

# 6 Deadlocks

- 1 Einführung
- 2 Prozesse und Threads
- 3 Speicherverwaltung
- 4 Dateisysteme
- 5 Eingabe und Ausgabe
- 6 Deadlocks**
- 7 Virtualisierung und die Cloud
- 8 Multiprozessorsysteme
- 9 IT-Sicherheit
- 10 Fallstudie 1: Linux
- 11 Fallstudie 2: Windows
- 12 Entwurf von Betriebssystemen

# 6 Deadlocks

6.1 Ressourcen

6.2 Einführung in Deadlocks

6.3 Der Vogel-Strauß-Algorithmus

6.4 Erkennen und beheben von Deadlocks

6.5 Verhinderung von Deadlocks (Avoidance)

6.6 Vermeidung con Deadlocks (Prevention)

6.7 Weitere Themen zu Deadlocks

# **6.1 Ressourcen**

**6.1.1 Unterbrechbare und nicht unterbrechbare  
Ressourcen**

**6.1.2 Ressourcenanforderung**

# **Unterbrechbare und nicht unterbrechbare Ressourcen**

**Notwendige Ereignisse für die Verwendung einer Ressource:**

- 1.** Ressource anfordern.
- 2.** Ressource verwenden.
- 3.** Ressource freigeben.

# Ressourcenanforderung (1)

```
typedef int semaphore;  
semaphore resource_1;  
  
void process_A(void) {  
    down(&resource_1);  
    use_resource_1( );  
    up(&resource_1);  
}
```

a

```
typedef int semaphore;  
semaphore resource_1;  
semaphore resource_2;  
void process_A(void) {  
    down(&resource_1);  
    down(&resource_2);  
    use_both_resources( );  
    up(&resource_2);  
    up(&resource_1);  
}
```

b

**Abbildung 6.1:** Jede Ressource wird durch ein Semaphor geschützt: (a) eine Ressource; (b) zwei Ressourcen.

# Ressourcenanforderung (2)

```
typedef int semaphore;  
semaphore resource_1;  
semaphore resource_2;  
  
void process_A(void) {  
    down(&resource_1);  
    down(&resource_2);  
    use_both_resources( );  
    up(&resource_2);  
    up(&resource_1);  
}  
  
void process_B(void) {  
    down(&resource_1);  
    down(&resource_2);  
    use_both_resources( );  
    up(&resource_2);  
    up(&resource_1);  
}
```

a

```
semaphore resource_1;  
semaphore resource_2;  
  
void process_A(void) {  
    down(&resource_1);  
    down(&resource_2);  
    use_both_resources( );  
    up(&resource_2);  
    up(&resource_1);  
}  
  
void process_B(void) {  
    down(&resource_2);  
    down(&resource_1);  
    use_both_resources( );  
    up(&resource_1);  
    up(&resource_2);  
}
```

b

**Abbildung 6.2:** (a) Deadlock-freier Code. (b) Code mit einem möglichen Deadlock.

## 6.2 Einführung in Deadlocks

6.2.1 Voraussetzungen für Ressourcen-Deadlocks

6.2.2 Modellierung von Deadlocks

# Definition Deadlock

**Eine Menge von Prozessen ist blockiert, wenn:**

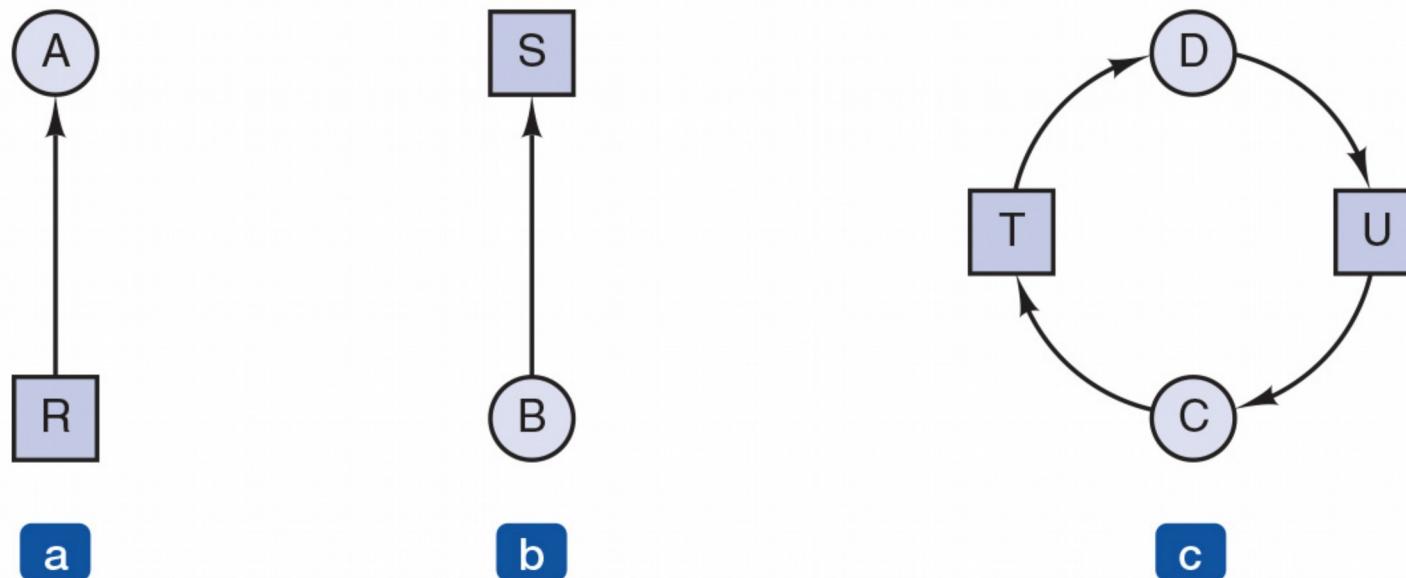
- Jeder Prozess in der Menge wartet auf ein Ereignis
- Dieses Ereignis kann nur durch einen anderen Prozess verursacht werden

# **Bedingungen für Ressourcen-Deadlocks**

**Vier Bedingungen, die erfüllt sein müssen:**

- 1.** Bedingung des wechselseitigen Ausschlusses
- 2.** Hold-and-Wait Bedingung
- 3.** Bedingung der Ununterbrechbarkeit
- 4.** Zyklischen Wartebedingung

# Modellierung von Deadlocks (1)



**Abbildung 6.3:** Ressourcen-Belegungsgraphen: (a) Belegung einer Ressource; (b) Anforderung einer Ressource; (c) Deadlock.

Tanenbaum, A. S.; Bos, H.: Moderne  
Betriebssysteme. Pearson Studium 2016

# Modellierung von Deadlocks (2)

A  
Anfrage R  
Anfrage S  
Freigabe R  
Freigabe S

a

B  
Anfrage S  
Anfrage T  
Freigabe S  
Freigabe T

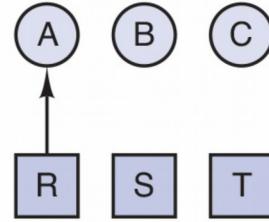
b

C  
Anfrage T  
Anfrage R  
Freigabe T  
Freigabe R

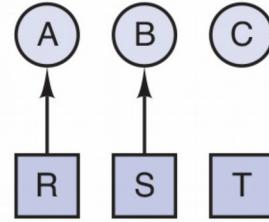
c

1. A verlangt R
  2. B verlangt S
  3. C verlangt T
  4. A verlangt S
  5. B verlangt T
  6. C verlangt R
- Deadlock

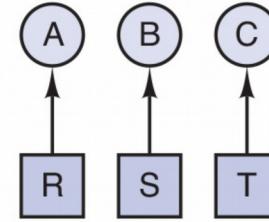
d



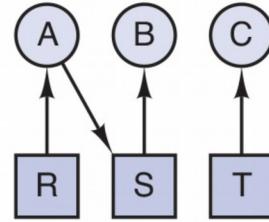
e



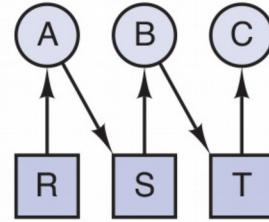
f



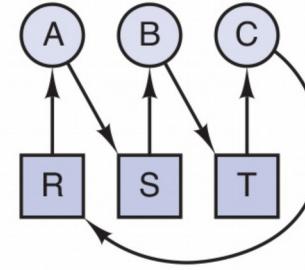
g



h



i



j

# Modellierung von Deadlocks (3)

1. A verlangt R
  2. C verlangt T
  3. A verlangt S
  4. C verlangt R
  5. A gibt R frei
  6. A gibt S frei
- kein Deadlock

k

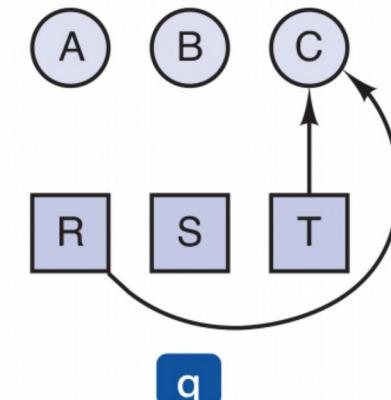
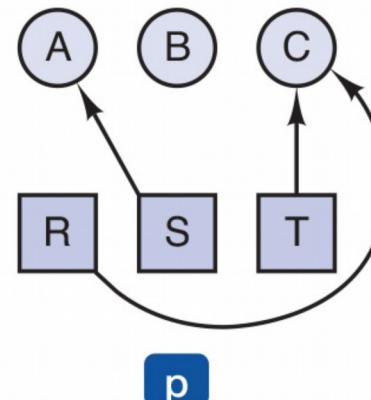
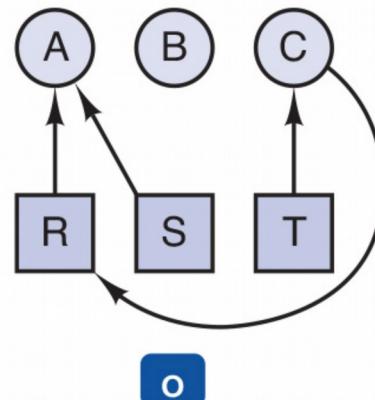
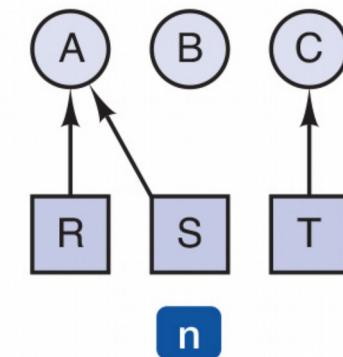
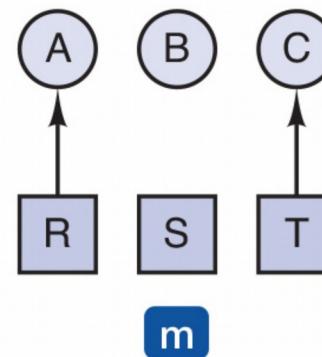
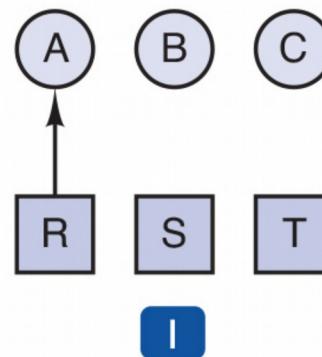


Abbildung 6.4: Beispiel, wie ein Deadlock entstehen kann und wie er sich verhindern lässt.

# Modellierung von Deadlocks (4)

Strategien für die Behandlung von Deadlocks:

1. Ignoriere das Problem, vielleicht geht es weg.\*
2. Erkennung und Beheben. Deadlocks zulassen, erkennen und etwas dagegen unternehmen.
3. Dynamische Verhinderung durch sorgfältige Ressourcenzuteilung.
4. Vermeidung von Deadlocks. Eine der vier notwendigen Bedingungen muss prinzipiell unerfüllbar werden.

\* Vogel-Strauß-Algorithmus

## 6.3 Der Vogel-Strauß-Algorithmus

Mehrere Prozesse kommen auf Grund zyklischer Bedingungen zwischen Ressourcen in einen Deadlock.

Der Vogel-Strauß-Algorithmus stellt hier den einfachsten Ansatz dar: Die Tatsache, dass ein Deadlock aufgetreten ist, wird einfach ignoriert:

- dieser Deadlock kann nicht aufgelöst werden
- Prozesse warten unendlich auf die Ressourcen oder werden „von außen“ aus dem Zustand befreit (Abbruch).

## 6.4 Erkennen und Beheben von Deadlocks

6.4.1 Deadlock-Erkennung bei einer Ressource je Typ

6.4.2 Deadlock-Erkennung bei mehreren Ressourcen

je Typ

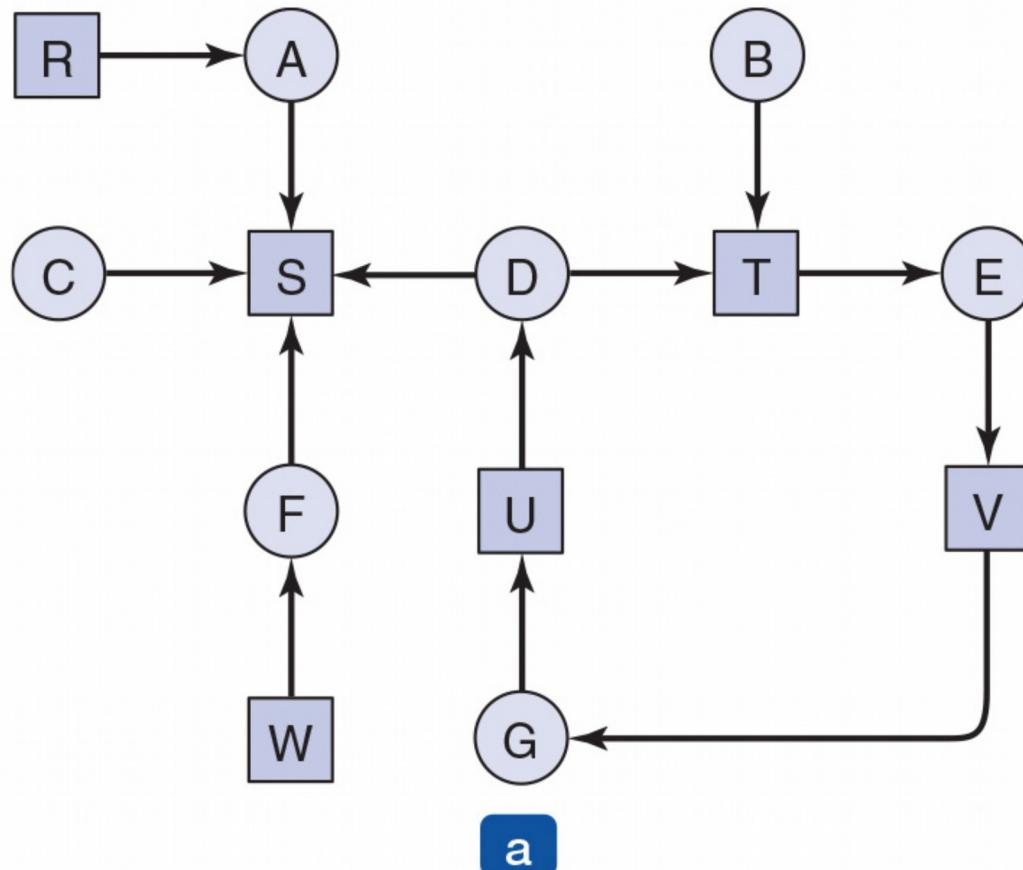
6.4.3 Beheben von Deadlocks

# Deadlock-Erkennung bei einer Ressource je Typ (1)

Beispiel für ein System - ist es blockiert?

1. Prozess A belegt R, verlangt S
2. Prozess B belegt nichts, verlangt T
3. Prozess C belegt nichts, verlangt S
4. Prozess D belegt U, verlangt S und T
5. Prozess E belegt T, verlangt V
6. Prozess F belegt W, verlangt S
7. Prozess G belegt V, verlangt U

# Deadlock-Erkennung bei einer Ressource je Typ (2)



**Abbildung 6.5:** (a) Ressourcen-Belegungsgraph. (b) Ein Zyklus aus (a).

Tanenbaum, A. S.; Bos, H.: Moderne Betriebssysteme. Pearson Studium 2016

# Algorithmus zur Erkennung von Deadlocks (1)

1. Für alle Knoten  $N$  im Graphen führe die folgenden Schritte aus. Benutze  $N$  als Startknoten.
2. Initialisiere  $L$  mit der leeren Liste und kennzeichne alle Kanten als „nicht markiert“.
3. Hänge den aktuellen Knoten an das Ende von  $L$  und überprüfe, ob er in  $L$  zweimal vorkommt. Wenn ja, enthält  $L$  den gesuchten Zyklus und der Algorithmus terminiert.

# Algorithmus zur Erkennung von Deadlocks (2)

4. Überprüfe, ob vom aktuellen Knoten unmarkierte Kanten wegführen. Wenn ja, gehe zu Schritt 5, sonst zu Schritt 6.
5. Wähle zufällig eine wegführende Kante und markiere sie. Folge der Kante zum neuen aktuellen Knoten und fahre mit Schritt 3 fort.
6. Wenn der aktuelle Knoten der Startknoten ist, enthält der Graph keine Zyklen und der Algorithmus terminiert. Sonst sind wir in einer Sackgasse. Lösche den aktuellen Knoten aus  $L$  und gehe zurück zum vorherigen Knoten.

# Deadlock-Erkennung bei mehreren Ressourcen je Typ (1)

Ressourcenvektor  
( $E_1, E_2, E_3, \dots, E_m$ )

aktuelle Belegungsmatrix

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

Reihe n gibt die aktuelle Belegung für Prozess n an

Ressourcenvektor  
( $A_1, A_2, A_3, \dots, A_m$ )

Anforderungsmatrix

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

Reihe 2 zeigt, was Prozess 2 benötigt

**Abbildung 6.6:** Die vier Datenstrukturen für den Deadlock-Erkennungsalgorithmus.

# Deadlock-Erkennung bei mehreren Ressourcen je Typ (2)

## Deadlock-Erkennungsalgoritmus:

1. Suche nach einem unmarkierten Prozess  $P_i$ , für den die i-te Reihe von  $R$  kleiner oder gleich A ist.
2. Wenn ein solcher Prozess gefunden wird, füge die i-te Zeile von C zu A hinzu, markiere den Prozess und gehe zurück zu Schritt 1.
3. Wenn kein solcher Prozess existiert, wird der Algorithmus terminiert.

# Deadlock-Erkennung bei mehreren Ressourcen je Typ (3)

	Bandlaufwerke	Plotter	Scanner	Blu-ray
E =	4	2	3	1
A =	2	1	0	0
C =	0	0	1	0
	2	0	0	1
	0	1	2	0
R =	2	0	0	1
	1	0	1	0
	2	1	0	0

**Abbildung 6.7:** Beispiel für den Deadlock-Erkennungsalgorithmus.

# Beheben von Deadlocks

Mögliche Methoden zum Beheben von Deadlocks  
(obwohl keine "attraktiv" ist):

## 1. Behebung durch Unterbrechung

(ob Unterbrechung möglich ist, hängt von Art der Ressource ab.  
Z.B. Laserdrucker anhalten, anderer Druckjob und manuell  
fortfahren)

## 2. Behebung durch Rollback

(Zustand Prozess regelmäßig an Checkpoints in Datei speichern,  
bei Unterbrechung zurück zum letzten Checkpoint)

## 3. Behebung durch Prozessabbruch

(Prozess abbrechen, vielleicht laufen die anderen dann weiter)

# 6.5 Verhinderung von Deadlocks

6.5.1 Ressourcenspuren

6.5.2 Sichere und unsichere Zustände

6.5.3 Der Bankier-Algorithmus für eine einzelne Ressource

6.5.4 Der Bankier-Algorithmus für mehrere Ressourcen

# Ressourcenspuren

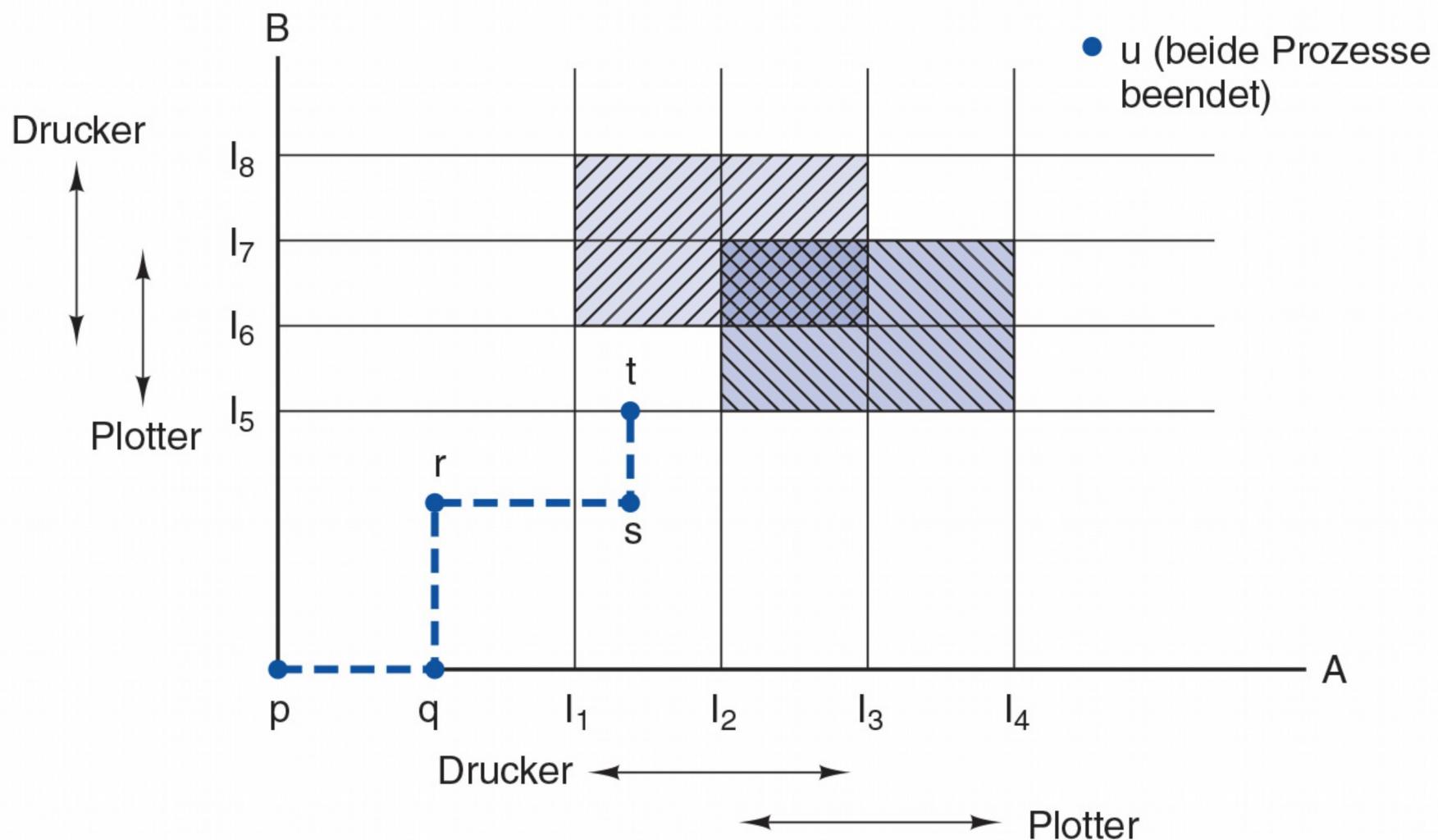


Abbildung 6.8: Ressourcenspur für zwei Prozesse.

# Sichere und unsichere Zustände

Belegt Max.		
A	3	9
B	2	4
C	2	7
Frei: 3		
Belegt Max.		
A	3	9
B	4	4
C	2	7
Frei: 1		
Belegt Max.		
A	3	9
B	0	-
C	2	7
Frei: 5		
Belegt Max.		
A	3	9
B	0	-
C	7	7
Frei: 0		
Belegt Max.		
A	3	9
B	0	-
C	0	-
Frei: 7		

**a**      **b**      **c**      **d**      **e**

**Abbildung 6.9:** Nachweis, dass der Zustand in (a) sicher ist.

# Sichere und unsichere Zustände

Belegt Max.		
A	3	9
B	2	4
C	2	7

Frei: 3

a

Belegt Max.		
A	4	9
B	2	4
C	2	7

Frei: 2

b

Belegt Max.		
A	4	9
B	4	4
C	2	7

Frei: 0

c

Belegt Max.		
A	4	9
B	-	-
C	2	7

Frei: 4

d

**Abbildung 6.10:** Nachweis, dass der Zustand in (b) unsicher ist.

# Der Bankier-Algorithmus für eine einzelne Ressource

	Belegt	Max.
A	0	6
B	0	5
C	0	4
D	0	7

Frei: 10

a

	Belegt	Max.
A	1	6
B	1	5
C	2	4
D	4	7

Frei: 2

b

	Belegt	Max.
A	1	6
B	2	5
C	2	4
D	4	7

Frei: 1

c

**Abbildung 6.11:** Drei Belegungszustände: (a) sicher; (b) sicher; (c) unsicher.

# Der Bankier-Algorithmus für mehrere Ressourcen (1)

	Prozess	Bandlaufwerk	Plotter	Drucker	Blu-ray
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	

zugewiesene Ressourcen

	Prozess	Bandlaufwerk	Plotter	Drucker	Blu-ray
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	

noch benötigte Ressourcen

$$\begin{aligned}E &= (6342) \\P &= (5322) \\A &= (1020)\end{aligned}$$

**Abbildung 6.12:** Der Bankier-Algorithmus für mehrere Ressourcenklassen.

# Der Bankier-Algorithmus für mehrere Ressourcen (2)

1. Suche eine Zeile aus  $R$ , deren ungedeckter Ressourcenbedarf kleiner oder gleich  $A$  ist. Wenn es keine solche Zeile gibt, kann kein Prozess beendet werden und das System wird in einen Deadlock laufen (vorausgesetzt, die Prozesse behalten immer alle Ressourcen, die sie einmal belegt haben).
2. Nimm an, dass der Prozess, der der gewählten Zeile entspricht, alle nötigen Ressourcen reserviert (was immer möglich ist) und seine Ausführung beendet. Markiere den Prozess als beendet und addiere seine Ressourcen zu  $A$ .
3. Wiederhole Schritt 1 und 2, bis entweder alle Prozesse markiert sind oder kein Prozess mehr übrig ist, dessen Ressourcenbedarf gedeckt werden kann. Im ersten Fall ist der Zustand sicher, im zweiten Fall tritt ein Deadlock auf.

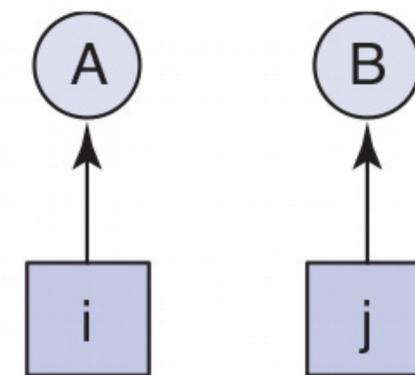
# 6.6 Vermeidung von Deadlocks

- 6.6.1 Unterlaufen der Bedingung des wechselseitigen Ausschlusses
- 6.6.2 Unterlaufen der Hold-and-Wait Bedingung
- 6.6.3 Unterlaufen der Bedingung der Ununterbrechbarkeit
- 6.6.4 Unterlaufen der zyklischen Wartebedingung

# Unterlaufen der zyklischen Wartebedingung (1)

1. Filmbelichter
2. Drucker
3. Plotter
4. Bandlaufwerk
5. Blu-ray-Laufwerk

a



b

**Abbildung 6.13:** (a) Numerisch geordnete Ressourcen. (b) Ressourcen-Belegungsgraph.

# Unterlaufen der zyklischen Wartebedingung (2)

Bedingung	Ansatz
Wechselseitiger Ausschluss	Spooling
Hold-and-Wait	Alle Ressourcen zu Beginn anfordern
Ununterbrechbarkeit	Ressourcen entziehen
Zyklisches Warten	Ressourcen numerisch ordnen

**Abbildung 6.14:** Die verschiedenen Ansätze zur Deadlock-Vermeidung.

# 6.7 Weitere Themen zu Deadlocks

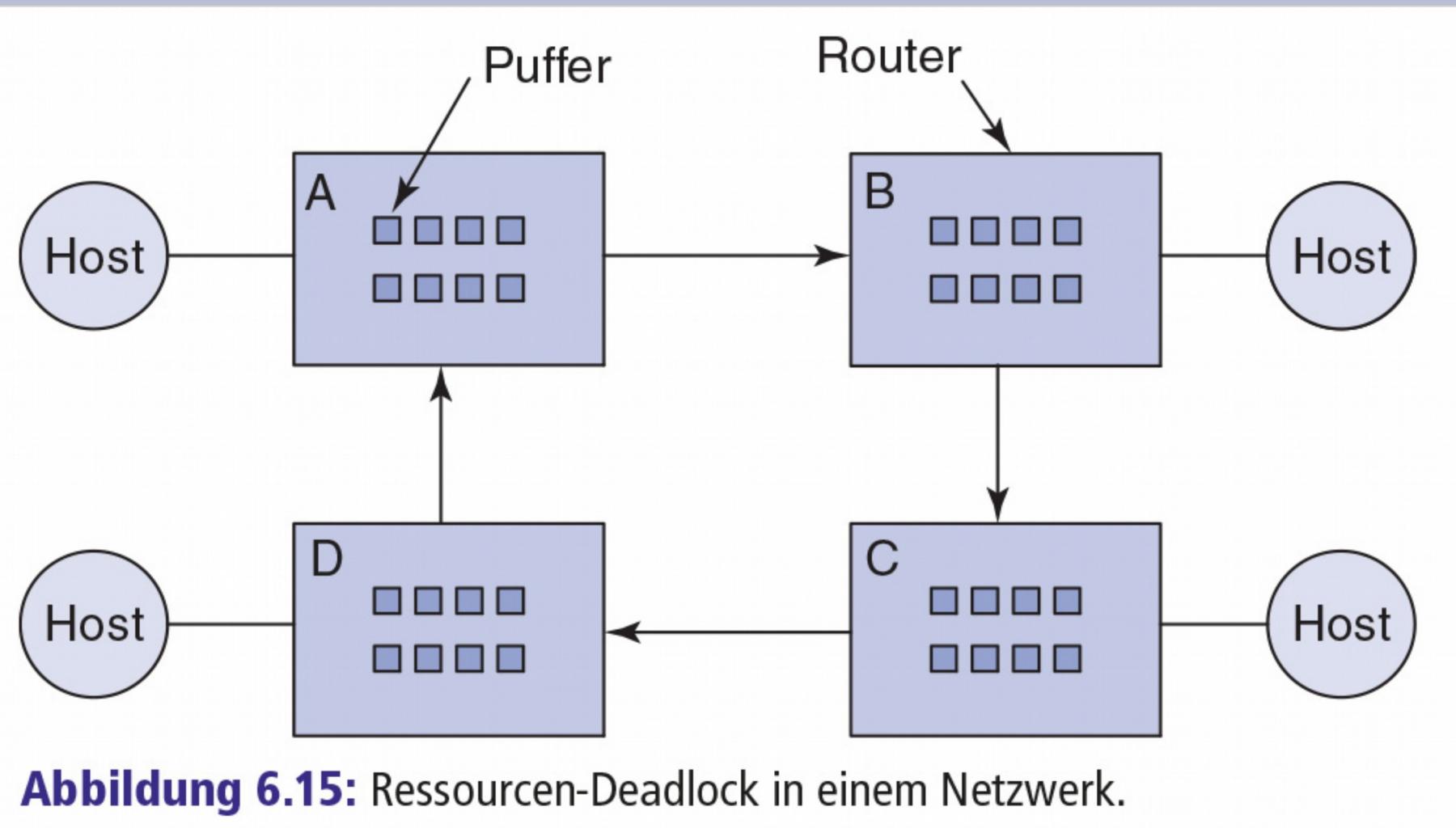
6.7.1 Zwei-Phasen-Sperren

6.7.2 Kommunikationsdeadlocks

6.7.3 Livelock

6.7.4 Verhungern

# Kommunikationsdeadlocks



**Abbildung 6.15:** Ressourcen-Deadlock in einem Netzwerk.

# Livelock

```
void process_A(void) {  
    acquire_lock(&resource_1);  
    while (try_lock(&resource_2) == FAIL) {  
        release_lock(&resource_1);  
        wait_fixed_time();  
        acquire_lock(&resource_1);  
    }  
    use_both_resources();  
    release_lock(&resource_2);  
    release_lock(&resource_1);  
}
```

```
void process_B(void) {  
    acquire_lock(&resource_2);  
    while (try_lock(&resource_1) == FAIL) {  
        release_lock(&resource_2);  
        wait_fixed_time();  
        acquire_lock(&resource_2);  
    }  
    use_both_resources();  
    release_lock(&resource_1);  
    release_lock(&resource_2);  
}
```

**Abbildung 6.16:** Höfliche Prozesse können einen Livelock verursachen.