

Betriebssysteme 1

SS 2019

Prof. Dr.-Ing. Hans-Georg Eßer
Fachhochschule Südwestfalen

Foliensatz E:

- Synchronisation
- Deadlocks

v1.1, 2018/06/14

Einführung (2)

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

```

erhoehe_zaehler( )   Ausgangssituation: w=10
{
  w=read(Adresse);   T1:                T2:
  w=w+1;              w=read(Adresse); // 10
  write(Adresse,w);  w=w+1;           // 11
}                    write(Adresse,w); // 11
                    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 (1)

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

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 (6)

Race Condition:

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

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 (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: Primitive Shell

```
for (;;) {
    read (command);
    f = creat ("/tmp/script"); // Datei erzeugen
    write (f, command);      // Befehl rein schreiben
    close (f);               // speichern/schließen
    chmod ("/tmp/script", 0x777); // ausführbar machen
    system ("/tmp/script");  // Skript ausführen
}
```

Annahme: Dateisystem ohne Zugriffsrechte (z. B. VFAT)

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

Einführung (10)

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

Einführung (9)

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

```
erhoehe_zaeher( ) {
    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

Inhaltsübersicht: Synchronisation

- Einführung, Race Conditions
- Kritische Abschnitte und gegenseitiger Ausschluss
- Synchronisationsmethoden, Standard-„Primitive“:
 - Mutexe
 - Semaphore
 - Monitore (nicht in dieser Vorlesung)

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
- Nicht den Code schützen, sondern die Daten
- Formulierung: kritischen Bereich „betreten“ und „verlassen“ (enter / leave critical section)

Kritische Bereiche (3)

- 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)

Kritische Bereiche (2)

- Bestimmen des kritischen Bereichs nicht ganz eindeutig:

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

- zwei kritische Bereiche oder nur einer?

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

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
```

Aktives / 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.

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

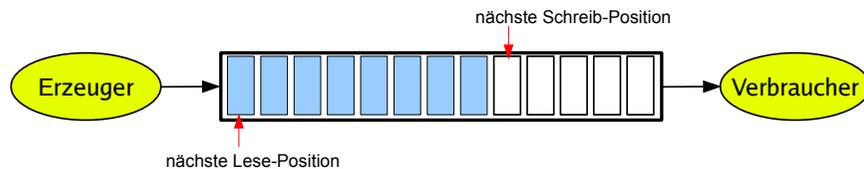
- TSL muss zwei Dinge leisten:
 - Interrupts ausschalten, damit der Test-und-Setzen-Vorgang nicht durch einen anderen Prozess unterbrochen wird
 - Im Falle mehrerer CPUs den Speicherbus sperren, damit kein Prozess auf einer anderen CPU (deren Interrupts nicht gesperrt sind!) auf die gleiche Variable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

Aktives / passives Warten (2)

- **Passives Warten (sleep and wake):**
 - Ein Thread wird schlafen gelegt und wartet auf ein Ereignis, das ihn wieder in den Zustand „bereit“ versetzt.
 - Schlafender 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 schlafende 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 (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 (`count == 0`), weckt er den Verbraucher; bei vollem Puffer blockiert er.
 - Wenn der Verbraucher eine Information abholt und der Puffer voll war (`count == max`), weckt er den Erzeuger; bei leerem Puffer blockiert er.

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 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 (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)

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

←

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 schlafen legen und in eine Warteschlange einreihen, wenn der Semaphor-Wert 0 ist.
- P = (niederl.) probeer

Semaphore (2)

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

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

Semaphore (3)

- Variante: Negative Semaphore-Werte
 - Semaphore zählt Anzahl der wartenden Threads
 - Anfordern (WAIT):
 - Semaphore-Wert um 1 erniedrigen
 - Thread blockieren und in eine Warteschlange einreihen, wenn der Semaphore-Wert ≤ 0 ist.
 - Freigabe (SIGNAL):
 - Thread aus der Warteschlange wecken (falls nicht leer)
 - Semaphore-Wert um 1 erhöhen

Semaphore (4)

- Pseudo-Code für Semaphore-Operationen

```
wait (sem) {  
    if (sem>0)  
        sem--;  
    else  
        self.state=SLEEPING;  
}  
  
signal (sem) {  
    if (P in QUEUE(sem)) {  
        wakeup (P);  
        remove (P, QUEUE);  
    }  
    else  
        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, wird schlafen gelegt → 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. Pseudo-Code:

```
wait (mutex) {                signal (mutex) {
  if (mutex==1)              if (P in QUEUE(mutex)) {
    mutex=0;                  wakeup (P);
  else                        remove (P, QUEUE);
    self.state=SLEEPING;     }
  }
  }                           else
                              mutex=1;
                              }
}
```

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

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

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

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
    }
}
```

Was ist ein Deadlock?

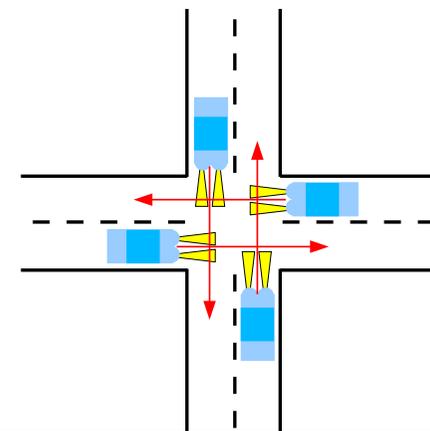
- Eine Menge von Prozessen befindet sich in einer **Deadlock-Situation**, wenn:
 - jeder Prozess auf eine Ressource wartet, die von einem anderen Prozess blockiert wird
 - keine der Ressourcen freigegeben werden kann, weil der haltende Prozess (indem er selbst wartet) blockiert ist
- In einer Deadlock-Situation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

Deadlocks – Gliederung

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

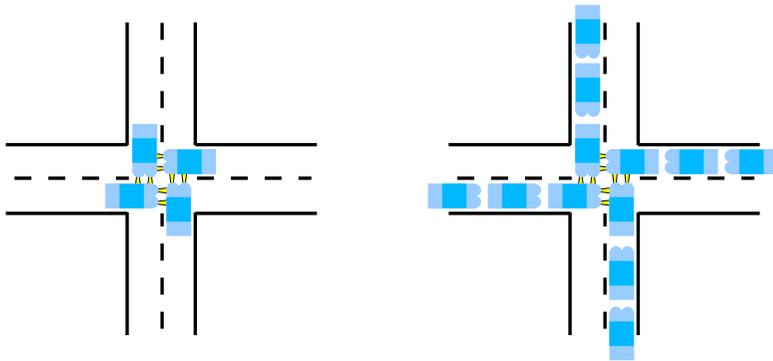
Deadlock: Rechts vor Links (1)

- Der Klassiker: Rechts-vor-Links-Kreuzung



Wer darf fahren?
Potenzieller Deadlock

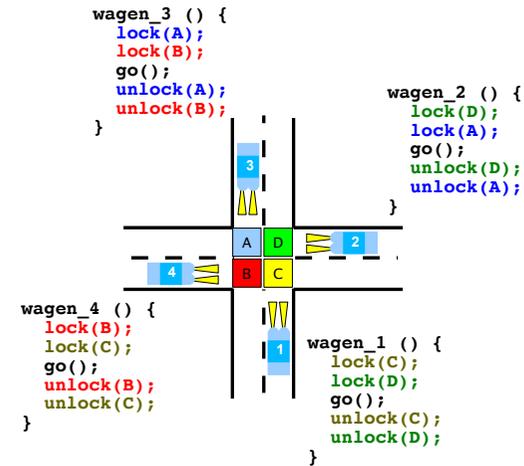
Deadlock: Rechts vor Links (2)



Deadlock, aber behebbar:
eines oder mehrere Autos
können zurücksetzen

Deadlock, nicht behebbar:
beteiligte Autos können nicht
zurücksetzen

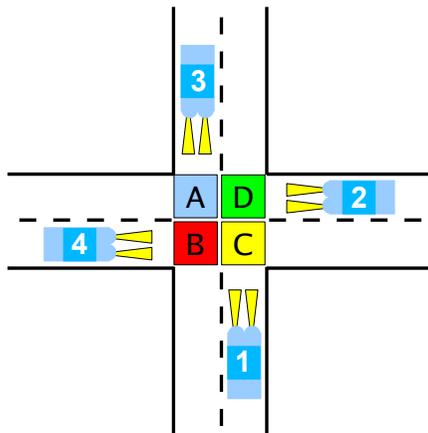
Deadlock: Rechts vor Links (4)



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: 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)

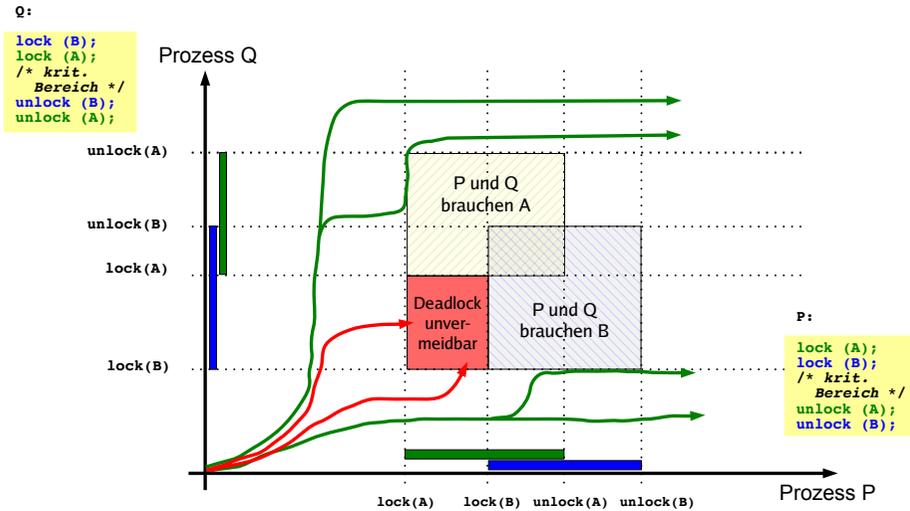
- Zwei Locks A und B
 - z. B. A = Scanner, B = Drucker,
Prozesse P, Q wollen beide eine Kopie erstellen
- Locking in verschiedenen Reihenfolgen

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

Problematische Reihenfolge:

P: lock(A)
Q: lock(B)
P: lock(B) ← blockiert
Q: lock(A) ← blockiert

Deadlock: kleinstes Beispiel (2)



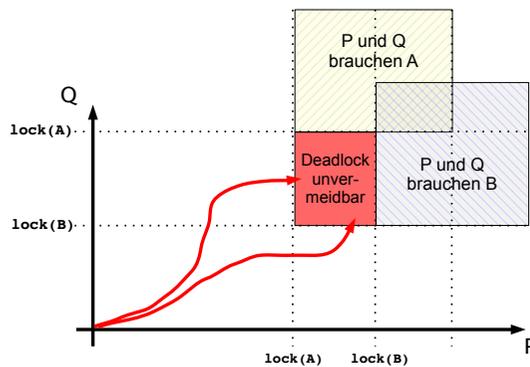
Deadlock: kleinstes Beispiel (4)

- Problem beheben:
P benötigt die Locks nicht gleichzeitig

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

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

Deadlock: kleinstes Beispiel (3)

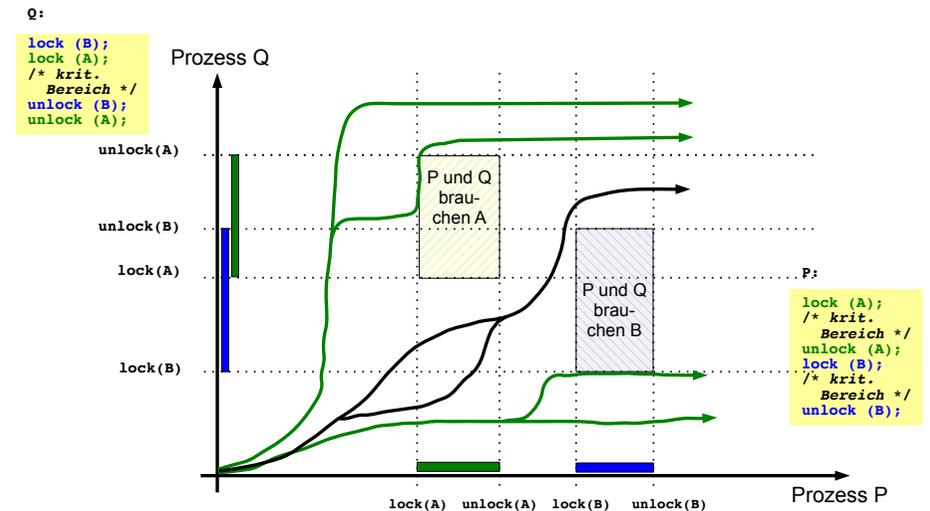


Programmverzahnungen,
die zwangsläufig in den
Deadlock führen:

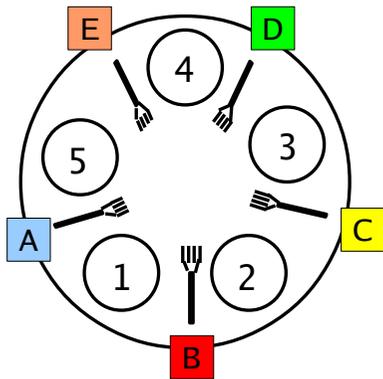
oberer roter Weg:
Q: lock(B)
P: lock(A)

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

Deadlock: kleinstes Beispiel (5)



Fünf-Philosophen-Problem



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

Problematische Reihenfolge:

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

Ressourcen-Typen (2)

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

Ressourcen-Typen (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

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)

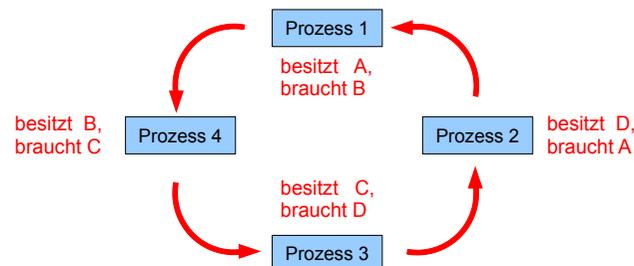
Deadlock-Bedingungen (4)

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

Deadlock-Bedingungen (3)

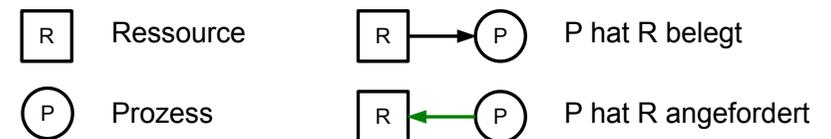
4. Zyklisches Warten

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

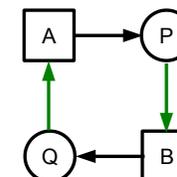


Ressourcen-Zuordnungs-Graph (1)

- Belegung und (noch unerfüllte) Anforderung grafisch darstellen:

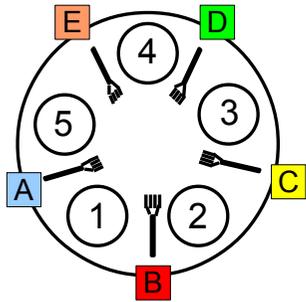


- P, Q aus Minimalbeispiel:
- Deadlock = Kreis im Graph

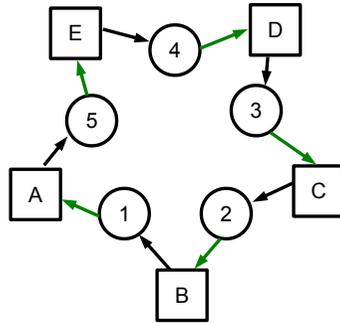


Ressourcen-Zuordnungs-Graph (2)

Philosophen-Beispiel

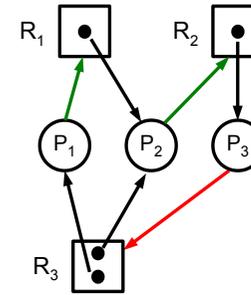


Situation, nachdem alle Philosophen ihre rechte Gabel aufgenommen haben

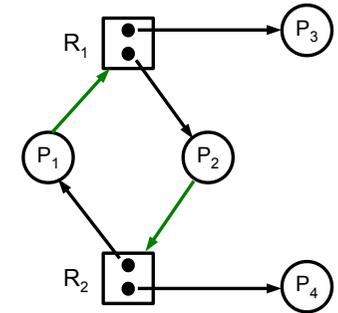


Ressourcen-Zuordnungs-Graph (4)

• 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!

Ressourcen-Zuordnungs-Graph (3)

• Variante für Ressourcen, die mehrfach vorkommen können

