



Betriebssysteme I

WS 2019/2020

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 5. Dezember 2019



Betriebssysteme I

WS 2019/2020

4 Verklemmungen (*Deadlocks*)



Inhalt:

- ➔ Einführung
- ➔ Formale Definition, Voraussetzungen
- ➔ Behandlung von Deadlocks:
 - ➔ Erkennung und Behebung
 - ➔ Vermeidung (*Avoidance*)
 - ➔ Verhinderung (*Prevention*)

- ➔ Tanenbaum 3
- ➔ Stallings 6.1-6.6
- ➔ Nehmer/Sturm 8

4.1 Einführung



- ➔ Wichtige Aufgabe des BSs: Verwaltung von Ressourcen
 - ➔ Ressourcen werden an Prozesse zugeteilt
 - ➔ Ressource kann nur jeweils von einem Prozeß genutzt werden
 - ➔ wechselseitiger Ausschluß
 - ➔ aber: mehrere identische Instanzen einer Ressource möglich
- ➔ Mögliches Problem: **Deadlock**, z.B.
 - ➔ Prozeß A hat Scanner belegt, benötigt CD-Brenner
 - ➔ Prozeß B hat CD-Brenner belegt, benötigt Scanner
 - ➔ A wartet auf B, B wartet auf A, ...



Unterbrechbare und ununterbrechbare Ressourcen

➔ Unterbrechbare Ressource

- ➔ kann einem Prozeß ohne Schaden wieder entzogen werden
- ➔ z.B. Prozessor (Sichern und Wiederherstellen des Prozessorzustands beim Threadwechsel)
- ➔ z.B. Hauptspeicher (Auslagern auf Platte)
- ➔ Deadlocks können verhindert werden

➔ Ununterbrechbare Ressource

- ➔ kann einem Prozeß nicht entzogen werden, ohne daß seine Ausführung fehlschlägt
- ➔ z.B. CD-Brenner

➔ Im Folgenden: ununterbrechbare Ressourcen



Anforderung von Ressourcen

➔ Systemabhängig, z.B.

- ➔ explizite Anforderung über speziellen Systemaufruf
- ➔ implizit beim Öffnen eines Geräts (Systemaufruf `open`)
- ➔ falls Ressource nicht verfügbar: Systemaufruf blockiert*

➔ Häufig wird „Zuteilung“ auch explizit programmiert

- ➔ z.B. durch Nutzung von Semaphoren

➔ Umgekehrt ist auch Belegung eines Semaphors als Ressourcenzuteilung interpretierbar

- ➔ dann i.d.R. Threads statt Prozesse

* Vereinfachung in diesem Kapitel: Prozesse mit nur einem Thread, d.h. gesamter Prozess blockiert!

Anmerkungen zu Folie 221:

Bei Prozessen mit mehreren Threads wird durch einen blockierenden Systemaufruf nur der aufrufende Thread blockiert, was die Betrachtung komplizierter macht. Daher gehen wir diesem Abschnitt davon aus, daß falls ein Prozeß eine bereits belegte Ressource anfordert, der Prozess vollständig blockiert wird, also z.B. nur einen Thread besitzt (klassischer Prozess).

Genau dieselbe Situation tritt ein, wenn mehrere Threads (desselben Prozesses oder auch verschiedener Prozesse) wechselseitigen Ausschluß, z.B. durch Semaphore, realisieren. Hier kann man die Semaphore-Einheiten als ununterbrechbare Ressource auffassen. Die P-Operation ist dann die Anforderung einer Einheit des entsprechenden Semaphors.

Daher wird im Folgenden manchmal von Prozessen (im Sinne von klassischen Prozessen mit einem Thread) und manchmal von Threads gesprochen.

221-1

4.1 Einführung ...



(Animierte Folie)

Beispiel: Nutzung von zwei Ressourcen

➔ Verklemmung möglich

```
Semaphore resource1 = 1;  
Semaphore resource2 = 1;
```

Thread A

```
P(resource1);  
"▷ P(resource2);  
UseBothResources();  
V(resource2);  
V(resource1);
```

Thread B

```
P(resource2);  
"▷ P(resource1);  
UseBothResources();  
V(resource1);  
V(resource2);
```



Definition

Eine Menge von Prozessen* befindet sich in einem **Deadlock-Zustand** (**Verklemmungs-Zustand**), wenn jeder Prozeß* aus der Menge auf ein Ereignis **wartet**, das nur ein anderer Prozeß* aus dieser Menge auslösen kann.

- ➔ Alle Prozesse* warten
- ➔ Die Ereignisse werden daher niemals ausgelöst
- ➔ Keiner der Prozesse* wird jemals wieder aufwachen

- ➔ Im BS-Kontext oft: Ereignis = Freigabe einer Ressource

* Bzw. Thread(s)



Bedingungen für einen Ressourcen-Deadlock

- 1. Wechselseitiger Ausschluß:**
Jede Ressource kann zu einem Zeitpunkt von höchstens einem Prozeß genutzt werden.
- 2. Hold-and-Wait (Besitzen und Warten):**
Ein Prozeß, der bereits Ressourcen besitzt, kann noch weitere Ressourcen anfordern.
- 3. Ununterbrechbarkeit (kein Ressourcenentzug):**
Einem Prozeß, der im Besitz einer Ressource ist, kann diese nicht gewaltsam entzogen werden.
- 4. Zyklisches Warten:**
Es gibt eine zyklische Kette von Prozessen, bei der jeder Prozeß auf eine Ressource wartet, die vom nächsten Prozeß in der Kette belegt ist.

Anmerkungen zu Folie 224:

Die auf Folie 223 angegebene Definition gilt allgemein, die Bedingungen auf dieser Folie gelten aber nur für Deadlocks, die durch das Sperren und Freigeben von Ressourcen entstehen.

Ausser durch Ressourcen-Belegungen (also wechselseitigen Ausschluss) können Deadlocks z.B. auch durch Reihenfolge-Synchronisation entstehen. Mit diesen Ursachen von Deadlocks befassen wir uns hier **nicht**.

224-1

4.2 Formale Beschreibung ...



Bedingungen für einen Deadlock ...

➔ Anmerkungen:

- ➔ Bedingungen 1-3 sind **notwendige** Bedingungen, aber nicht hinreichend
- ➔ Bedingung 4 ist eine potentielle Konsequenz aus 1-3
 - ➔ die Unauflösbarkeit des zyklischen Wartens ist eine Folge aus 1-3
- ➔ Alle vier Bedingungen zusammen sind **notwendig** und **hinreichend** für eine Verklemmung

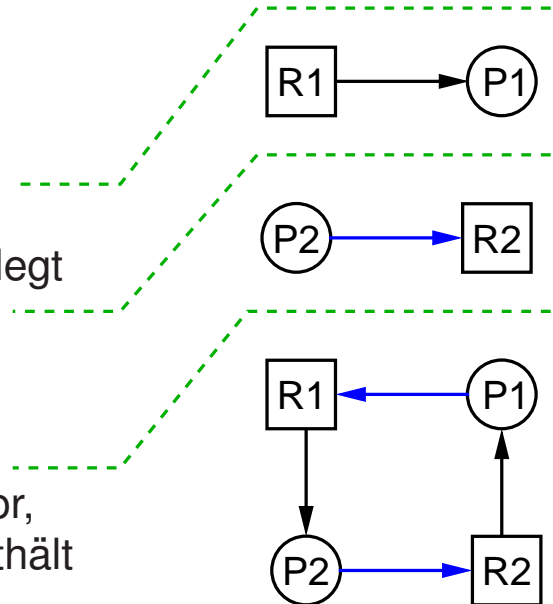
➔ Konsequenz:

- ➔ wenn eine der Bedingungen unerfüllbar ist, können keine Deadlocks auftreten

Modellierung von Deadlocks

➔ Belegungs-Anforderungs-Graph:

- ➔ gerichteter Graph
- ➔ zwei Arten von Knoten:
 - ➔ Prozesse (○)
 - ➔ Ressourcen (□)
- ➔ Kante Ressource → Prozeß: Ressource ist vom Prozeß belegt
- ➔ Kante Prozeß → Ressource: Prozeß wartet auf Zuteilung der Ressource



- ➔ Ein Deadlock liegt genau dann vor, wenn der Graph einen Zyklus enthält

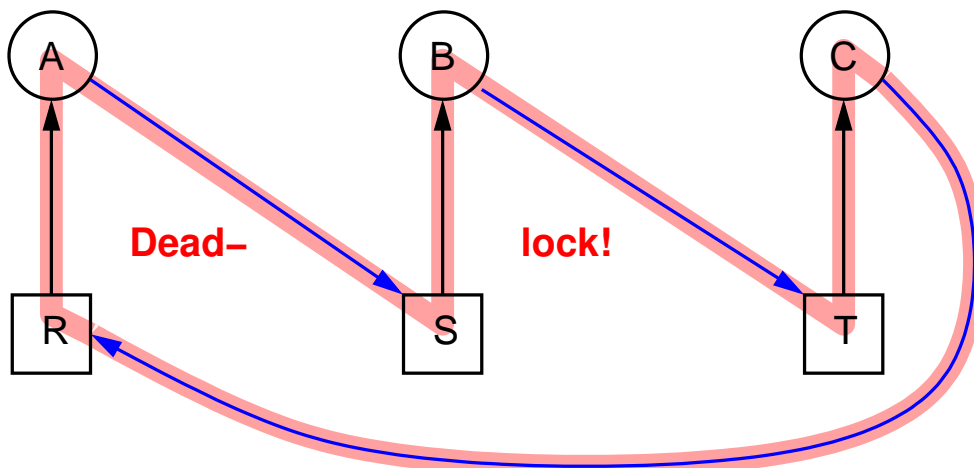
(Animierte Folie)

Beispiel: Ablauf mit Verklemmung

➔ Anfrage R
➔ Anfrage S
➔ Freigabe R
➔ Freigabe S

➔ Anfrage S
➔ Anfrage T
➔ Freigabe S
➔ Freigabe T

➔ Anfrage T
➔ Anfrage R
➔ Freigabe T
➔ Freigabe R



Reihenfolge: A-B-C-A-B-C...

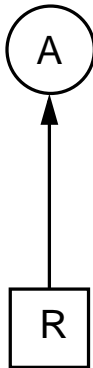
4.2 Formale Beschreibung ...



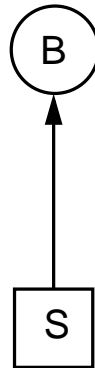
(Animierte Folie)

Beispiel: Ablauf ohne Verklemmung

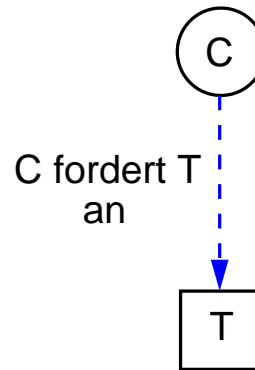
→ Anfrage R
Anfrage S
Freigabe R
Freigabe S



→ Anfrage S
Anfrage T
Freigabe S
Freigabe T



→ Anfrage T
Anfrage R
Freigabe T
Freigabe R



Betriebssystem erkennt, daß Belegung von T durch C zu Deadlock führen könnte ==> C wird blockiert

4.3 Behandlung von Deadlocks



Prinzipiell vier Möglichkeiten

- Vogel-Strauß-Algorithmus
 - nichts tun und hoffen, daß alles gut geht
- **Deadlock-Erkennung und -Behebung**
 - lässt zunächst alle Anforderungen zu, löst ggf. Deadlock auf
- **Deadlock-Avoidance** (Deadlock-Vermeidung)
 - BS lässt Ressourcenanforderung nicht zu, wenn dadurch ein Deadlock entstehen könnte
- **Deadlock-Prevention** (Deadlock-Verhinderung)
 - macht eine der 4 Deadlock-Bedingungen unerfüllbar



Achtung: Begriffsverwirrung bei Verhinderung vs. Vermeidung!

Anmerkungen zu Folie 229:

Im Tanenbaum-Buch ist *Deadlock-Avoidance* mit Verhinderung und *Deadlock-Prevention* mit Vermeidung übersetzt.

Im Stallings-Buch ist *Deadlock-Avoidance* mit Vermeidung und *Deadlock-Prevention* mit Verhinderung übersetzt.

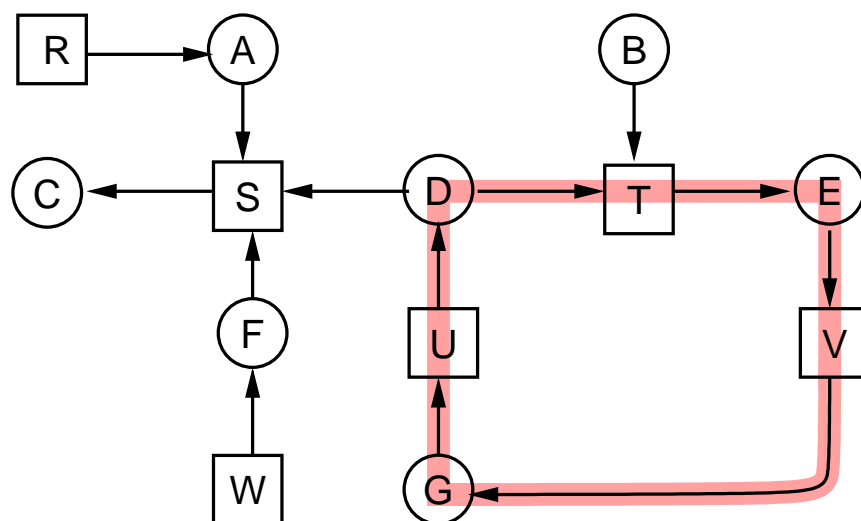
229-1

4.3.1 Deadlock-Erkennung und -Behebung



Deadlock-Erkennung bei einer Ressource pro Typ

➔ Deadlock \Leftrightarrow Belegungs-Anforderungs-Graph enthält Zyklus



➔ Benötigt Algorithmus, der Zyklen in Graphen findet



Deadlock-Erkennung bei mehreren Ressourcen pro Typ

- ➔ Belegungs-Anforderungs-Graph nicht mehr adäquat
 - Prozeß wartet nicht auf **bestimmte** Ressource, sondern auf irgendeine Ressource des passenden Typs
- ➔ Im Folgenden: **Matrix-basierter Algorithmus**
- ➔ Das System sei wie folgt modelliert:
 - n Prozesse P_1, \dots, P_n
 - m Klassen von Ressourcen
 - Klasse i ($1 \leq i \leq m$) enthält E_i Ressource-Instanzen

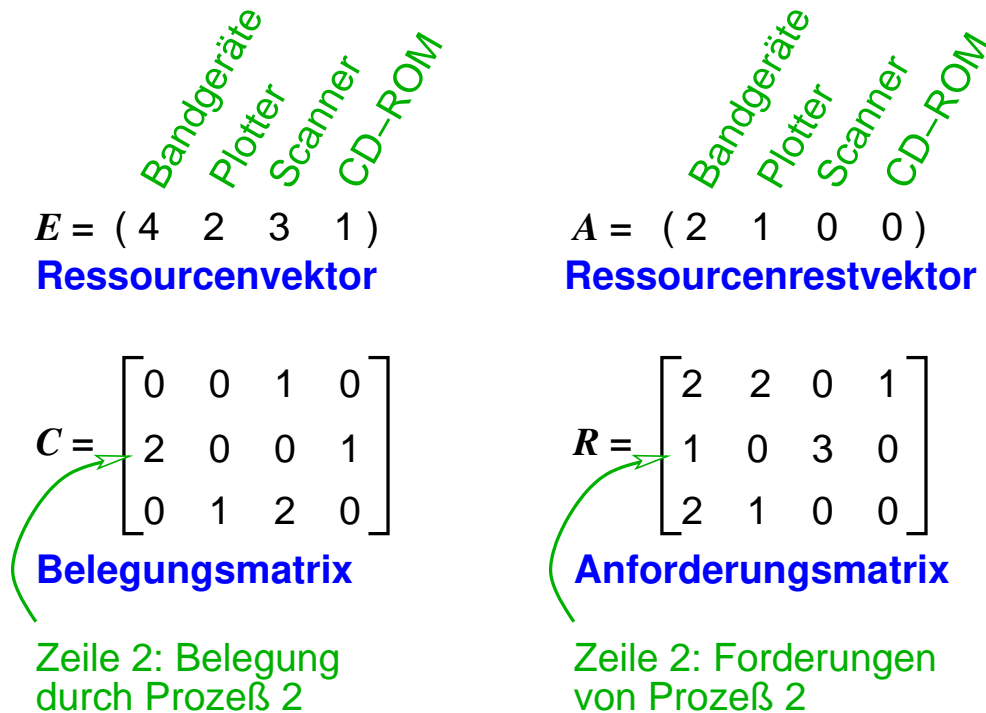


Datenstrukturen des Matrix-Algorithmus

- ➔ **Ressourcenvektor E**
 - Anzahl verfügbarer Ressourcen für jede Klasse
- ➔ **Ressourcenrestvektor A**
 - gibt für jede Klasse an, wieviele Ressourcen noch frei sind
- ➔ **Belegungsmatrix C**
 - C_{ij} = Anzahl der Ressourcen der Klasse j , die Prozeß P_i belegt
 - Invariante: $\forall j = 1 \dots m : \sum_{i=1}^n C_{ij} + A_j = E_j$
- ➔ **Anforderungsmatrix R**
 - R_{ij} = Anzahl der Ressourcen der Klasse j , die Prozeß P_i im Moment anfordert bzw. auf deren Zuteilung er wartet



Beispiel:



Matrix-Algorithmus

1. Suche unmarkierten Prozeß P_i , so daß i -te Zeile von R kleiner oder gleich A ist, d.h. $R_{ij} \leq A_j$ für alle $j = 1 \dots m$
 2. Falls ein solcher Prozeß existiert:
addiere i -te Zeile von C zu A , markiere Prozeß P_i , gehe zu 1
 3. Andernfalls: Ende
- ➔ Alle am Ende nicht markierten Prozesse sind an einem Deadlock beteiligt
 - ➔ Schritt 1 sucht Prozeß, der zu Ende laufen könnte
 - ➔ Schritt 2 simuliert Freigabe der Ressourcen am Prozeßende

4.3.1 Deadlock-Erkennung und -Behebung ...



(Animierte Folie)

Beispiel:

$$E = (4 \quad 2 \quad 3 \quad 1)$$

Ressourcenvektor

$$A = (2 \quad 1 \quad 0 \quad 0)$$

Ressourcenrestvektor

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Belegungsmatrix

$$R = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 0 & 3 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Anforderungsmatrix

1. $R_{3,j} \leq A_j$ für $j = 1 \dots 4$
Anforderungen von Prozeß 3 erfüllbar

4.3.1 Deadlock-Erkennung und -Behebung ...



(Animierte Folie)

Beispiel:

$$E = (4 \quad 2 \quad 3 \quad 1)$$

Ressourcenvektor

$$A = (2 \quad 2 \quad 2 \quad 0)$$

Ressourcenrestvektor

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Belegungsmatrix

$$R = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 0 & 3 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Anforderungsmatrix

1. $R_{1,4} > A_4$ Prozeß 1 kann auch nicht weiterlaufen **Deadlock zwischen Prozeß 1 und 2!**



Wann wird Deadlock-Erkennung durchgeführt?

- ➔ Bei jeder Ressourcen-Anforderung:
 - ➔ Deadlocks werden schnellstmöglich erkannt
 - ➔ ggf. zu hoher Rechenaufwand
- ➔ In regelmäßigen Abständen
- ➔ Wenn Prozessorauslastung niedrig ist
 - ➔ Rechenkapazität ist verfügbar
 - ➔ niedrige Auslastung kann Hinweis auf Deadlock sein



Möglichkeiten zur Behebung von Deadlocks

- ➔ Temporärer Entzug einer Ressource
 - ➔ schwierig bis unmöglich
 - ➔ (wir betrachten hier ununterbrechbare Ressourcen ...)
- ➔ Rücksetzen eines Prozesses
 - ➔ Prozeßzustand wird regelmäßig gesichert (*Checkpoint*)
 - ➔ bei Verklemmung: Rücksetzen auf Checkpoint, bei dem Prozeß Ressource noch nicht belegt hatte
 - ➔ auch schwierig
- ➔ Abbruch eines Prozesses
 - ➔ wähle Prozeß, der problemlos neu gestartet werden kann
 - ➔ z.B. in Datenbanksystemen (Transaktionen)

Betriebssysteme I

WS 2019/2020

12.12.2019

Roland Wismüller
 Betriebssysteme / verteilte Systeme
 roland.wismueller@uni-siegen.de
 Tel.: 0271/740-4050, Büro: H-B 8404

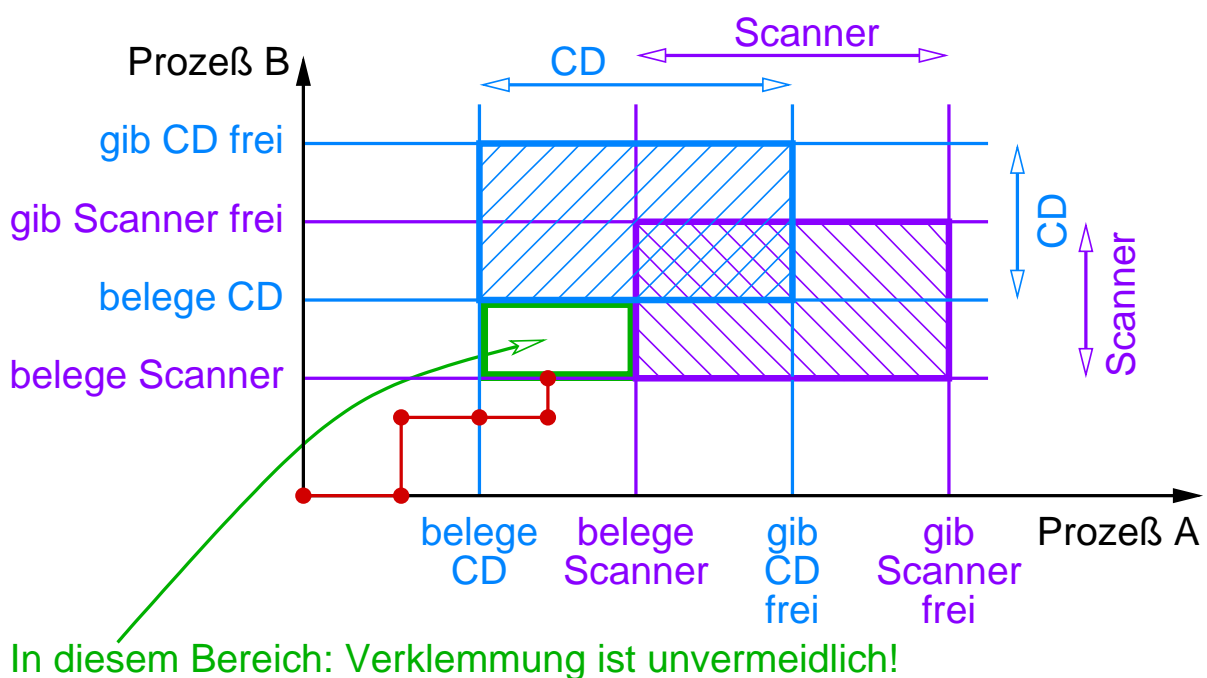
Stand: 5. Dezember 2019

4.3.2 Deadlock-Avoidance



(Animierte Folie)

Vorbemerkung: Ressourcenspur





Sichere und unsichere Zustände:

- ➔ Ein Zustand heißt **sicher**, wenn es eine Ausführungsreihenfolge der Prozesse gibt, in der alle Prozesse (ohne Deadlock) zu Ende laufen, selbst wenn alle Prozesse sofort die maximalen Ressourcenanforderungen stellen
- ➔ Sonst heißt der Zustand **unsicher**
- ➔ Anmerkung:
 - ein unsicherer Zustand muß nicht zwangsläufig in eine Verklemmung führen
 - es müssen ja nicht alle Prozesse (sofort) ihre Maximalforderungen stellen!



Bankiers-Algorithmus

- ➔ BS teilt Ressource nur dann zu, wenn dadurch auf keinen Fall ein Deadlock entstehen kann
 - dazu: BS prüft bei jeder Ressourcenanforderung, ob der Zustand **nach** der Anforderung noch **sicher** ist
 - falls ja, wird die Ressource zugeteilt
 - falls nein, wird der anfordernde Prozeß blockiert
 - sehr vorsichtige (restriktive) Ressourcenzuteilung!
- ➔ Das BS muß dazu die (maximalen) zukünftigen Forderungen aller Prozesse kennen!
 - in der Praxis i.a. nicht erfüllbar
 - Verfahren nur für spezielle Anwendungsfälle tauglich



Bestimmung, ob ein Zustand sicher ist

- ➔ Durch angepaßten Matrix-Algorithmus zur Deadlock-Erkennung (☞ 4.3.1, Folie 231)
- ➔ Der aktuelle Zustand des Systems ist durch E , A , und C beschrieben
 - ➔ vorhandene, freie und belegte Ressourcen
- ➔ R beschreibt nun die **maximalen zukünftigen Forderungen**
- ➔ Der Zustand ist genau dann sicher, wenn der Algorithmus für diese Werte von E , A , C und R keine Verklemmung erkennt
 - ➔ d.h., auch wenn im aktuellen Zustand (E , A , C) alle Prozesse sofort ihre maximalen Forderungen (R) stellen, gibt es einen Ablauf, der alle Prozesse zu Ende führt

4.3.3 *Deadlock-Prevention*



Ziel: Vorbeugendes Verhindern von Deadlocks

- ➔ Eine der vier Deadlock-Bedingungen wird unerfüllbar gemacht
 - ➔ damit sind Deadlocks unmöglich
 - ➔ häufigste Lösung in der Praxis!

Wechselseitiger Ausschluß

- ➔ Ressourcen nur dann direkt an einzelne Prozesse zuteilen, wenn dies unvermeidlich ist
- ➔ Beispiel: Zugriff auf Drucker nur über Drucker-Spooler
 - ➔ es gibt keine Konkurrenz mehr um den Drucker
 - ➔ aber: evtl. Deadlocks an anderer Stelle möglich (Konkurrenz um Platz im Spooler-Puffer)



Hold-and-Wait-Bedingung

- ➔ Forderung: jeder Prozeß muß alle seine Ressourcen bereits beim Start anfordern
 - Nachteil: i.d.R. sind Forderungen nicht bekannt
 - Weiterer Nachteil: mögliche Ressourcen-Verschwendung
 - Anwendung bei Transaktionen in Datenbank-Systemen:
Two Phase Locking (Sperrphase, Zugriffsphase)
- ➔ Alternative: vor Anforderung einer Ressource alle belegten Ressourcen kurzzeitig freigeben
 - i.a. auch nicht praktikabel (wechselseitiger Ausschluß!)

Ununterbrechbarkeit (kein Ressourcenentzug)

- ➔ i.a. unpraktikabel, siehe Deadlock-Behebung



Zyklisches Warten

- ➔ Durchnummerieren aller Ressourcen
- ➔ Anforderungen von Ressourcen zu beliebigem Zeitpunkt, aber nur in aufsteigender Reihenfolge!
 - d.h. Prozeß darf nur Ressource mit größerer Nummer als bereits belegte Ressourcen anfordern
- ➔ Deadlocks werden damit unmöglich:
 - zu jedem Zeitpunkt: betrachte reservierte Ressource mit der höchsten Nummer
 - Prozeß, der diese Ressource belegt, kann zu Ende laufen
 - er wird nur Ressourcen mit höherer Nummer anfordern
 - Induktion: alle Prozesse können zu Ende laufen



Zyklisches Warten ...

- ➔ In der Praxis erfolgversprechendster Ansatz zur Verhinderung von Deadlocks
- ➔ Probleme:
 - Festlegen einer für alle Prozesse tauglichen Ordnung nicht immer praktikabel
 - kann zu Ressourcenverschwendung führen
 - Ressourcen müssen evtl. bereits reserviert werden, bevor sicher ist, daß sie benötigt werden

4.4 Verhungering



- ➔ Situation: ein Prozeß P bekommt eine Ressource nie zugeteilt, obwohl kein Deadlock vorliegt
 - wegen ständiger Ressourcenanforderungen anderer Prozesse, die vor der von P erfüllt werden
- ➔ Beispiel: Lösung des Leser/Schreiber-Problems aus **3.1.5**
- ➔ Vermeidung des Verhungerns: z.B. durch **FIFO-Strategie**
 - FIFO: *First In First Out*
 - teile Ressource an den Prozeß zu, der am längsten wartet
- ➔ Anmerkung: manchmal ist die Möglichkeit des Verhungerns „gewollt“
 - wenn bestimmten Prozessen Priorität gegeben werden soll

4.5 Zusammenfassung / Wiederholung



- ➔ Deadlock: eine Menge von Prozessen wartet auf Ereignisse, die nur ein anderer Prozeß der Menge auslösen kann
- ➔ Bedingungen für Ressourcen-Deadlocks:
 - Wechselseitiger Ausschluß
 - *Hold-and-Wait* (Besitzen und Warten)
 - Ununterbrechbarkeit (kein Ressourcenentzug)
 - Zyklisches Warten
- ➔ Behandlung von Deadlocks:
 - Erkennung und Behebung
 - Erkennung: Zyklus im Belegungs-Anforderungs-Graph, Matrix-basierter Algorithmus
 - Behandlung: meist Abbruch eines Prozesses
 - z.B. bei Datenbank-Transaktionen

4.5 Zusammenfassung / Wiederholung ...



- ➔ Behandlung von Deadlocks ...:
 - *Avoidance* (Vermeidung)
 - Ressource nur dann vergeben, wenn der entstehende Zustand **sicher** ist
 - sicherer Zustand: kein Deadlock, selbst wenn alle Prozesse sofort ihre Maximalforderungen stellen
 - *Prevention* (Verhinderung)
 - eine der vier Bedingungen unerfüllbar machen
 - z.B. *Spooling*: vermeidet wechselseitigen Ausschluß
 - oft auch: Festlegung einer Reihenfolge der Ressourcen
 - wenn alle Prozesse die Ressourcen nur in dieser Reihenfolge belegen, können keine Deadlocks entstehen