

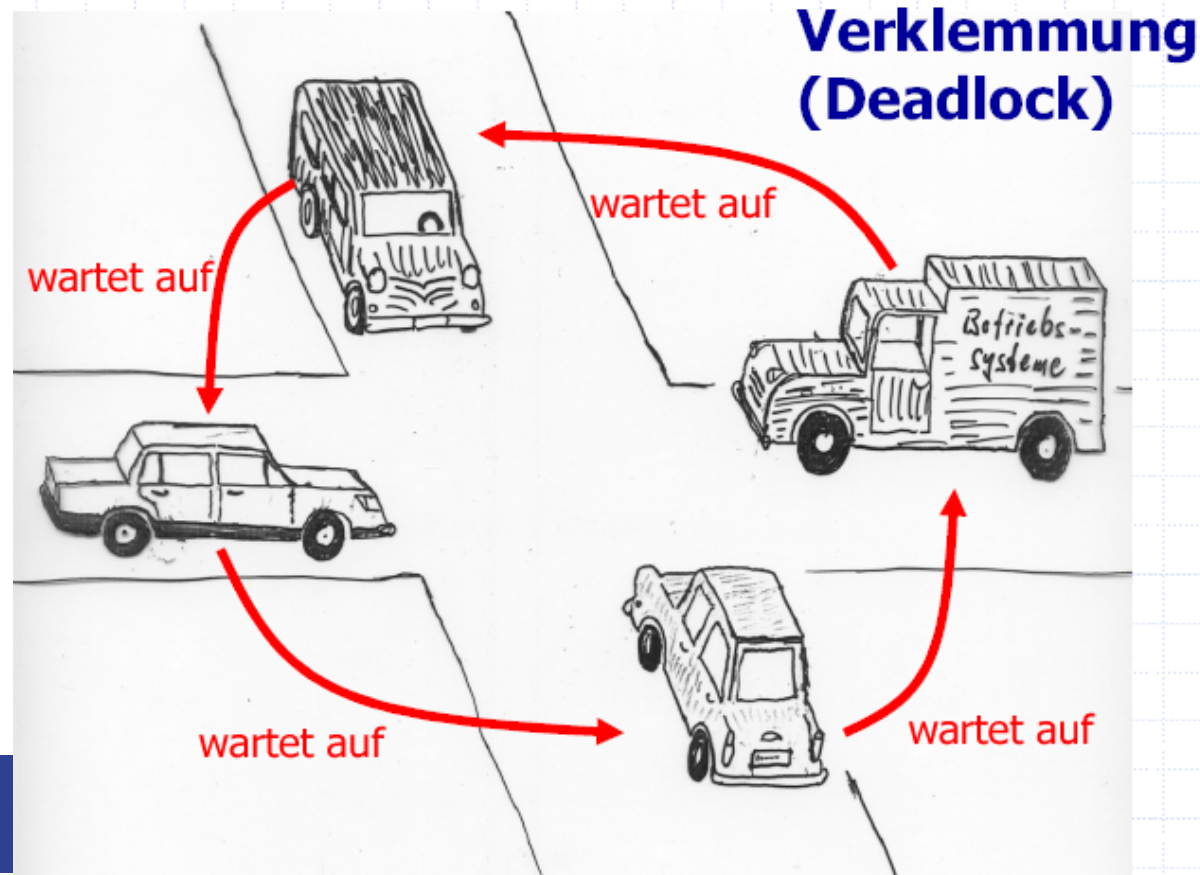
Betriebssysteme

Deadlocks

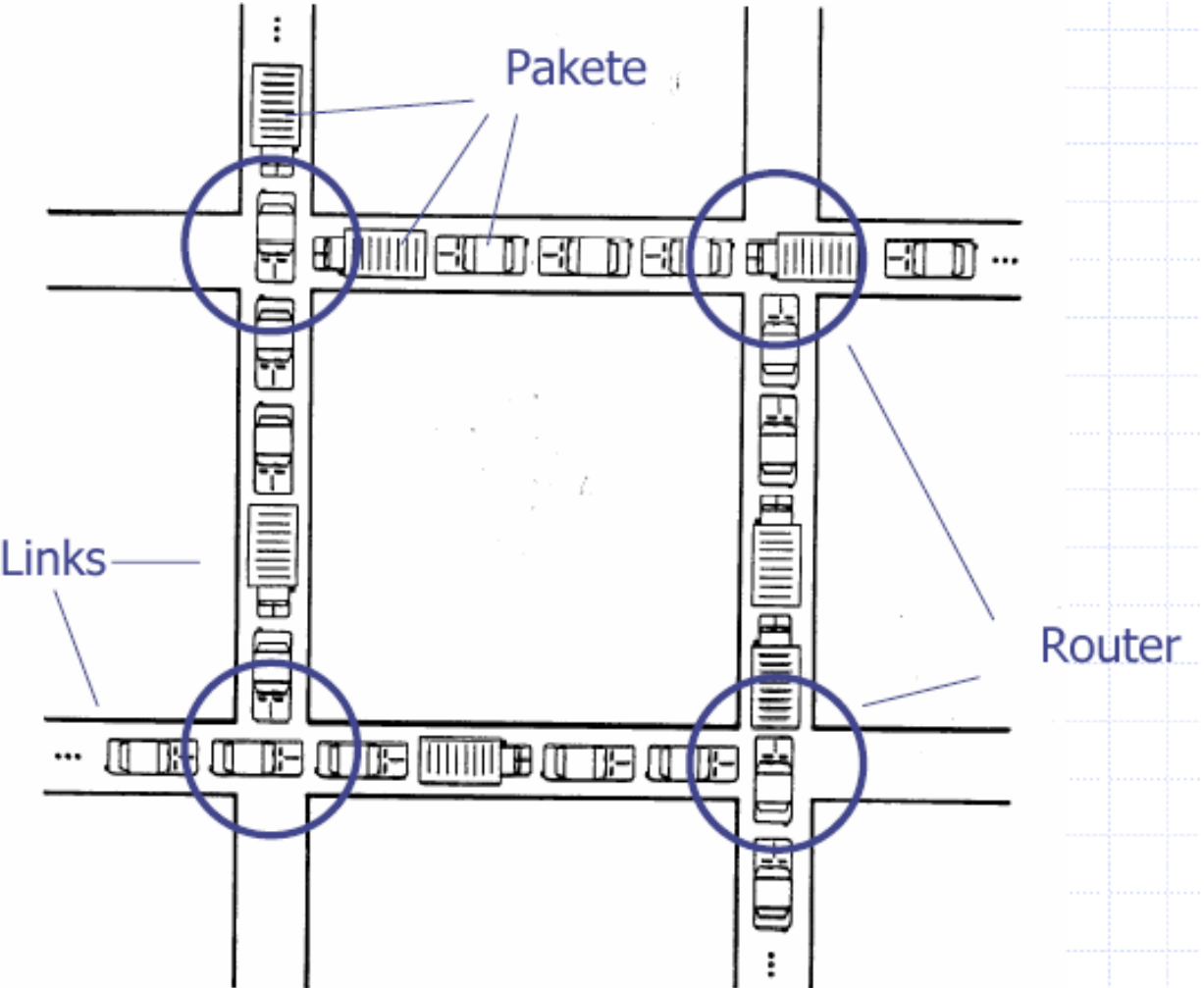
Prof. Dr.-Ing. Torben Weis
Universität Duisburg-Essen

Deadlocks und Ressourcen

- Konkurrierende Belegung von Ressourcen kann zu einer Verklemmung (Deadlock) der Prozesse führen



Deadlocks in Netzwerken ?



Ressourcen

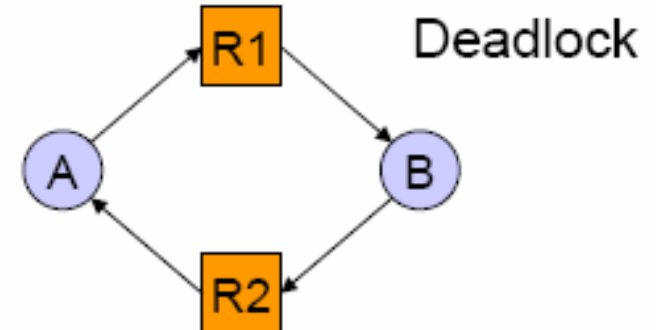
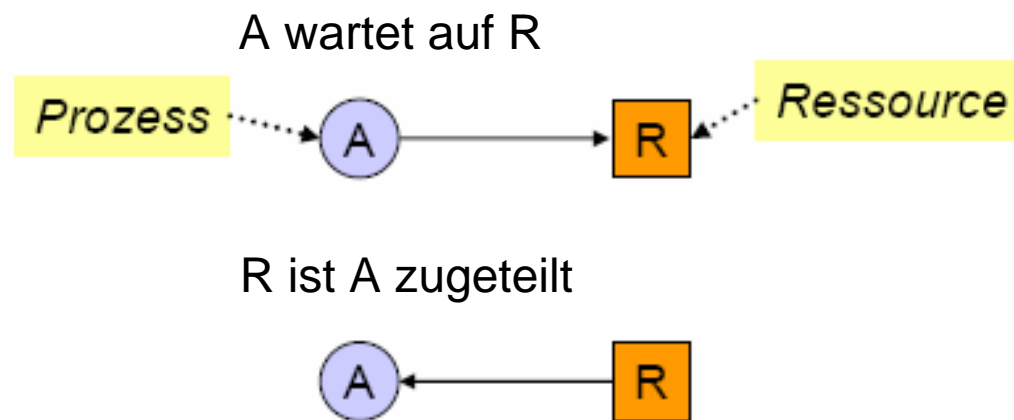
- Ressourcen (Betriebsmittel)
 - Unterbrechbar (*preemptable*)
 - Nicht unterbrechbar (*non-preemptable*)
 - Exklusiver Zugriff (*mutual exclusion*)
 - Gemeinsamer Zugriff (*shared access*) Beispiel?
- Benutzung einer Ressource
 1. Anfordern
 - Der Prozess blockiert, bis er die Ressource bekommt
 2. Belegen
 - Der Prozess benutzt die Ressource
 3. Freigeben
 - Das BS bekommt die Kontrolle über die Ressource zurück

Deadlock

- Definition
 - „Eine Menge von Prozessen ist im Deadlock, wenn jeder Prozeß auf ein Ereignis wartet, das nur ein anderes Mitglied der Menge verursachen kann.“
- Vier notwendige Bedingungen
 1. Wechselseitiger Ausschluß (*mutual exclusion*)
 2. Halten-und-Warten (*hold and wait*)
 3. Nicht unterbrechbar (*non-preemptable*)
 4. Zirkuläre Wartesituation (*circular wait*)
- Alle vier Bedingungen müssen erfüllt sein, damit es zu Deadlocks kommen kann

Wartegraph

- Möglichkeiten / Aufgaben des Betriebssystems
 1. Vorbeugung (*prevention*)
 2. Vermeidung (*avoidance*)
 3. Erkennung (*detection*)
 4. Auflösung (*recovery*)

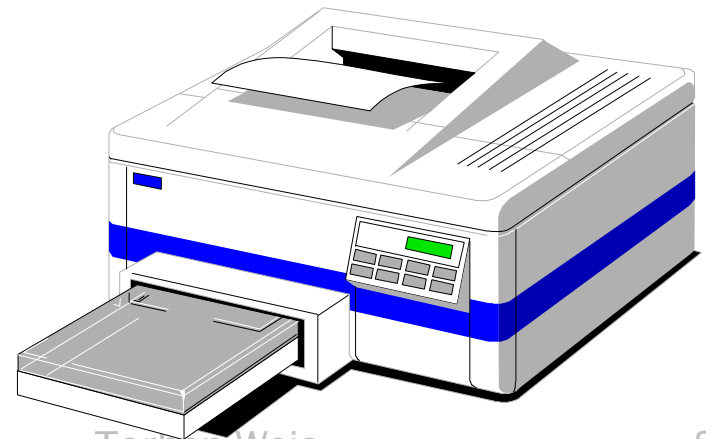


Vorbeugung

- Durchbrechen einer der vier Deadlock-Bedingungen
 - Deadlocks werden prinzipiell unmöglich
- 1. Exklusiver Zugriff
 - Ressource gemeinsam benutzbar (*sharable*) machen
- 2. Halten-und-Warten
 - alle Res. müssen vor Start angefordert werden & belegt sein
- 3. Nicht-Unterbrechbarkeit
 - Ressource entziehen und anderem Prozeß geben
- 4. Zirkuläres Warten
 - Reihenfolge der Ressourcenbelegung einschränken

Vorbeugung

- Kein Exklusiver Zugriff
- Beispiel Drucker
 - Prozesse drucken in den Spooler
 - Spooler-Prozess ist der einzige, der direkt mit dem Drucker redet
 - Der Spooler kann parallel durch mehrere Prozesse genutzt werden
- Haken an obigem Beispiel
 - Zwei Prozesse drucken große Dateien
 - In der Mitte ist die Spooling-Platte voll
 - Wir haben einen Deadlock ☹️



Vorbeugung

- Halten und Warten verhindern
- Idee 1:
 - Alle Ressourcen auf einmal anfordern, idealerweise bei Programmstart
 - Leider unrealistisch
- Idee 2:
 - Kein Warten zulassen
 - Wenn Ressource nicht sofort verfügbar ist, dann mit einer Fehlermeldung abbrechen
 - Das ist die Semantik vieler BS Funktionen

Vorbeugung

- Unterbrechbarkeit herstellen
- Keine allgemein gangbare Möglichkeit
 - Beispiel Drucker: Unterbrechung ist physikalisch unmöglich
- Drastische Methode
 - Den Prozess komplett unterbrechen
 - Zurücksetzen auf einen sicheren Checkpoint
 - Seit dem Checkpoint allokierte Ressourcen freigeben
 - Leider oft nicht möglich, da der Prozess bereits Seiteneffekte produziert hat
 - Bei Datenbanken hingegen ist dergleichen möglich

Vorbeugung

- Keine Zyklische Wartebedingung
- Methode 1
 - Ein Prozeß belegt immer nur eine Ressource
 - Damit sind Zyklen im Graph ausgeschlossen
 - Leider sehr starke Einschränkung
- Methode 2
 - Nummeriere alle Ressourcen 1 ... M
 - Anforderungen nur in numerisch aufsteigender Reihenfolge
 - Das Resultat ist auch hier ein azyklischer Graph
 - Wie soll man eine allgemeine und sinnvolle Nummerierung finden?
 - Kennt man alle Ressourcen a priori?

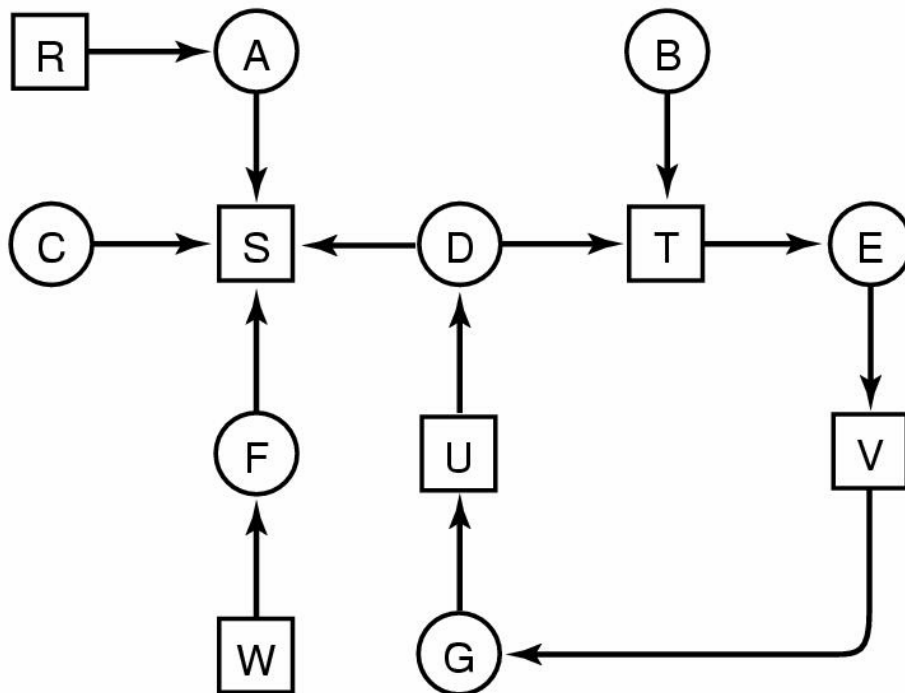
Deadlocks Erkennen

- Mangels Vorbeugung und Vermeidung lassen wir Deadlocks zu
 - Wir wollen sie aber erkennen
 - Danach kann man sich um Auflösung des Deadlocks Gedanken machen
- Grundannahmen
 - Wir kennen alle Prozesse
 - Wir kennen alle Ressourcen
 - Wir wissen, welcher Prozess welche Ressourcen hält
 - Wir wissen, welcher Prozess auf welche Ressource wartet
 - Wir wissen, wie viele Ressourcen es von jeder Sorte gibt

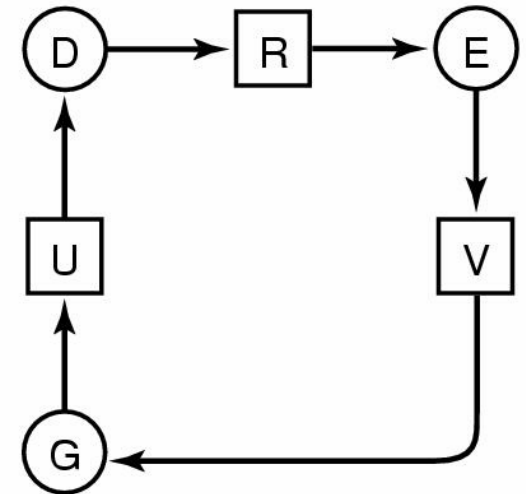
Deadlock-Erkennung bei einer Ressource pro Typ

- Algorithmus
 1. Konstruiere den (gerichteten) Wartegraphen
 - Ist das realistischer Weise möglich ?
 2. Suche Zyklen im Wartegraphen
 - Von jedem Knoten ausgehend starten wir eine Tiefensuche. Sieht man einen Knoten 2 mal, dann hat man einen Zyklus gefunden
 3. Zyklus gefunden \rightarrow Alle Prozesse und Ressourcen des Zyklus sind im Deadlock
 - Nach Auflösen eines Zyklus können immer noch weitere Zyklen bestehen
 - Man muss den Algorithmus solange wiederholen, bis keine Zyklen mehr gefunden werden

Deadlock-Erkennung bei einer Ressource pro Typ



(a)



(b)

Deadlock-Erkennung bei mehreren Ressourcen pro Typ

Resources in existence
($E_1, E_2, E_3, \dots, E_m$)

Resources available
($A_1, A_2, A_3, \dots, A_m$)

Current allocation matrix

Request matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row n is current allocation
to process n

Row 2 is what process 2 needs

Resources in existence
($E_1, E_2, E_3, \dots, E_m$)

Resources available
($A_1, A_2, A_3, \dots, A_m$)

Current allocation matrix

C_{11}	C_{12}	C_{13}	\dots	C_{1m}
C_{21}	C_{22}	C_{23}	\dots	C_{2m}
\vdots	\vdots	\vdots		\vdots
C_{n1}	C_{n2}	C_{n3}	\dots	C_{nm}

Row n is current allocation
to process n

Request matrix

R_{11}	R_{12}	R_{13}	\dots	R_{1m}
R_{21}	R_{22}	R_{23}	\dots	R_{2m}
\vdots	\vdots	\vdots		\vdots
R_{n1}	R_{n2}	R_{n3}	\dots	R_{nm}

Row 2 is what process 2 needs

- Jede Ressource E_i ist entweder in A_i (also frei) oder in einem C_i verbucht (also belegt)

$$C_{1,j} + \dots + C_{n,j} + A_j = E_j$$

- Ein Prozess k ist ausführbar, wenn alle $R_{k,i} < A_i$
- Starte alle ausführbaren Prozesse und addiere deren belegte Ressourcen zu den freien: $A_i += C_{k,i}$

Deadlock-Erkennung bei mehreren Ressourcen pro Typ

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Tape drives
Plotters
Scanners
CD Roms

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

Tape drives
Plotters
Scanners
CD Roms

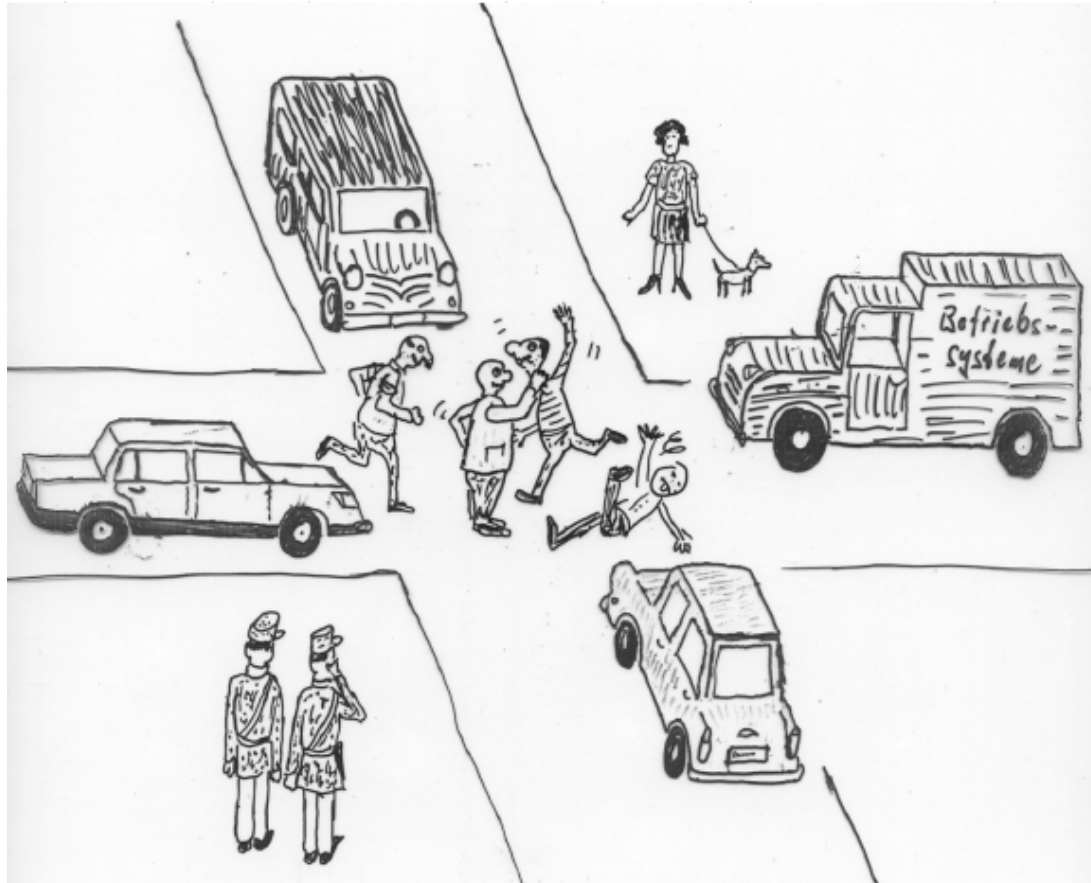
Current allocation matrix

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

Request matrix

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

Deadlock Auflösung



Deadlock Auflösung

- Behebung durch Unterbechung
 - Einem Prozess wird eine Ressource entzogen
 - Der Prozess erfährt das durch einen Fehlercode, wenn er die Ressource wieder benutzen möchte (d.h. einen Systemaufruf tätigt, der nicht erfolgreich ist)
- Behebung durch Wiederholung (Rollback)
 - Das BS setzt den Prozess auf einen Checkpoint zurück, an dem er die Ressource noch nicht hatte
- Behebung durch Prozessabbruch
 - Der Prozess wird „abgeschossen“ und die Ressource damit frei

Deadlock Vermeidung

- Nachteile von Vorbeugung
 - Ärgerliche Einschränkungen und daher keine allgemeingültige Lösung
- Nachteile von Erkennen & Auflösen
 - Abbrechen, Zurücksetzen oder Entzug von Ressourcen ist ein drastischer Eingriff
 - Teils unabsehbaren Nebenwirkungen
- Gibt es eine allgemeine Lösung, die nicht (oder zumindest weniger) einschränkt und dennoch eingreift bevor ein Deadlock entsteht?
 - Idee: Wir prüfen vor jeder Zuteilung einer Ressource, ob wir uns damit einen potentiellen Deadlock einhandeln

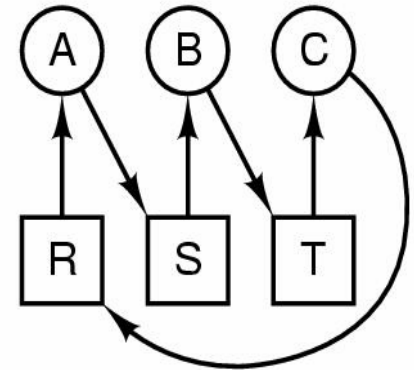
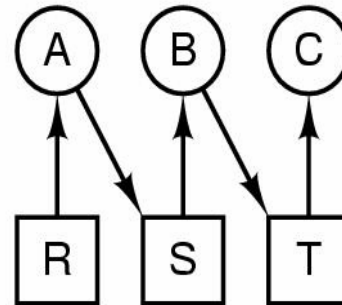
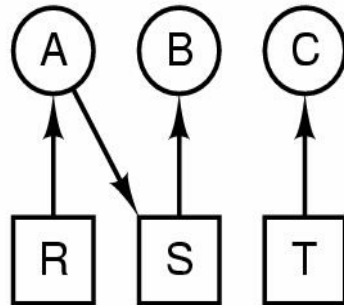
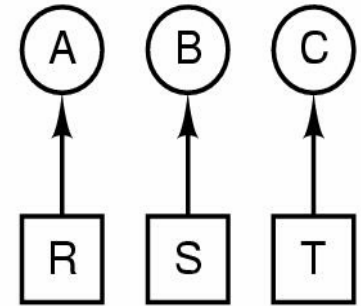
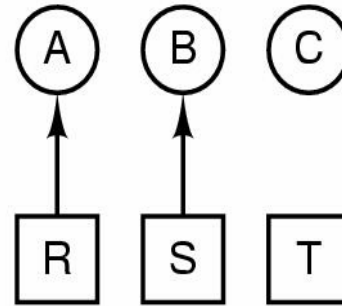
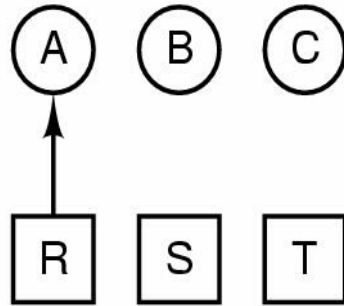
Deadlock Vermeidung

- Grundidee
 - Lasse nie zu, dass kein Prozess zu Ende laufen kann
 - Anders ausgedrückt: Mindestens ein Prozess muss sein Maximum an gewünschten Ressourcen erhalten können
- Annahmen
 - Wir müssen wissen, wie viele Ressourcen jeder Prozess maximal irgendwann einmal verlangen wird
 - Jeder Prozess läuft zu Ende, wenn man ihm alle Ressourcenwünsche erfüllt
 - Vergleiche Deadlock Erkennung: Wir müssen nur wissen, auf wie viele Ressourcen jeder Prozess gerade wartet

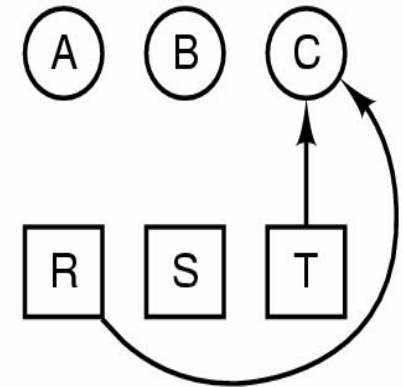
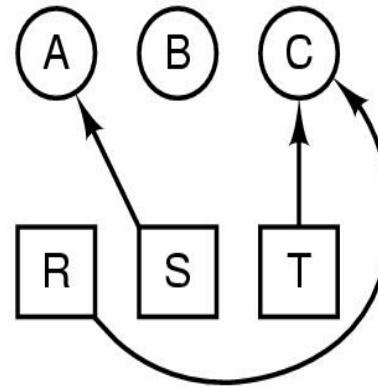
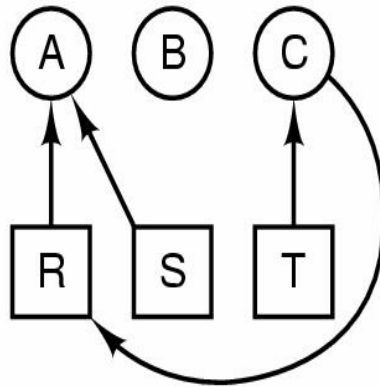
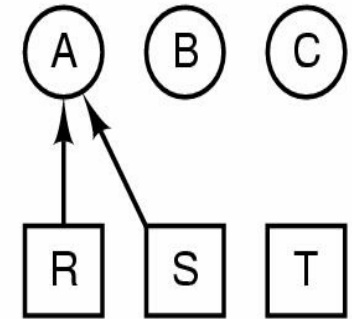
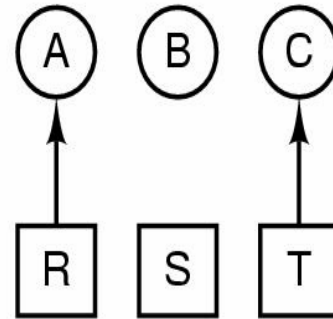
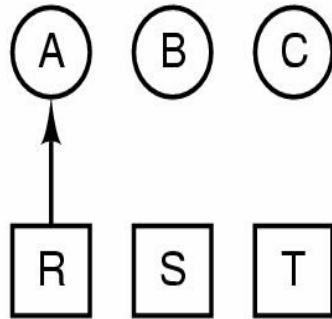
Deadlock Vermeidung

- „Die Reihenfolge macht ´s“
 - Einige Ressourcenanforderungen müssen wir zurückhalten. Der Prozess muss dann warten
 - Andere können wir sofort bedienen
- Wir teilen Ressourcen nur dann zu wenn
 - a) die Zuteilung „ungefährlich“ ist, also noch mindestens ein Prozess fertig werden kann
 - b) Der Prozess nach der Zuteilung fertig laufen kann

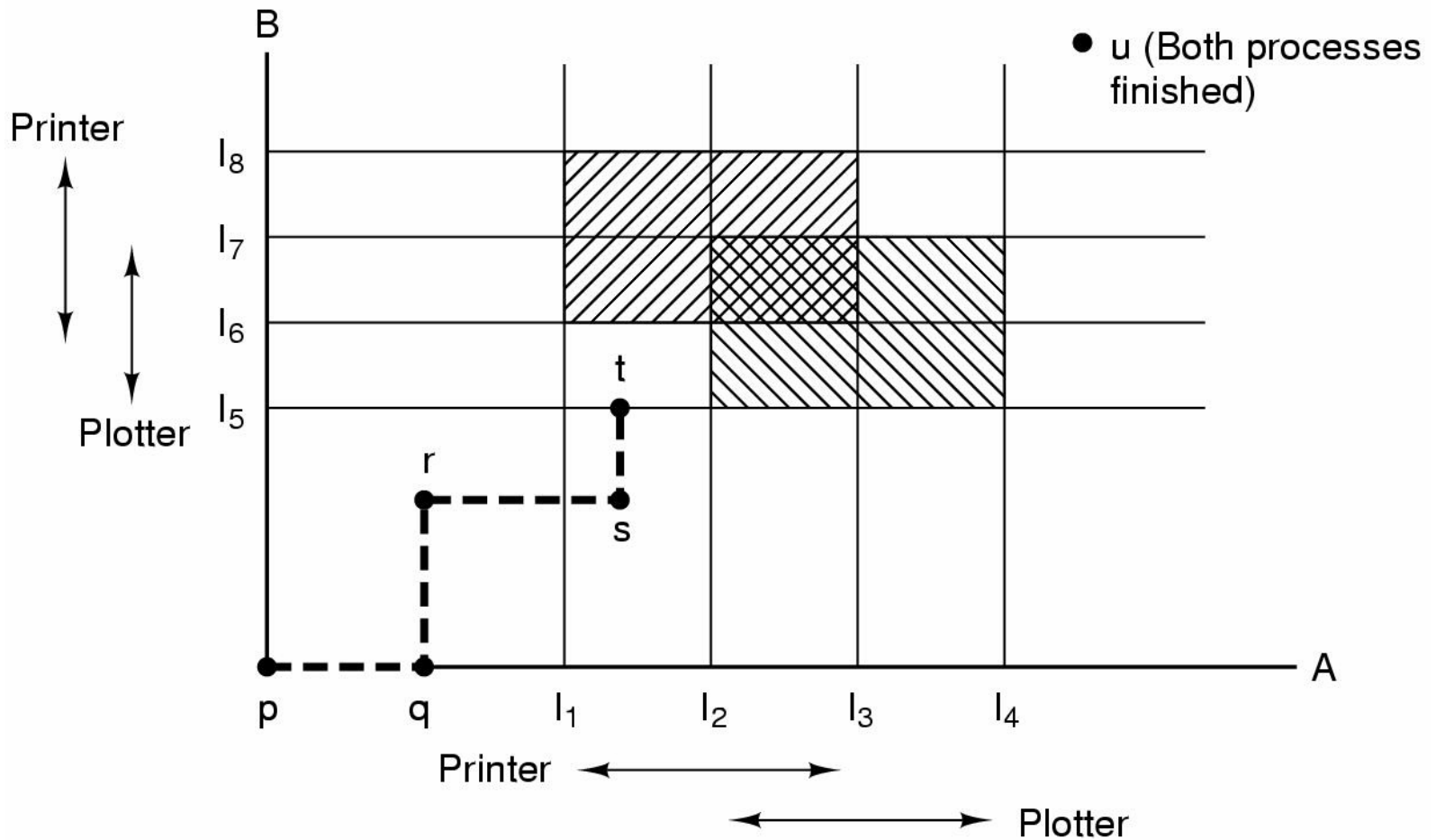
1. A requests R
 2. B requests S
 3. C requests T
 4. A requests S
 5. B requests T
 6. C requests R
- deadlock



1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
no deadlock



Ressourcenspur



Sichere Zustände

	Has	Max		Has	Max		Has	Max		Has	Max		Has	Max
A	3	9	A	3	9	A	3	9	A	3	9	A	3	9
B	2	4	B	4	4	B	0	-	B	0	-	B	0	-
C	2	7	C	2	7	C	2	7	C	7	7	C	0	-
	Free: 3			Free: 1			Free: 5			Free: 0			Free: 7	
	(a)			(b)			(c)			(d)			(e)	

- Der Zustand (a) ist sicher
- Es gibt eine Ausführungsreihenfolge mit der Eigenschaft
 - Alle Prozesse terminieren erfolgreich
 - Jeder Prozess könnte sein Maximum an Ressourcen voll ausschöpfen

Unsichere Zustände

Has Max

A	3	9
B	2	4
C	2	7

Free: 3

(a)

Has Max

A	4	9
B	2	4
C	2	7

Free: 2

(b)

Has Max

A	4	9
B	4	4
C	2	7

Free: 0

(c)

Has Max

A	4	9
B	—	—
C	2	7

Free: 4

(d)

- Zustand (a) ist noch sicher
 - Durch Zuteilung einer Ressource an A gibt es Probleme
- Der Zustand (b) ist nicht sicher
 - Nur Prozess B kann erfolgreich terminieren
 - Danach kann weder A noch C sein Maximum an Ressourcen ausschöpfen

Bankier-Algorithmus für einen Ressourcen Typ

- Vor jeder Zuteilung prüfe
 - Ist der Zustand nach der Zuteilung sicher?
Dann teile die Ressource zu
 - Ansonsten muss der Prozess warten
- Prüf-Algorithmus
 - Es gibt immer einen Prozess, der mit den verfügbaren Ressourcenanzahl fertig laufen kann
 - Entferne ihn aus der Liste. Alle ihm zugeteilten Ressourcen werden wieder verfügbar
 - Wiederhole bis
 - a) Alle Prozesse aus der Liste entfernt sind → sicher
 - b) Kein Prozess mehr entfernt werden kann → unsicher

Beispiel für den Bankier-Algorithmus mit einem Ressourcen Typ

Has Max

A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

Has Max

A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

Has Max

A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

- Zustand (a) ist der Initialzustand und sicher
- Zustand (b) ist sicher
 - Danach wird C eine Ressource zugeteilt
- Zustand (c) ist nicht sicher

Bankier-Algorithmus für mehrere Ressourcen Typen

- Mathematisch etwas komplexer
 - 2 Matrizen: Jeder Prozess hat eine Zeile und jede Ressource eine Spalte
 - Matrix C: Die belegten Ressourcen
 - Matrix R: Die maximal noch benötigten Ressourcen
- Konzeptionell gleich
 - Suche eine Zeile k in R, so dass alle benötigten Ressourcen verfügbar sind
 - Gib die Ressourcen aus Zeile k in C frei und entferne Zeile k aus R und C
 - Wiederhole bis
 - a) Alle Prozesse (d.h. Zeilen) aus R entfernt sind → sicher
 - b) Kein Prozess mehr entfernt werden kann → unsicher

Beispiel für den Bankier-Algorithmus mit mehreren Ressourcen Typen

	Process	Tape drives	Plotters	Scanners	CD ROMs
A	3	0	1	1	
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	0	

Resources assigned

	Process	Tape drives	Plotters	Scanners	CD ROMs
A	1	1	0	0	
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	

Resources still needed

E = (6342)
 P = (5322)
 A = (1020)

Kritik am Bankier-Algorithmus

- Man muss zuviel a priori wissen
 - Wie viele und welche Ressourcen?
 - USB Geräte „kommen und gehen“
 - Prozesse laufen zu Ende, wenn sie nur genügend Ressourcen bekommen
 - Nicht zutreffend für interaktive Anwendungen
- Der Algorithmus ist sehr konservativ
 - Er hält Ressourcen zurück, für den Fall, dass alle Prozesse
 - a) Das Maximum an Ressourcen ausschöpfen wollen
 - b) Kein Prozess eine Ressource vorzeitig frei gibt