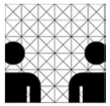
C Nebenläufigkeit und Verteilung

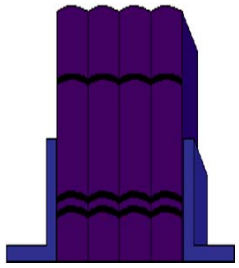


C 1.1: Einführung

Zunächst wurde ein Berechnungsvorgang als zeitliche Folge einzelner Berechnungsschritte modelliert (*sequentieller* Prozess). In realen Systemen können sich Prozesse zeitlich überlappen und interagieren – d.h. sie sind *nebenläufig*.

Wir befassen uns mit wichtigen Aspekten nebenläufiger Systeme:

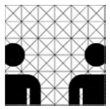
- Anwendungsprobleme
- formale Beschreibung und Analyse
- Architekturen und Entwurf
- Programmierung in Java



R.G. Herrtwich, G. Hommel: Nebenläufige Programme, Springer, 2. Aufl., 1994

J. Magee, J. Kramer: Concurrency - State Models & Java Programs, Wiley, 2nd Edition, 2006

A. Kemper, A. Eickler: Datenbanksysteme - Eine Einführung, 9. Auflage, Oldenbourg 2013, 848 S.



Beispiel Kontoführung

Prozess 1: Umbuchung eines Betrages von Konto A nach Konto B

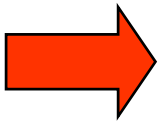
Prozess 2: Zinsgutschrift für Konto A

Umbuchung

```
read (A, a1)
a1 := a1 - 300
write (A, a1)
read (B, b1)
b1 := b1 + 300
write (B, b1)
```

Zinsgutschrift

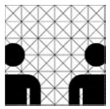
```
read (A, a2)
a2 := a2 * 1.03
write (A, a2)
```



Möglicher verzahnter Ablauf:

Umbuchung	Zinsgutschrift
read (A, a1)	
a1 := a1 - 300	
	read (A, a2)
	a2 := a2 * 1.03
	write (A, a2)
write (A, a1)	
read (B, b1)	
b1 := b1 + 300	
write (B, b1)	

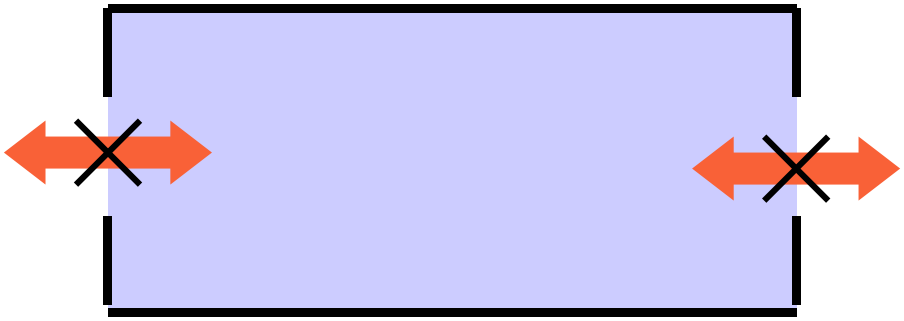
Wo ist die Zinsgutschrift geblieben??



Beispiel Besucherzählung

Drehkreuz1:

```
loop {
read (Counter, c1)
if (c1 >= MaxN) lock
if (c1 < MaxN) open
if enter incr(c1)
if leave decr(c1)
write (Counter, c1)
}
```



Drehkreuz2:

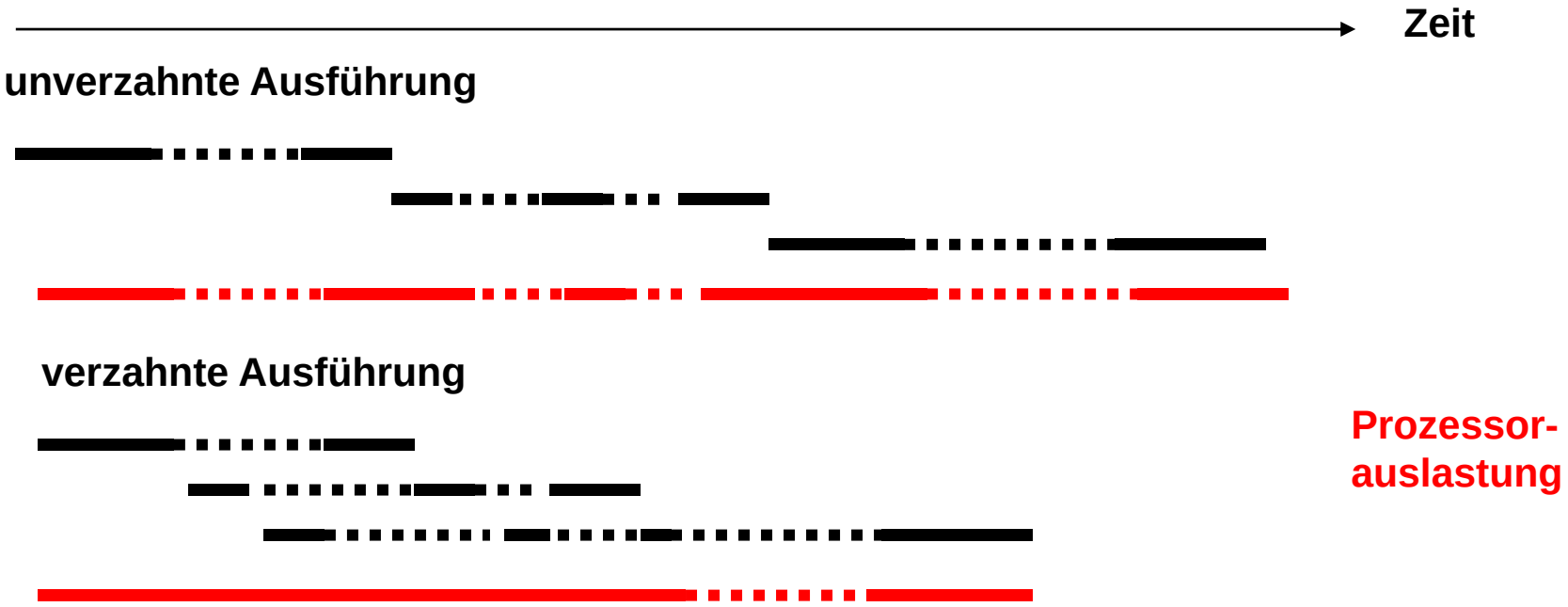
```
loop {
read (Counter, c2)
if (c2 >= MaxN) lock
if (c2 < MaxN) open
if enter incr(c2)
if leave decr(c2)
write (Counter, c2)
}
```

Verzahnte Ausführung der zwei Prozesse Drehkreuz1 und Drehkreuz2 mit Zugriff auf gemeinsamen Counter kann inkorrekte Besucherzahl ergeben!

=> Überfüllung, Panik, Katastrophen durch Studium der Nebenläufigkeit vermeiden

Mehrbenutzersynchronisation

Die nebenläufige Ausführung mehrerer Prozesse auf einem Rechner kann grundsätzlich zu einer besseren Ausnutzung des Prozessors führen, weil Wartezeiten eines Prozesses (z.B. auf ein I/O-Gerät) durch Aktivitäten eines anderen Prozesses ausgefüllt werden können.



Prozesse synchronisieren = partielle zeitliche Ordnung herstellen

Mehrbenutzerbetrieb mit Zugriff auf gemeinsame Daten

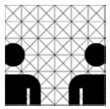
Um Probleme durch unerwünschte Verzahnung nebenläufiger Zugriffe (s. Beispiel Kontoführung) zu vermeiden, werden atomare Aktionen zu größeren Einheiten geklammert - diese nennt man *"Transaktionen"*.

Eine Transaktion ist eine Folge von Aktionen (Anweisungen), die (u.a.) *ununterbrechbar* ausgeführt werden soll.

Da Fehler während einer Transaktion auftreten können, muss eine Trans-aktionsverwaltung dafür sorgen, dass unvollständige Transaktionen ggf. zurückgenommen werden können.

Befehle für Transaktionsverwaltung:

- ***begin of transaction* (BOT)** — ***Beginn der Anweisungsfolge einer Transaktion***
- ***commit* / *end of TA* (EOT)** — ***Einleitung des Endes einer Transaktion, Änderungen der Datenbasis werden festgeschrieben***
- ***abort* bzw. *rollback*** — ***Abbruch der Transaktion, Datenbasis wird in den Zustand vor der Transaktion zurückversetzt***



Eigenschaften von Transaktionen

„ACID-Paradigma“ steht für die vier wichtigsten Eigenschaften:

Atomicity (Atomarität)

Eine Transaktion wird als unteilbare Einheit behandelt ("alles-oder-nichts").

Consistency (Konsistenz)

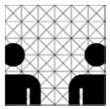
Eine Transaktion hinterlässt nach (erfolgreicher oder erfolgloser) Beendigung eine konsistente Datenbasis.

Isolation

Nebenläufig ausgeführte Transaktionen beeinflussen sich nicht gegenseitig.

Durability (Dauerhaftigkeit)

Eine erfolgreich abgeschlossene Transaktion hat dauerhafte Wirkung auf die Datenbasis, auch bei Hardware- und Software-Fehlern (nach EOT).

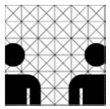


Problembereiche bei Mehrbenutzerbetrieb auf gemeinsamen Daten

Synchronisation mehrerer nebenläufiger Transaktionen:

- Bewahrung der intendierten Semantik einzelner Transaktionen
- Sicherung von Rücksetzmöglichkeiten im Falle von Abbrüchen
- Vermeidung von Schneeballeffekten beim Rücksetzen
- Protokolle zur Sicherung der Serialisierbarkeit
- Behandlung von Verklemmungen

Wir können hier nur einige Themen anschneiden, Vertiefung in weiterführenden Lehrveranstaltungen - insbesondere in den Vorlesungen *DIS (Datenbanken&Info.syst.)* und *VIS (Verteilte Systeme).*



C1.2: Prozesssychronisation und -kommunikation
Synchronisation bei Mehrbenutzerbetrieb

Nebenläufige Transaktionsausführungen sind serialisierbar, gdw. ihr Ergebnis dem irgendeiner (!) seriellen Ausführungsreihenfolge entspricht

Synchronisationsproblem : *Welche „verzahnt sequentielle“ Ausführung nebenläufiger Transaktionen entspricht der Wirkung einer unverzahnten ("seriellen") Hintereinanderausführung der Transaktionen?*

Konfliktursache: *read* und *write* von Prozessen i und k auf dasselbe Datum A:

$read_i(A)$	$read_k(A)$	Reihenfolge irrelevant, kein Konflikt
$read_i(A)$	$write_k(A)$	Reihenfolge muss spezifiziert werden, Konflikt
$write_i(A)$	$read_k(A)$	analog
$write_i(A)$	$write_k(A)$	Reihenfolge muss spezifiziert werden, Konflikt

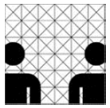
Serialisierbarkeitsgraph:

Knoten = atomare Operationen (read, write)

Kanten = Ordnungsbeziehung (Operation i vor Operation k)

Serialisierbarkeitstheorem:

Eine partiell geordnete Menge nebenläufiger Operationen ist genau dann serialisierbar, wenn der Serialisierungsgraph zyklenfrei ist.



Beispiel für nicht(?) serialisierbare Historie

T1	T2
BOT	
read(A)	
write(A)	
	BOT
	read(A)
	write(A)
	read(B)
	write(B)
	commit
read(B)	
write(B)	
commit	

verzahnte Historie

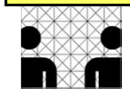
T1	T2
BOT	
read(A)	
write(A)	
read(B)	
write(B)	
commit	
	BOT
	read(A)
	write(A)
	read(B)
	write(B)
	commit

mögliche Serialisierung 1

T1	T2
	BOT
	read(A)
	write(A)
	read(B)
	write(B)
	commit
BOT	
read(A)	
write(A)	
read(B)	
write(B)	
commit	

mögliche Serialisierung 2

Der Effekt dieser Verzahnung entspricht keiner der 2 möglichen Serialisierungen: *T1 vor T2* oder *T2 vor T1*: d.h. die Historie ist nicht serialisierbar !



Synchronisation durch *Sperren*

Viele Transaktions-Scheduler verwenden Sperranweisungen zur Erzeugung konfliktfreier Abläufe paralleler Transaktionen:

- **Sperrmodus S („shared“, read lock, Lesesperre)**

Wenn Transaktion T_i eine S-Sperre für ein Datum A besitzt, kann T_i *read(A)* ausführen. Mehrere Transaktionen können gleichzeitig eine S-Sperre für dasselbe Objekt A besitzen.

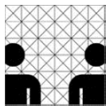
- **Sperrmodus X („exclusive“, write lock, Schreibsperre)**

Nur eine einzige Transaktion , die eine X-Sperre für A besitzt, darf *write(A)* ausführen.

Verträglichkeit der Sperren untereinander:

(NL = no lock, keine Sperrung)

	NL	S	X
S	ok	ok	-
X	ok	-	-



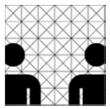
Beispiel für Sperrverzahnung

Beispiel:

T1: Modifikation von A und B (z.B. Umbuchung)

T2: Lesen von A und B (z.B. Addieren der Salden)

t	T1	T2	
	BOT		
	lockX(A)		
	read(A)		
	write(A)		
		BOT	
		lockS(A)	*T2 muss warten*
	lockX(B)		
	read(B)		
	unlockX(A)		*T2 wecken*
		read(A)	
		lockS(B)	*T2 muss warten*
	write(B)		
	unlock(B)		*T2 wecken*
		read(B)	
	commit		
		unlockS(A)	
		unlockS(B)	
		commit	



Mögliches Problem dabei: *Verklemmungen* (Deadlocks)

Sperrbasierte Synchronisationsmethoden können (unvermeidbar) zu Verklemmungen führen: z.B. gegenseitiges Warten auf Freigabe von Sperren

Beispiel wie eben:

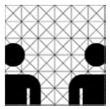
T1: Modifikation von A und B (z.B. Umbuchung)

T2: Lesen von B und A (z.B. Addieren der Salden)

Transaktionsablauf nur leicht modifiziert:

T1	**T2**	
BOT		
lockX(A)		
	BOT	
	lockS(B)	
	read(B)	
read(A)		
write(A)		
lockX(B)		*T1 muss auf T2 warten*
	lockS(A)	*T2 muss auf T1 warten*

=> Deadlock !



Strategien für den Umgang mit (potentiellen) Deadlocks

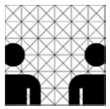
Variante 1. Vermeiden von Deadlocks – z.B. durch

Variante 1a: Preclaiming –

Vorab-Anforderung *aller* Sperren

Beginn einer Transaktion erst nachdem die für diese Transaktion insgesamt erforderlichen Sperren erfolgt sind
(-> „2-Phasen-Sperren“ / „*2-phase-locking*“, 2PL)

Problem: Wie vorab die erforderlichen Sperren erkennen?



Variante 1a: „Zwei-Phasen-Sperrprotokoll“

(Englisch: „Two-phase locking“, 2PL)

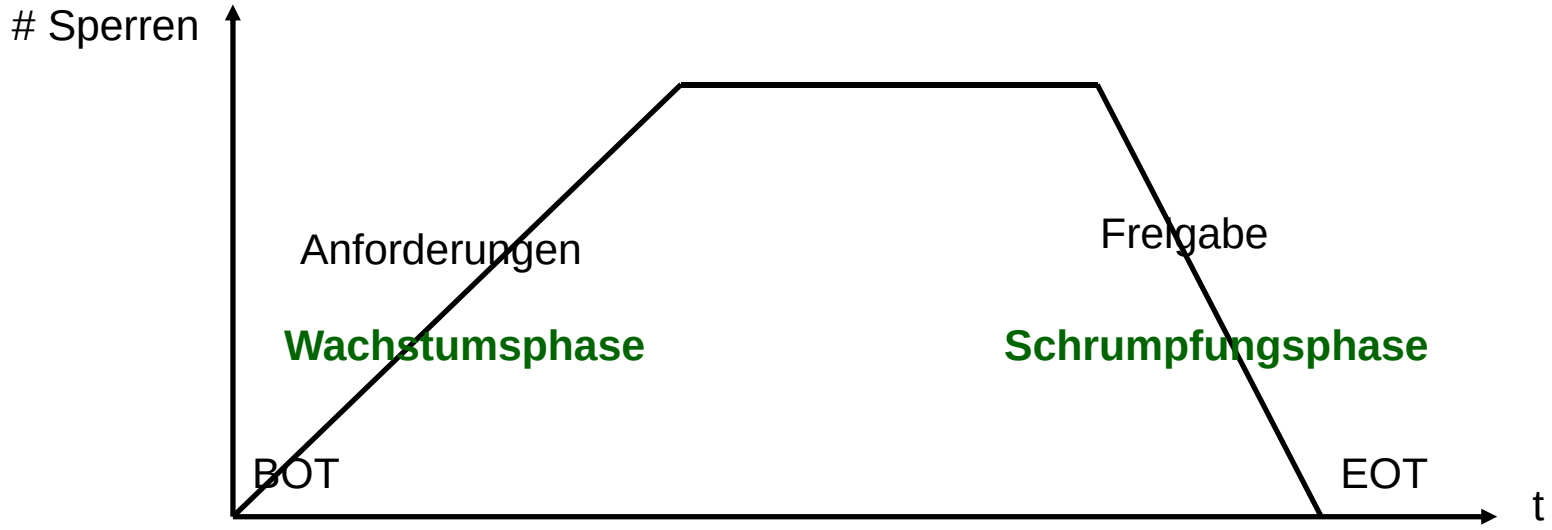
Das 2PL-Protokoll gewährleistet die Serialisierbarkeit von Transaktionen. Für jede individuelle Transaktion muss gelten:

1. **Jedes von einer Transaktion betroffene Objekt muss vor Beginn des Zugriffs von der Transaktion entsprechend gesperrt werden.**
2. **Eine Transaktion fordert eine Sperre, die sie schon einmal besessen hat, niemals wieder erneut an.**
3. **Eine Transaktion muss – bei jedem Zugriff – so lange warten, bis sie alle erforderlichen Sperren entsprechend der Verträglichkeitstabelle erhalten kann.**
5. **Spätestens wenn die erste Sperre frei gegeben wurden, darf keine neue mehr angefordert werden**
4. **Spätestens bei EOT (Transaktionsende) muss eine Transaktion alle ihre Sperren zurück geben.**

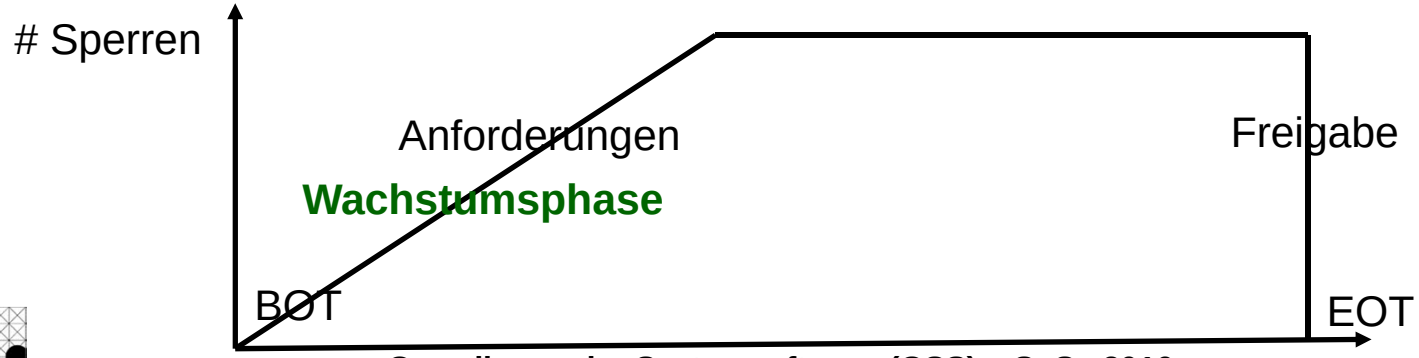
D.h.: Jede Transaktion durchläuft 2 Phasen:

- **in der Wachstumsphase werden Sperren angefordert, aber nicht freigegeben**
- **in der Schrumpfungsphase werden Sperren freigegeben, aber nicht mehr angefordert**

Variante 1a (2PL) graphisch dargestellt:



Verschärfung zum „strengen 2PL-Protokoll" zur Vermeidung nicht rücksetzbarer Abläufe: Keine Schrumpfungsphase, alle Sperren werden bei EOT freigegeben.



Deadlock-Vermeidungsstrategien

Variante 1b: Zeitstempel - Verfahren

Transaktionen werden durch Zeitstempel

(z.B. Zeit des BOT) **priorisiert. - Beispiel:**

T_1 hält eine exklusive (!) Sperre auf A – dann kommt

T_2 und fordert auch eine Sperre auf A.

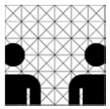
Allgemeine Strategien für parallele Transaktionen, die auf dasselbe A zugreifen wollen:

- **„*wound-wait*“:**

  ***Abbruch* von T_1, falls T_1 *jünger* ist als T_2, sonst wartet T_2**

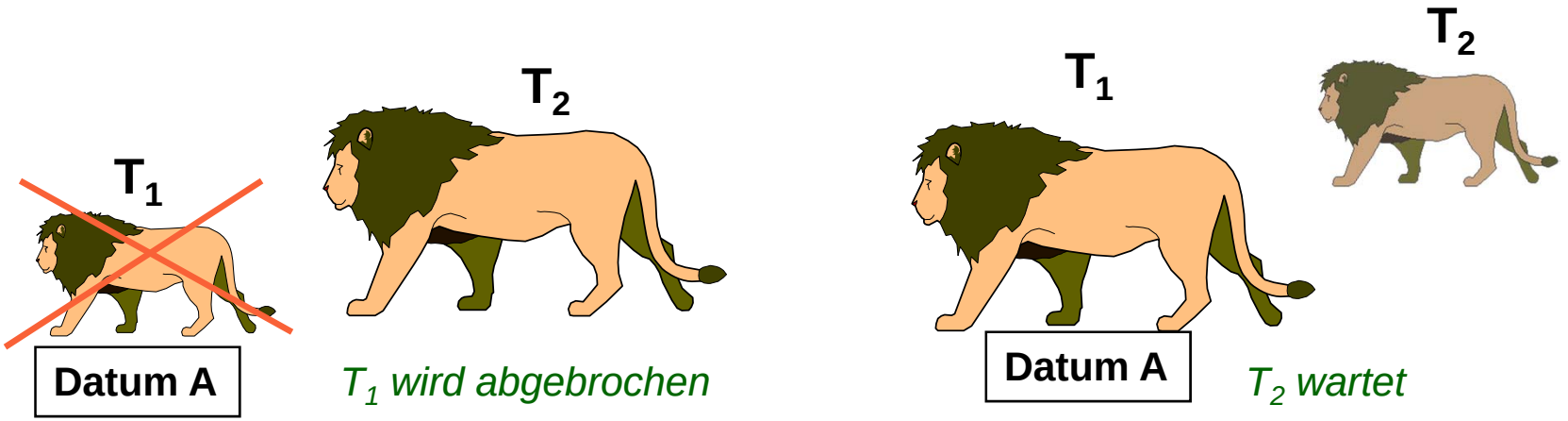
- **„*wait-die*“:**

  ***Abbruch* von T_2, wenn T_2 *jünger* ist als T_1, sonst wartet T_2**

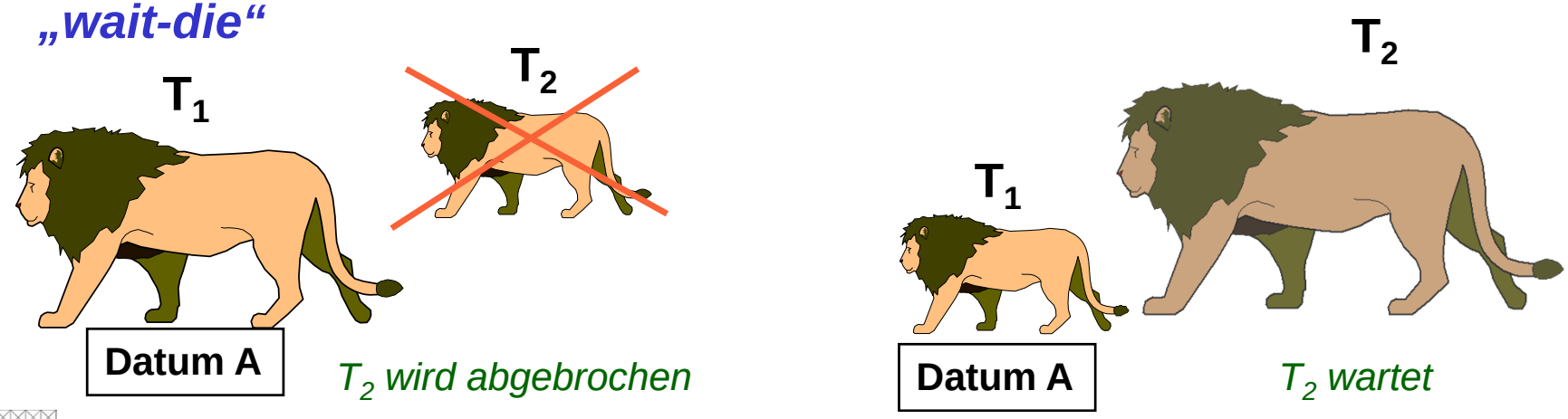


Variante 1b einmal bildlich dargestellt...

„wound-wait“



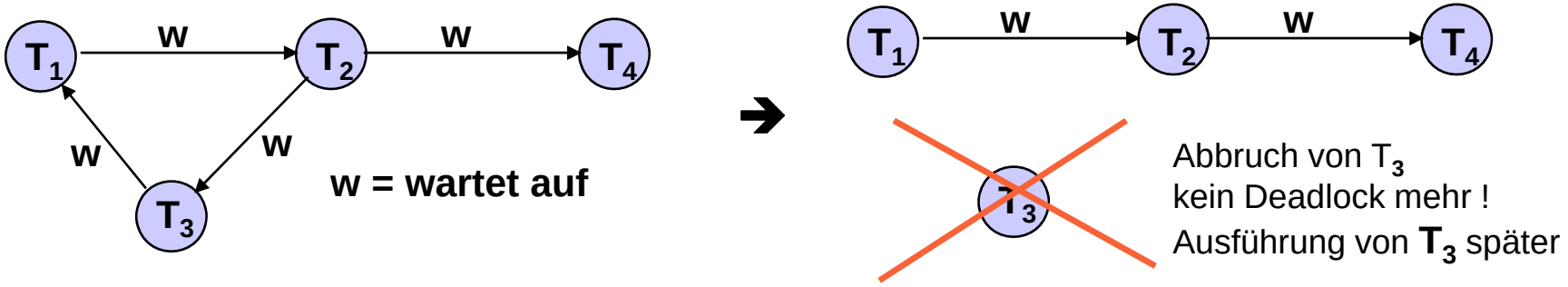
„wait-die“



Alternative: Deadlocks zunächst zulassen... - und dann:

Variante 2. Erkennen von Deadlocks **– z.B. durch**

Wartegraph mit Zyklen – sowie danach ...



... Beseitigen der Verklemmung durch Zurücksetzen bzw. zeitliches Verschieben einer „geeigneten“ Transaktion (z.B. der jüngsten)

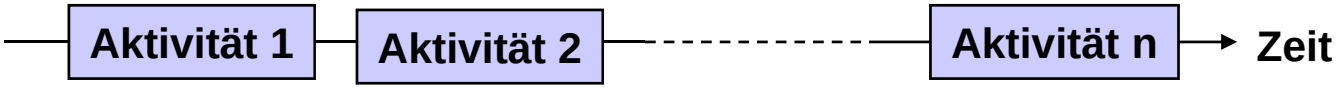
C 1.3: Prozesse in der Informatik

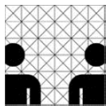
Allgemein:

Ein Prozess ist eine Folge von Vorgängen und Systemzuständen.

Informatik:

Prozess	**sequentieller Ablauf von Aktivitäten**
Zustand eines Prozesses	**Werte expliziter und impliziter Prozessgrößen, qualitative Aussagen über Prozessgrößen**
(atomare) Aktivität	**Veränderung eines Zustands durch (unteilbaren) Vorgang**





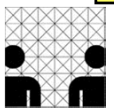
Verallgemeinerungen

Klassische Annahmen für Programmausführung:

- **Es geht um Programme, die auf Rechnern ausgeführt werden**
- **Ein Rechner führt genau ein Programm aus**
- **Ein Programm wird auf genau einem Rechner ausgeführt**
- **Ein Programm erfüllt seine Funktion unabhängig von Start-zeitpunkt und benötigter Bearbeitungszeit**

Fortlassen dieser Annahmen ergibt:

- **Es geht um Aktivitäten in Prozessen**
- **Prozesse können nebenläufig *(concurrent)* sein**
- **Prozesse können verteilt *(distributed)* sein**
- **Prozesse können echtzeitabhängig *(real-time dependent)* sein**



Nebenläufig vs. parallel

"Aktivitäten sind nebenläufig":

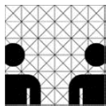
- **Die Aktivitäten können von mehreren Prozessoren ausgeführt werden**
- **Die Aktivitäten können in beliebiger Folge sequentiell von einem Prozessor ausgeführt werden**

"Aktivitäten werden parallel ausgeführt":

- **Aktivitäten werden auf mehreren Prozessoren zeitüberlappend ausgeführt**
- **Parallelität ist Spezialfall von Nebenläufigkeit**

"Aktivitäten werden quasi-parallel ausgeführt":

- **Aktivitäten werden auf einem Prozessor sequentiell aber ohne vorgeschriebene Reihenfolge ausgeführt**

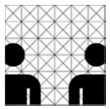


Nichtdeterminismus und Determiniertheit

Bei nebenläufigen Prozessen laufen Aktivitäten in *nicht-deterministischer*, d.h. beliebiger, nicht vorher bestimmter Reihenfolge ab.

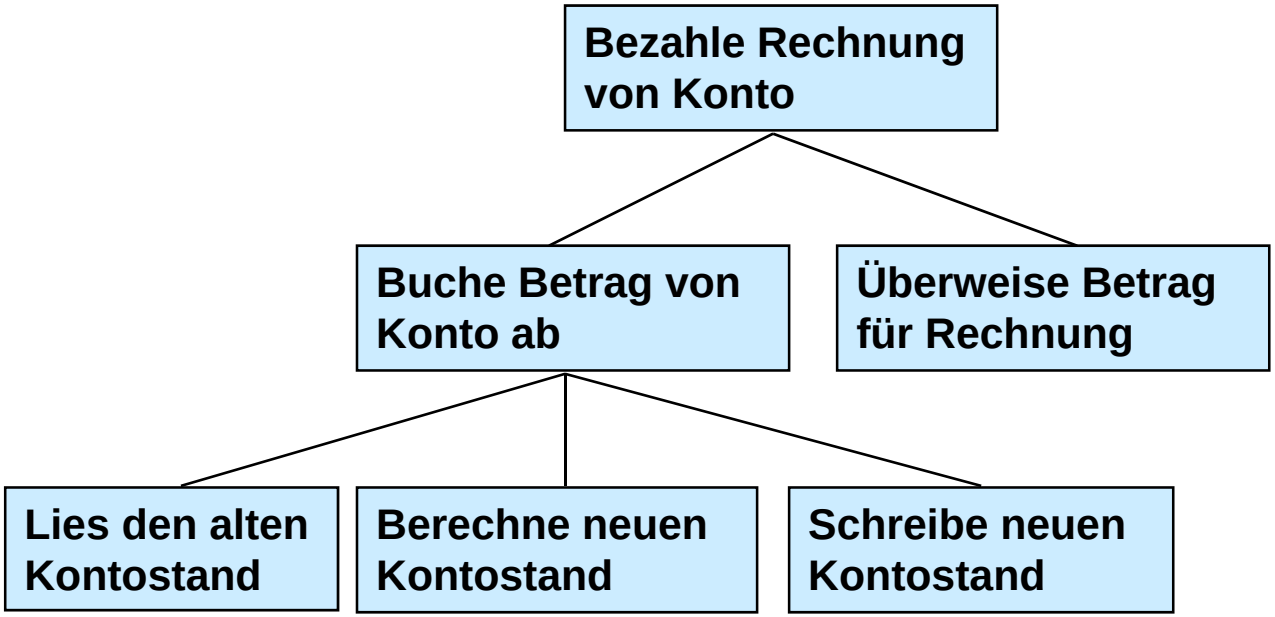
Man ist jedoch i.A. daran interessiert, dass nebenläufige Prozesse ein *determiniertes* Ergebnis haben, egal wie verzahnt sie ausgeführt werden.

Auch *nicht-determinierte* Ergebnisse können gefragt sein, z.B. Bestimmen des kürzesten Pfades in einem Graphen durch nebenläufige Prozesse: Im Falle von mehreren kürzesten Pfaden ist es egal, welcher Prozess das Ergebnis liefert



Unteilbarkeit

Aktivitäten eines Prozesses können je nach Abstraktionsebene in gröbere oder feinere Einheiten zerlegt werden.



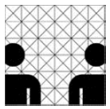
Bei nebenläufigen Prozessen kann es wichtig sein, unteilbare (atomare) Einheiten zu spezifizieren - siehe Transaktionskonzept.

Verzahnung von Zuweisungen

Die nebenläufige Ausführung von zwei Zuweisungen kann zu unerwünschten Ergebnissen führen, wenn die Verzahnung auf der Ebene von Maschinenbefehlen erfolgt:

Zuweisungsebene		
Prozess 1	**Prozess 2**	**x**
		i
	x := x + 1	i+1
x := x + 1		i+2

Befehlsebene				
Prozess 1	**Prozess 2**	**x**	**r1**	**r2**
		i	?	?
load x,r1		i	i	?
incr r1		i	i+1	?
	load x,r2	i	i+1	i
	incr r2	i	i+1	i+1
store r1,x		i+1	i+1	i+1
	store r2,x	i+1	i+1	i+1

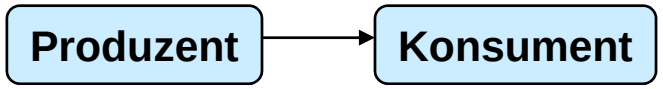


Kooperation und Konkurrenz

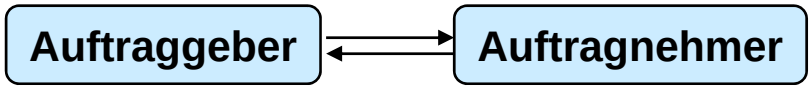
Nebenläufige Prozesse sind nur dann interessant (für uns), wenn sie voneinander *abhängig* sind.

1. Grundform der Abhängigkeit: Kooperation

Durch Kooperation werden gemeinsame Ziele verfolgt (und erreicht).



Produzenten/Konsumenten-System (producer/consumer system)



Auftraggeber/Auftragnehmer-System (client/server system)

2. Grundform der Abhängigkeit: Konkurrenz

Prozesse behindern sich durch Nutzung begrenzter Ressourcen.

Bsp.: Wettbewerb um Betriebsmittel

Synchronisation und Kommunikation

Synchronisation = zeitliche Koordination von kooperierenden und konkurrierenden Prozessen

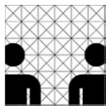
Beispiele:

- Konsument greift erst dann auf Daten zu, wenn Produzent fertig ist.
- Prozess 1 benutzt Drucker erst wenn Prozess 2 Drucker freigegeben hat

Kommunikation = Informationsaustausch zwischen Prozessen

Beispiele:

- Zugriffe auf gemeinsamen Datenbereich
- Datentransport von einem Prozess zum anderen



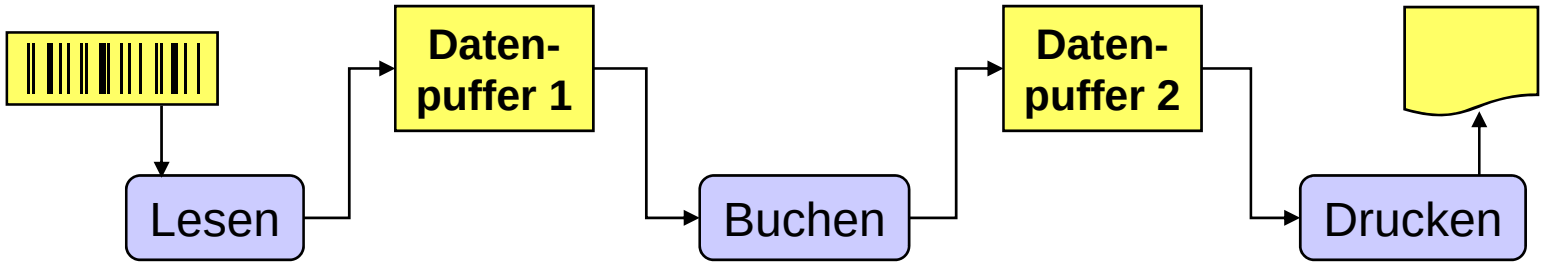
Einseitige Synchronisation

Einseitige Synchronisation von zwei Aktivitäten A1 und A2 mit der Relation

A1→ A2 „A1 geschieht vor A2“

A1→ A2 beeinflusst nur die Aktivität A2

Beispiel: Einfaches Buchungssystem (Registrierkasse)



Ablegen auf Datenpuffer 1 → Abnehmen von Datenpuffer 1

Ablegen auf Datenpuffer 2 → Abnehmen von Datenpuffer 2

Mehrseitige Synchronisation

Mehrseitige Synchronisation zweier Aktivitäten A1 und A2 mit der Relation

A1 ↔ A2 „A1 und A2 sind gegenseitig ausgeschlossen“

Die Relation ↔ ist symmetrisch aber nicht transitiv.

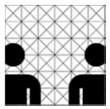
Aktivitäten (Anweisungen), deren Ausführung einen gegenseitigen Ausschluss erfordern, heißen „kritische Abschnitte“.

Beispiel: Lese- und Schreibzugriffe auf eine Variable

Schreiben durch Prozess 1 ↔ Schreiben durch Prozess 2

Schreiben durch Prozess 1 ↔ Lesen durch Prozess 2

Lesen durch Prozess 1 ↔ Schreiben durch Prozess 2



Beidseitiger Ausschluss mit Schlossvariablen 1. Version

Idee: Schlossvariable *locked* ist Schlüssel für kritischen Abschnitt

locked* = *false **Schlüssel vorhanden, kritischer Abschnitt offen**

locked* = *true **Schlüssel fehlt, kritischer Abschnitt gesperrt**

```
public class lock {
boolean locked = false;
public boolean isLocked () {return locked;}
public void setLocked (lockValue) {
locked = lockValue;}
}
```

Gegenseitiger Ausschluss funktioniert so nicht, weil Lesen und Schreiben der Schlossvariablen nicht ununterbrechbar sind - (Alternative s.u.)

```
class process1 extends thread {
...
public void run (lock commonLock) {
...
while (commonLock.isLocked ()) { };
commonLock.setLocked (true);
<Aktivität1>
commonLock.setLocked (false);
... }
}
```

```
class process2 extends thread {
...
public void run (lock commonLock) {
...
while (commonLock.isLocked ()) { };
commonLock.setLocked (true);
<Aktivität2>
commonLock.setLocked (false);
... }
}
```

Beidseitiger Ausschluss mit Schlossvariablen 2. Version

Idee:

- **Jeder Prozess hat eigene Schlossvariable, sichtbar auch für anderen Prozess**
- **Gemeinsame Prioritätsvariable löst Vorrangproblem**
- **Betreten des kritischen Abschnittes, wenn Schlossvariable des anderen Prozesses dies zulässt und die Prioritätsvariable den Prozess favorisiert**

```
...
lock1.setLocked (true);
while (lock2.isLocked ()) {
      if (favorit.getValue () <> 1){
            lock1.setLocked (false);
            while (favorit.getValue () <> 1) { };
            lock1.setLocked (true);
      }
}
<Aktivität1>
favorit.setValue (2);
lock1.setLocked (false);
...
```

→ ***Prozess meldet Eintrittswunsch***

→ ***falls anderer Prozess auch Eintrittswunsch gemeldet hat, entscheidet Prioritätsvariable*** favorit ***über Vorrang***

→ ***kritischer Abschnitt***

→ ***Prozess meldet Austritt und gibt anderem Prozess Vorrang für nächsten Eintritt***

Semaphore

Semaphor ist Zähler mit Prozessverwaltungskompetenz: Statt aktiv zu warten wird ein Prozess durch ein Semaphor ggf. blockiert und deblockiert.

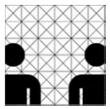
Traditionelle Operationen (Dijkstra 68):

P (passeeren, passieren)	**bei Zähler = 0 Prozess blockieren, vor Passage dekrementieren**
V (vrijgeven, freigeben, verlassen)	**Zähler inkrementieren, wartenden Prozess deblockieren**

Grundsätzliche Verwendung für beidseitigen Ausschluss:

```
s = new Semaphore(1)

class P1 extends thread {          class P2 extends thread {
...                                ...
s.P ();                            s.P ();
<kritischer Abschnitt>             <kritischer Abschnitt>
s.V ();                            s.V ();
...                                ...
}                                  }
```



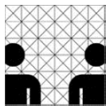
C 1.4: Prozesse & Threads
Nebenläufiger Prozesse

Vorgestellte Synchronisationsmethoden verwenden meist Ausdrucksmöglichkeiten klassischer Programmiersprachen auf niedriger Abstraktionsebene:

- **kritische Abschnitte**
- **Semaphore**
- **Monitore (s.u.)**

Problematisch bei komplexen Synchronisierungsaufgaben!

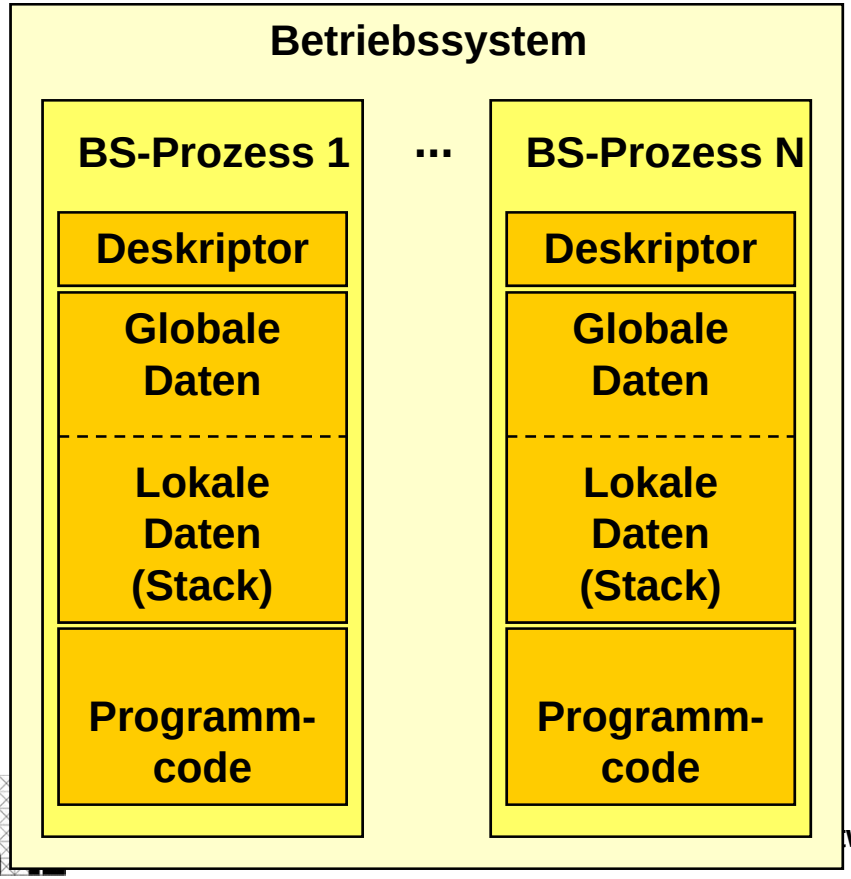
Moderne Programmiersprachen (wie z.B. Java) bieten vorgefertigte Möglichkeiten, nebenläufige Prozesse und Synchronisationsverfahren auf höherer Abstraktionsebene zu definieren.



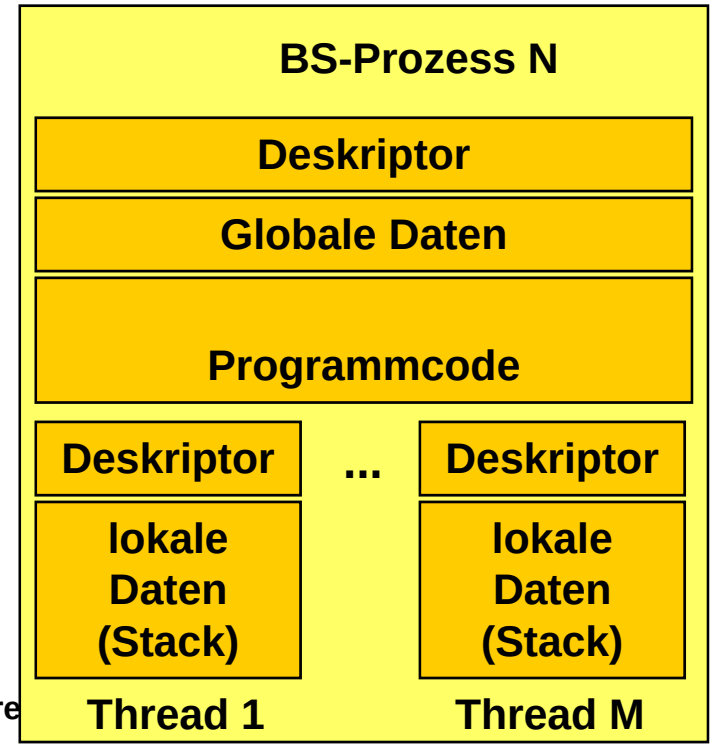
Schwergewichtige und leichtgewichtige Prozesse

Schwergewichtige Prozesse eines Betriebssystems (BS):

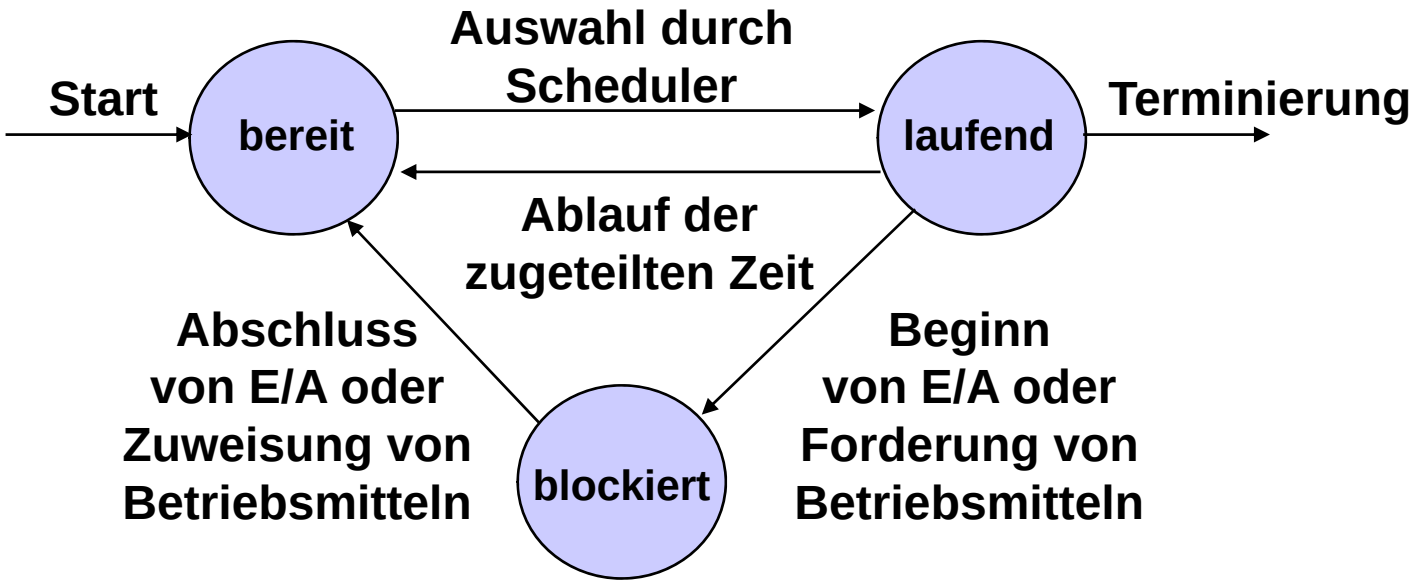
Aufträge mit Ressourcenbedarf



Leichtgewichtige Prozesse (Threads) als Teile eines BS-Prozesses



Prozessorzuteilung durch Scheduler



- Status "bereit" kann mehrere Warteschlangen mit verschiedener Priorität besitzen
- Einordnung von Prozessen nach dem "Round-Robin"-Verfahren