

# Betriebssysteme und Netzwerke

## Vorlesung 15

Artur Andrzejak

# Deadlocks - Verklemmungen

# Deadlocks - Verklemmungen

---

- ▶ Ein Gesetz, das in US-Staat Kansas anfangs des 20. Jahrhunderts verabschiedet wurde, besagt:

*„When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.“*

- ▶ Generell, ein **Deadlock** (**Verklemmung**) entsteht, wenn mehrere Prozesse aufeinander warten ...
- ▶ Und keiner die Ausführung fortsetzen kann
  - ▶ Weil andere wartende Prozesse Ressourcen gespermt haben, die benötigt sind, um weiterzumachen

# Deadlock - Beispiel

- ▶ Ein System hat zwei Festplatten (FP), **A** und **B**
- ▶ Eine Situation mit einem Deadlock:
  - ▶ **Prozess P1** hat A gesperrt (durch Betreten einer kritischen Region) und braucht nun B
  - ▶ **Prozess P2** hat B gesperrt und braucht nun A
- ▶ Mögliche „Implementierung“ durch Semaphore?
  - ▶ Semaphore A und B, initialisiert jeweils zu 1

P1	P2
wait (A) wait (B)	wait (B) wait (A)

Dieser Code führt nicht zwangsläufig zum Deadlock! Nur bei einer ungünstigen Ausführung entsteht Deadlock, sequentiell:  
P1:wait(A) ... P2:wait(B) ... P1:wait(B) ... P2:wait(A)

# Schutz durch Semaphore

- ▶ Zwei Ressourcen, zwei Semaphore, zwei Prozesse
- ▶ Ist dieser Code *korrekt*?

```
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);  
}
```

Ja! Dieser Code aber nicht:

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

# Deadlocks – Strikt Definiert

---

- ▶ Eine Gruppe von Prozessen befindet sich in einem **Deadlock-Zustand**, wenn jeder Prozess aus der Gruppe auf ein Ereignis wartet, das nur ein anderer Prozess aus der Gruppe auslösen kann
- ▶ Meistens ist das Ereignis, auf welches ein Prozess wartet: **Die Freigabe einer Ressource**
  - ▶ Diese Art von Deadlocks werden **Ressourcen-Deadlocks** genannt
- ▶ Andere Formen: **Kommunikationsdeadlocks**, **Livelocks**: Wie Deadlocks, nur der „Zustand“ der Prozesse ändert sich laufend - aber ohne Fortschritt

# Systemmodell

---

- ▶ 1. Wir betrachten **nicht-unterbrechbare Ressourcen** (**nonpreemptable resources**)
  - ▶ Solche können dem aktuellen Besitzer nicht entzogen werden, ohne dass die Ausführung fehlschlägt
  - ▶ Beispiele: CD-Brenner, Drucker
  - ▶ Beispiel unterbrechbare Ressource: Speicher
- ▶ 2. Ein Prozess verwendet jede Ressource wie folgt:
  1. Die Ressource anfordern
  2. Die Ressource benutzen
  3. Die Ressource freigeben

# Voraussetzungen für Ressourcen-Deadlocks

---

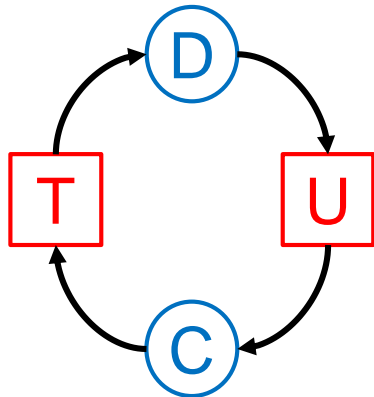
- ▶ **Wechselseitiger Ausschluss (mutual exclusion)**
  - ▶ Jede Ressource ist entweder verfügbar oder genau einem Prozess zugeordnet
- ▶ **Hold-and-Wait-Bedingung (hold and wait)**
  - ▶ Prozesse, die schon Ressourcen reserviert haben, können noch weitere Ressourcen anfordern
- ▶ **Bedingung der Ununterbrechbarkeit (no preemption)**
  - ▶ Ressourcen, die einem Prozess bewilligt wurden, können diesem nicht gewaltsam wieder entzogen werden; der Prozess muss sie explizit freigeben
- ▶ **Zyklische Wartebedingung (circular wait)**
  - ▶ Es muss eine zyklische Kette von Prozessen geben, von denen jeder auf eine Ressource wartet, die dem nächsten Prozess in der Kette gehört, d.h. in  $P_0, P_1, P_2, \dots, P_n$ :  $P_0$  wartet auf  $P_1$ ,  $P_1$  auf  $P_2$ , ...,  $P_n$  auf  $P_0$

Gut ist: die Bedingungen sind notwendig aber nicht hinreichend und müssen gleichzeitig erfüllt sein!



# Ressourcen-Belegungs-Graph (R-B-Graph)

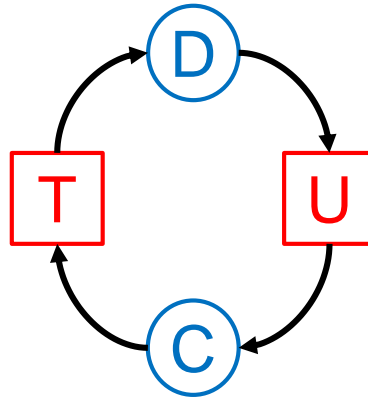
- ▶ Modellierung von Deadlocks als **R-B-Graph**
- ▶ **Prozesse**  $P_0, P_1, \dots, P_n$  (oder A, B, C, D,...)
  - ▶ Knoten des Graphen dargestellt als Kreise
- ▶ **Ressourcen**  $R_0, \dots, R_m$  (oder R, S, T, U, ...)
  - ▶ Knoten des Graphen dargestellt als Quadrate
- ▶ Gerichtete Kanten (dargestellt als Pfeile)
  - ▶ Von Prozessen zu den Ressourcen
    - ▶ Interpretation: „Prozess verlangt ein Ressource und wartet darauf“
  - ▶ Von Ressourcen zu den Prozessen
    - ▶ Interpretation: „Ressource wurde dem Prozess zugeordnet“



Ein Deadlock oder keiner?

# Ein Deadlock oder keiner?

---



Nochmals:

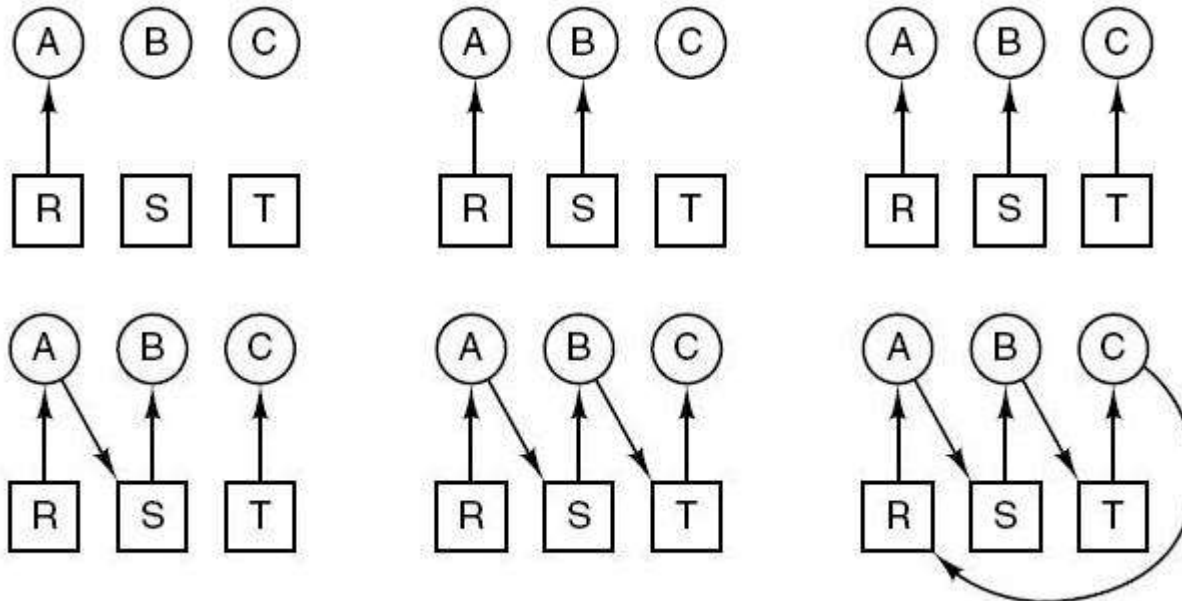
- ▶ Wechselseitiger Ausschluss (mutual exclusion)
- ▶ Hold-and-Wait-Bedingung (hold and wait)
- ▶ Bedingung der Ununterbrechbarkeit (no preemption)
- ▶ Zyklische Wartebedingung (circular wait)
  - ▶ Es muss eine zyklische Kette von Prozessen geben, von denen jeder auf eine Ressource wartet, die dem nächsten Prozess in der Kette gehört
  - ▶ D.h. in  $P_0, P_1, \dots, P_n$ :  $P_0$  wartet auf  $P_1$ ,  $P_1$  auf  $P_2$ , ...,  $P_n$  auf  $P_0$

# Beispiel - Entstehung eines Deadlocks

- ▶ Drei Prozesse: A, B, C
- ▶ Drei Ressourcen: R, S, T
- ▶ Ablauf

Wir zeichnen den Graphen gemeinsam

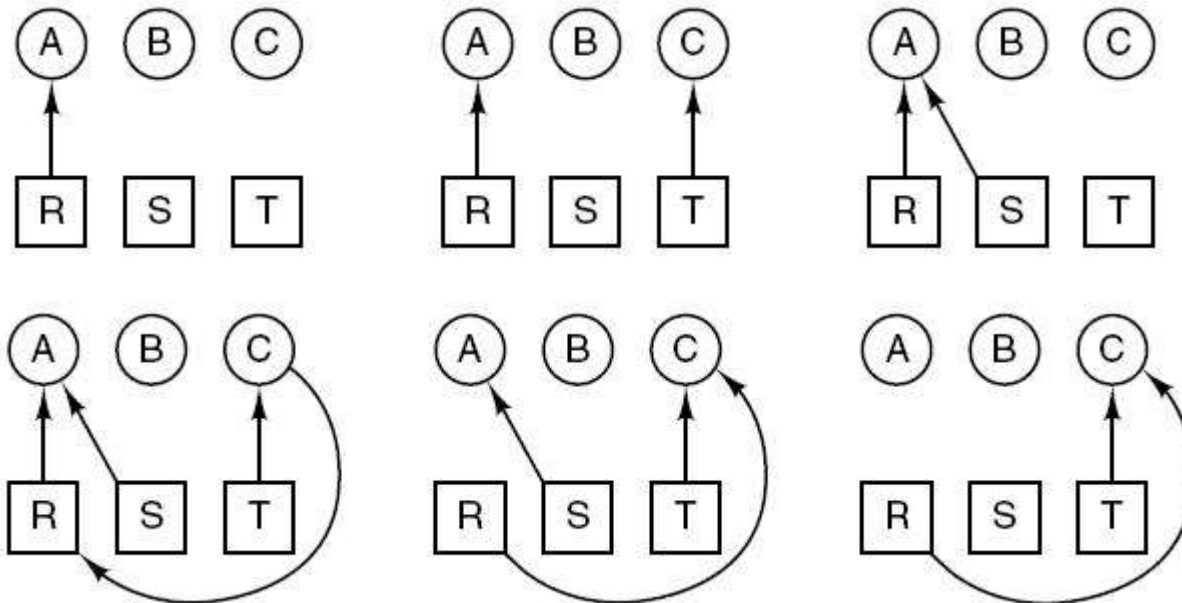
A verlangt und bekommt R  
B verlangt und bekommt S  
C verlangt und bekommt T  
A verlangt S  
B verlangt T  
C verlangt R



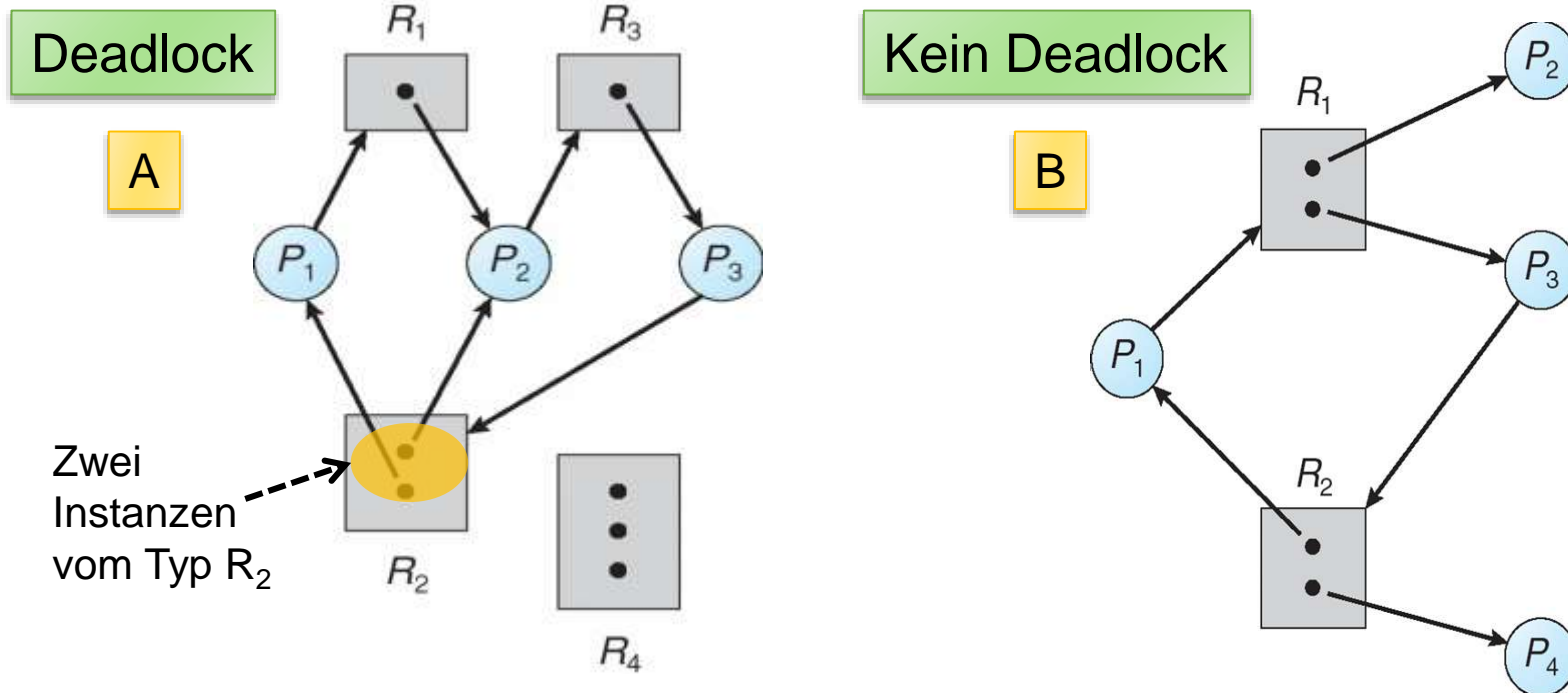
# Verhindern eines Deadlocks

- ▶ Das BS kann einen anderen Ablauf wählen und Deadlock **verhindern**
- ▶ Ablauf —————→

A verlangt und ... R  
C verlangt und ... T  
A verlangt S; C gibt T frei  
C verlangt R und T  
A gibt R frei; C bekommt T  
A gibt S frei; C hält R und T



# Mehrere Instanzen einer Ressource



- ▶ Wir wissen: Kein Zyklus => **kein Deadlock**
- ▶ Wenn aber ein Zyklus existiert, dann unterscheiden wir:
  - ▶ Eine Instanz pro Ressourcentyp => **Deadlock**
  - ▶ Mehrere Instanzen pro Ressourcentyp => **Deadlock ist möglich, aber nicht zwingend** (siehe Beispiele A, B)

# Vier Strategien, die Deadlocks zu behandeln

---

- ▶ **Einfach ignorieren:** der „Vogel-Strauß-Algorithmus“
  - ▶ Die meisten BS wählen diesen Ansatz, da Deadlocks (im Vergleich zu anderen Problemen) sehr selten sind
- ▶ **Erkennen und beheben**
  - ▶ Deadlocks zulassen, erkennen und etwas dagegen unternehmen
- ▶ **Dynamische Verhinderung**
  - ▶ Durch vorsichtige Ressourcenzuteilung
- ▶ **Vermeidung von Deadlocks**
  - ▶ Eine der vier Bedingungen sollte prinzipiell unerfüllbar sein

# Erkennung von Deadlocks

## Strategien zur Behandlung von Deadlocks:

- Einfach ignorieren
- **Erkennen und beheben**
  - Deadlocks zulassen, erkennen und beheben
- **Dynamische Verhinderung**
  - Durch vorsichtige Ressourcenzuteilung
- **Vermeidung von Deadlocks**

# Erkennen von Deadlocks

---

- ▶ Annahme: unser BS weiß, welche Prozesse welche Ressourcen belegen oder angefordert haben
  - ▶ Z.B. Via Kontrolle von Semaphoren und Systemaufrufen
- ▶ Zunächst der einfache Fall: **nur eine Instanz pro Ressourcentyp**
  - ▶ Wir haben 4 Bedingungen (*Wechselseitiger Ausschluss, Hold-and-Wait-Bedingung, Bedingung der Ununterbrechbarkeit, Zyklische Wartebedingung*)
  - ▶ Welche soll man dynamisch überprüfen, welche nicht?
  - ▶ => Was muss das BS laufend testen?
- ▶ Das BS muss nur prüfen, ob der Ressourcen-Belegungs-Graph (**RBG**) einen Zyklus enthält



# Ein Algorithmus zur Zyklus-Erkennung?

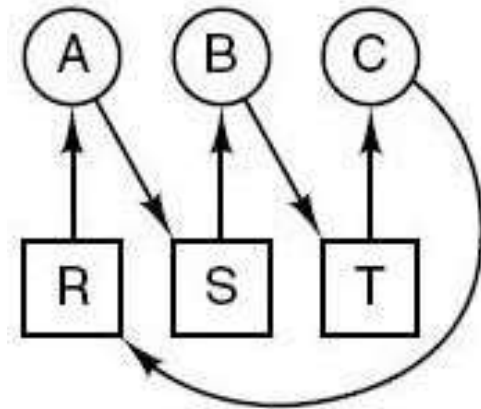
---

- ▶ Wir wollen testen, ob der gerichtete Ressource-Belegungs-Graph  $G$  irgendeinen Zyklus enthält
- ▶ Es reicht es, für jeden Knoten  $N$  (Prozess oder Ressource) zu testen, ob dieser sich in einem Zyklus befindet („Zyklus-Test für  $N$ “)
  - ▶ Falls das für keinen der Knoten  $N$  gilt, ist  $G$  zyklensfrei
- ▶ Mögliche Implementierung des „Zyklus-Tests für  $N$ “?
  - ▶ Führe Tiefensuche (DFS) mit  $N$  als Wurzel aus und markiere alle zum 1. Mal besuchten Knoten
  - ▶ Falls wir einen Knoten zwei Mal sehen, gibt es einen Zyklus

# Beispiel zur Zyklus-Erkennung

---

- ▶ Für welche Knoten würde hier T1 Zyklen anzeigen?
- ▶ Kleine Optimierung der Erkennung?



- ▶ Optimierung: breche ab, sobald man den 1. Zyklus gefunden hat => es gibt ein Deadlock
- ▶ In diesem Fall: Für jeden Knoten gibt es einen Zyklus

# Erkennung von Deadlocks: Genereller Ansatz

## Strategien zur Behandlung von Deadlocks:

- Einfach ignorieren
- **Erkennen und beheben**
  - Deadlocks zulassen, erkennen und beheben
- **Dynamische Verhinderung**
  - Durch vorsichtige Ressourcenzuteilung
- **Vermeidung von Deadlocks**

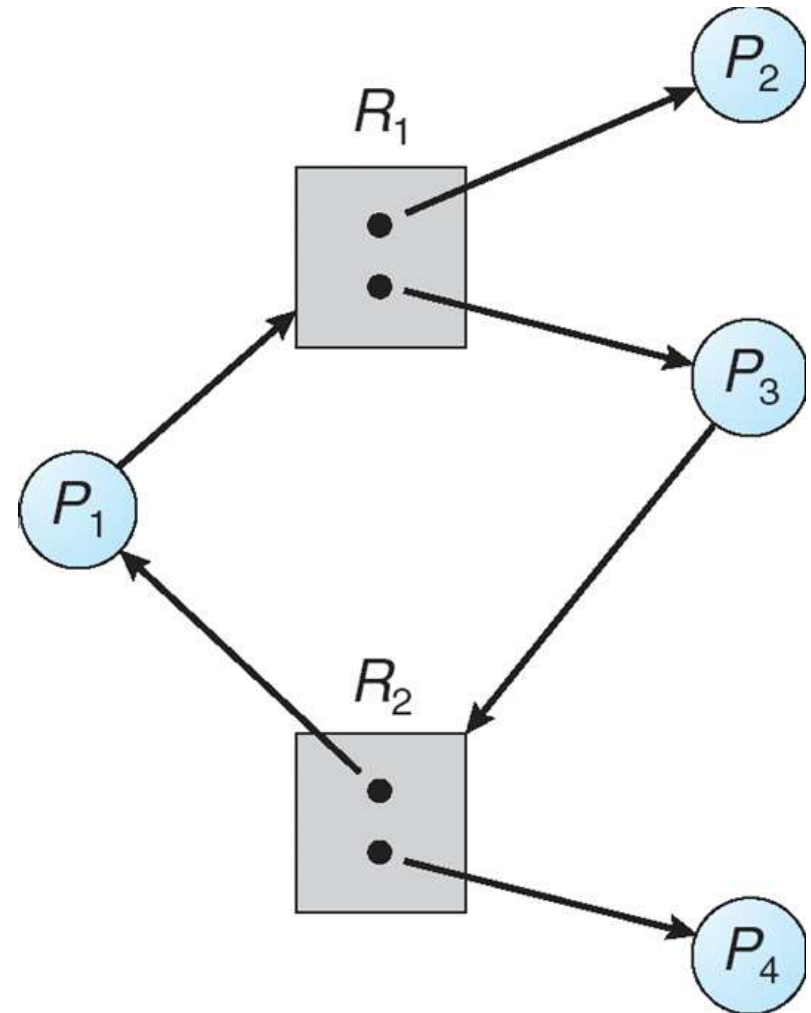
# Erkennung mit mehreren Instanzen pro R.-Typ

---

- ▶ Wir nutzen einen matrixbasierten Algorithmus
- ▶ Wir haben  $n$  Prozesse  $P_1, \dots, P_n$  und  $m$  Ressourcen(typen) mit Indices  $1, \dots, m$
- ▶ Die Anzahl der verfügbaren Instanzen vom Typ  $i$  wird mit  $E_i$  angegeben
- ▶  $(E_1, E_2, \dots, E_m)$  ist der **Ressourcenvektor**
- ▶ Für jeden Ressourcen(typ)  $i$  bezeichnen wir mit  $A_i$  die Anzahl der noch freien (nicht belegten) Instanzen
- ▶  $(A_1, \dots, A_m)$  bilden den **Ressourcenrestvektor** (available resource vector)

# Beispiel Ressourcen{rest}vektoren

- ▶ Bei diesem RBG: was ist der Ressourcenvektor  $(E_1, E_2, \dots, E_m)$ , was ist der Ressourcenrestvektor  $(A_1, \dots, A_m)$ ?
- ▶  $(E_1, E_2, \dots, E_m): (2, 2)$
- ▶  $(A_1, \dots, A_m): (0, 0)$



# Mehrere Instanzen pro Typ – Definitionen

---

- ▶ **Aktuelle Belegungsmatrix (current allocation matrix)  $C$**

- ▶ Der Eintrag  $C_{i,j}$  ist die Anzahl der Ressourcen vom Typ  $j$ , die Prozess  $i$  aktuell belegt

- ▶ Was gibt die  $i$ -te Zeile / was die  $j$ -te Spalte an?

- ▶ Zeile  $i$  gibt die aktuelle Belegung für Prozess  $i$  an

- ▶ Spalte  $j$  gibt aktuelle Belegung des Ressourcentyps  $j$  an

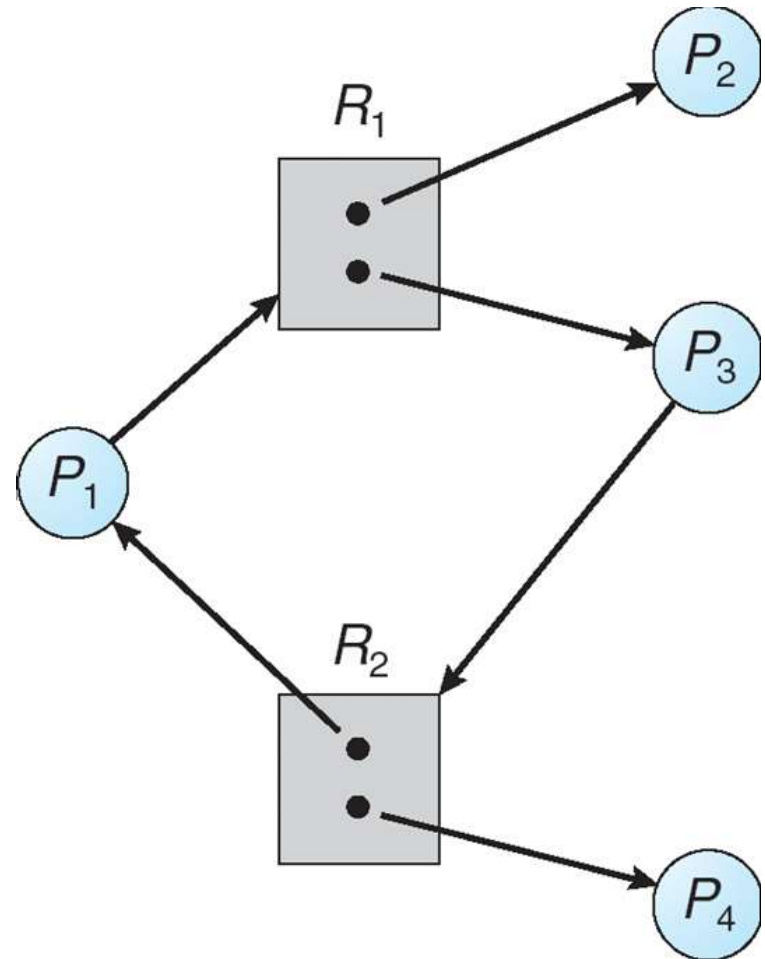
$$\begin{bmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \dots & C_{nm} \end{bmatrix}$$

# Beispiel Aktuelle Belegungsmatrix

- Was ist die aktuelle Belegungsmatrix hier?

$$\begin{bmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \dots & C_{nm} \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$



# Anforderungsmatrix

- ▶ **Anforderungsmatrix (request matrix)  $R$**

- ▶ Der Eintrag  $R_{ij}$  ist die Anzahl der Ressourcen der Klasse  $j$ , die Prozess  $i$  angefordert hat (aber noch nicht belegt)

- ▶ Was gibt die  $i$ -te Zeile / was die  $j$ -te Spalte an?

- ▶ Zeile  $i$  gibt die Anforderung für Prozess  $i$  an

- ▶ Spalte  $j$  gibt die ausstehenden Anforderungen an Instanzen der Ressource von Typ  $j$  an

$$\begin{bmatrix} R_{11} & R_{12} & \dots & R_{1m} \\ R_{21} & R_{22} & \dots & R_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ R_{n1} & R_{n2} & \dots & R_{nm} \end{bmatrix}$$




# Zusammenfassung: C, R, E, A

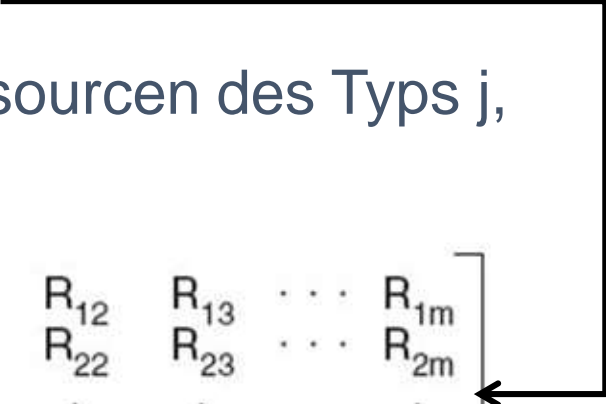
## ► Aktuelle Belegungsmatrix C

- Der Eintrag  $C_{ij}$  ist die Anzahl der Ressourcen des Typs  $j$ , die Prozess  $i$  belegt

## ► Anforderungsmatrix R

- Der Eintrag  $R_{ij}$  ist die Anzahl der Ressourcen des Typs  $j$ , die Prozess  $i$  anfordert


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


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

## ► Ressourcenvektor ( $E_1, E_2, \dots, E_m$ )

## ► Ressourcenrestvektor ( $A_1, A_2, \dots, A_m$ )

# Ein Beispiel

---

- ▶ Ressourcen sind: 4 Bandlaufwerke, 2 Plotter, 3 Scanner, 1 CD-ROM
- ▶  $E = (4, 2, 3, 1)$  (gleiche Reihenfolge)
- ▶  $A = (2, 1, 0, 0)$
- ▶ Kann das stimmen und warum / warum nicht?
  1. Wir können testen, ob die Summe jeder Spalte von  $C$  nicht größer als jeweilige Komponente von  $E$  ist
  2. Vektor der Spaltensummen von  $C$  plus  $A$  muss  $E$  ergeben

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

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

# Algorithmus-Idee /1

---

- ▶ Man nimmt an, dass alle Prozesse „ihre“ Ressourcen behalten, bis sie terminieren
- ▶ Es wird angenommen, dass ein **Prozess i**, der die die gewünschten Ressourcen bekommt, erfolgreich weitermachen kann und irgendwann terminiert
- ▶ Dadurch gibt **i** alle Ressourcen frei:
  - ▶ Die bis jetzt belegten (in **Zeile i** der Matrix C angegeben)
  - ▶ Und die neu angeforderten (in **Zeile i** der Matrix R ...)
- ▶ Dadurch stehen nach der Terminierung von **i** insgesamt mehr Ressourcen zur Verfügung
  - ▶ Denn auch die bis jetzt von i belegten werden frei
  - ▶ Somit erhöht man die Chance, dass auch andere Prozesse erfolgreich terminieren können

# Algorithmus-Idee /2

---

- ▶ Am Anfang gelten alle Prozesse als „unmarkiert“
- ▶ Wiederhole, bis die Suche fehlschlägt:
  - ▶ Suche nach einem Prozess  $i$ , für den die Ressourcen-anfrage ausgeführt werden kann und markiere ihn
  - ▶ Nehme an, dass er irgendwann terminiert und die aktuell belegten Ressourcen freigibt
    - ▶  $\Rightarrow$  Addiere seine aktuell belegten Ressourcen (aus  $C_i$ ) zu  $A$
- ▶ Prüfe zum Schluß: Gibt es noch unmarkierte Prozesse, so haben wir einen Deadlock
  - ▶ Die unmarkierten Prozesse sind an einem Deadlock beteiligt
- ▶ Technische Anmerkung:
  - ▶ Vergleich von Vektoren:  $\mathbf{X} \leq \mathbf{Y}$  gdw  $X_i \leq Y_i$  für alle Komponenten (d.h.  $i = 1, \dots, m$ ) gilt

# Algorithmus - Bankieralgorithmus

1. Suche einen unmarkierten Prozess  $i$ , für den die  $i$ -te Zeile von  $R$  kleiner oder gleich  $A$  ist:  $R_i \leq A$
2. Wenn ein solcher Prozess existiert:
  - ▶ Addiere die  $i$ -te Zeile von  $C$  zu  $A$ :  $A = A + C_i$
  - ▶ Markiere den Prozess  $i$
  - ▶ Gehe zum Schritt 1 zurück
3. Anderenfalls beende den Algorithmus:
  - ▶ Falls es noch unmarkierte Prozesse gibt, haben wird ein Deadlock
  - ▶ Gibt es keine unmarkierten Prozesse mehr => kein Deadlock

Warum?

Erläuterung zu  $A = A + C_i$  : Da  $i$  nun ausführen und terminieren kann, wird  $i$  am Ende alle soeben angeforderten Ressourcen (angegeben in  $R_i$ ) zurückgeben und zusätzlich alle noch bis jetzt gehaltenen Ressourcen (angegeben in  $C_i$ ). Wir verschmelzen beide Schritte => damit kann der Ressourcenrestvektor  $A$  sofort um  $C_i$  vergrößert werden!

# Woher der Name „Bankieralgorithmus“?

---

- ▶ Angenommen, mehrere EU-Länder haben Schulden
- ▶ Sie brauchen aber erstmal noch mehr Geld, um zu überleben und alte Schulden zurückzuzahlen
- ▶ Die EZB hat nur begrenzte Geldmenge („Vektor A“)
- ▶ EZB wählt ein Land aus (sagen wir „G“), das nicht mehr braucht als EZB gerade hat, und leiht ihm Geld
- ▶ Nun kann „G“ fröhlich weiterwirtschaften, und irgendwann („am Ende“?) zahlt es alles zurück: Alte Schulden und das neue Geld von EZB ...
- ▶ Dadurch hat EZB noch mehr Geld (alte Schulden von „G“ sind zurückgezahlt), und hilft einem anderen Land – sagen wir „S“ ...

So viel zur Theorie 😊

# Nach der Erkennung: Deadlocks Beheben

---

- ▶ Behebung durch **Unterbrechung**
  - ▶ Beim Drucker: bereits ausgedruckte Blätter entnehmen und den Prozess suspendieren
  - ▶ Insgesamt schwierig und ressourcenspezifisch
- ▶ Behebung durch **Rollback**
  - ▶ Bei der Gefahr von Deadlocks wird regelmäßig der Zustand des Prozesses abgespeichert (**Checkpointing**)
  - ▶ Beim Deadlock wird ein beteiligter Prozess in den letzten Zustand zurückgesetzt, bei dem die Ressource noch nicht reserviert war (aber ein Teil der Arbeit geht verloren)
- ▶ Behebung durch **Prozessabbruch**
  - ▶ Brutal aber einfach

# Video: Deadlocks und Dining Philosophers

---

- ▶ MIT 6.004 L21: Processes, Synchronization & Deadlock
  - ▶ <https://www.youtube.com/watch?v=TVkQ1VeRKt4>
  - ▶ Von 41:50 bis ca. 48:00 (min:sec)



# Verhinderung und Vermeidung von Deadlocks

## Strategien zur Behandlung von Deadlocks:

- Einfach ignorieren
- Erkennen und beheben
  - Deadlocks zulassen, erkennen und beheben
- **Dynamische Verhinderung**
  - Durch vorsichtige Ressourcenzuteilung
- **Vermeidung von Deadlocks**

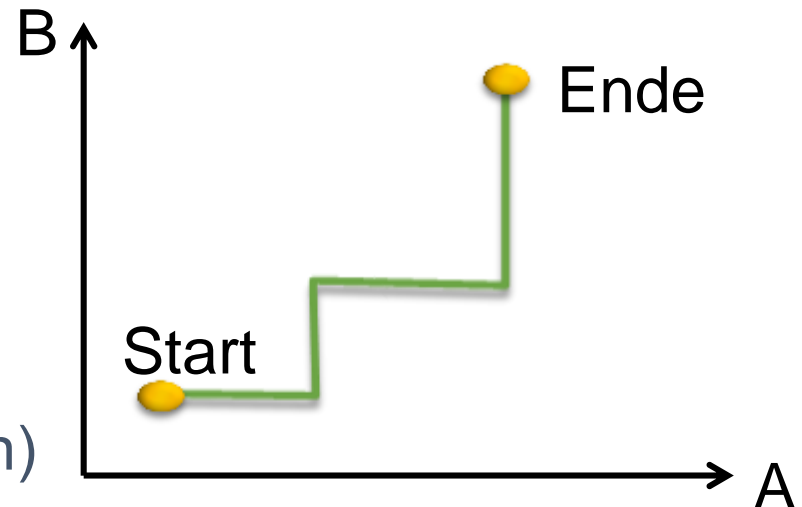
# Verhinderung von Deadlocks

---

- ▶ Meistens werden die Ressourcen nicht auf einmal, sondern nach und nach zugeteilt
- ▶ Das BS kann entscheiden, ob die nächste Ressource zugeteilt wird oder doch nicht
  - ▶ Das BS sollte das nur tun, nur es nicht gefährlich ist
- ▶ Gibt es einen Algorithmus, der das zuverlässig entscheiden kann (und somit Deadlocks verhindert)?
- ▶ Ja, aber nur wenn bestimmte Informationen im Voraus zur Verfügung stehen
  - ▶ U.a.: Maximale Anzahl der Ressourceninstanzen von jedem Typ, die ein Prozess reservieren wird

# Visualisierung eines Ausführungsablaufs

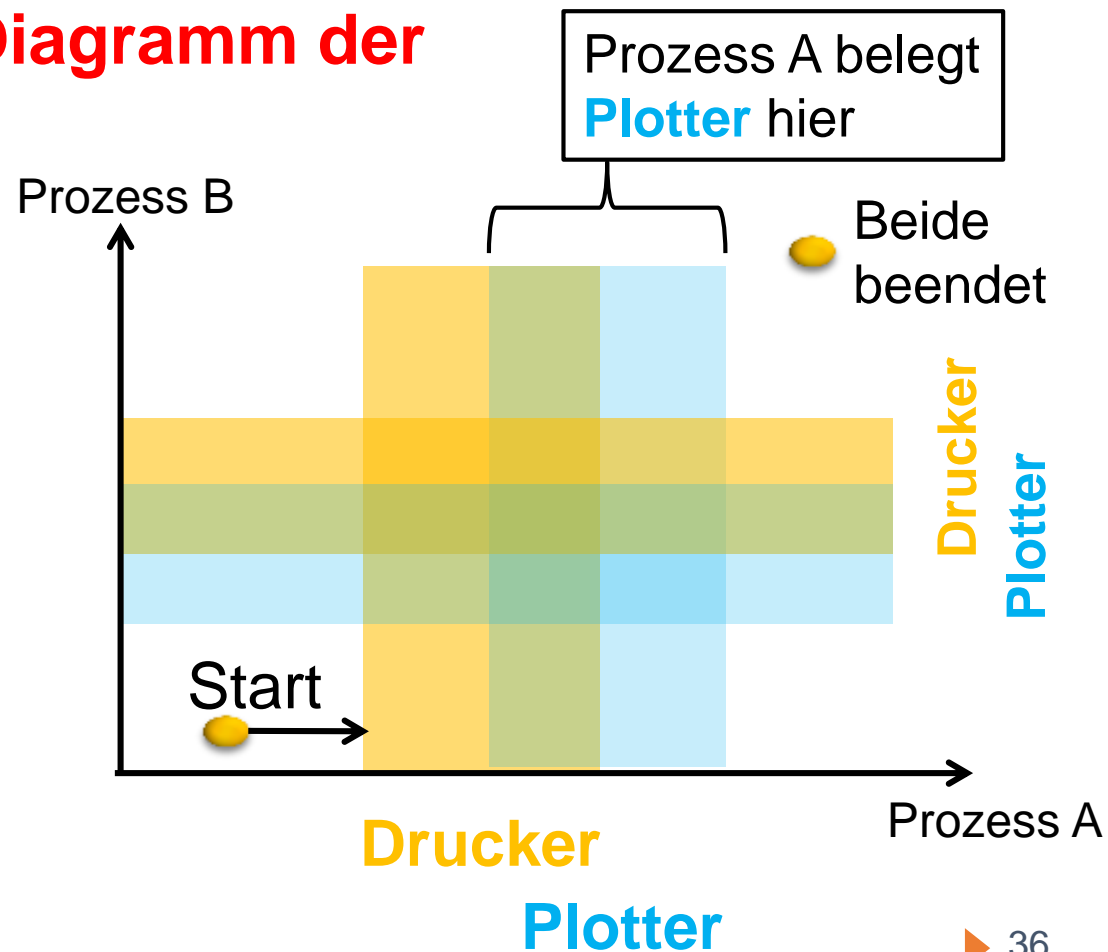
- ▶ Das BS darf das System nicht in eine Lage bringen, bei der ein Deadlock unvermeidlich ist
  - ▶ Dazu kann das BS eine Ressourcenanfrage so lange zu verzögern, bis ihre Erfüllung zu keinem Deadlock führen kann
- ▶ Wie kann eine „unvermeidliche“ Situation entstehen?
- ▶ Dazu visualisieren wir die Ausführung von zwei Prozessen A und B
  - ▶ A führt „horizontal“ aus (nach rechts)
  - ▶ B führt „vertikal“ aus (nach oben)



# Ressourcenspuren

- ▶ Wir können auch einzeichnen, wann ein Prozess ein Ressource benutzt (und diese belegt hat)
- ▶ Man nennt das ein **Diagramm der Ressourcenspuren**

- ▶ Innerhalb eines gefärbten Bereiches ist die Ressource belegt
- ▶ Am Rande eines gefärbten Bereiches gibt es evtl. eine Res.-Anfrage

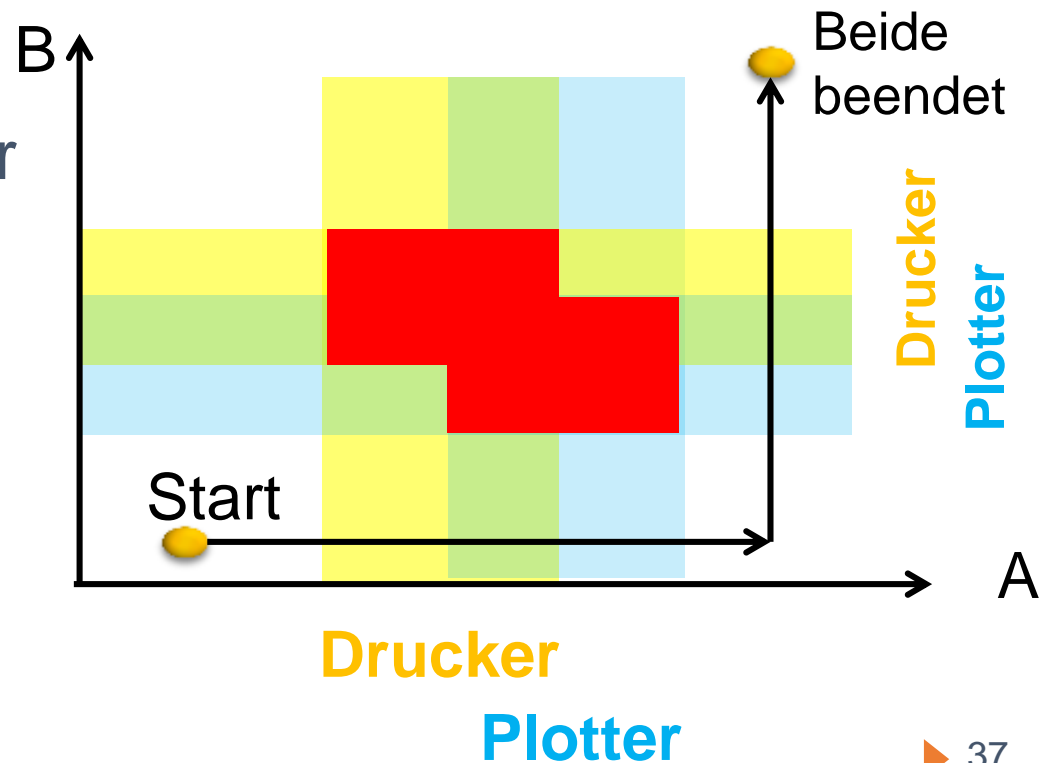


# Erfolgreiche Ausführung

- ▶ Angenommen, den jeweils 1 (einen) Drucker / Plotter kann nur 1 Prozess auf einmal verwenden
- ▶ Welche Bereiche sind dann verboten (unmöglich), und warum?

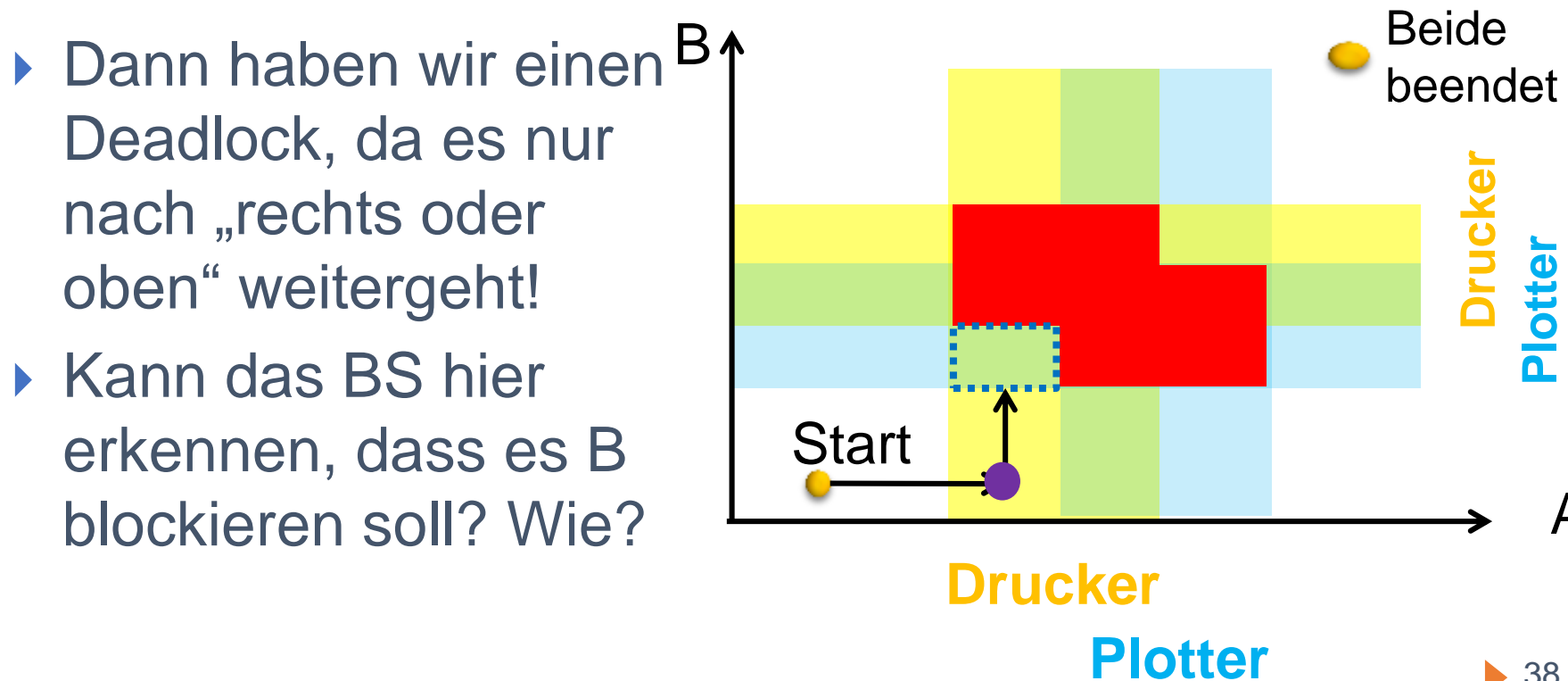
- ▶ Verboten: Gleichzeitige Belegung einer Ressource durch A und B (doppelt gefärbt mit gleicher Farbe)

- ▶ Mutual exclusion verletzt!



# Visualisierung von Deadlocks

- ▶ In der dargestellten Situation ● fragt der Prozess B beim BS nach dem Plotter an
- ▶ Was passiert, wenn das BS zustimmt, und wir in den blau umrandeten Zustand kommen?

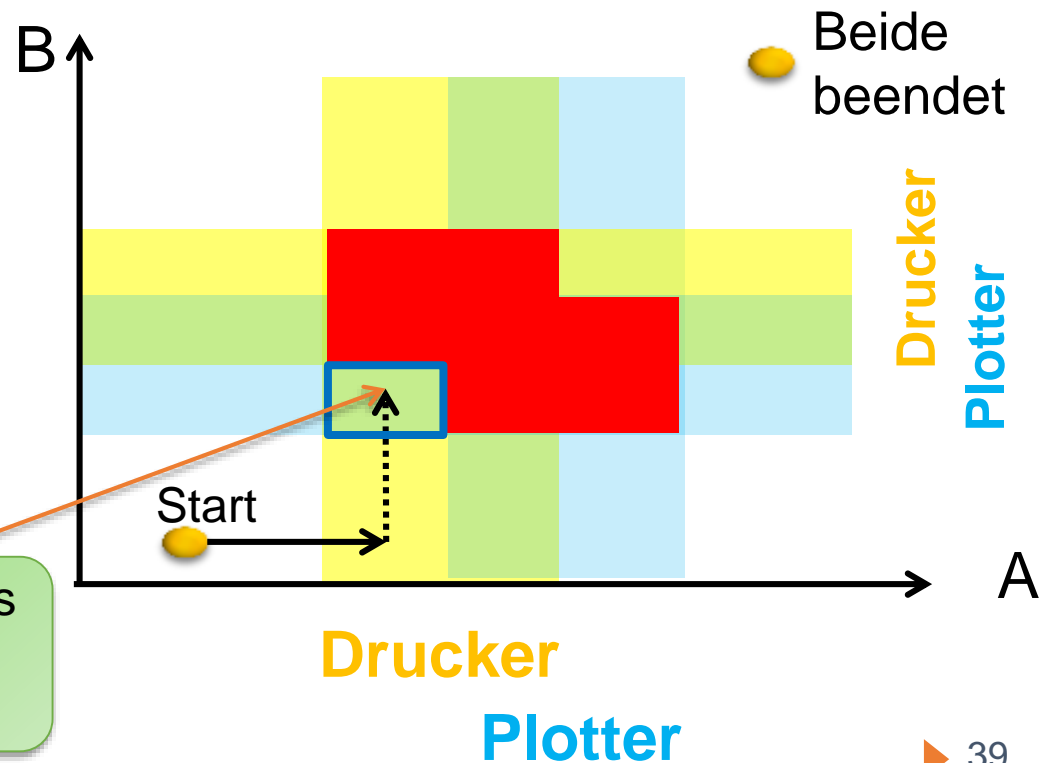


# Vermeidung von Deadlocks

- ▶ Wie kann das BS erkennen, dass es B blockieren soll?
- ▶ **Wir tun so, als ob das BS die Anfrage gebilligt hätte, und überprüfen dann, ob es ein Deadlock gäbe** (Hauptidee der Vermeidung)

- ▶ Wie können wir das implementieren?
- ▶ Mit dem „alten Freund“, dem Bankiersalgorithmus!

Lasse den Bankiersalgorithmus auf diesem hypothetischen Zustand laufen



# Nochmals der Bankieralgorithmus

---

- ▶ Der Bankieralgorithmus kann entscheiden, ob der Zustand des System zu einem Deadlock führt
- ▶ Wiederholung:
  - ▶ **A**: Vektor der Anzahlen aktuell verfügbarer Ressourcen
  - ▶ **E**: Vektor der Anzahlen vorhandener Ressourcen
  - ▶ **Aktuelle Belegungsmatrix C**:  $c_{i,j}$  = Anzahl Instanzen der Ressourcen aus der Klasse  $j$ , die von dem Prozess  $i$  belegt sind
  - ▶ **Anforderungsmatrix R**:  $r_{i,j}$  = Anzahl der Instanzen der Ressourcen von Typ  $j$ , der Prozess  $i$  noch insgesamt anfordern wird
    - ▶ Achtung: Modifizierte Bedeutung von R



# Erzeugen eines zu testenden Zustands

---

- ▶ Das BS nimmt also testweise an, dass es die angefragten Ressourcen an Prozess  $i$  zugeteilt hat und modifiziert (temporär) den Zustand wie folgt
- ▶ Sei  $Req_i$  der Vektor mit Anfrage von  $i$  ( $Req_i \leq R_i$ )
  - ▶  $A = A - Req_i$  (Vektoraddition)
    - ▶ D.h. reduziere den Ressourcenrestvektor um die Anfrage
  - ▶  $C_i = C_i + Req_i$  (Vektoraddition)
    - ▶ D.h. erhöhe die aktuelle Belegung durch  $P_i$
  - ▶  $R_i = R_i - Req_i$  (Vektorsubtraktion)
    - ▶ D.h. reduziere die restliche „Anfragen-Kapazität“ von  $P_i$

# Entscheidung über die Zuteilung der Ressourcen

---

- ▶ Das BS lässt nun den Bankiersalgorithmus (**BA**) auf diesem hypothetischen Zustand laufen
  - ▶ Falls **BA** antwortet, dass der Zustand **sicher** ist (zu keinem Deadlock führt), wird die Anfrage gebilligt und der hypothetische Zustand wird zum neuen Zustand
  - ▶ Falls **BA** antwortet, dass der Zustand zum Deadlock führt, lehnt das BS die Anfrage ab, oder blockiert den Prozess i, bis andere Prozesse Ressourcen freigeben
    - ▶ Der Deadlock müsste nicht wirklich auftreten, da z.B. andere Prozesse terminieren können

# Verhinderung und Vermeidung von Deadlocks

## Strategien zur Behandlung von Deadlocks:

- Einfach ignorieren
- Erkennen und beheben
  - Deadlocks zulassen, erkennen und beheben
- Dynamische Verhinderung
  - Durch vorsichtige Ressourcenzuteilung
- **Vermeidung von Deadlocks**

# Vermeidung von Deadlocks

---

- ▶ Unterlaufen von „Wechselseitiger Ausschluss“
  - ▶ Schwierig, und ggf. unmöglich (z.B. Drucker, DVD-Writer)
- ▶ Unterlaufen von „Hold-and-Wait“-Bedingung
  - ▶ Prozesse sollten alle benötigten Ressourcen im Voraus angeben
  - ▶ Oder: Vor neuer Anforderung alle Ressourcen kurzzeitig freigeben
- ▶ Unterlaufen der „Ununterbrechbarkeit“
  - ▶ Hier kann man mit Virtualisierung arbeiten
- ▶ Unterlaufen der zyklischen Wartebedingung
  - ▶ Im Prinzip durch die Verhinderung (soeben vorgestellt)

# Zusammenfassung

---

- ▶ Deadlocks – Grundlagen
- ▶ Erkennung von Deadlocks
  - ▶ Durch Graphen-Analyse (Eine Instanz pro R.-Typ)
- ▶ Erkennung von Deadlocks: Bankiersalgorithmus
  - ▶ Erkennung bei einer vs. mehreren Instanzen pro R.-Typ
- ▶ Verhinderung und Vermeidung von Deadlocks
- ▶ Quellen:
  - ▶ Deadlocks: Silberschatz et al. Kapitel 7; Tanenbaum Kap. 6