FH Bielefeld
University of
Applied Sciences

Vorlesung

Betriebssysteme

Teil 6

Deadlocks

FH Bielefeld
University of
Applied Sciences

Inhalt

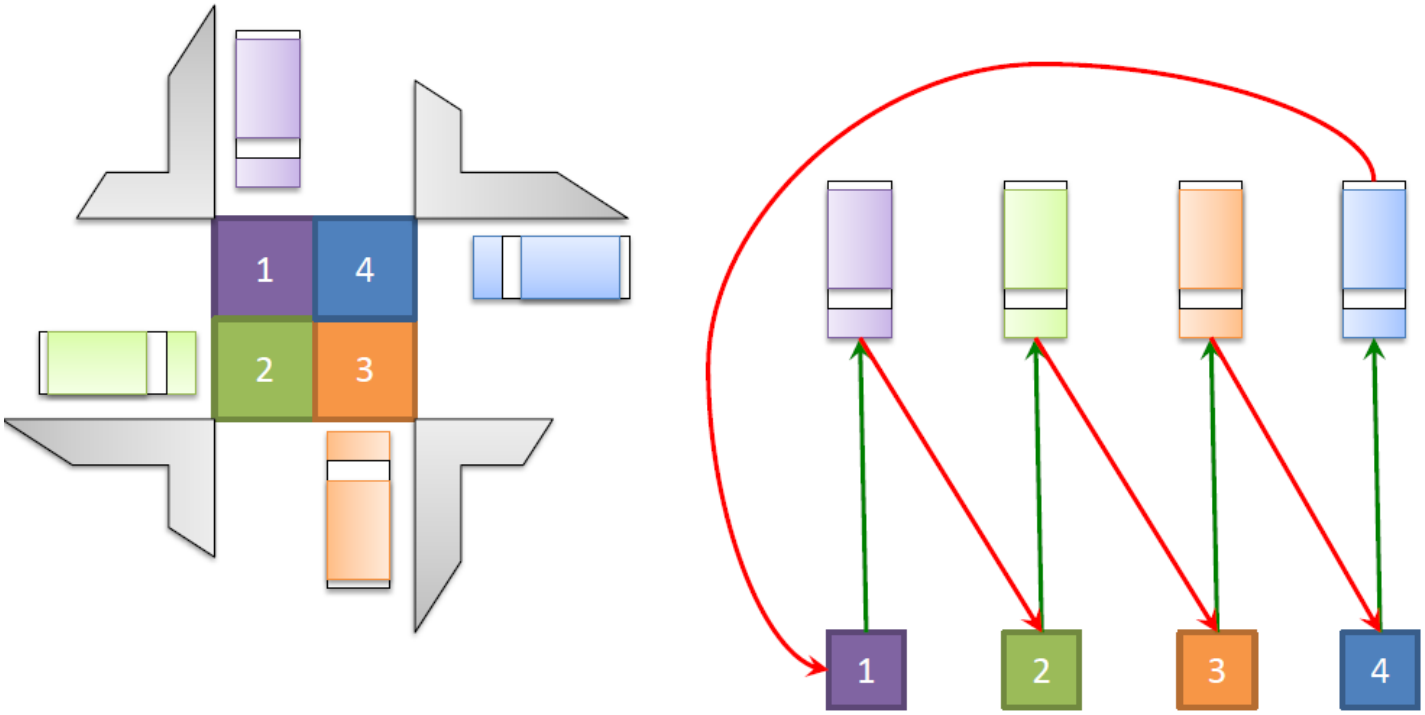
Letzte Vorlesung

- Interprozesskommunikation
  - Pipes, IPC, Shared Memory
- Mutexe und Semaphore
  - Schutz von kritischen Abschnitten

Heute

- Verklemmungen
  - Modellieren
  - Erkennen und Beheben
  - Verhindern (Banker Algorithm)

Belegungs-Anforderungs-Graph: Modellierung Straßenkreuzung



- Jedes Auto **fährt ein Stück** auf die Kreuzung (Areal = *Ressource*). Ein weiteres Areal wird beansprucht, um die Kreuzung **gerade passieren** (= *Prozess*) zu können. Dies wird aber von einem **anderen Fahrzeug belegt** →*Deadlock*.

FH Bielefeld
University of
Applied Sciences

Deadlocks: Bedingungen

Vier **Bedingungen**, die zum Auftreten einer Verklemmung *notwendig* sind (*Coffman / Elphick / Shoshani*, 1971):

1. *Wechselseitiger Ausschluss*:
   - Jede Ressource ist entweder verfügbar oder genau einem Prozess zugeordnet.
2. *Hold-and-wait-Bedingung*:
   - Prozesse, die schon Ressourcen reserviert haben, können noch weitere Ressourcen anfordern.
3. *Nichtunterbrechbarkeit*:
   - Ressourcen, die einem Prozess bewilligt wurden, können diesem nicht wieder entzogen werden (*no preemption*).
4. *Zyklische Wartebedingung*:
   - Es muss eine zyklische Kette von Prozessen geben, von denen jeder auf eine Ressource wartet, die dem nächsten Prozess in der Kette gehört.

FH Bielefeld
University of
Applied Sciences

Behandlung von Deadlocks

Dem Deadlock Problem kann auf **vier möglichen Weisen** begegnet werden:

1. **Ignorieren**: Erscheint unangemessen, aber:
   - Die meisten Betriebssysteme inklusive Windows und Unix verfahren so.
   - Kein zusätzlicher Overhead für selten auftretende Ereignisse.
2. **Erkennen und Beheben**: Lasse Deadlocks passieren und behebe sie dann.
3. **Dynamische Verhinderung:** durch vorsichtiges Ressourcenmanagement.
4. **Vermeidung von Deadlocks**: Eine der vier notwendigen Bedingungen muss prinzipiell unerfüllbar werden.

FH Bielefeld
University of
Applied Sciences

Deadlocks ignorieren

FH Bielefeld
University of
Applied Sciences

Deadlocks: Erkennen und Beheben

Zwei Phasen des Algorithmus:

- **Erkennungsphase** (*detection algorithm*): Das Betriebssystem überprüft periodisch, ob ein Deadlock vorliegt.
- **Behebungsphase** (*recovery algorithm*): Im Fall eine Deadlocks ergreift das Betriebssystem Maßnahmen, um den Deadlock aufzulösen.

- Notwendige **Randbedingungen zur Erkennung**:
- Die notwendigen Informationen über belegte und angeforderte Ressourcen müssen einfach zugänglich sein.
- Der Aufwand zur Entdeckung von Deadlocks muss vertretbar sein. (z.B. während der Ausführung des Erkennungsverfahrens dürfen keine Betriebsmittel angefordert oder freigegeben werden.)

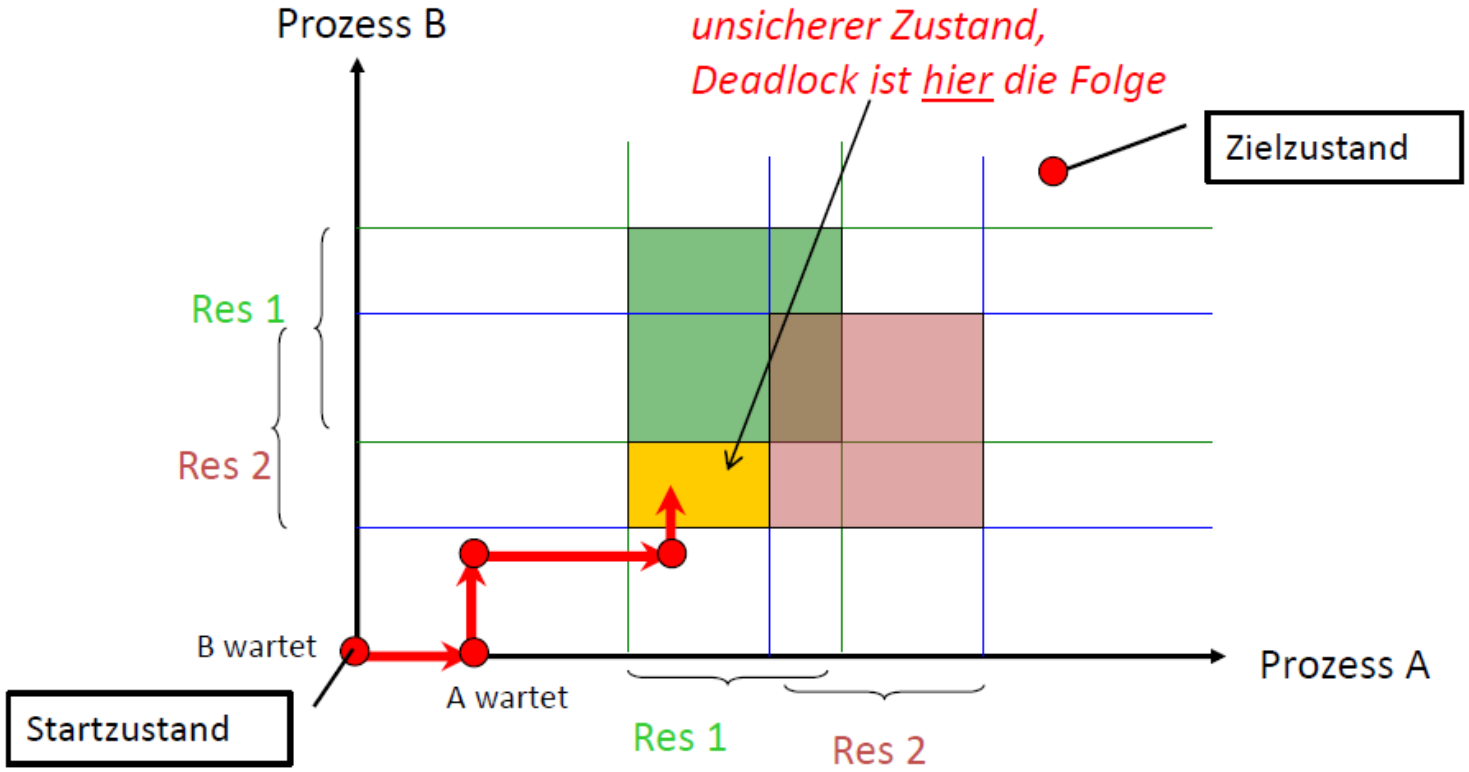
Deadlocks: Beheben

- Behebung durch **Unterbrechung**: Einem Prozess wird eine Ressource zeitweise entzogen, Zuteilung an blockierten Prozess.
  - Auswahl des Prozesses schwierig.
  - Nicht jede Ressource kann zeitweise entzogen werden.
- Behebung durch **teilweise Wiederholung** (Rollback):
  - Prozess schreibt seinen Zustand (belegte Ressourcen) regelmäßig (Checkpoints) in Log-Dateien.
  - Prozess, der einen Deadlock durch die Anforderung einer Ressource verursacht, wird auf einen Checkpoint zurückgesetzt, in dem er die Ressource noch nicht benötigt hat.
  - Arbeit hinter dem Checkpoint ist verloren.
- Behebung durch **Prozessabbruch** (und Neustart):
  - Am einfachsten und wirkungsvollsten (aber: am brutalsten).
  - Auswahl des zu beendenden Prozesses schwierig.
  - Kann zu Inkonsistenzen bei globalen Daten führen.

FH Bielefeld
University of
Applied Sciences

Deadlocks: Verhindern

- Besser als das Erkennen und Beheben von Deadlocks ist das **Verhindern**!
- Der wichtigste Algorithmus zur Verhinderung basiert auf dem Konzept der ‚**sicheren Zustände**'.



FH Bielefeld
University of
Applied Sciences

Deadlocks: Sichere Zustände

Definition:

- Ein Zustand ist ***sicher***, wenn er **kein Deadlock-Zustand** ist und es eine Scheduling-Reihenfolge gibt, die alle Prozesse zum Ende führt, selbst **wenn alle Prozesse gleichzeitig die maximal benötigten Ressourcen** anfordern.

Beispiel:

- 3 Prozesse A, B und C sowie eine Ressourcenklasse mit 10 Instanzen

	nutzt	max.
A	3	9
B	2	4
C	2	7

a) Frei: 3

	nutzt	max.
A	3	9
B	4	4
C	2	7

b) Frei: 1

	nutzt	max.
A	3	9
B	0	-
C	2	7

c) Frei: 5

	nutzt	max.
A	3	9
B	0	-
C	7	7

d) Frei: 0

	nutzt	max.
A	3	9
B	0	-
C	0	-

e) Frei: 7

Der Zustand a) ist sicher!

FH Bielefeld
University of
Applied Sciences

Deadlocks: Unsichere Zustände

- Beispiel wie oben (3 Prozesse A, B und C sowie eine Ressourcenklasse mit 10 Instanzen) bei **unsicherer** Ressourcen-Zuordnung:

	nutzt	max.
A	3	9
B	2	4
C	2	7

a) Frei: 3

	nutzt	max.
A	4	9
B	2	4
C	2	7

b) Frei: 2

	nutzt	max.
A	4	9
B	4	4
C	2	7

c) Frei: 0

	nutzt	max.
A	4	9
B	0	-
C	2	7

d) Frei: 4

Der Zustand b) ist unsicher!

FH Bielefeld
University of
Applied Sciences

Deadlocks: Sichere / unsichere Zustände

Bemerkungen:

- Ein Zustand ist **unsicher**, wenn
  - **kein Deadlock** vorliegt, und
  - es **eine Scheduling-Reihenfolge** gibt, die **zum Deadlock** führt, wenn **alle Prozesse sofort die maximale Anzahl** an benötigten Ressourcen anfordern.
- Ein ***unsicherer*** Zustand ist noch kein Deadlock-Zustand und muss auch **nicht zwangsläufig** zu einem Deadlock führen:
  - Prozesse müssen nicht alle simultan maximale Ressourcen-Anzahl anfordern.
  - Prozesse können u.U. im unsicheren Zustand auch wieder Ressourcen freigeben.
- Im Gegensatz zu einem ***sicheren*** Zustand kann aber nicht mehr garantiert werden, dass es eine Scheduling Reihenfolge gibt, die einen Deadlock vermeidet.

Ein Algorithmus zur Verhinderung von unsicheren Zuständen und damit Deadlocks ist der **Bankier-Algorithmus** (*bankers algorithm*) von Dijkstra (1965).

Deadlocks: Erkennen und Beheben

Annahmen über das System:

- Es gibt *n* Prozesse
- Es existieren *m* Ressourcenklassen mit jeweils E_j Ressourcen (Instanzen) in der Klasse *j* ($1 \leq j \leq m$)

Das **Betriebssystem überprüft zyklisch**, ob ein Deadlock vorliegt.

Dafür werden folgende Datenstrukturen benötigt:

- *Ressourcenvektor* **E**: Gibt an, wie viele Ressourcen E_i pro Klasse insgesamt existieren.
- *Ressourcenrestvektor* **A**: gibt an, wie viele freie Instanzen A_i pro Klasse existieren.
- *Belegungsmatrix* **C***:* C_{ij} gibt an, wie viele Instanzen der Prozess *i* von der Ressourcenklasse *j* belegt.
- *Anforderungsmatrix* **R**: R_{ij} gibt an, wie viele Instanzen der Prozess *i* von der Ressourcenklasse *j* angefordert hat (ohne sie zu bekommen).

Deadlocks: Erkennen und Beheben (nach Tanenbaum)

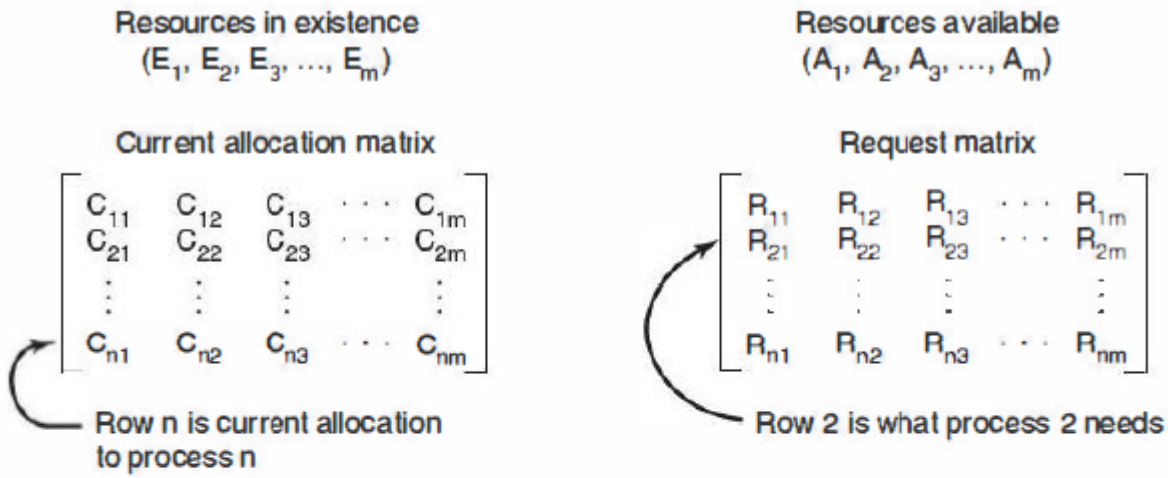


Figure 6-6. The four data structures needed by the deadlock detection algorithm.

An important invariant holds for these four data structures. In particular, every resource is either allocated or is available. This observation means that

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

FH Bielefeld
University of
Applied Sciences

Deadlocks: Algorithmus zur Erkennung

1. Zu Beginn sind **alle** Prozesse **unmarkiert**
2. Suche einen unmarkierten Prozess P_i, für den
   die i-te Reihe von R ≤ A ist (Komponentenvergleich)
3. Wenn es einen solchen Prozess gibt, dann
   addiere die i-te Zeile von C auf die i-te Zeile von A und gehe zu Schritt 2
4. Wenn es keinen solchen Prozess gibt, terminiert der Algorithmus

Wenn alle Prozesse markiert wurden, existiert **kein** Deadlock.

Bleibt ein unmarkierter Prozess übrig, liegt ein Deadlock vor.

FH Bielefeld
University of
Applied Sciences

Deadlocks: Beispiel zur Erkennung (nach Tanenbaum)

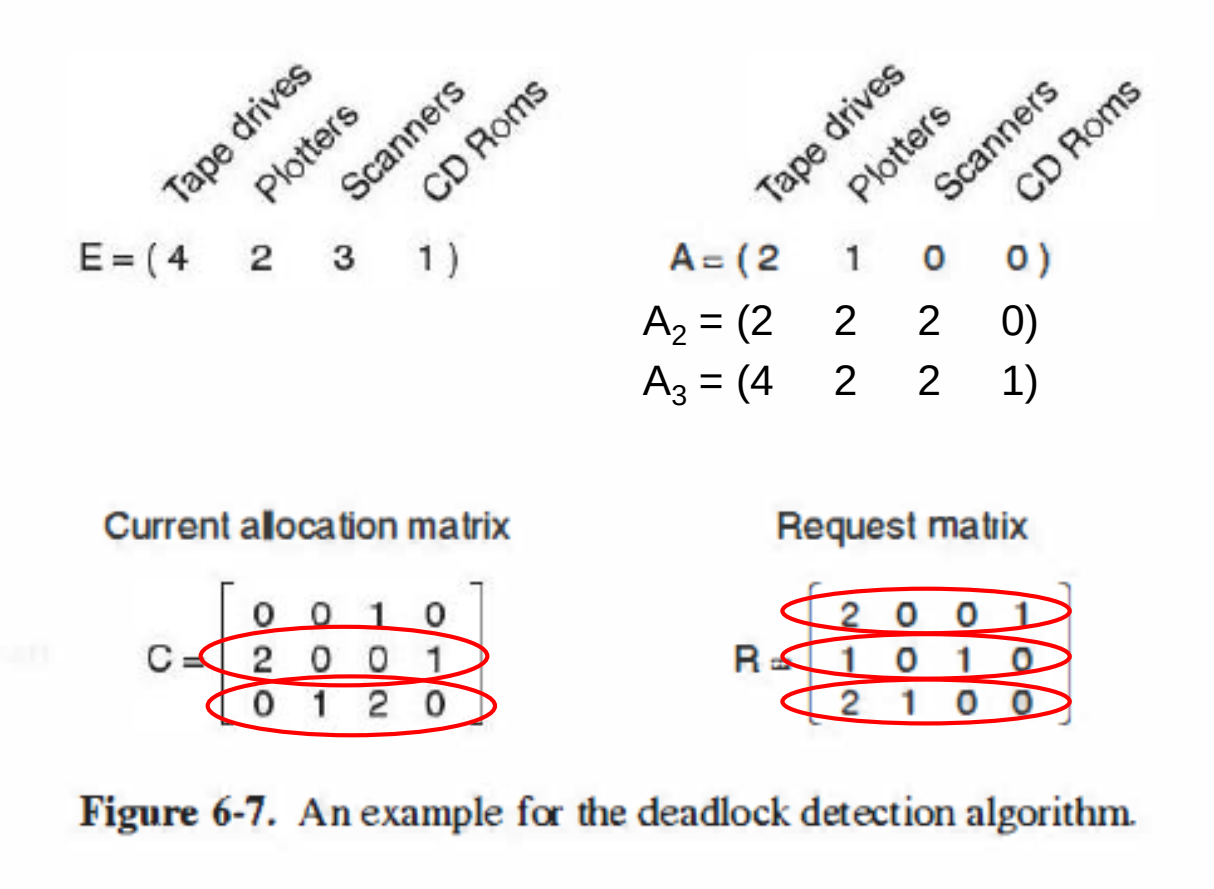


Figure 6-7. An example for the deadlock detection algorithm.

FH Bielefeld
University of
Applied Sciences

Deadlocks: Verhinderung

Bemerkungen:

- Der Bankier-Algorithmus zur Deadlock-*Verhinderung* setzt **Wissen** voraus, das in der Praxis **meist nicht gegeben** ist:
  - Max. Ressourcen eines Prozesses nicht bekannt
  - Prozesse werden dynamisch erzeugt
  - Ressourcen können auch verschwinden
  - …
- Daher kann eigentlich nur versucht werden, **Deadlocks prinzipiell unmöglich** zu machen!

Deadlocks: Bankier-Algorithmus (Dijkstra)

Der Algorithmus basiert auf dem Algorithmus zur **Deadlock Erkennung** und benutzt die gleichen Datenstrukturen. Allerdings: in **R** werden die **maximal zusätzlich** benötigten Instanzen eingetragen.

Wird eine Ressource angefordert oder freigegeben, so wird folgender Algorithmus durchlaufen:

1. Suche einen Prozess *i* (*i* = 1,..., *n*), für den gilt,
die i-te Zeile von R ≤ A

Wenn kein *i* gefunden wird, endet das System möglicherweise in einem Deadlock.

2. Nimm an, der Prozess *i* reserviert alle geforderten Ressourcen (was sicher geht) und wird beendet. Markiere ihn und addiere seine Ressourcen zur i-ten Zeile von A

3. Wiederhole Schritt 1 und 2 bis alle Prozesse markiert sind oder ein unsicherer Zustand auftritt. Im ersten Fall ist der Zustand sicher.

Wird bei der Anforderung einer Ressource ein **unsicherer Zustand** detektiert, dann muss die **Zuteilung der Ressource** verweigert werden.

FH Bielefeld
University of
Applied Sciences

Übungsaufgabe

- s. Handout

FH Bielefeld
University of
Applied Sciences

Inhalt

Letzte Vorlesung

- Interprozesskommunikation
  - Pipes, IPC, Shared Memory
- Mutexe und Semaphore
  - Schutz von kritischen Abschnitten

Heute

- Verklemmungen
  - Modellieren
  - Erkennen und Beheben
  - Verhindern (Banker Algorithm)

FH Bielefeld
University of
Applied Sciences

Vorlesung

Vielen Dank für ihre Aufmerksamkeit

Dozent

Prof. Dr.-Ing.
Martin Hoffmann
martin.hoffmann@fh-bielefeld.de