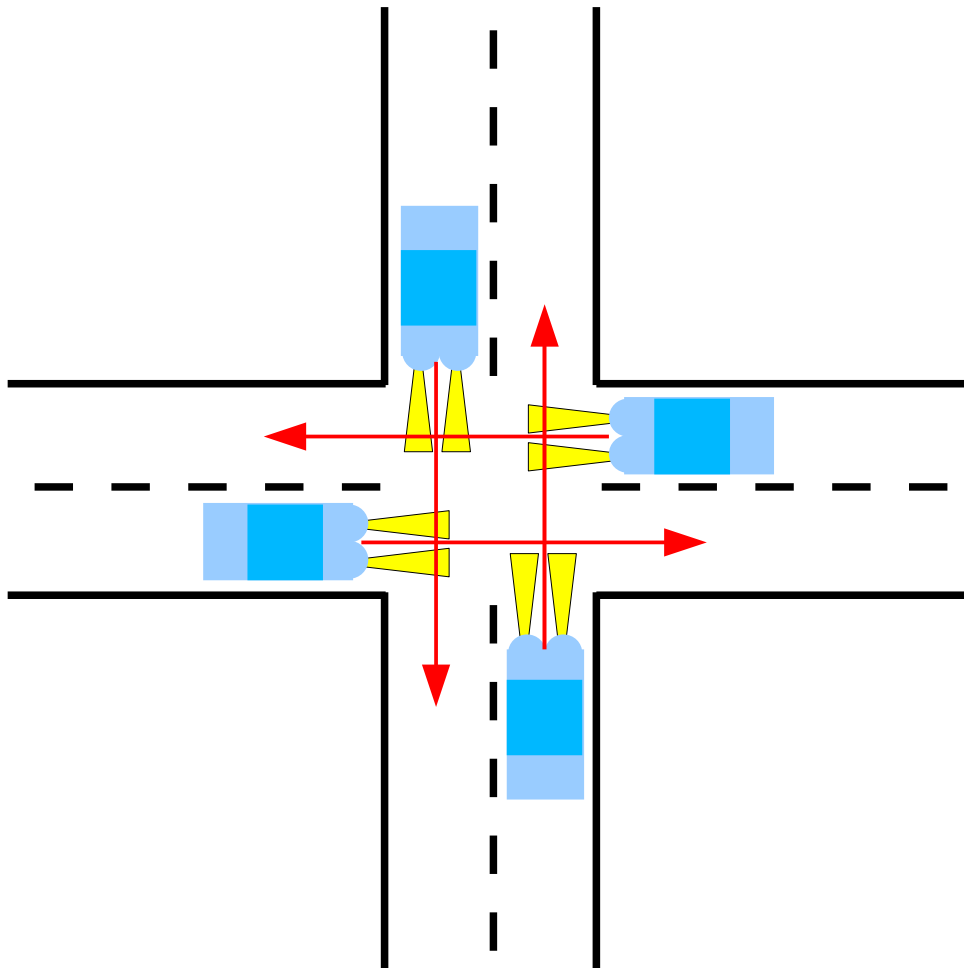
- Einführung
- Ressourcen-Typen
- Hinreichende und notwendige Deadlock-Bedingungen
- Deadlock-Erkennung und -Behebung
- Deadlock-Vermeidung (avoidance):
Banker-Algorithmus
- Deadlock-Verhinderung (prevention)

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

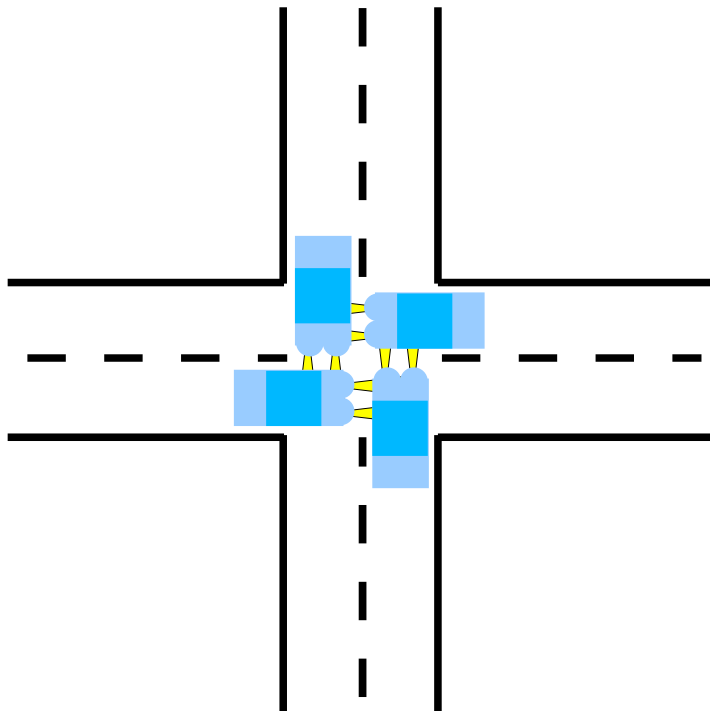
Deadlock: Rechts vor Links (1)

- Der Klassiker: Rechts-vor-Links-Kreuzung

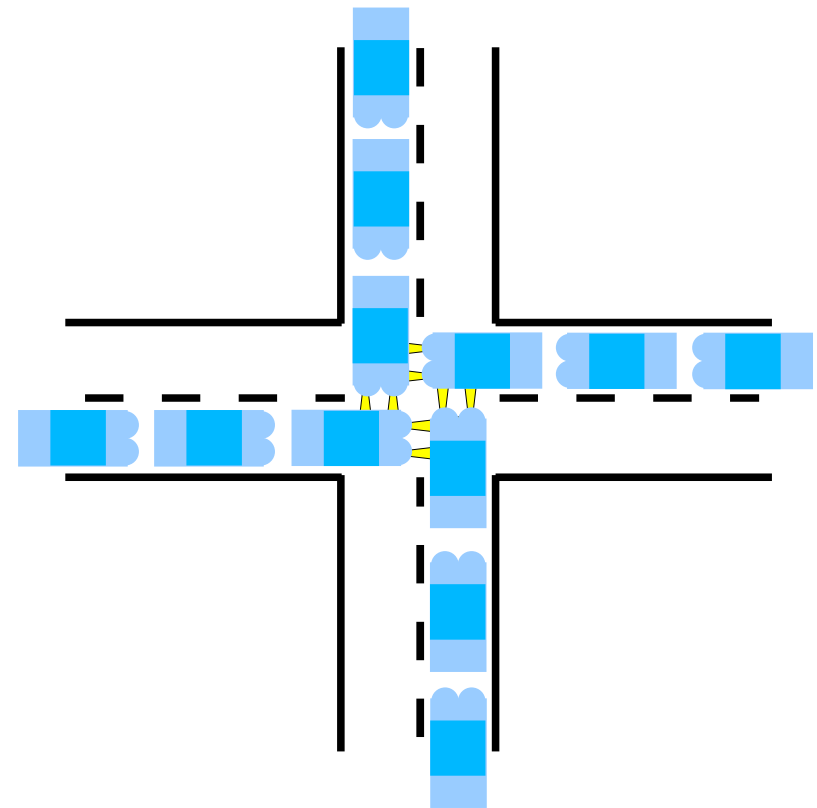


Wer darf fahren?
Potenzieller Deadlock

Deadlock: Rechts vor Links (2)



Deadlock, aber behebbbar:
eines oder mehrere Autos
können zurücksetzen

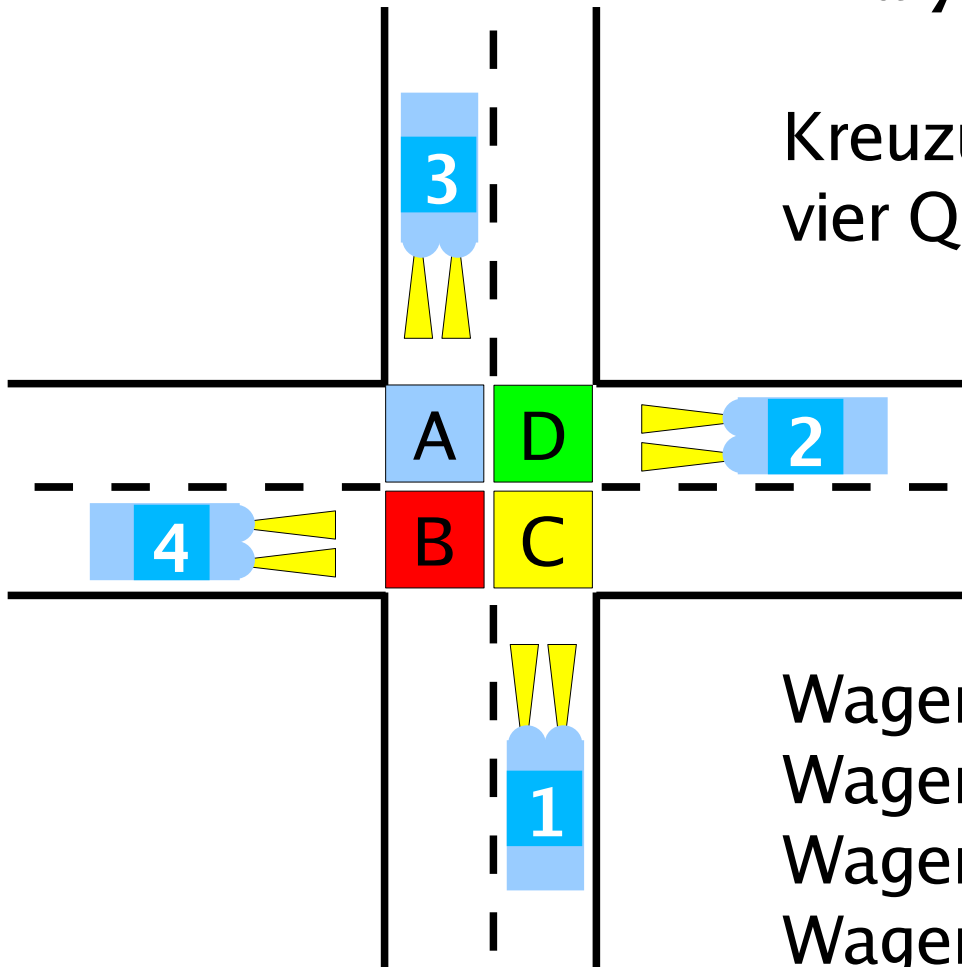


Deadlock, nicht behebbbar:
beteiligte Autos können nicht
zurücksetzen

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

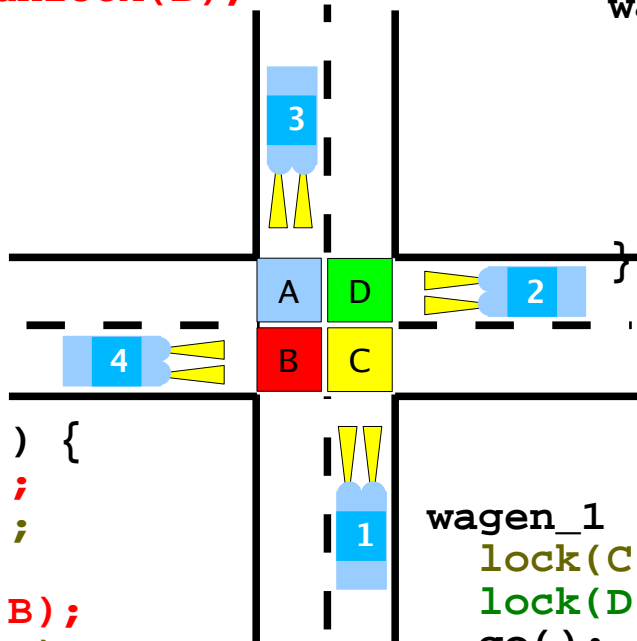
Deadlock: Rechts vor Links (4)

```
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

Prozess P

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

```
/* krit. Bereich */
```

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

Prozess Q

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

```
/* krit. Bereich */
```

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

Problematische
Reihenfolge:

P: lock(A)

Q: lock(B)

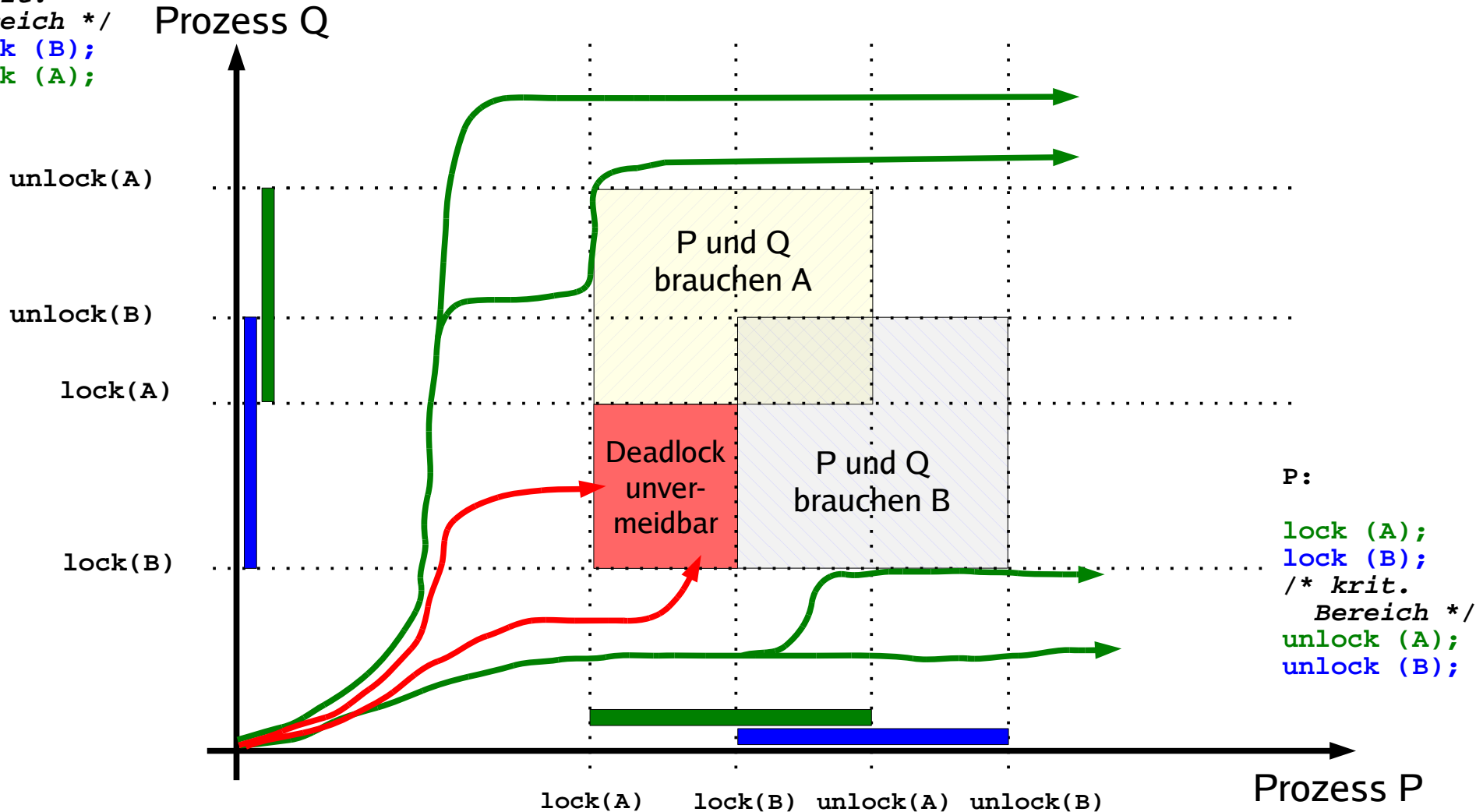
P: lock(B) ← blockiert

Q: lock(A) ← blockiert

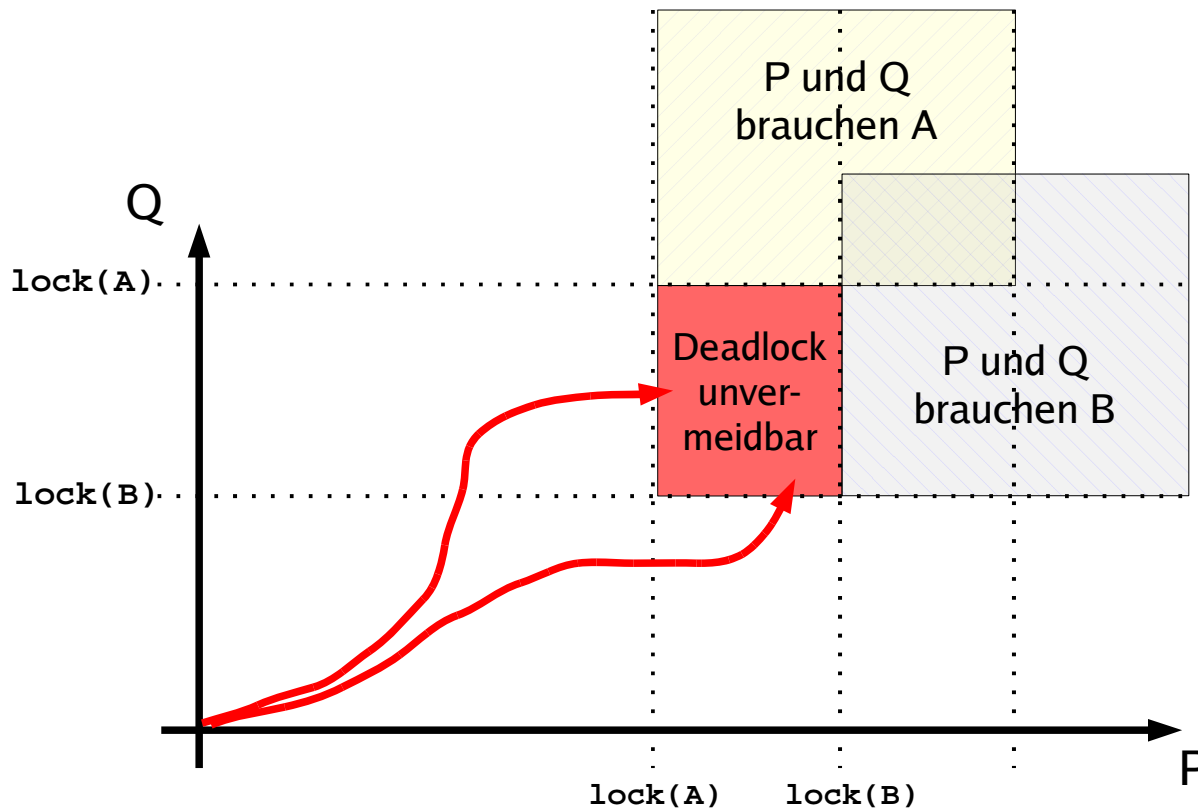
Deadlock: kleinstes Beispiel (2)

Q:

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



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 (4)

- Problem beheben:
P benötigt die Locks nicht gleichzeitig

Prozess P

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

Prozess Q

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

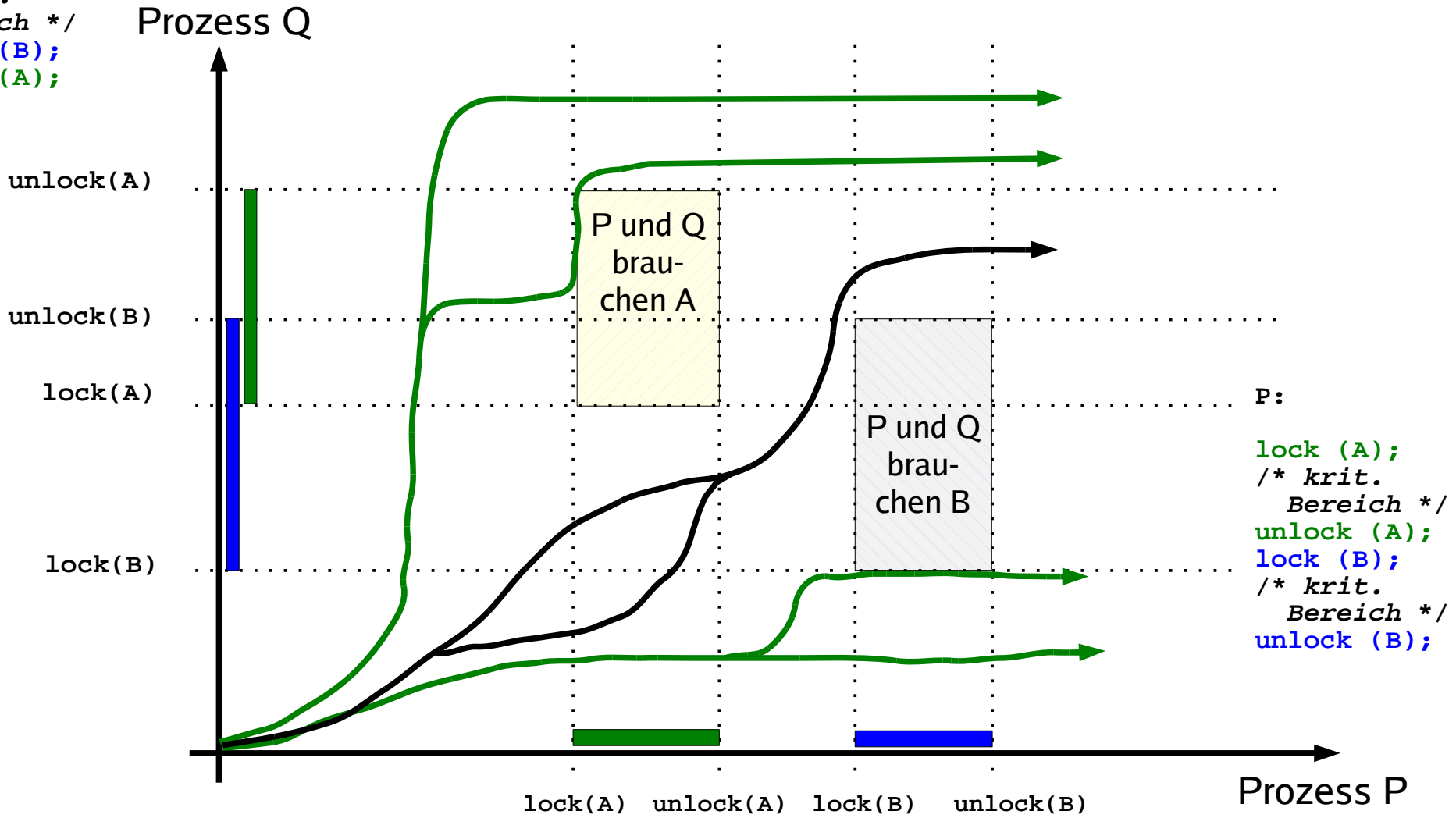
Jetzt kann kein Deadlock mehr auftreten

- Andere Lösung: P und Q fordern A, B in gleicher Reihenfolge an

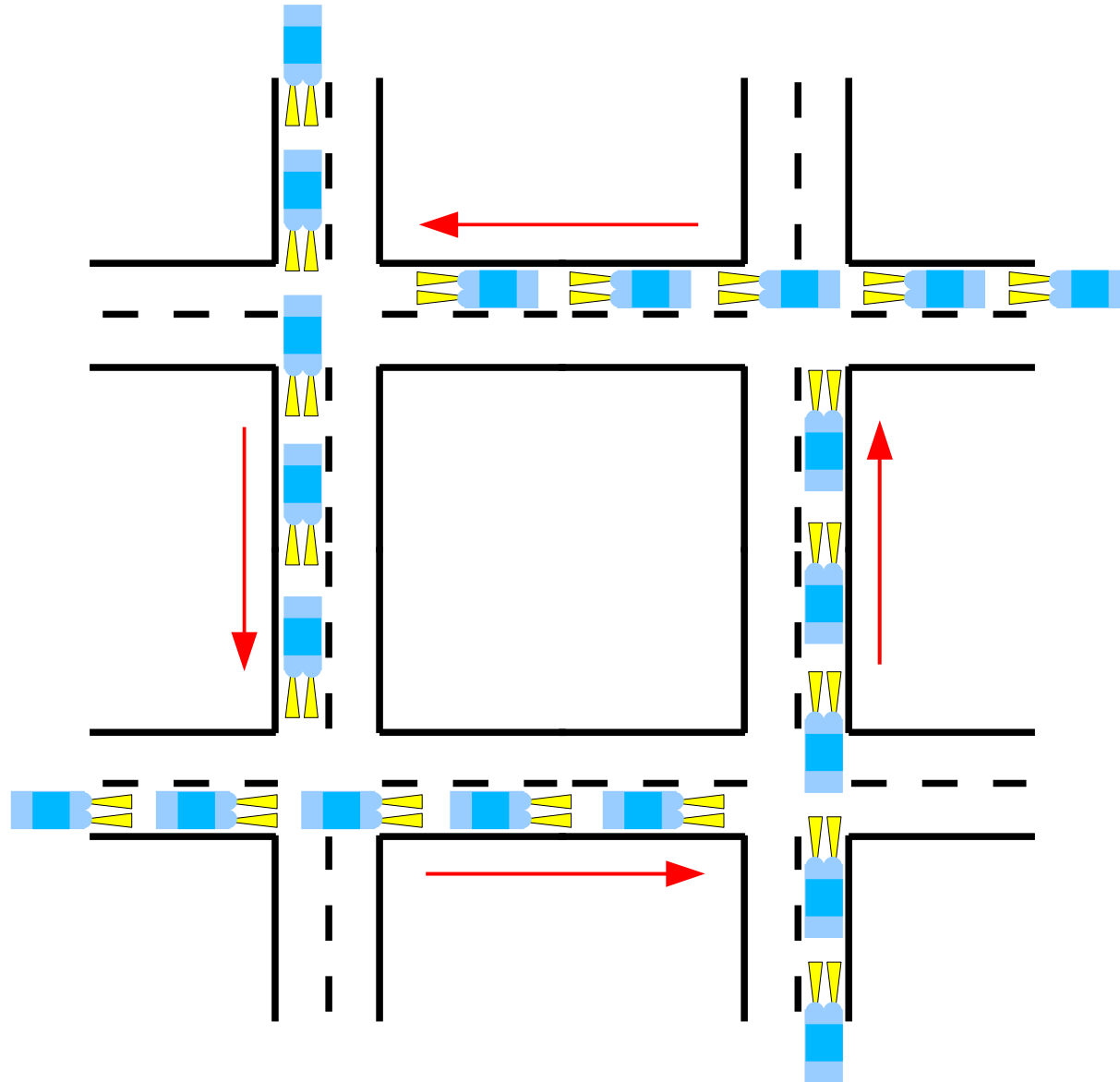
Deadlock: kleinstes Beispiel (5)

Q:

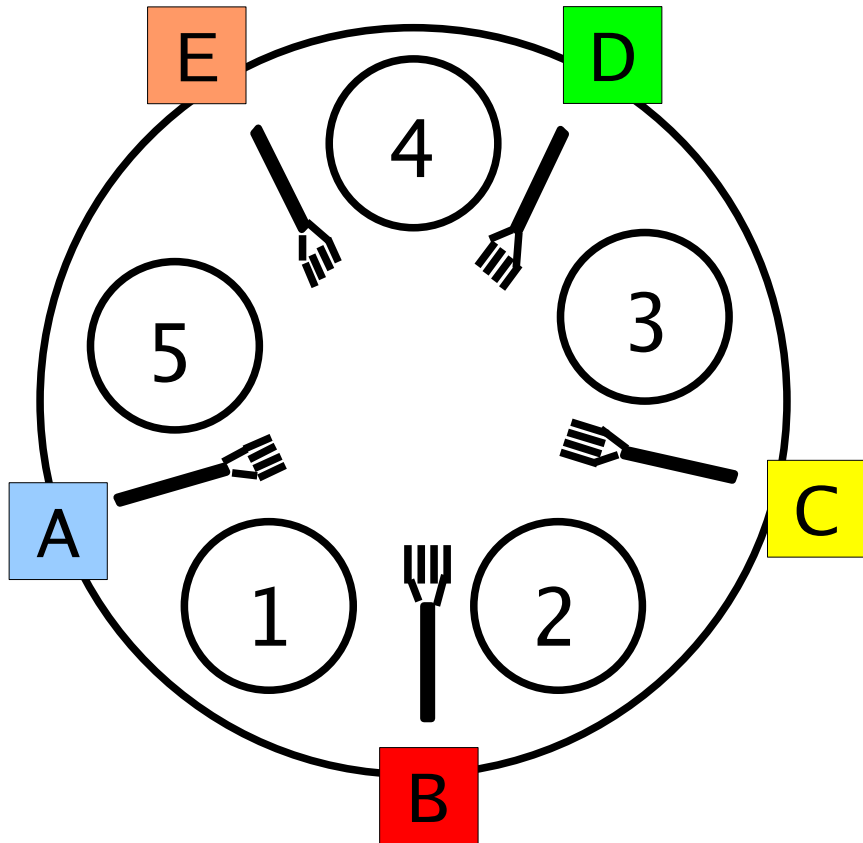
```
lock (B);
lock (A);
/* krit.
Bereich */
unlock (B);
unlock (A);
```



Grid Lock



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

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 (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 wg. 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 (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

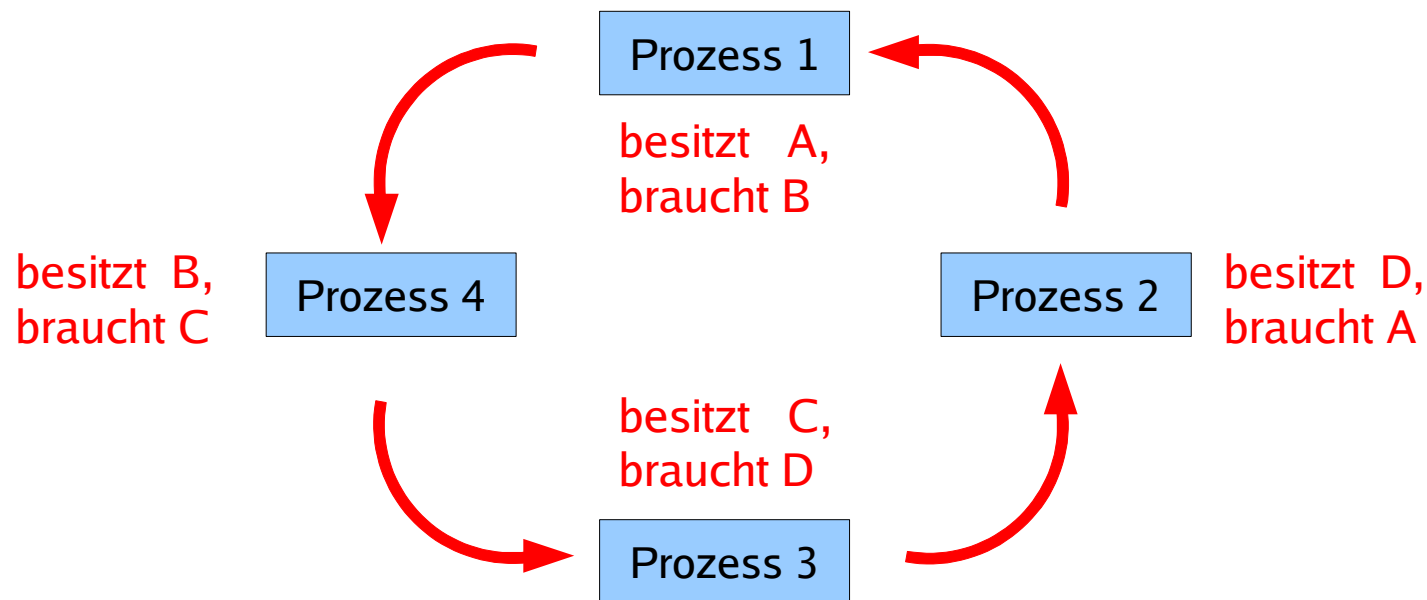
Deadlock-Bedingungen (2)

- (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 (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)

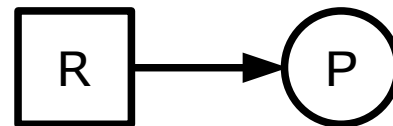
1. **Gegenseitiger Ausschluss**
 2. **Hold and Wait**
 3. **Ununterbrechbarkeit der Ressourcen**
 4. **Zyklisches Warten**
- (1) bis (4) sind **notwendige und hinreichende** Bedingungen für einen Deadlock
 - Das zyklische Warten (4) (und dessen Unauflösbarkeit) sind Konsequenzen aus (1) bis (3)
 - (4) ist der erfolgversprechendste Ansatzpunkt, um Deadlocks aus dem Weg zu gehen

Ressourcen-Zuordnungs-Graph (1)

- Belegung und (noch unerfüllte) Anforderung grafisch darstellen:



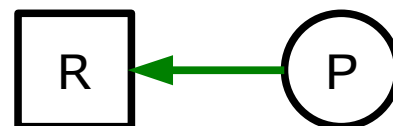
Ressource



P hat R belegt

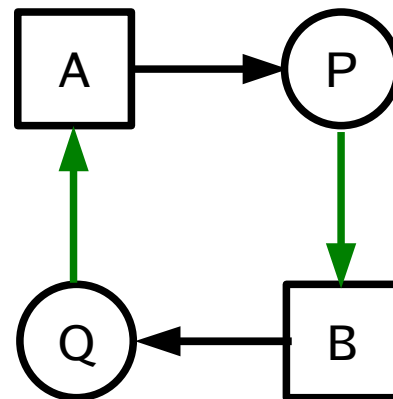


Prozess



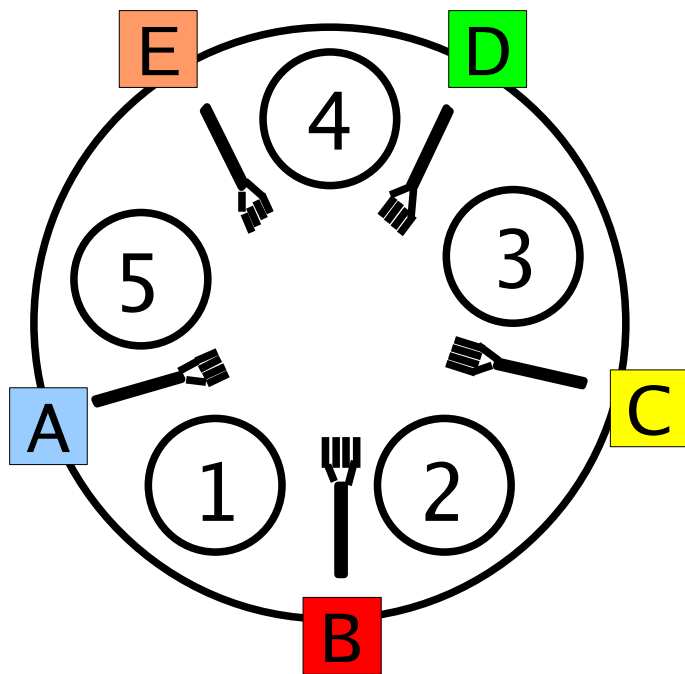
P hat R angefordert

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

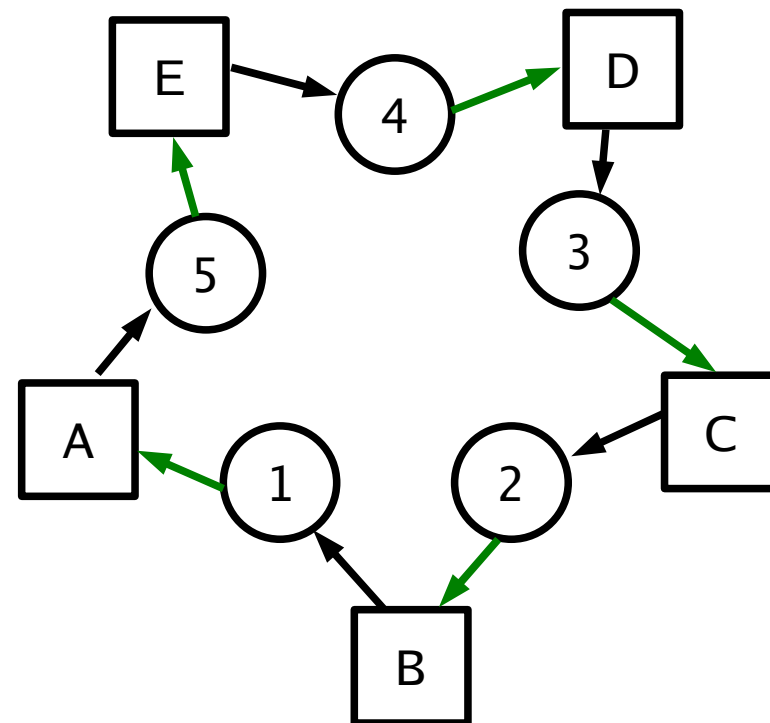


Ressourcen-Zuordnungs-Graph (2)

Philosophen-Beispiel

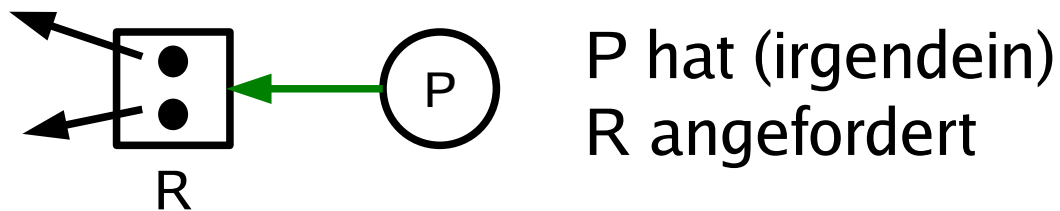


Situation, nachdem alle Philosophen ihre rechte Gabel aufgenommen haben



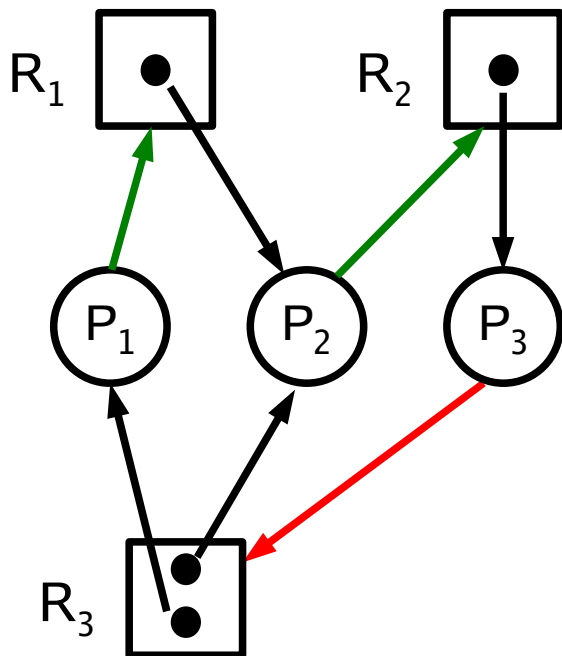
Ressourcen-Zuordnungs-Graph (3)

- Variante für Ressourcen, die mehrfach vorkommen können

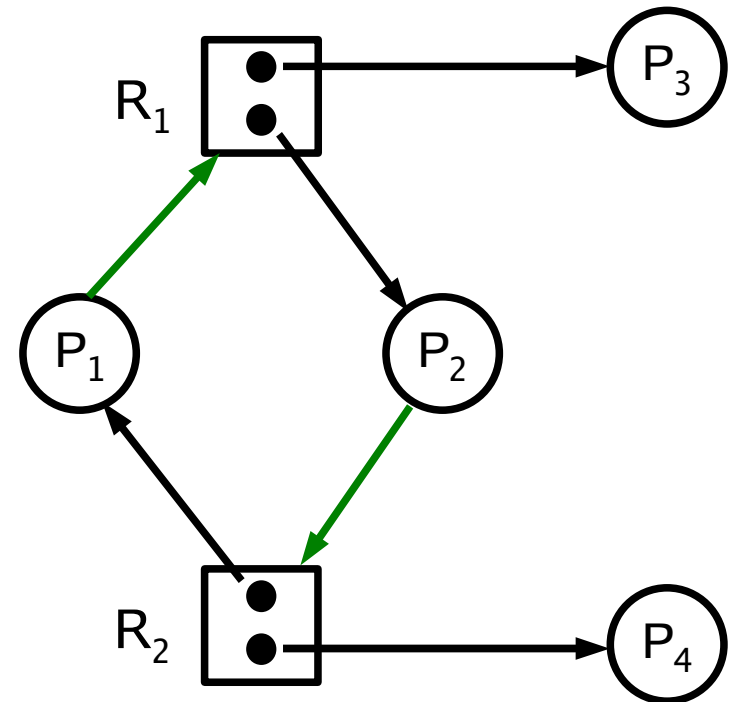


Ressourcen-Zuordnungs-Graph (4)

- 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) (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)

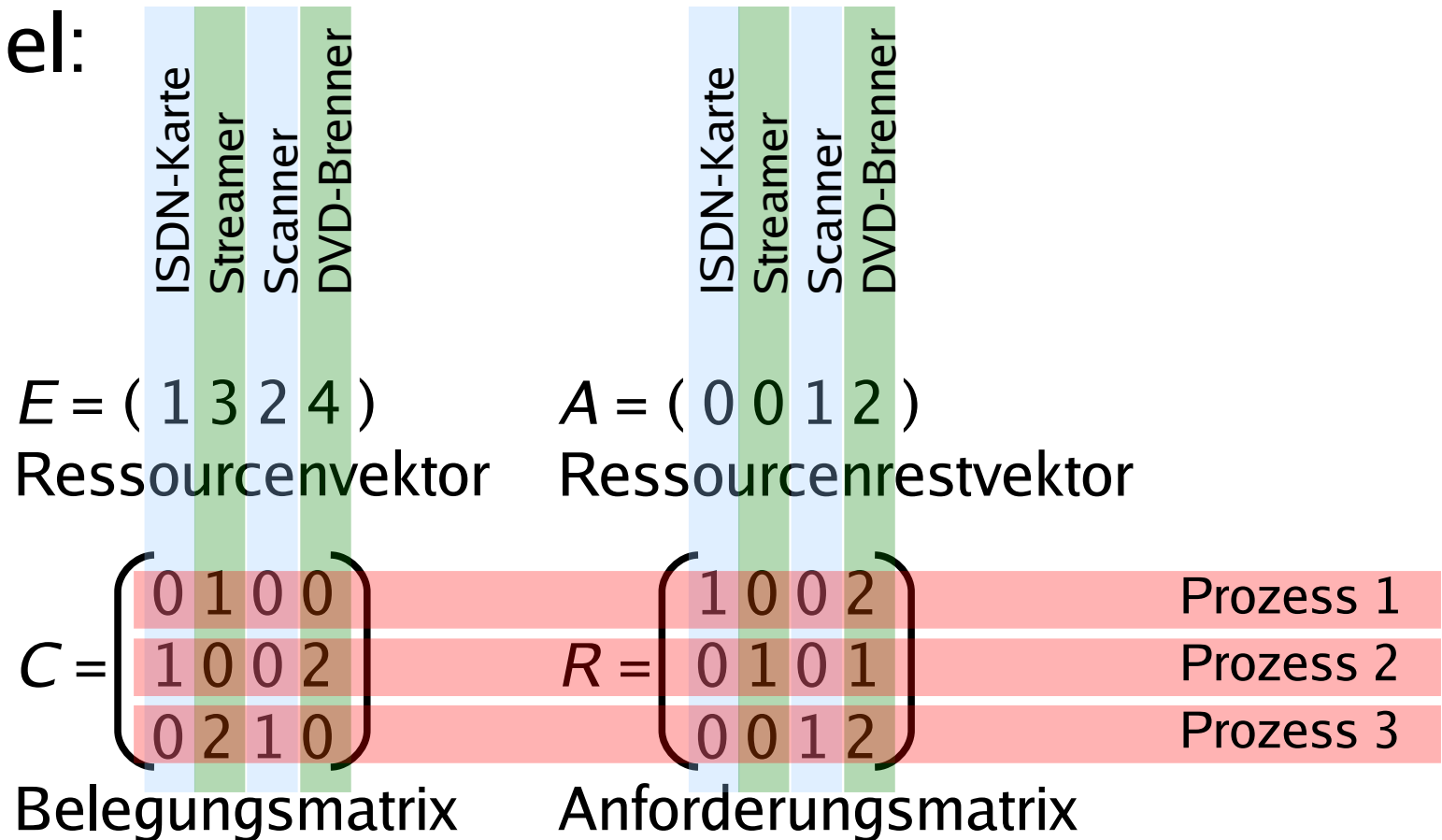
- n Prozesse P_1, \dots, P_n
- m Ressourcentypen R_1, \dots, R_m
Vom Typ R_i gibt es E_i Ressourcen-Instanzen ($i=1, \dots, m$)
-> **Ressourcenvektor** $E = (E_1 \ E_2 \ \dots \ E_m)$
- **Ressourcenrestvektor** A (wie viele sind noch frei?)
- **Belegungsmatrix** C
 C_{ij} = Anzahl Ressourcen vom Typ 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) (4)

-Vermeidung (avoidance)

-Verhinderung (prevention)

Algorithmus

1. Suche einen unmarkierten Prozess P_i , dessen verbleibende Anforderungen vollständig erfüllbar sind, also $R_{ij} \leq A_j$ für alle j
2. Gibt es keinen solchen Prozess, beende Algorithmus
3. Ein solcher Prozess könnte erfolgreich abgearbeitet werden. Simuliere die Rückgabe aller belegten Ressourcen:
 $A := A + C_i$ (i -te Zeile von C)
Markiere den Prozess – er ist nicht Teil eines Deadlocks
4. Weiter mit Schritt 1

Deadlock-Erkennung (detection) (5)

-Vermeidung (avoidance)

-Verhinderung (prevention)

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

- Beispiel

$$E = (1\ 3\ 2\ 4) \quad A = (0\ 0\ 1\ 2)$$
$$C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 2 & 1 & 0 \end{pmatrix} \quad R = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

$$E = (1\ 3\ 2\ 4) \quad A = (0\ 2\ 2\ 2)$$
$$C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 2 & 1 & 0 \end{pmatrix} \quad R = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

$$E = (1\ 3\ 2\ 4) \quad A = (1\ 2\ 2\ 4)$$
$$C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 2 & 1 & 0 \end{pmatrix} \quad R = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

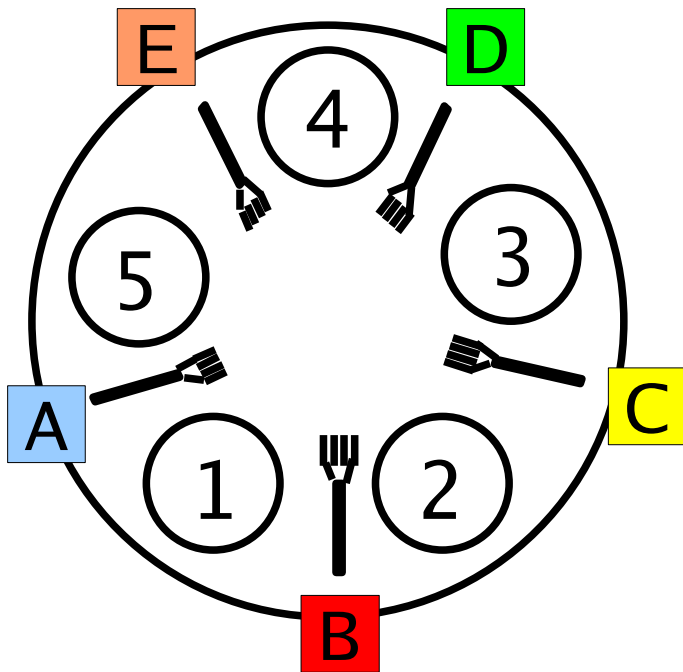
$$E = (1\ 3\ 2\ 4) \quad A = (1\ 3\ 2\ 4)$$
$$C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 2 & 1 & 0 \end{pmatrix} \quad R = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

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	$A =$	0	0	0	0	0
$C =$	1	0	0	0	0	$R =$	0	0	0	0	1
	0	1	0	0	0		1	0	0	0	0
	0	0	1	0	0		0	1	0	0	0
	0	0	0	1	0		0	0	1	0	0
	0	0	0	0	1		0	0	0	1	0

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

Deadlock-Erkennung (detection) (7)

-Vermeidung (avoidance)

-Verhinderung (prevention)

Deadlock-Behebung:

Was tun, wenn ein Deadlock erkannt wurde?

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

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

Deadlock Avoidance (Vermeidung)

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

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

Sichere vs. unsichere Zustände

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

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

Banker-Algorithmus (1)

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

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

Banker-Algorithmus (2) – Beispiel

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



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

sicher, denn es gibt folgende Auszahlungs-/Rückzahlungsreihenfolge:

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

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



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

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

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

(letzte zwei unmöglich)

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

Banker-Algorithmus (3) – Beispiel

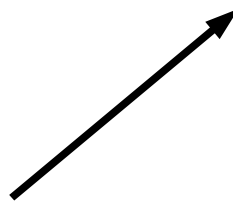
Bank: 1200 €,
 900 € verliehen, 300 € Cash



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



*Kunde 3 fordert
 100 € an*



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 €



-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) (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 111)

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

Banker-Algorithmus (6) – Beispiel

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

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

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

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

Deadlock-Vermeidung (avoidance) (9)

1

$$E = (1057) \quad A' = (230)$$

$$C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$$

4

$$E = (1057) \quad A' = (753)$$

$$C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$$

2

$$E = (1057) \quad A' = (532)$$

$$C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$$

5

$$E = (1057) \quad A' = (1055)$$

$$C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$$

3

$$E = (1057) \quad A' = (743)$$

$$C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$$

6

$$E = (1057) \quad A' = (1057)$$

$$C' = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad R' = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$$

OK!

-Erkennung (detection)

-Vermeidung (avoidance)

Deadlock-Verhinderung (prevention) (1)

Deadlock-Verhinderung (prevention): Vorbeugendes Verhindern

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

-Erkennung (detection)

-Vermeidung (avoidance)

Deadlock-Verhinderung (prevention) (2)

1. Gegenseitiger Ausschluss

- Ressourcen nur dann exklusiv Prozessen zuteilen, wenn es keine Alternative dazu gibt
- Beispiel: Statt mehrerer konkurrierender Prozesse, die einen gemeinsamen Drucker verwenden wollen, 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?)

-Erkennung (detection)

-Vermeidung (avoidance)

Deadlock-Verhinderung (prevention) (3)

2. Hold and Wait

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

-Erkennung (detection)

-Vermeidung (avoidance)

Deadlock-Verhinderung (prevention) (4)

3. Ununterbrechbarkeit der Ressourcen

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

-Erkennung (detection)

-Vermeidung (avoidance)

Deadlock-Verhinderung (prevention) (5)

4. Zyklisches Warten (1)

- Ressourcen 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

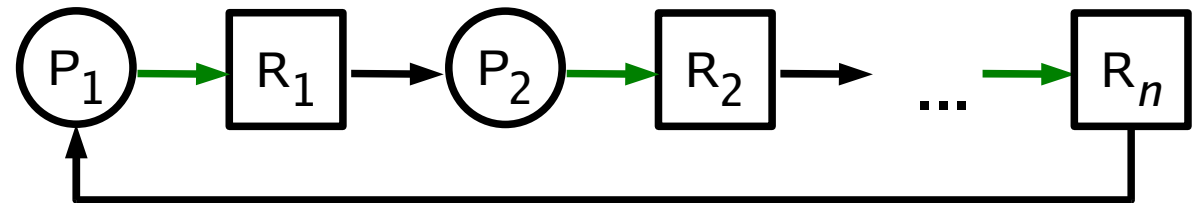
-Erkennung (detection)

-Vermeidung (avoidance)

Deadlock-Verhinderung (prevention) (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)