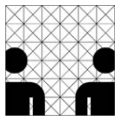
C Nebenläufigkeit und Verteilung

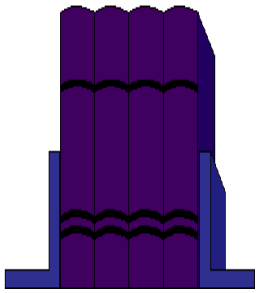


C 1.1: Einführung

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:

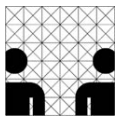
- Anwendungsprobleme
- formale Beschreibung und Analyse
- Architekturen und Entwurf
- Programmierung in Java



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, 8. Auflage, Oldenbourg 2011, 792 S.



Beispiel Kontoführung

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

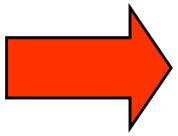
Prozess 2: Zinsgutschrift für Konto A

Umbuchung

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

Zinsgutschrift

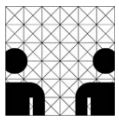
```
read (A, a2)
a2 := a2 * 1.03
write (A, a2)
```



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)	

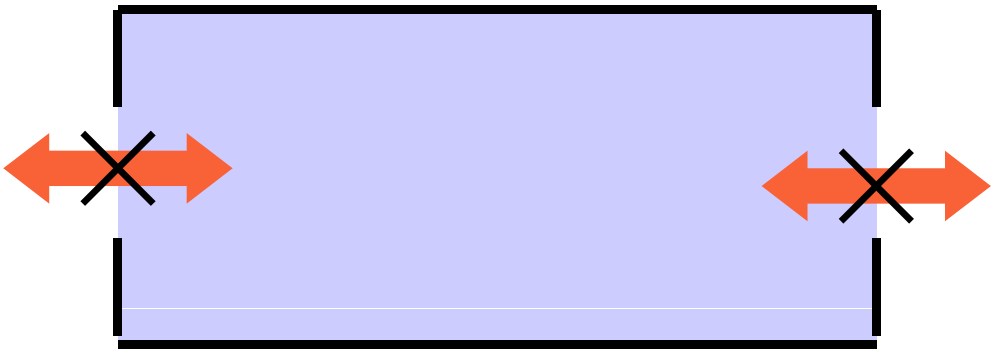
Wo ist die Zinsgutschrift geblieben??



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)
}
```

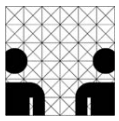


Drehkreuz2:

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

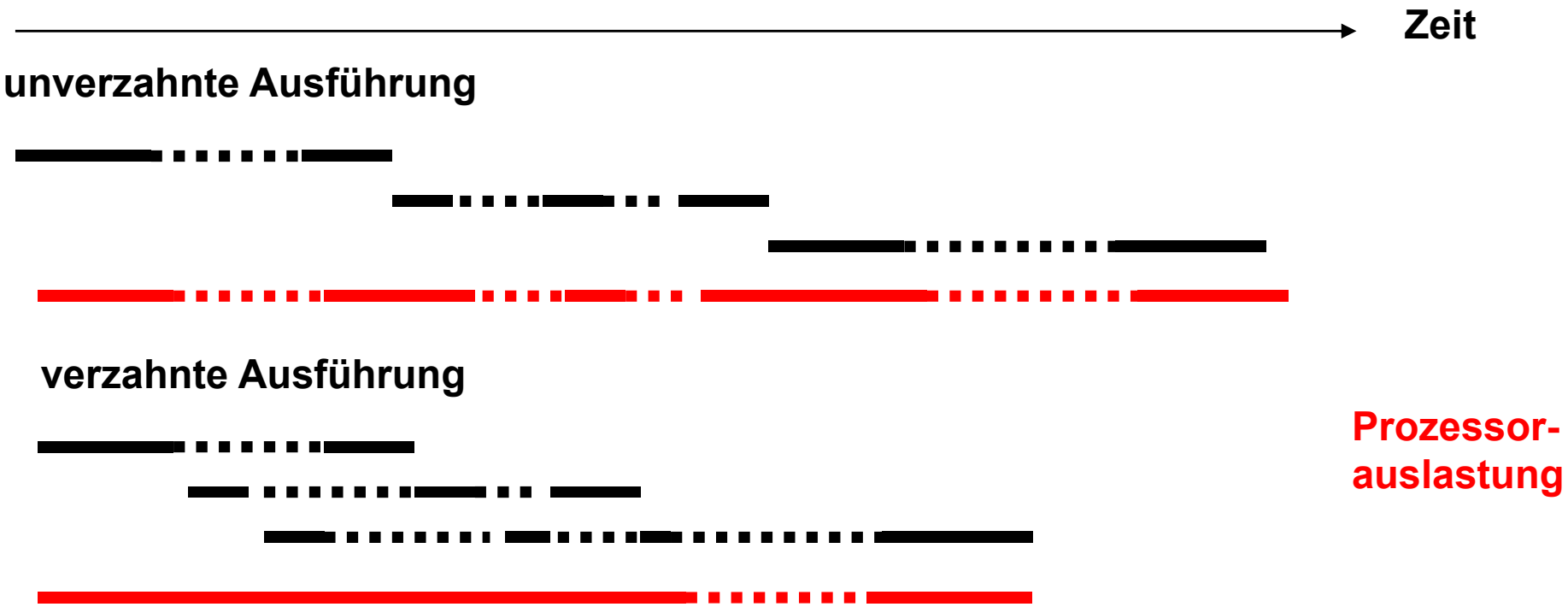
Verzahnte Ausführung der zwei Prozesse Drehkreuz1 und Drehkreuz2 mit Zugriff auf gemeinsamen Counter kann inkorrekte Besucherzahl ergeben!

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

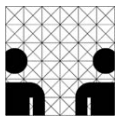


Mehrbenutzersynchronisation

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

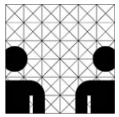
Um Probleme durch unerwünschte Verzahnung nebenläufiger Zugriffe (s. Beispiel Kontoführung) zu vermeiden, werden atomare Aktionen zu größeren Einheiten geklammert - diese nennt man *"Transaktionen"*.

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)

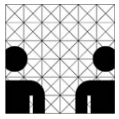
Eine Transaktion hinterlässt nach (erfolgreicher oder erfolgloser) Beendigung eine konsistente Datenbasis.

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).

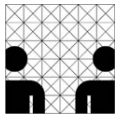


Problembereiche bei Mehrbenutzerbetrieb auf gemeinsamen Daten

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)* 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

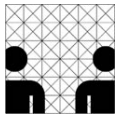
Serialisierbarkeitsgraph:

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

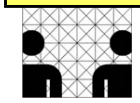
T1	T2
BOT	
read(A)	
write(A)	
read(B)	
write(B)	
commit	
	BOT
	read(A)
	write(A)
	read(B)
	write(B)
	commit

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.

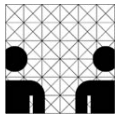
- **Sperrmodus X („exclusive“, write lock, Schreibsperre)**

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	-	-

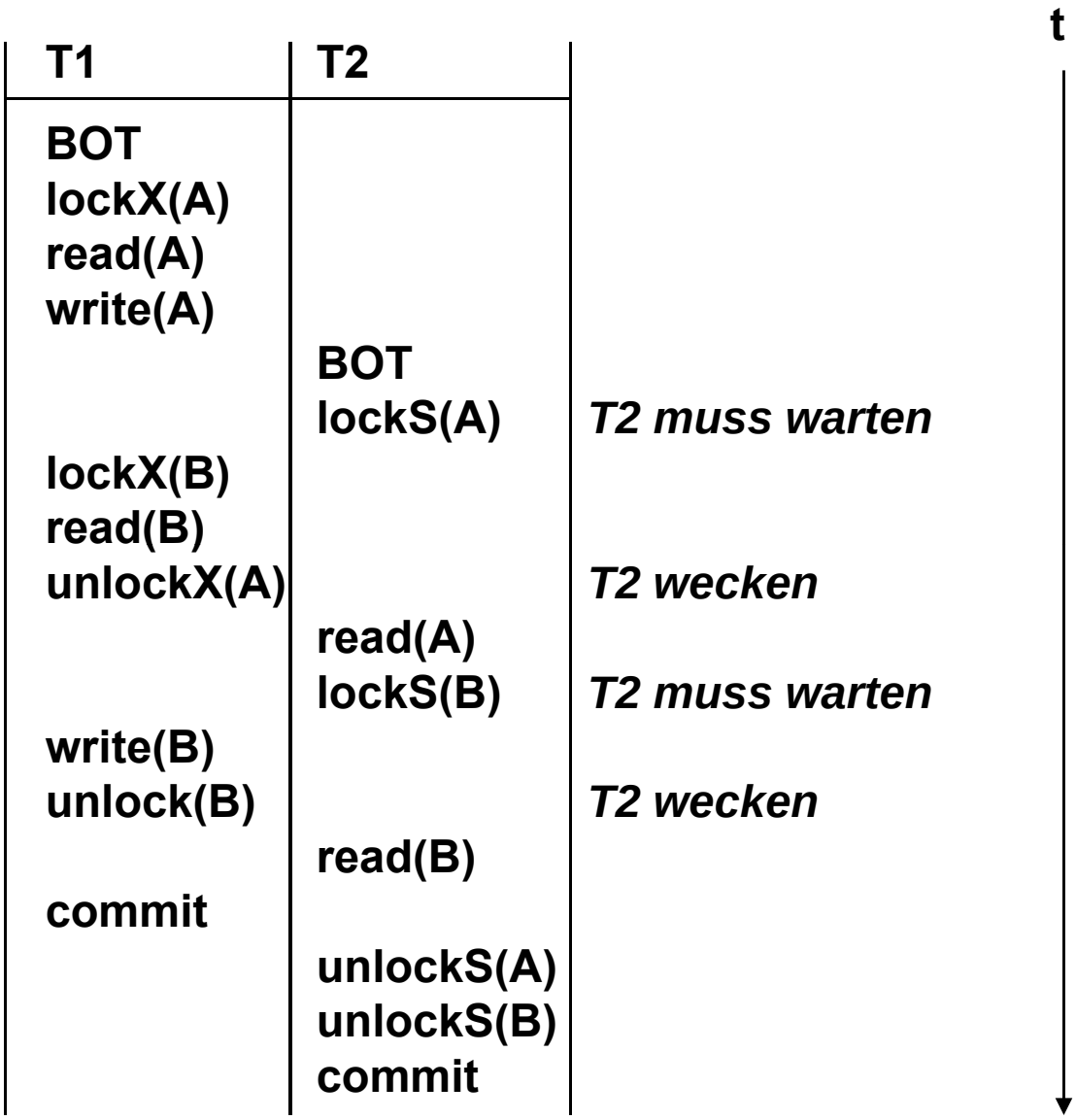


Beispiel für Sperrverzahnung

Beispiel:

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

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



Mögliches Problem dabei: *Verklemmungen* (Deadlocks)

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

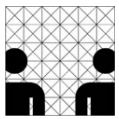
Beispiel wie eben:

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

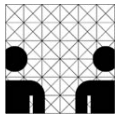
Variante 1. Vermeiden von Deadlocks – z.B. durch

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)

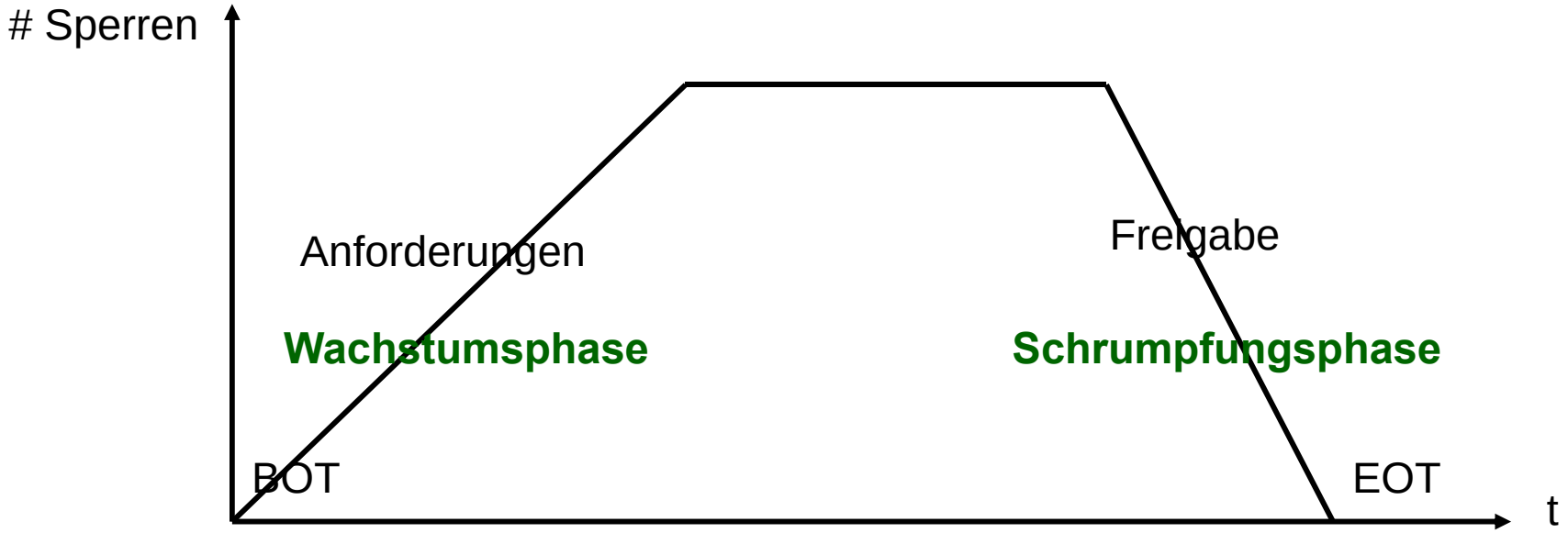
Das 2PL-Protokoll gewährleistet die Serialisierbarkeit von Transaktionen. Für jede individuelle Transaktion muss gelten:

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.**

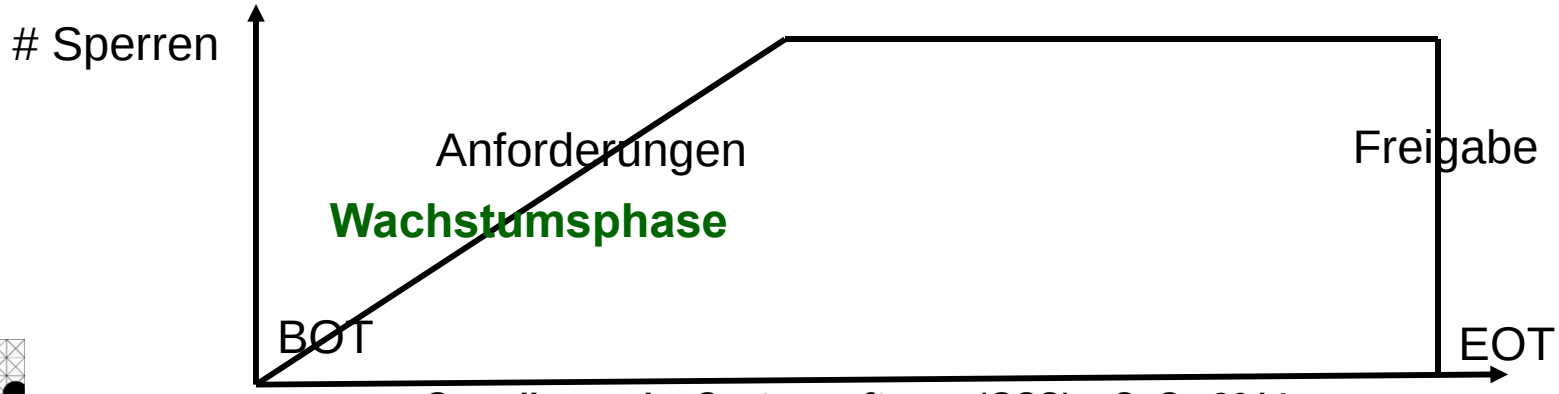
D.h.: Jede Transaktion durchläuft 2 Phasen:

- **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.



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.

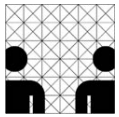
Allgemeine Strategien für parallele Transaktionen, die auf dasselbe A zugreifen wollen:

- **„*wound-wait*“:**

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

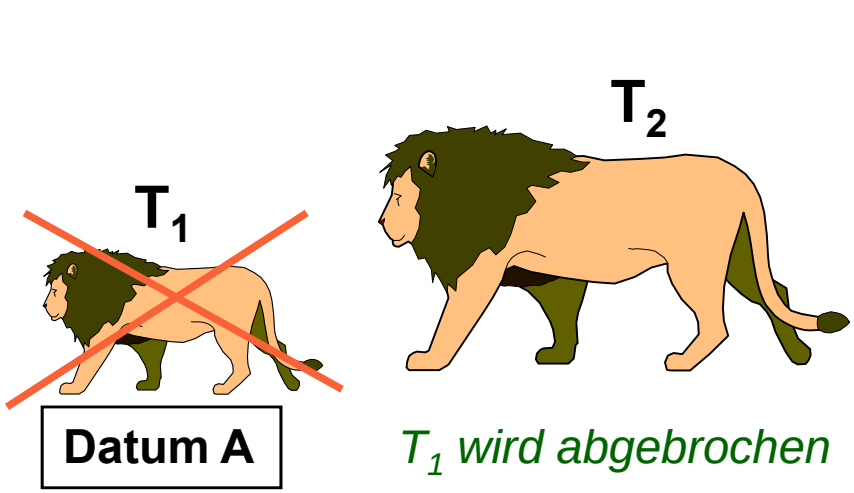
- **„*wait-die*“:**

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

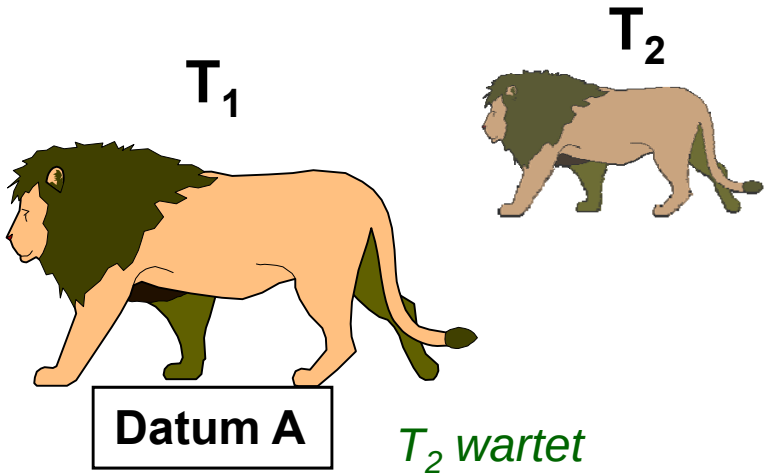


Variante 1b einmal bildlich dargestellt...

„wound-wait“

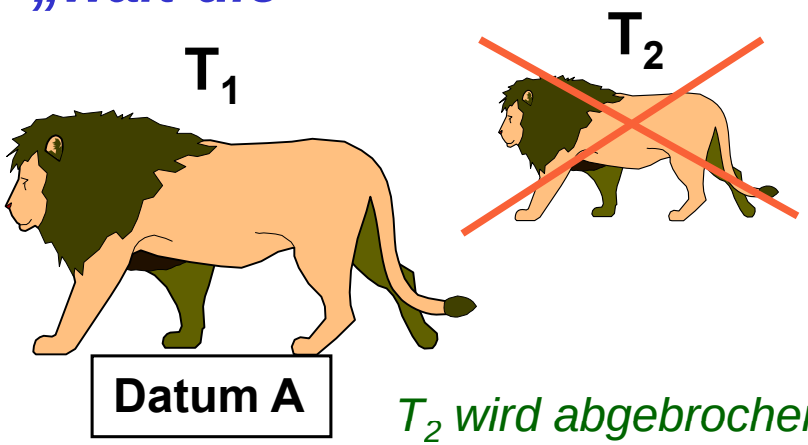


T_1 *wird abgebrochen*

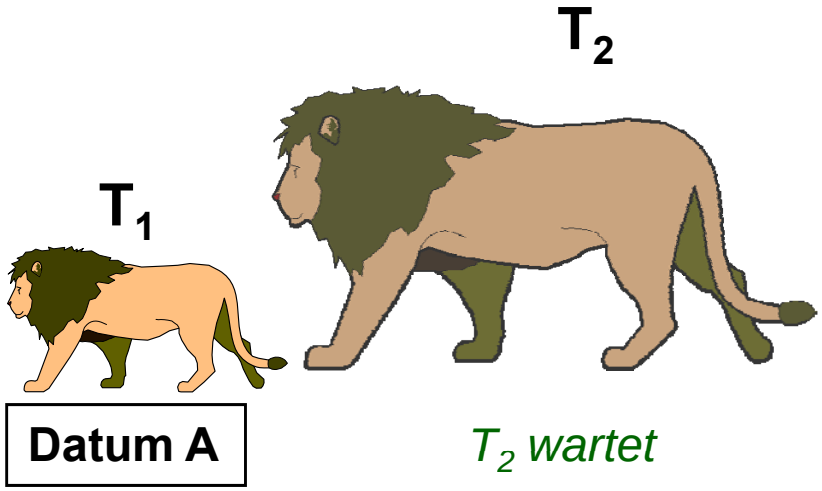


T_2 *wartet*

„wait-die“



T_2 *wird abgebrochen*

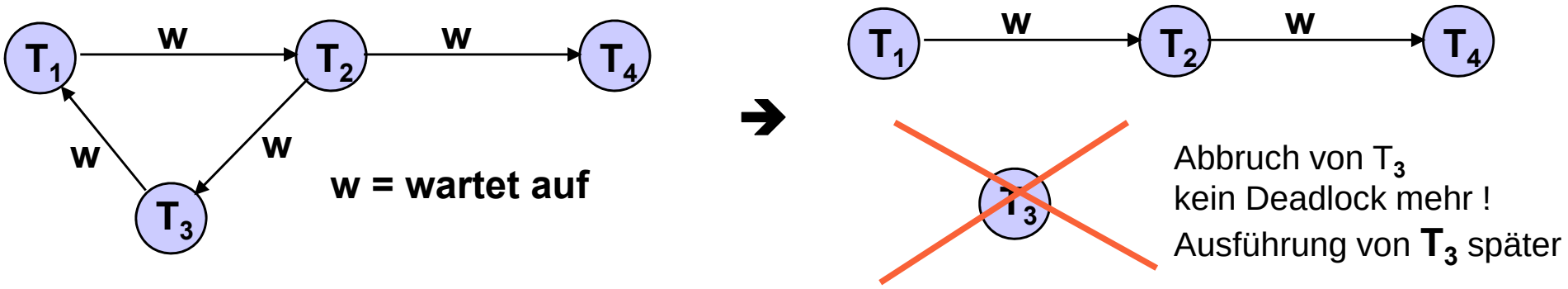


T_2 *wartet*

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)

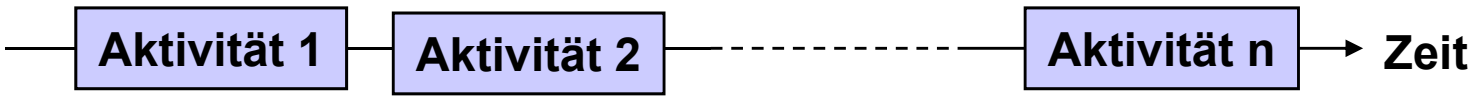
C 1.3: Prozesse in der Informatik

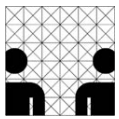
Allgemein:

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**





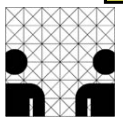
Verallgemeinerungen

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":

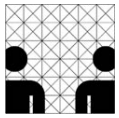
- **Die Aktivitäten können von mehreren Prozessoren ausgeführt werden**
- **Die Aktivitäten können in beliebiger Folge sequentiell von einem Prozessor ausgeführt werden**

"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**

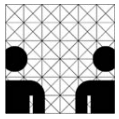


Nichtdeterminismus und Determiniertheit

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

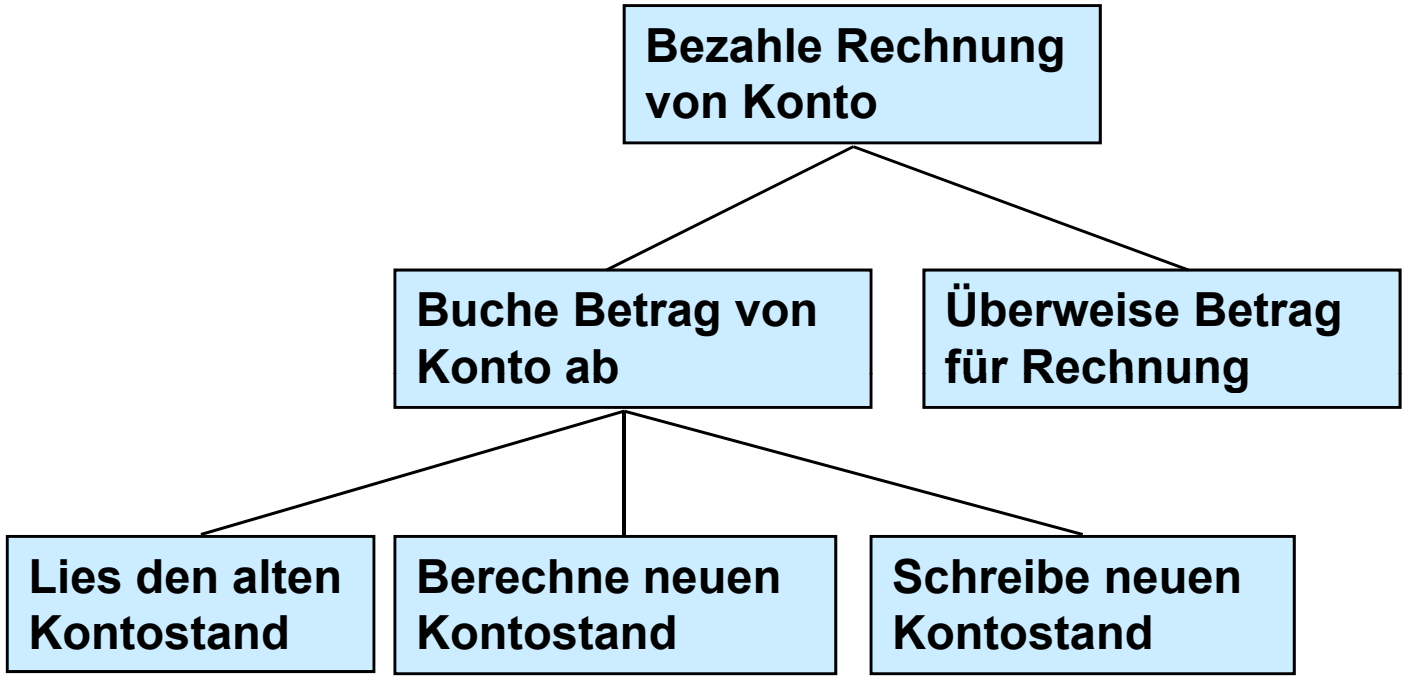
Man ist jedoch i.A. daran interessiert, dass nebenläufige Prozesse ein *determiniertes* Ergebnis haben, egal wie verzahnt sie ausgeführt werden.

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.



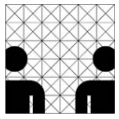
Bei nebenläufigen Prozessen kann es wichtig sein, unteilbare (atomare) Einheiten zu spezifizieren - siehe Transaktionskonzept.

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

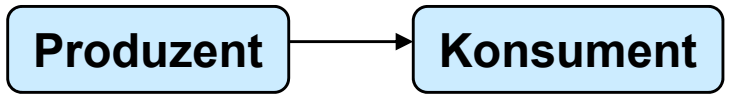


Kooperation und Konkurrenz

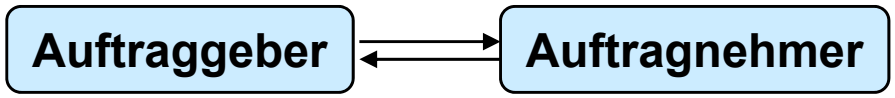
Nebenläufige Prozesse sind nur dann interessant (für uns), wenn sie voneinander *abhängig* sind.

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

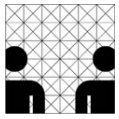
Beispiele:

- 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



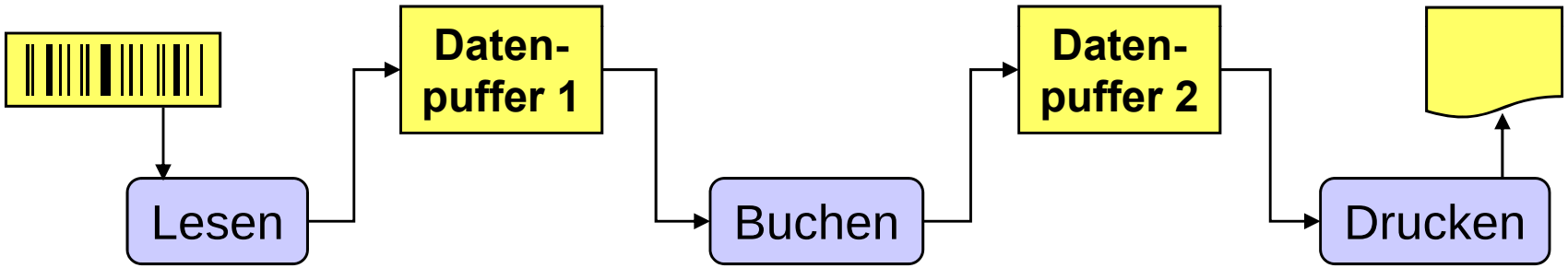
Einseitige Synchronisation

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.

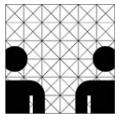
Aktivitäten (Anweisungen), deren Ausführung einen gegenseitigen Ausschluss erfordern, heißen „kritische Abschnitte“.

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.

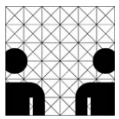
Traditionelle Operationen (Dijkstra 68):

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 ();
...                                ...
}                                  }
```



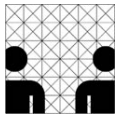
C 1.4: Prozesse & Threads
Nebenläufiger Prozesse

Vorgestellte Synchronisationsmethoden verwenden meist Ausdrucksmöglichkeiten klassischer Programmiersprachen auf niedriger Abstraktionsebene:

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

Problematisch bei komplexen Synchronisierungsaufgaben!

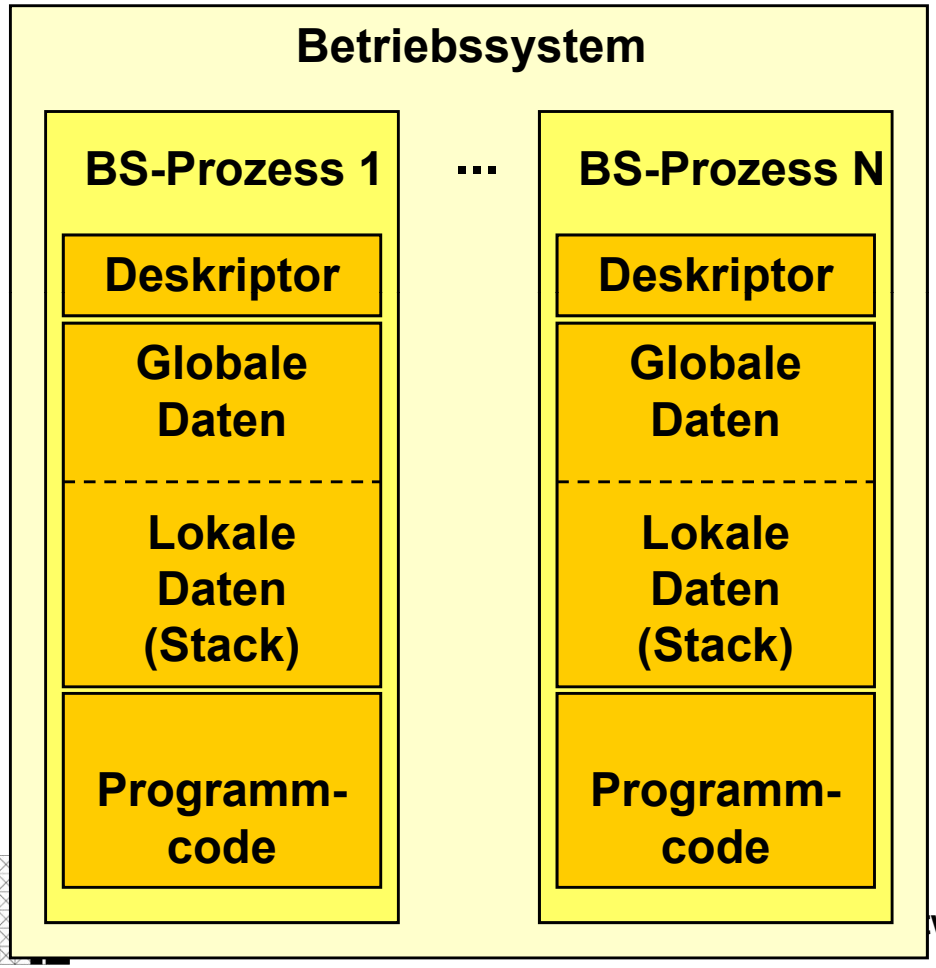
Moderne Programmiersprachen (wie z.B. Java) bieten vorgefertigte Möglichkeiten, nebenläufige Prozesse und Synchronisationsverfahren auf höherer Abstraktionsebene zu definieren.



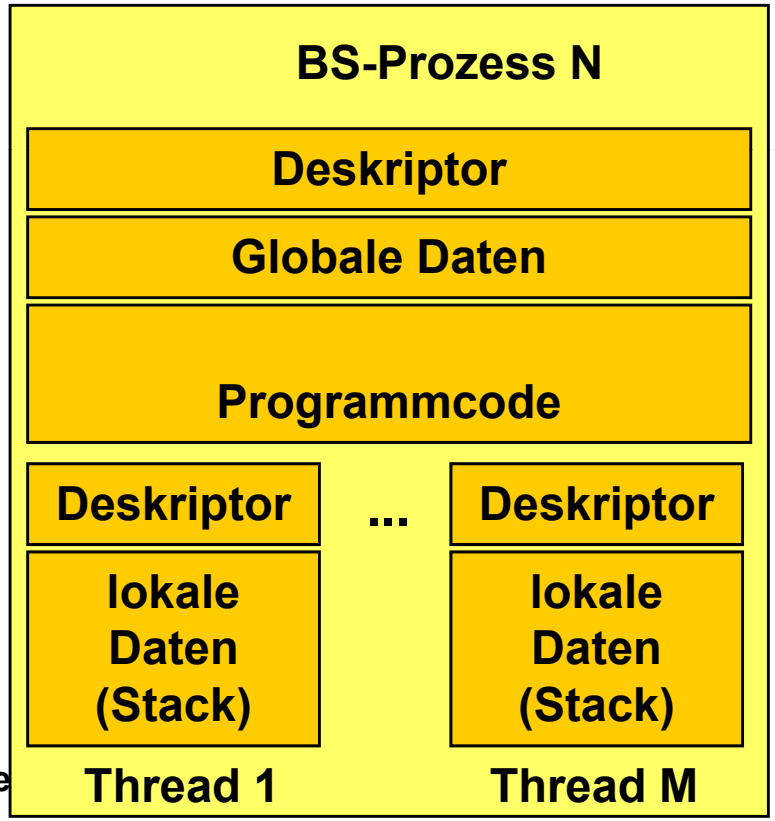
Schwergewichtige und leichtgewichtige Prozesse

Schwergewichtige Prozesse eines Betriebssystems (BS):

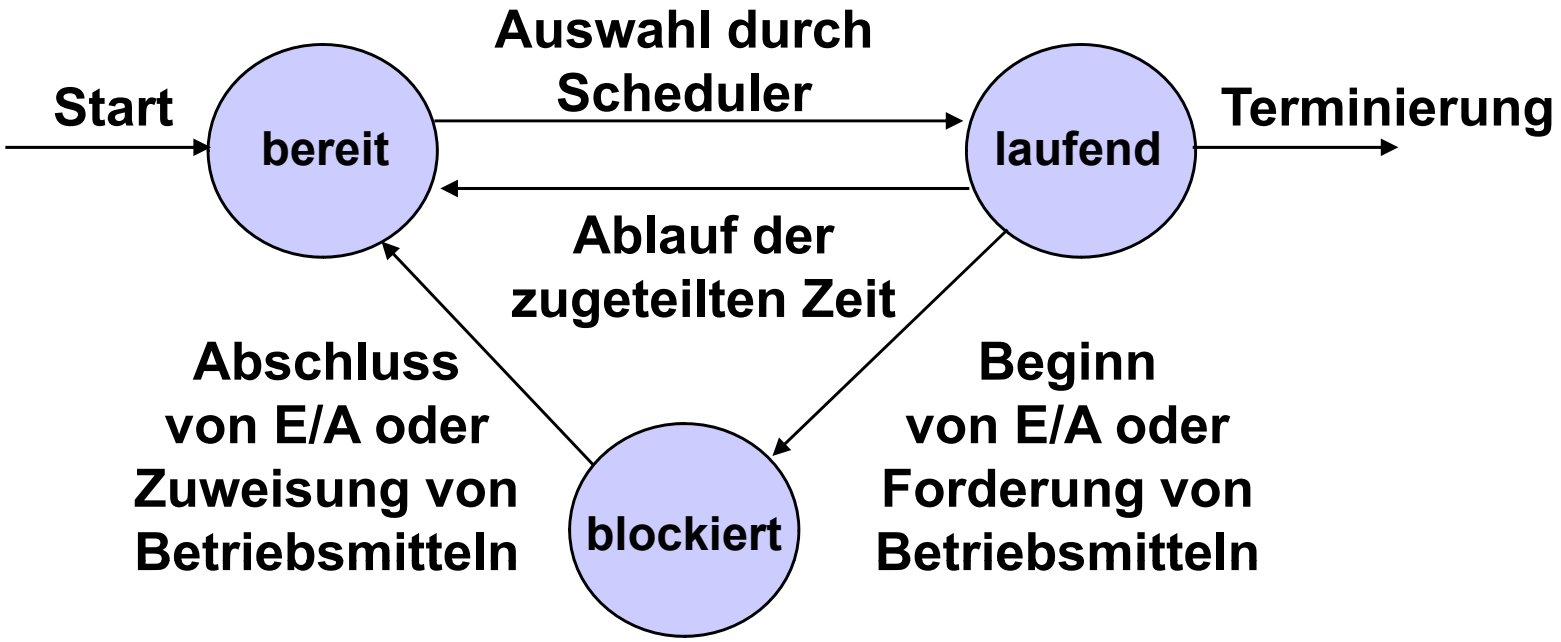
Aufträge mit Ressourcenbedarf



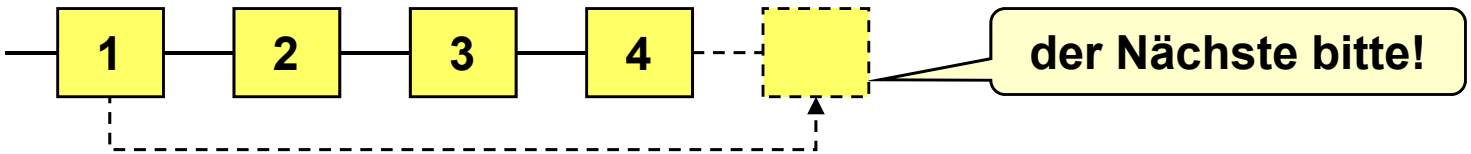
Leichtgewichtige Prozesse (Threads) als Teile eines BS-Prozesses



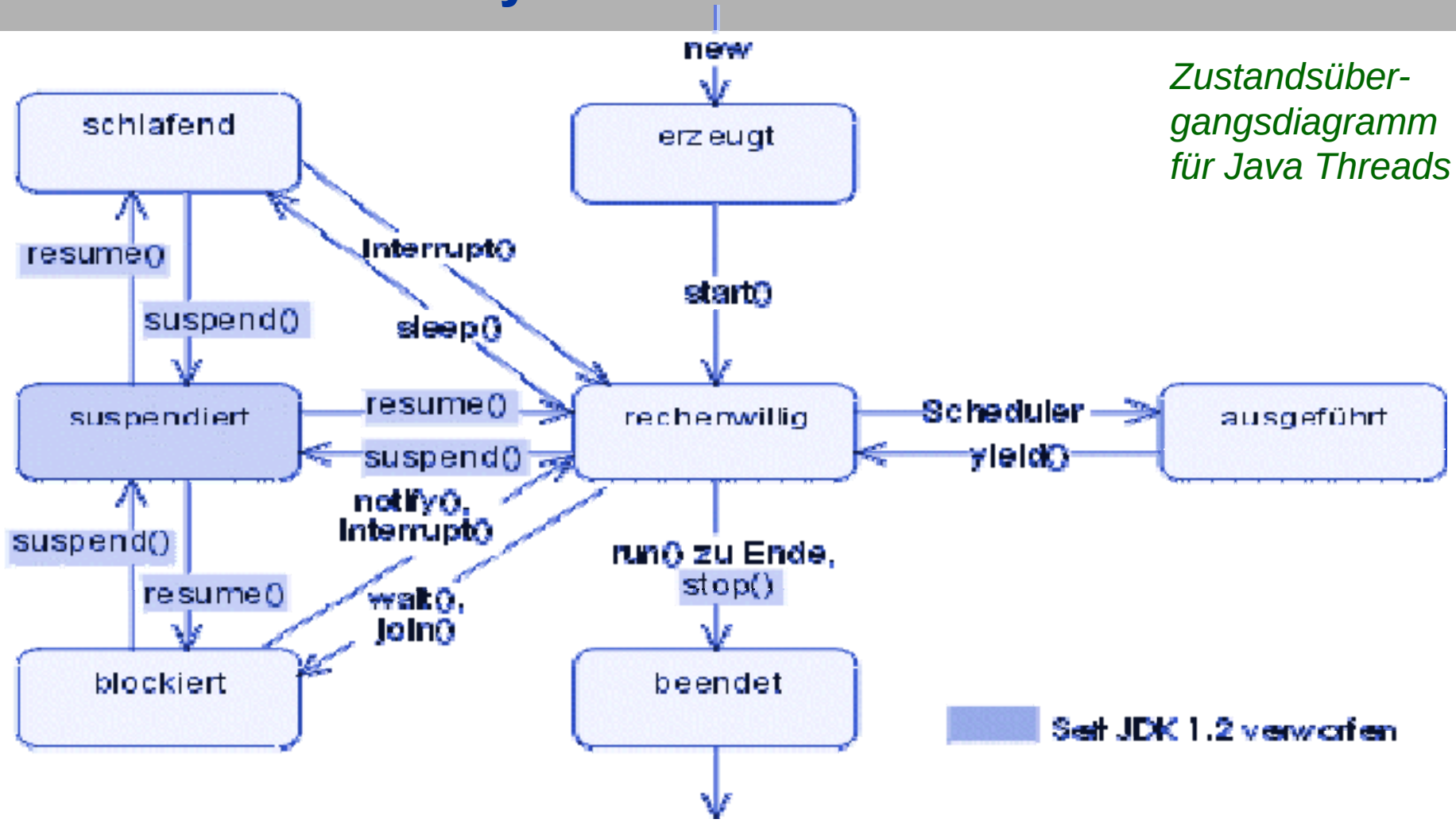
Prozessorzuteilung durch Scheduler



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



Lebenszyklus von Java-Threads



- start() *bewirkt Aufruf der Methode* run() *und nebenläufige Ausführung des Threads*
- *Thread terminiert, wenn* run() *terminiert oder* stop() *ausgeführt wird*
- *Prädikat* isAlive() *liefert true, wenn Thread gestartet und noch nicht terminiert ist*
- *Laufender Thread kann Prozessor durch* yield() *aufgeben*
- *Thread kann durch* suspend() *blockiert und durch* resume() *de-blockiert werden*
- *durch* sleep() *wird Thread auf bestimmte Dauer blockiert*

Synchronisation in Java

Schlüsselwort *synchronized* bewirkt gegenseitigen Ausschluss von nebenläufigen Aktivierungen einer Methode in Java.

Prozessoperationen *wait* und *notify* ermöglichen Prozessverwaltung.

Prozesssynchronisation in Java

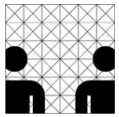
```
public void aMethod () {
        synchronized (anObject) {
        // kritischer Abschnitt
                }
}
```

Beispiel: Interferenz von nebenläufigen Zählerinkrementen verhindern

```
class Counter {
    int value = 0;
    synchronized void increment() {
        ++value;
    }
}
```

Java realisiert gegenseitigen Ausschluss von Methoden verschiedener Threads. Methoden gleicher Threads schließen sich nicht aus.

Mit *synchronized* werden kritische Abschnitte realisiert!



Java-Implementierung eines Semaphors

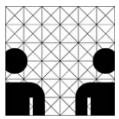
Ein Semaphor ist ein traditioneller Baustein für komplexere Synchronisierungsaufgaben, grundsätzlich in Java entbehrlich, weil beidseitiger Ausschluss durch *synchronized* geregelt werden kann.

```
public class Semaphore {
    private int value;
    public Semaphore (int initial)
        {value = initial;}
        synchronized public void P ()
        throws InterruptedException {
        while (value == 0) wait ();
        --value;}
        synchronized public void V () {
        ++value;
        notify ();}
}
```

Initialwert entspricht Zahl von Passagen vor Blockade

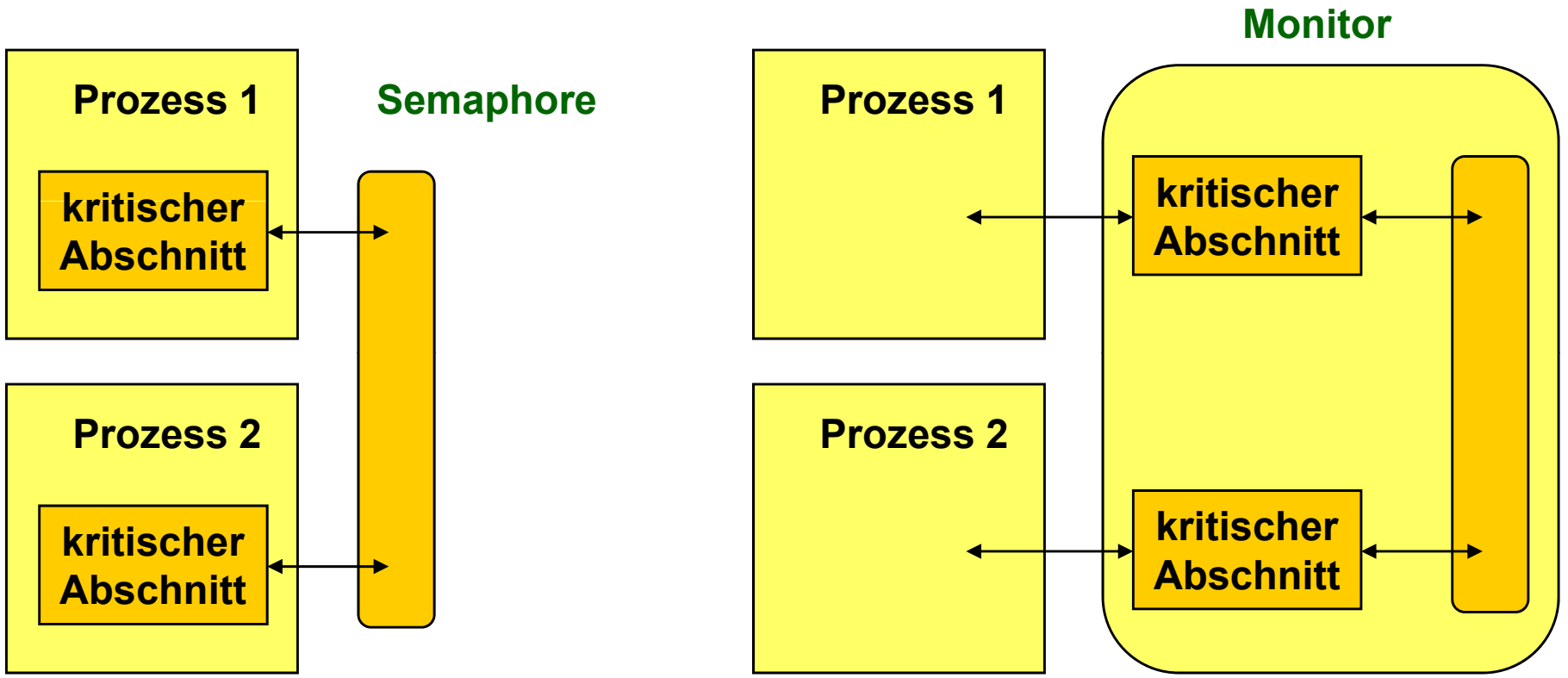
passives Warten, bis Semaphor positiven Wert hat, dann dekrementieren und Passage

nach Inkrementieren nächsten wartenden Prozess aktivieren



Semaphore versus Monitore

Kapselung der kritischen Abschnitte in gemeinsamer abstrakter Datenstruktur („Monitor“) kann größere Klarheit schaffen



Implementierung des Produzenten-Konsumenten-Problems (1)

```
class Produkt {
    private int Ware;
    private boolean verfügbar = false;
    public synchronized int konsumiert () {
        while (! verfügbar) {
            try {wait ();}
            catch (InterruptedException e) { }
        }
        verfügbar = false;
        notify ();
        return Ware;
    }
    public synchronized void produziert (int Warennummer)
{
        while (verfügbar) {
            try {wait();}
            catch (InterruptedException e) { }
        }
        Ware = Warennummer;
        verfügbar = true;
        notify ();
    }
}
```

Klasse* Produkt *muss bei Zugriff auf Ware durch Produzenten und Konsumenten:

1. gegenseitigen Ausschluss garantieren

2. zugreifende Prozesse blockieren und deblockieren

3. über Warenbestand Buch führen

Implementierung des Produzenten-Konsumenten-Problems (2)

```
class Produzent extends Thread {
    private Produkt eineWare;
    Produzent (Produkt c) {eineWare = c;}
    public void run () {
        for (int i = 0; i < 10; i++) {
            eineWare.produziert (i);
            System.out.println (
                i + "produziert");
        }
    }
}
```

```
class Konsument extends Thread {
    private Produkt eineWare;
    Verbraucher (Produkt c) {eineWare = c;}
    public void run() {
        for (int i = 0; i < 10; i++) {
        System.out.println (
            eineWare.konsumiert () +
            "konsumiert");
        }
    }
}
```

Testprogramm

```
class ProduzentKonsument {
    public static void main(String[ ] args) {
        Produkt c = new Produkt ();
        (new Produzent (c)).start ();
        (new Konsument (c)).start ();
    }
}
```

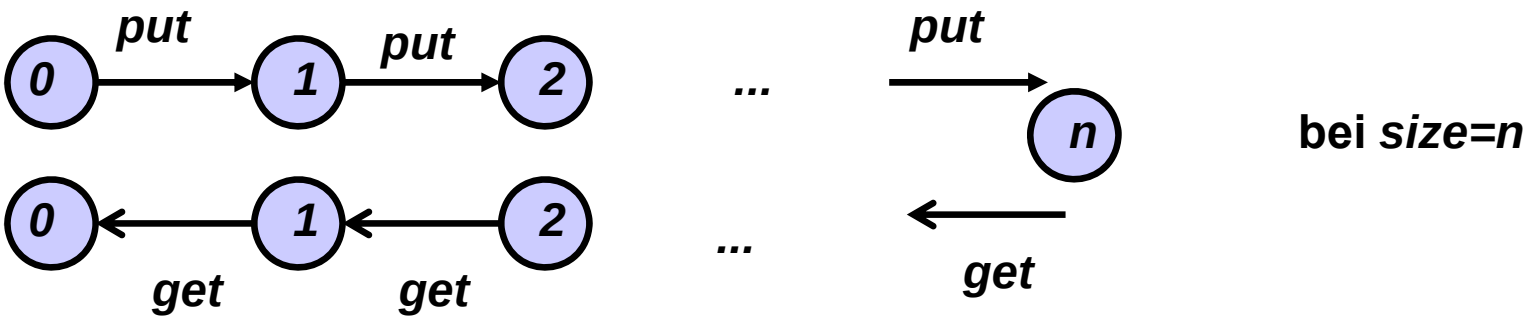
Ausdruck bei Testlauf zeigt abwechselnde Produktion und Konsumption:

```
0  produziert
0  konsumiert
1  produziert
1  konsumiert
2  produziert
2  konsumiert
...
```

Erweiterung des Produzenten/Konsumenten-Problems zum Pufferverwaltungsprogramm

- *Puffer nimmt maximale Zahl von Objekten auf*
- *Produzent füllt Puffer stückweise*
- *Konsument leert Puffer stückweise*

Automatenmodell

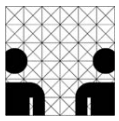


Java-Programme für Pufferverwaltung

- Puffer nimmt begrenzte Zahl von Objekten auf
- Produzent füllt Puffer stückweise
- Konsument leert Puffer stückweise

```
public interface Buffer {
    public void put (Object o)
        throws InterruptException;
    public Object get ()
        throws InterruptException;
}
```

- *Interface ist abgetrennt, um alternative Implementierungen zu ermöglichen*
- *Puffer hat feste Größe* size*, nimmt beliebige Objekte auf, ist als Ringpuffer organisiert*
- notify *nach* put*, falls abnehmender Prozess wartet*
- notify *nach* get*, falls liefernder Prozess wartet*



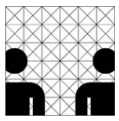
```
class BufferImpl implements Buffer {
    protected Object[ ] buf;
    protected int in = 0;
    protected int out = 0;
    protected int count = 0;
    protected int size;
    BufferImpl (int size) {
        this.size = size; buf = new Object[size];
    }
    public synchronized void put (Object o)
            throws InterruptException {
        while (count == size) wait ();
        buf [in] = o;
        ++count;
        in = (in + 1) mod size;
        notify ();
    }
    public synchronized Object get ()
            throws InterruptException {
        while (count == 0) wait ();
        Object o = buf[out];
        buf [out] = null;
        --count;
        out = (out + 1) mod size;
        notify ();
        return (o);
    }
}
```

Java-Programme für Puffer-Zugriff

```
class Producer implements runnable {
    Buffer buf;
    Object item;
    Producer (Buffer b) {buf = b};
    public void run () {
        try {
            while (true) {
                buf.put (new item);
            }
            catch (InterruptedException e){ }
        }
    }
}
```

```
class Consumer implements runnable {
    Buffer buf;
    Object item;
    Consumer (Buffer b) {buf = b};
    public void run () {
        try {
            while (true) {
                item = buf.get ();
            }
            catch (InterruptedException e){ }
        }
    }
}
```

- *Lieferant erzeugt Objekte* (new item) *in Endlosschleife und legt sie im Puffer ab*
- *Abnehmer entfernt Objekte aus Puffer in Endlosschleife (und tut hier nichts weiter damit)*



Prozesskommunikation

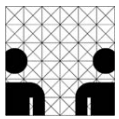
Bisher haben Prozesse über *gemeinsam zugreifbare Variable* interagiert. Sind keine gemeinsamen Datenbereiche vorhanden, müssen Informationen als Nachrichten oder Botschaften (messages) ausgetauscht werden.

Entsprechungen:	**Schreiben**	⟷	**Senden**
	Lesen	⟷	**Empfangen**
	gemeinsamer Datenbereich	⟷	**Kommunikations-kanal**

Nachrichtenaustausch ist eine mächtige Metapher für Synchronisierung, denn implizit gilt: (sende Nachricht) → (empfange Nachricht)

Ein Kommunikationskanal kann als abstrakter Datentyp realisiert werden und unterscheidet sich dann kaum von einem gemeinsamen Datenbereich:

send wert to kanal <=> kanal.send (wert)



Relevante Eigenschaften für Synchronisierung

Senden von Nachrichten

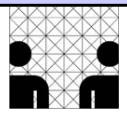
blockierend:	***Prozess wartet nach Sendeoperation auf Empfangsbestätigung***
nicht blockierend:	***Prozess läuft nach Sendeoperation weiter***

Empfangen von Nachrichten

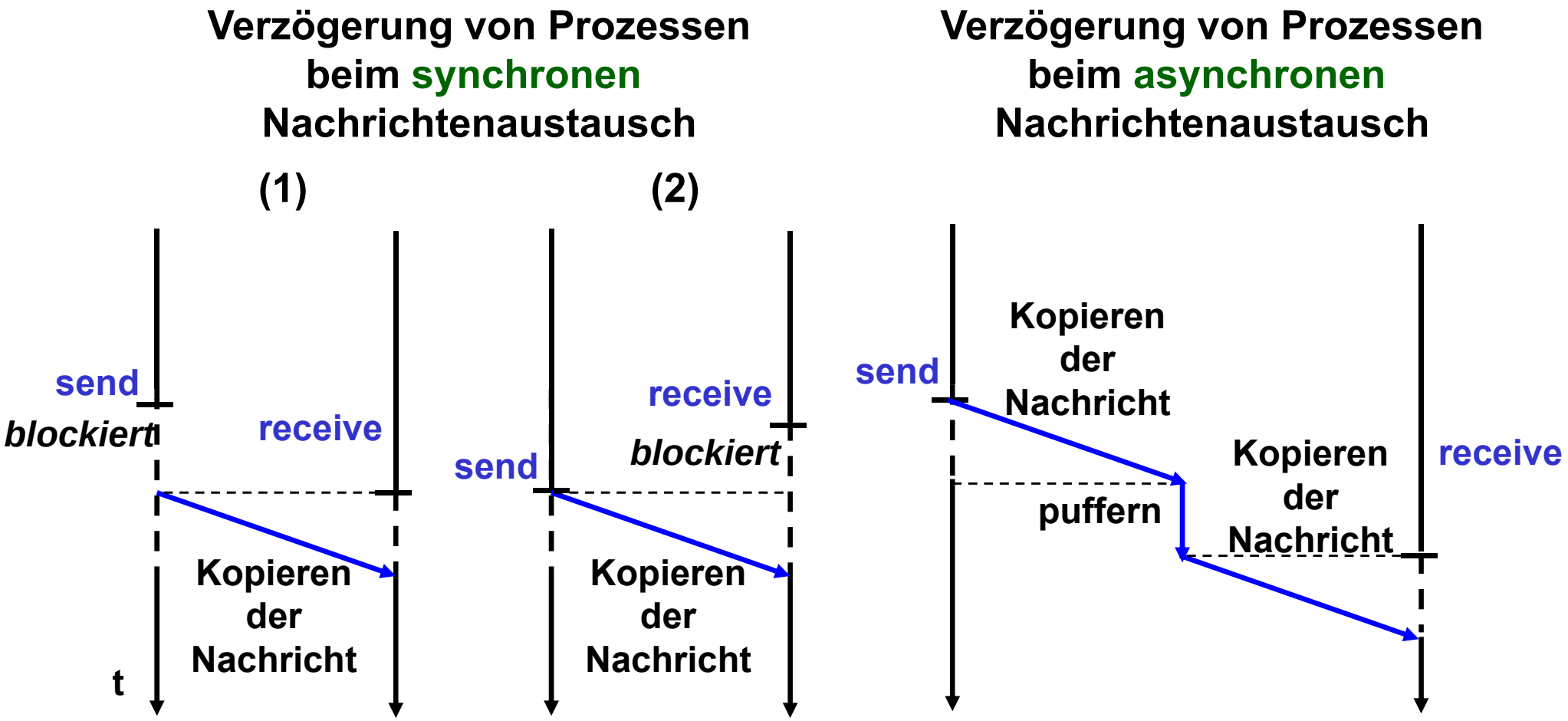
blockierend (üblich):	***Prozess wartet auf Nachrichtenempfang***
nicht blockierend:	***Prozess bleibt bei fehlender Nachricht aktiv (z.B. Test auf zu aktualisierende Werte)***

Kommunikationskanal

gepuffert:	***Nachrichten werden entsprechend der Sendefolge zwischengelagert***
ungepuffert:	***Nachrichten werden direkt vom Sender zum Empfänger kopiert***



Synchroner und asynchroner Nachrichtenaustausch



asynchrones Verhalten geht verloren, wenn
- Puffer voll ist und Sender blockiert wird
- Puffer leer ist und Empfänger blockiert wird

Nachrichtenaustausch zwischen mehr als 2 Prozessen

Rundsendung (broadcast)

Nachricht wird an alle denkbaren Empfänger gesendet

```
broadcast wert
```

Mehrfachsendung (multicast)

Nachricht wird an mehrere spezifizierte Empfänger gesendet

```
multicast wert to (kanal1, kanal2, kanal3)
```

Selektives Empfangen

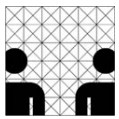
Nichtdeterministische Auswahl von eingetroffenen Nachrichten

```
select
    receive variable1 from kanal1 → anweisung1
    receive variable2 from kanal2 → anweisung2
    receive variable3 from kanal3 → anweisung3
end select
```

Bedingtes selektives Empfangen

Auswahl zwischen Nachrichten, für die eine Bedingung zutrifft

```
select
    (when B1 and receive variable1 from kanal1)
        → anweisung1
    (when B2 and receive variable2 from kanal2)
        → anweisung2
end select
```

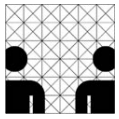


Konstrukte für Nachrichtenaustausch zwischen Java-Prozessen

Java bietet keine besonders eleganten Sprachelemente zum Nachrichten-austausch zwischen Prozessen.

Methoden der Basisklassen *Select* und *Selectable* steuern Auswahl aus Warteschlangen synchronisierter Objekte.

select.add	***fügt ein* selectable *Objekt in Warteschlange für selektives Empfangen ein***
select.choose	***führt selektives Empfangen von* selectable *Objekten aus, die in der Warteschlange sind***
selectable.guard	***testet* selectable *Objekt in Warteschlange für selektives Empfangen***



Java-Programm für selektiven Nachrichtenempfang

```
class Channel extends Selectable {
    public synchronized void send (Object v)
        throws InterruptException { ... }
    public synchronized Object receive ( )
        throws InterruptException { ... }
```

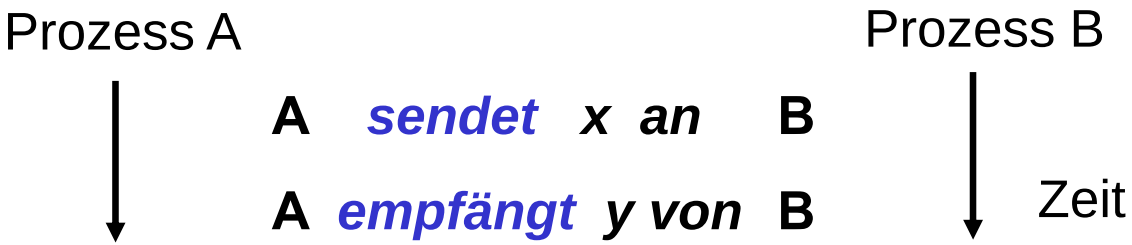
Implementierung eines Nachrichtenkanals **Channel** *mit Hilfe der Klasse selectable*

```
class MessageReceiver implements Runnable {
    private Channel arrive1, arrive2;
    public void run ( ) {
        try {
            Select sel = new Select ( );
            sel.add (arrive1);
            sel.add (arrive2)
            while (true) {
                arrive1.guard (<Bedingung1>);
                arrive2.guard (<Bedingung2>);
                switch (sel.choose ( )) {
                    case 1: arrive1.receive ( ); ...; break;
                    case 2: arrive2.receive ( ); ... ; break;
                }
            }
        } catch InterruptException{ }
    }
}
```

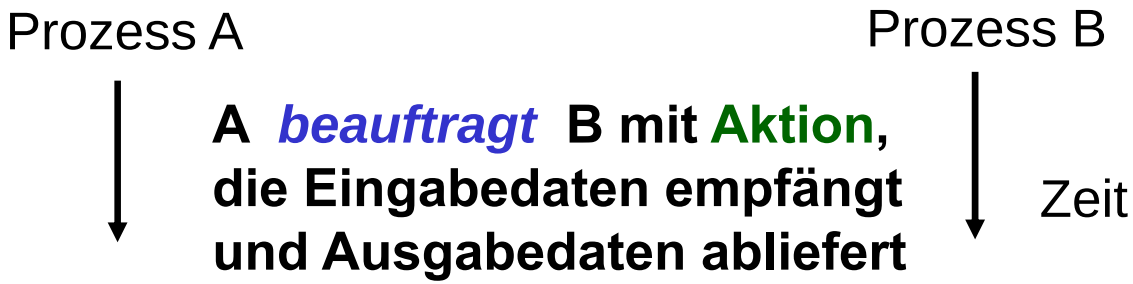
Selektive Auswahl von auswahlbereiten Nachrichten

Abstraktion vom Nachrichtenaustausch

Bisher datenorientierter Nachrichtenaustausch mit typischem Muster:



Aktionsorientierter Nachrichtenaustausch bietet Abstraktionsmöglichkeit durch Zusammenfassen der Aktivitäten von B als Aktion:



Fernaufruf von Prozeduren

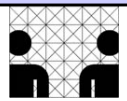
Die Beziehung zwischen Auftraggeber und Auftragnehmer lässt sich durch einen entfernten Prozedurfernaufruf (Remote Procedure Call, RPC) in vertrauter Weise (d.h. - fast - wie im lokalen System) modellieren.

Unterschied zum lokalen Prozeduraufruf (u.a.):

- **Auftraggeber und Auftragnehmer sind verschiedene Prozesse in verschiedenen Datenräumen**
- **Auftraggeber und Auftragnehmer sind nebenläufig, Synchronisationsbedarf je nach Art des Auftrags**

```
auftraggeber: process
eing: eTyp
ausg: aTyp
   repeat
   ...
   auftragnehmer.auftrag (eing, ausg);
   ...
   end repeat
end process
```

```
auftragnehmer: process
   export auftrag;
   auftrag: procedure (ein: eTyp; out aus: aTyp)
   ... // Auftrag bearbeiten
   end procedure
end process
```



Rendezvous

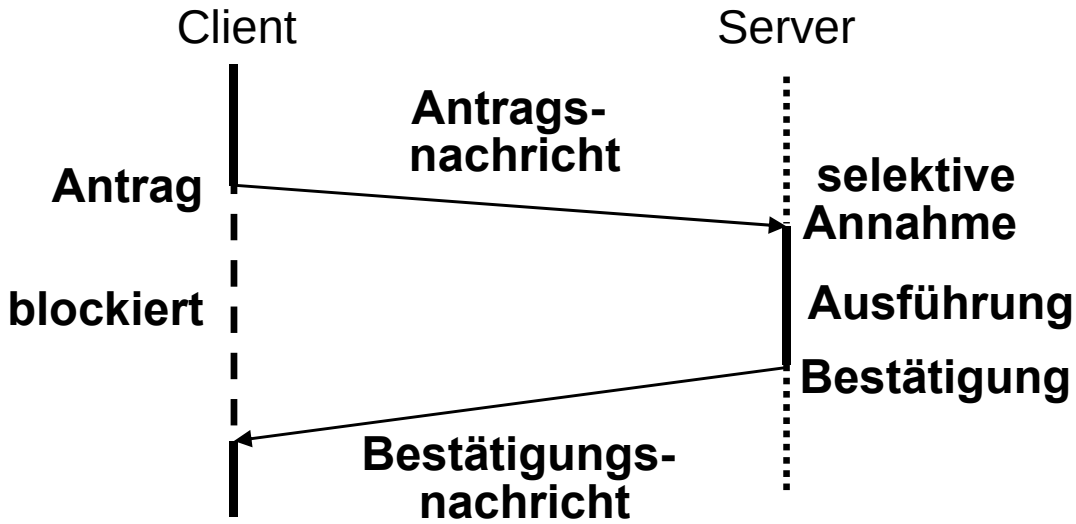
Rendezvous = Prozedurfernaufruf mit größerer Autonomie des aufgerufenen Prozesses (bestimmt selbst über Ausführung des Auftrags)

Sprachgebrauch:

Client beantragt einen Dienst (request)

Server
- bietet Dienst an (offer)
- nimmt Dienstauftrag an (accept)
- führt Dienst aus (execution)
- bestätigt Dienst (reply)

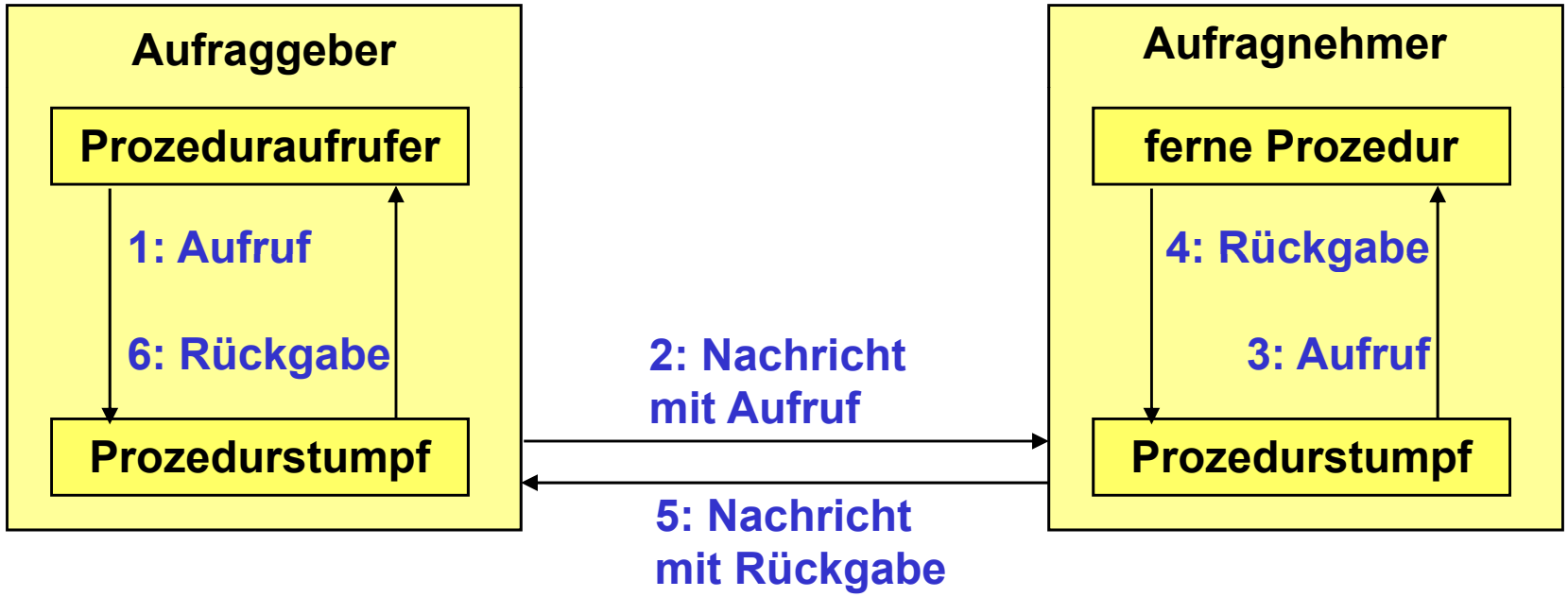
Rendezvous werden vom Server in der Regel selektiv eingegangen. Client ist während der Ausführung des Dienstes meist blockiert.



Implementierung von Prozedurfernaufrufen

Prozedurstumpf („stub“) auf Auftraggeber**seite repräsentiert ferne Prozedur im Adressraum des Auftraggebers und sorgt für Nachrichtenaustausch mit Auftragnehmer.**

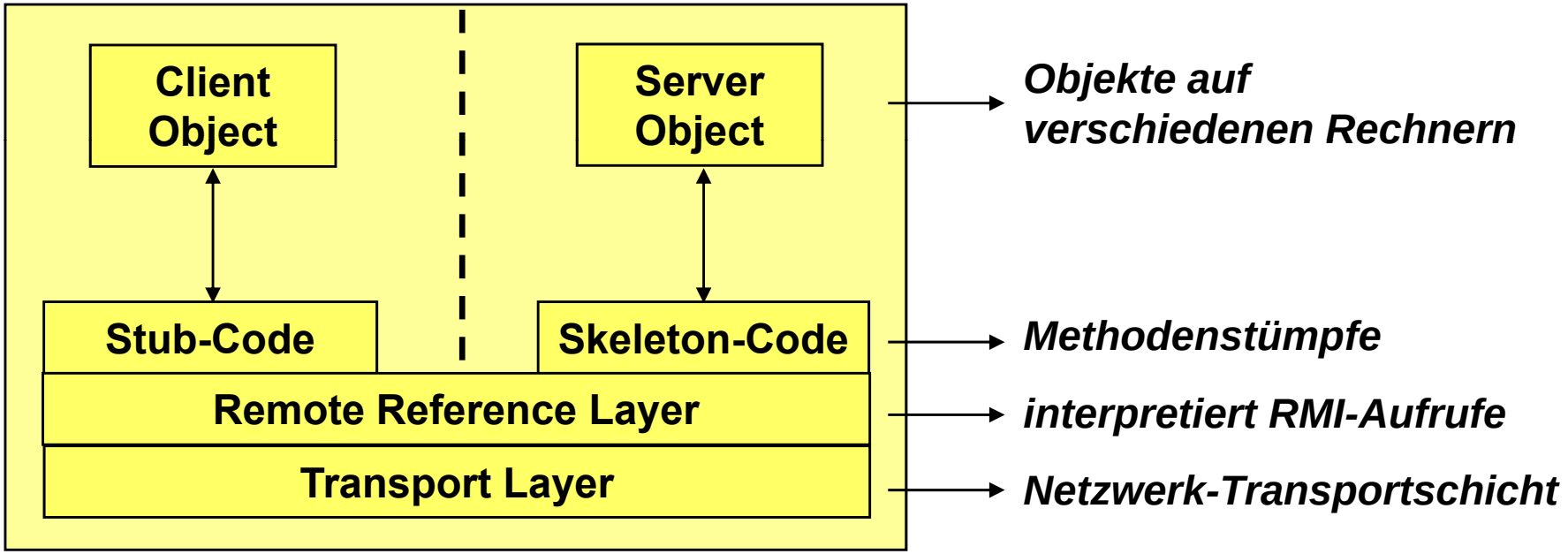
Prozedurstumpf auf Auftragnehmer**seite sorgt für Prozeduraufruf und Nachrichtenaustausch mit Auftraggeber.**



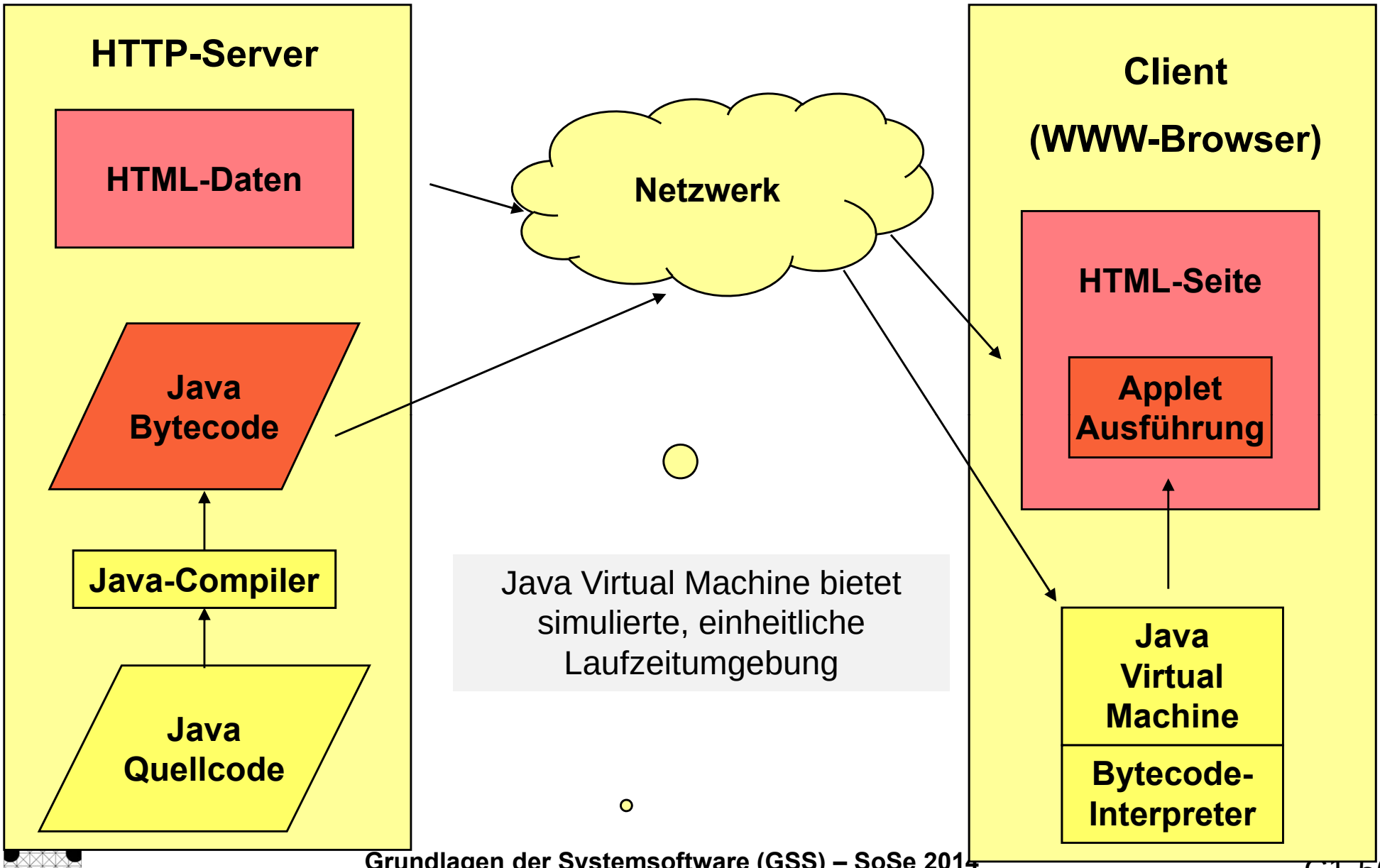
Methodenfernaufruf in Java

Java-Objekt auf Rechner A (Client) kann Methoden eines entfernten Objektes auf Rechner B (Server) durch „Remote Method Invocation“ (RMI) aufrufen.

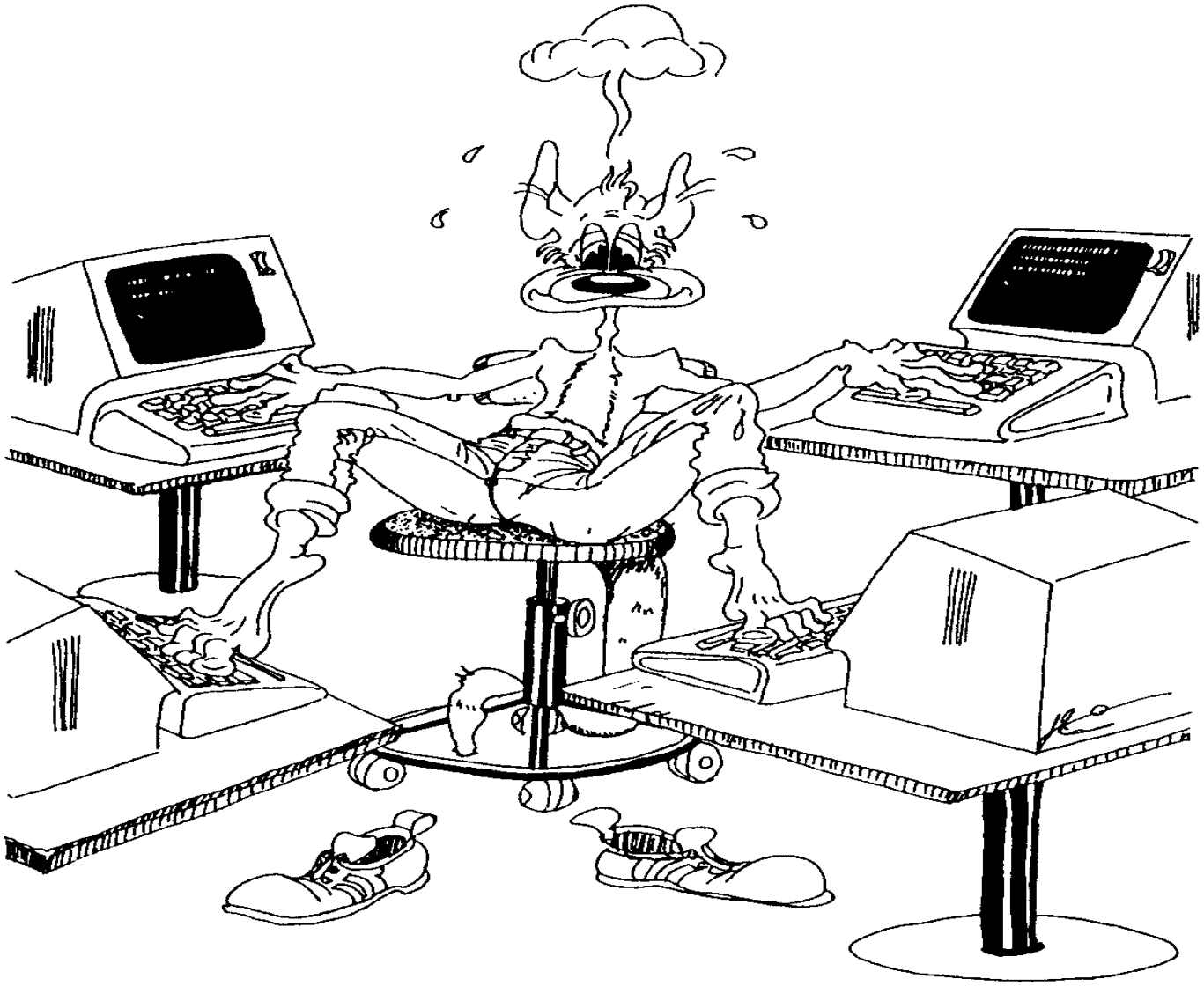
Architektur des RMI-Systems:

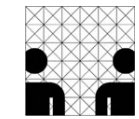


Fernausführung von Java-Applets im WWW



Aber: Nebenläufigkeitsprobleme erfordern oft abstrakte Modellierung



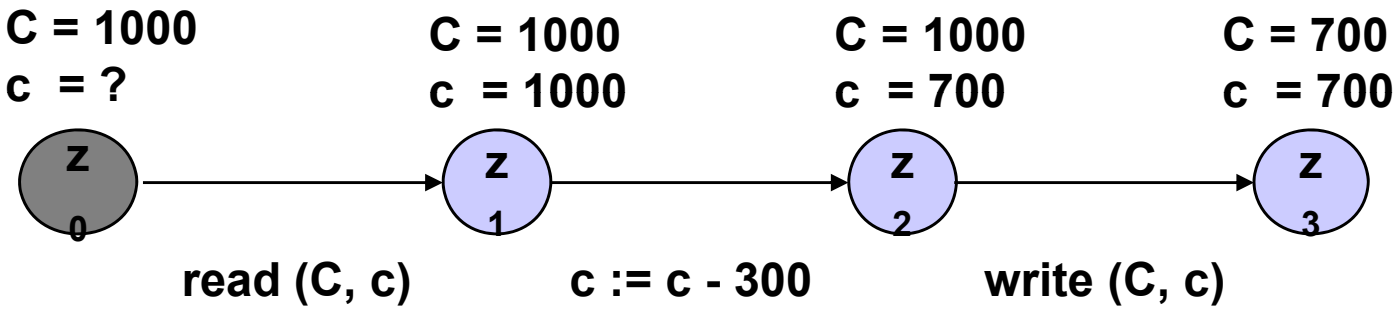


C 1.5: Abstrakte Modellierung
Modellierung von Prozessen durch endliche Automaten

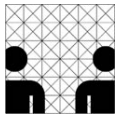
Prozesszustände — **Zustände eines Automaten**

Aktivitäten — **Eingaben des Automaten, bewirken Zustandsübergänge**

Beispiel:



$Z = \{ z_0 \ldots z_N \}$ Menge der Zustände
$A = \{ a_0 \ldots a_M \}$ Menge der Aktivitäten
$E = \{ (z_i\ a_i\ z_k) \}$ Zustandsübergänge des endlichen Automaten

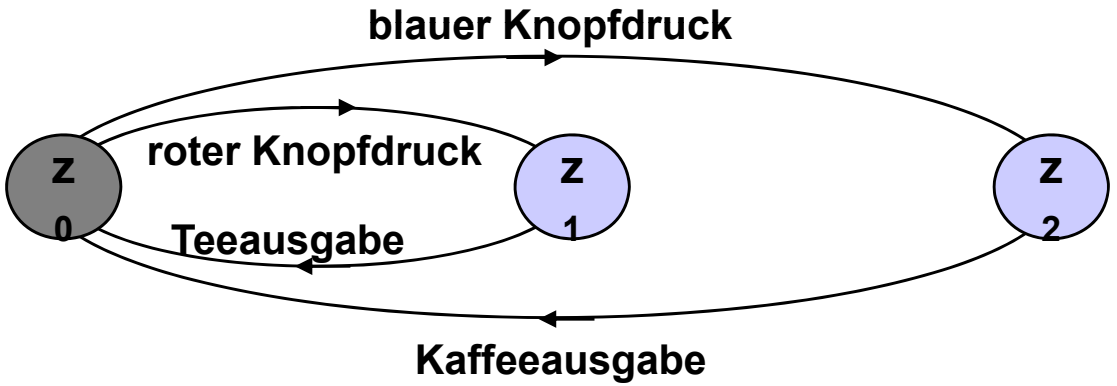


Deterministische Wahl

Eine deterministische Wahl besteht, wenn ein Prozess von einem Zustand aus durch unterschiedliche Aktivitäten in verschiedene – aber dann jeweils *eindeutige* – Folgezustände übergehen kann.

Beispiel:

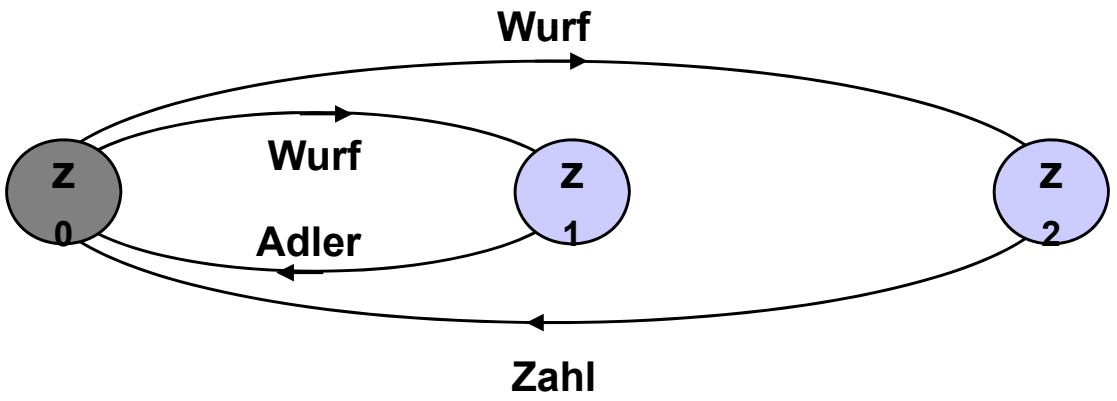
Getränkeautomat hat roten Knopf für Tee und blauen Knopf für Kaffee



Nichtdeterministische Wahl

Ein Prozess ist nicht-deterministisch, wenn er bei *gleicher* Aktivität in *verschiedene* Zustände übergehen kann.

Beispiel: Münzwurf



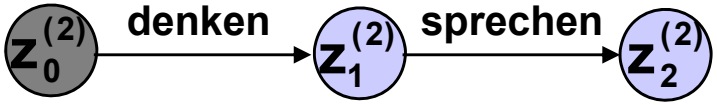
Parallele Ausführung nebenläufiger Prozesse

Beschreibung der möglichen Verzahnungen zweier nebenläufiger Prozesse durch einen Produktautomaten.

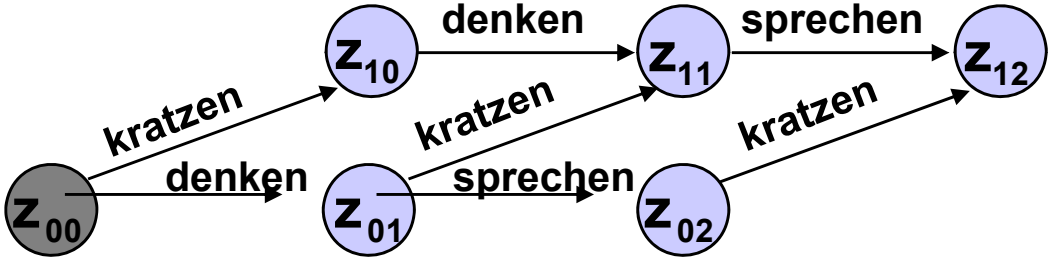
Beispiel:

Jucken

Vortragen



Jucken || Vortragen

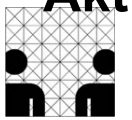


Definition Produktautomat:

Zustände $Z = \{ z_{ij} \mid z_i^{(1)} \in Z^{(1)} \wedge z_j^{(2)} \in Z^{(2)} \}$

Zustandsübergänge $E = \{ (z_{ij}\ a_k\ z_{mn}) \mid (z_i\ a_k\ z_m) \in E^{(1)} \vee (z_j\ a_k\ z_n) \in E^{(2)} \}$

Aktivitäten $A = A^{(1)} \cup A^{(2)}$



Nebenläufige Prozesse mit gemeinsamen Aktivitäten

Enthalten Prozesse gemeinsame Aktivitäten, so müssen diese gleichzeitig ausgeführt werden.

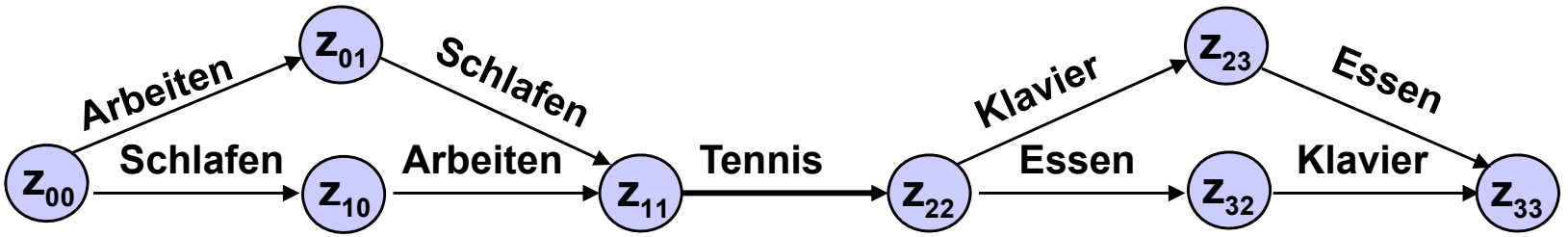
gemeinsame Aktivität "Tennis"

Beispiel:

$$\text{Felix} = \{(z_0^{(1)}\ \text{Schlafen}\ z_1^{(1)})(z_1^{(1)}\ \text{Tennis}\ z_2^{(1)})(z_2^{(1)}\ \text{Essen}\ z_3^{(1)})\}$$

$$\text{Marietta} = \{(z_0^{(2)}\ \text{Arbeiten}\ z_1^{(2)})(z_1^{(2)}\ \text{Tennis}\ z_2^{(2)})(z_2^{(2)}\ \text{Klavier}\ z_3^{(2)})\}$$

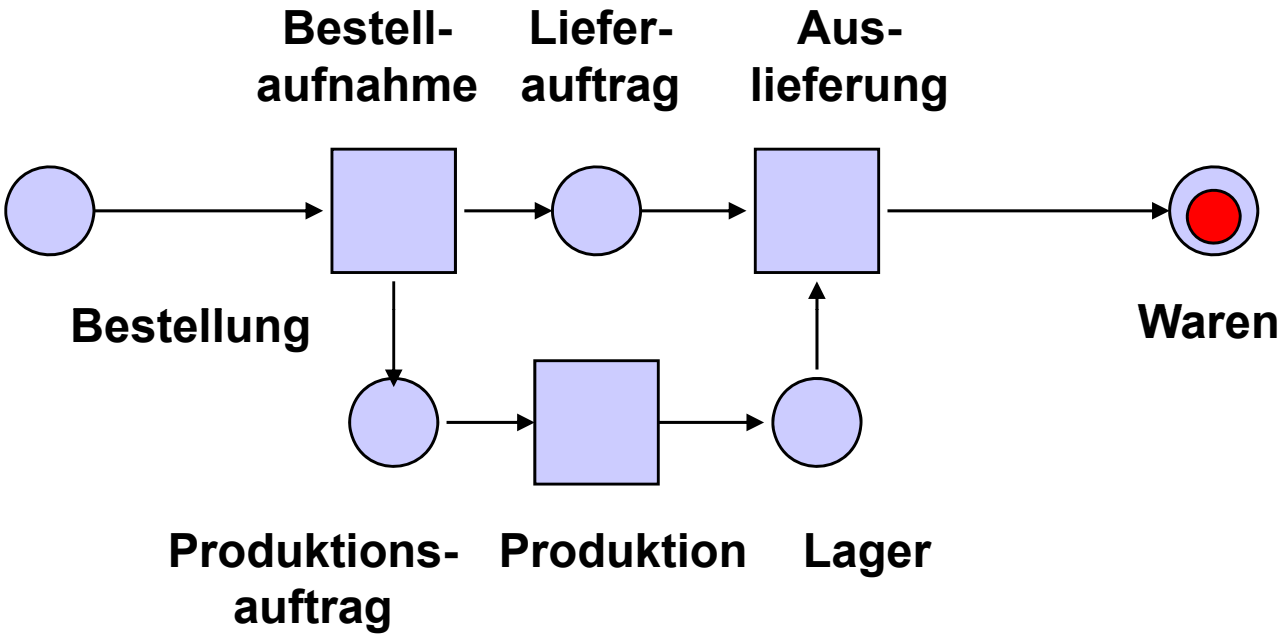
$$\text{Felix} \mid\mid \text{Marietta} = \{(z_{00}\ \text{Schlafen}\ z_{10})(z_{01}\ \text{Schlafen}\ z_{11})(z_{00}\ \text{Arbeiten}\ z_{01})(z_{10}\ \text{Arbeiten}\ z_{11})$$
$$(z_{11}\ \text{Tennis}\ z_{22})$$
$$(z_{22}\ \text{Essen}\ z_{32})(z_{23}\ \text{Essen}\ z_{33})(z_{22}\ \text{Klavier}\ z_{23})(z_{32}\ \text{Klavier}\ z_{33})\}$$

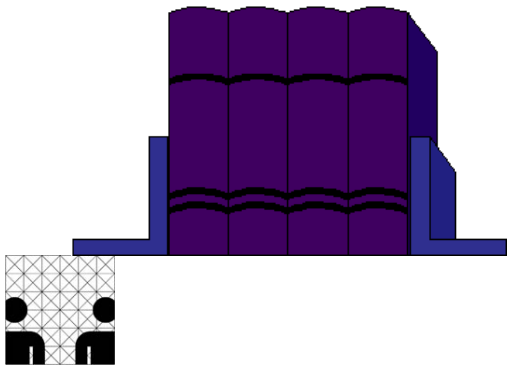


Petri-Netze

Anschauliche Modellierungsmethode für nebenläufige Prozesse und ihre Synchronisation.

Beispiel:
Materialverwaltung

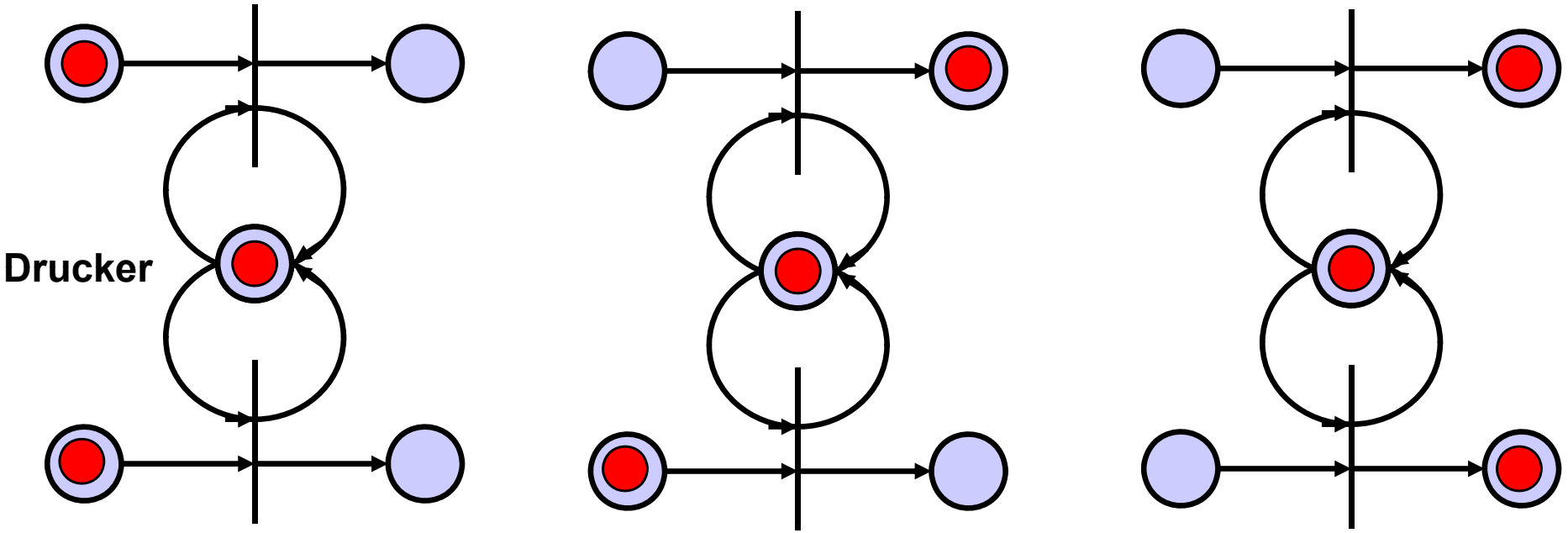




W. Reisig: Petri-Netze - Modellierungstechnik, Analysemethoden, Fallstudien, Vieweg/Teubner, 2010

Benutzung eines Betriebsmittels durch zwei Prozesse

Zwei Prozesse wollen einen Drucker benutzen ...



beide Transitionen feuerbereit

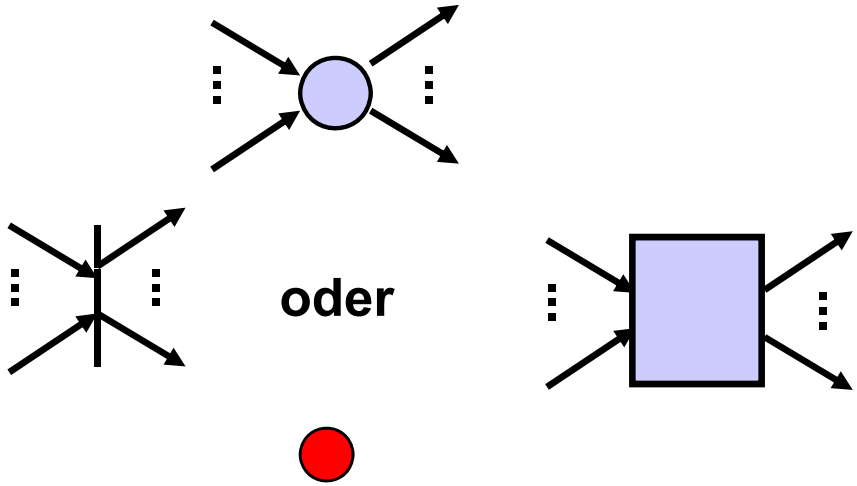
nur eine Transition kann feuern

die zweite Transition kann danach feuern

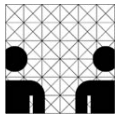
Erinnerung: Grundelemente von Petri-Netzen

Abstraktion interagierender nebenläufiger Prozesse durch S/T-Netz aus

- **Stellen (Plätzen)**
- **Transitionen (Übergängen)**
- **Marken, die nach bestimmten Regeln verschoben werden können**



- **Stellen sind nur mit Transitionen, Transitionen nur mit Stellen verbunden**
- **Eine Transition kann feuern, wenn alle Eingangsstellen mit Marken besetzt sind**
- **Beim Feuern einer Transition werden alle Eingangsstellen freigemacht, alle Ausgangsstellen besetzt**

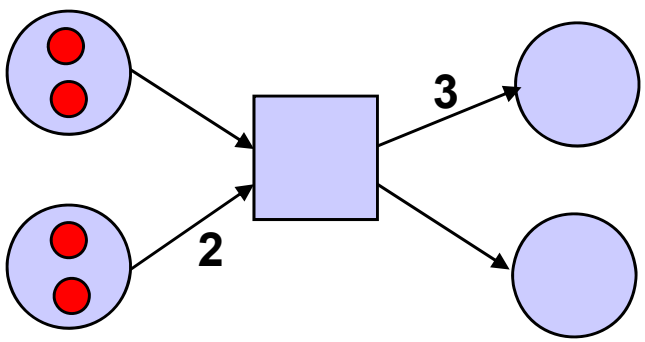


Kapazität und Gewichtung

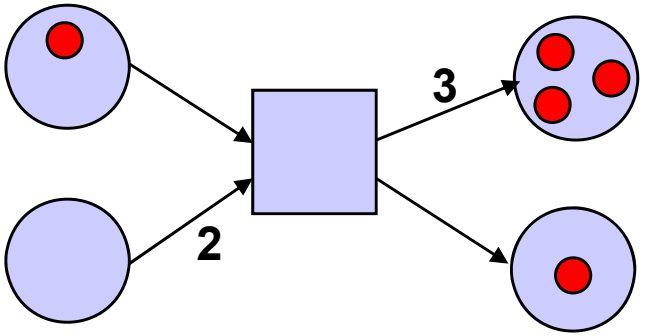
Die *Kapazität* einer Stelle ist die Zahl der maximal aufnehmbaren Marken dieser Stelle. Ohne Angabe ist die Kapazität ∞.

Kanten können eine *Gewichtung* tragen:

- **Zahl der Marken, die beim Schalten der Transition von einer Eingangsstelle entfernt werden müssen**
- **Zahl der Marken, die beim Schalten der Transition einer Ausgangsstelle zugefügt werden müssen**



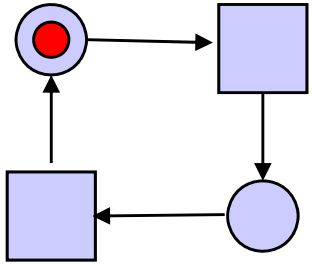
vorher



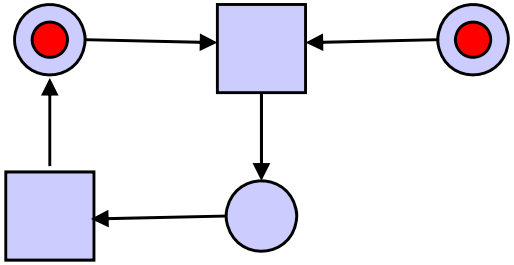
nachher

Lebendige und sichere Netze

Ein (Teil-) Netz heißt *lebendig*, wenn es keinen Zustand geben kann, wo es
- wegen zu wenig Stellen im Vorbereich, oder
- wegen zu viel Stellen im Nachbereich
nicht mehr schalten kann. Andernfalls heißt es *todesgefährdet*.

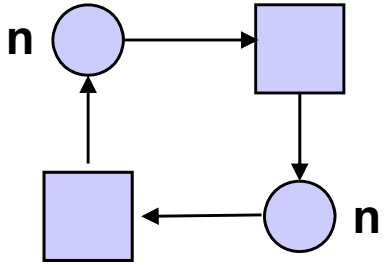


lebendig

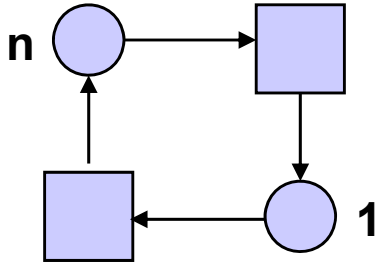


todesgefährdet

Ein Netz heißt *sicher*, wenn eine Erhöhung von Kapazitäten nicht zu mehr Schaltmöglichkeiten führt.



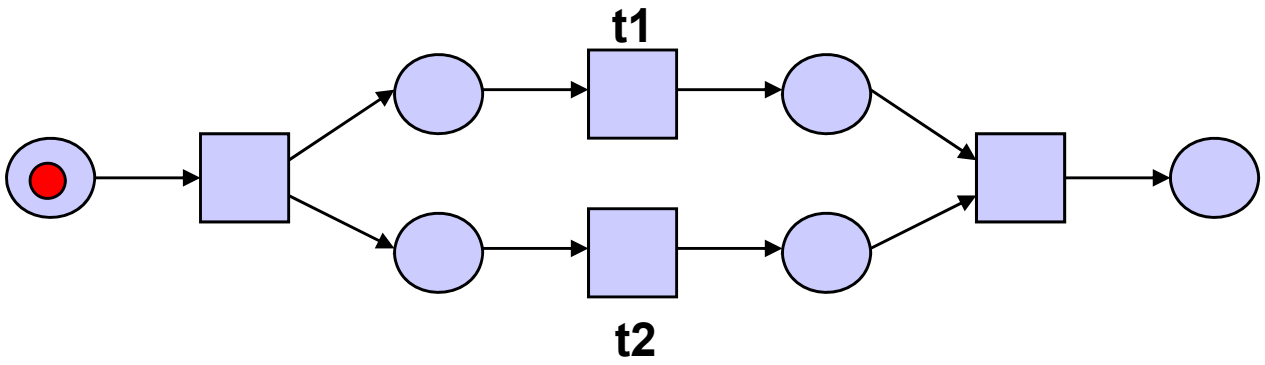
sicher



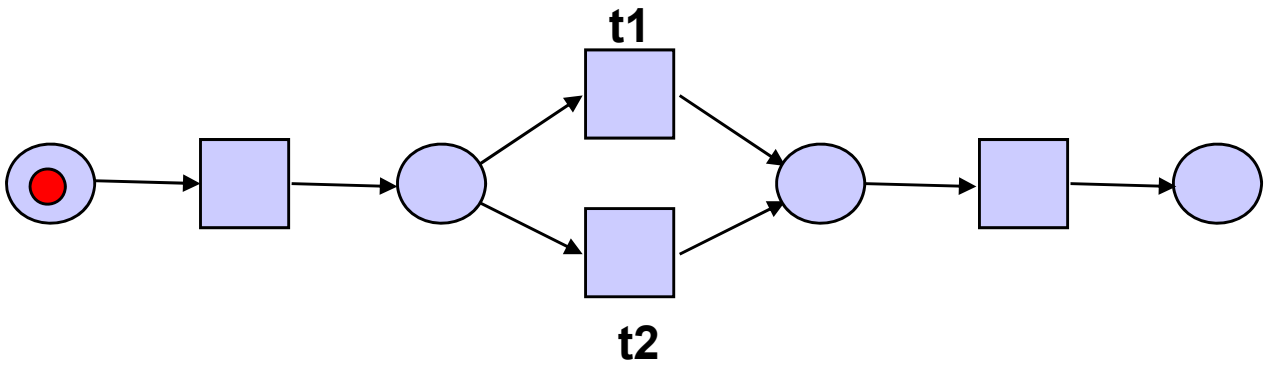
unsicher

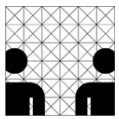
Netzmuster (1)

Nichtdeterministische Reihenfolge von t1 und t2



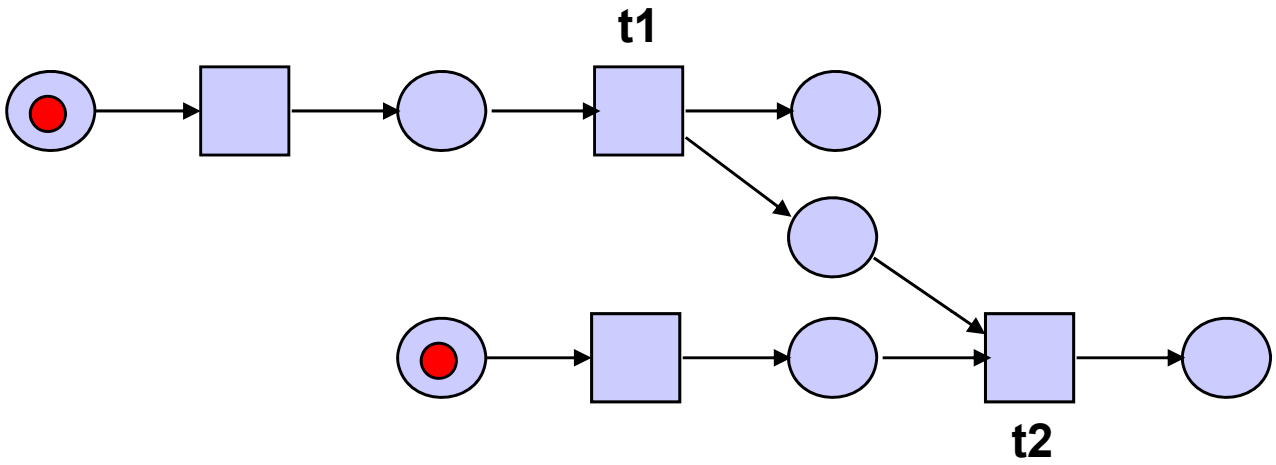
Nichtdeterministische Auswahl von t1 und t2





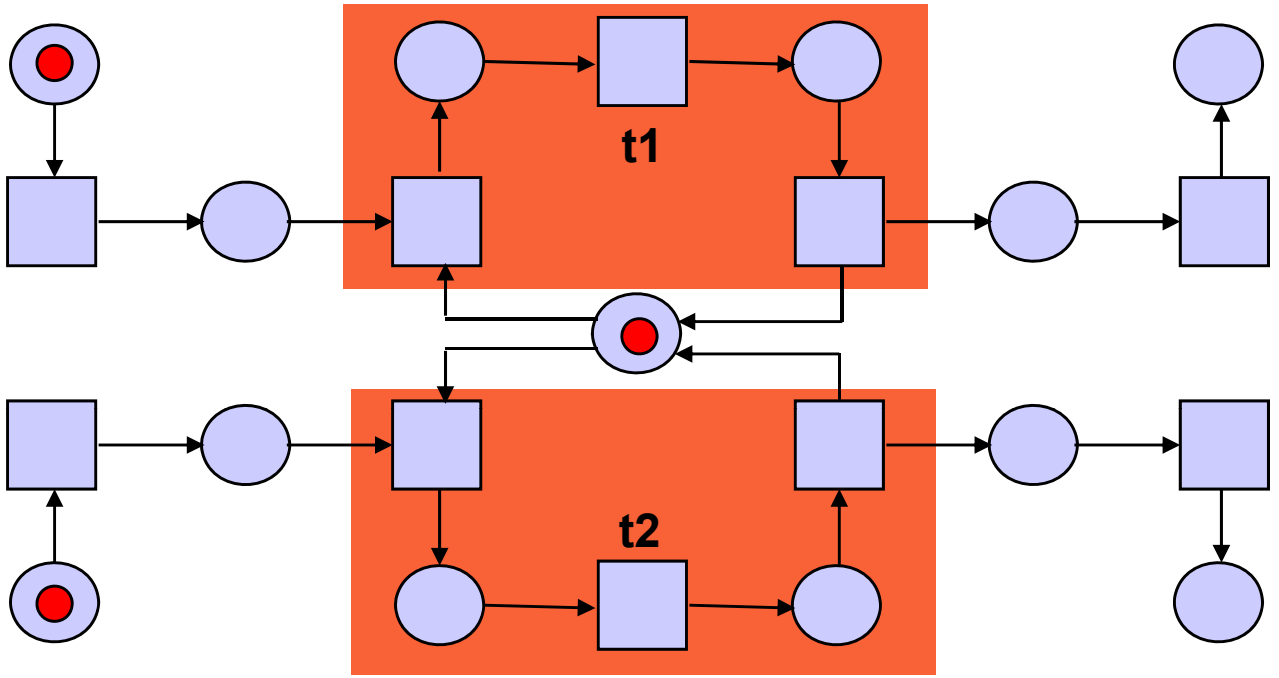
Netzmuster (2)

Einseitige Synchronisierung t1→ t2



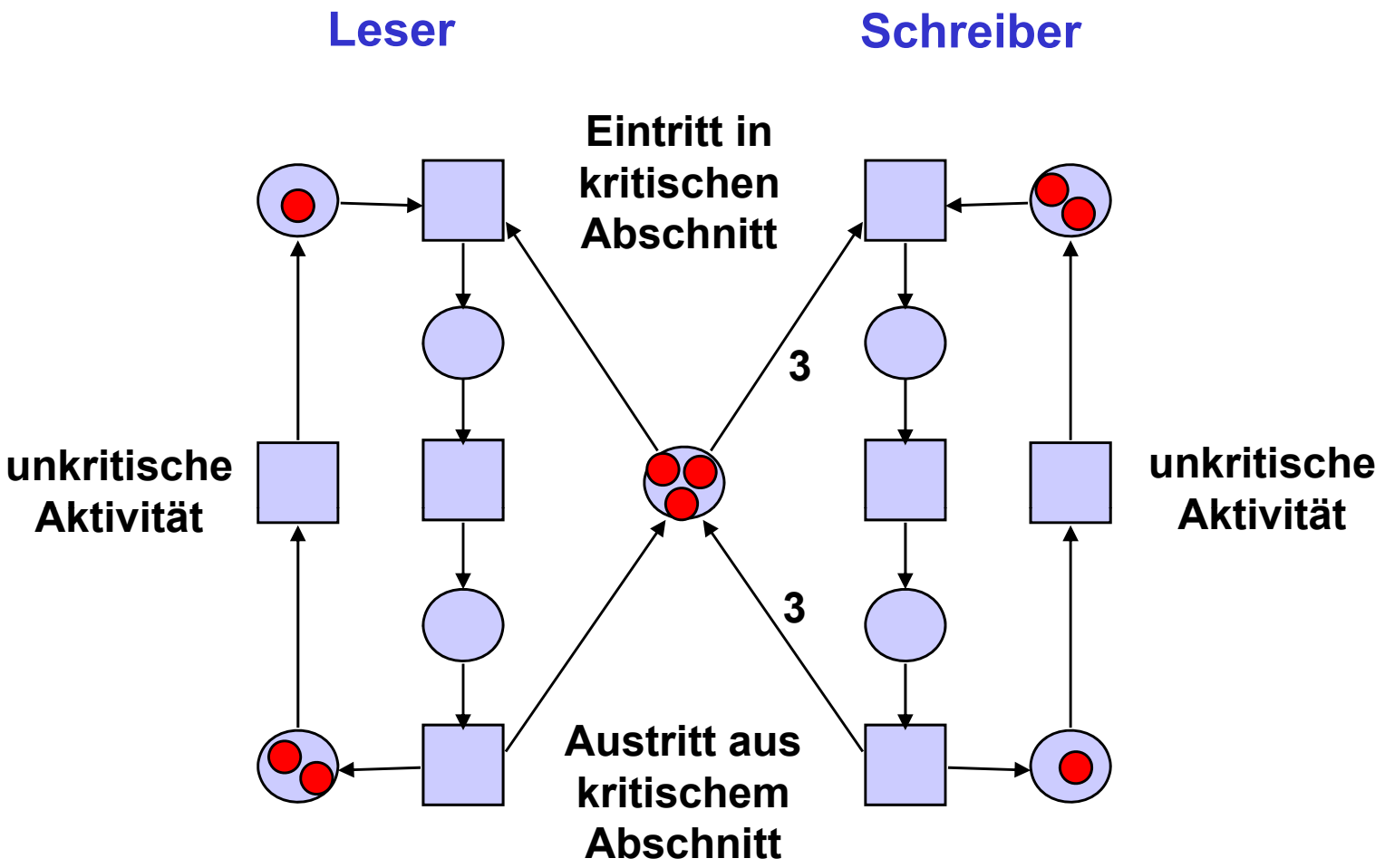
Netzmuster (3)

Gegenseitiger Ausschluss t1↔ t2



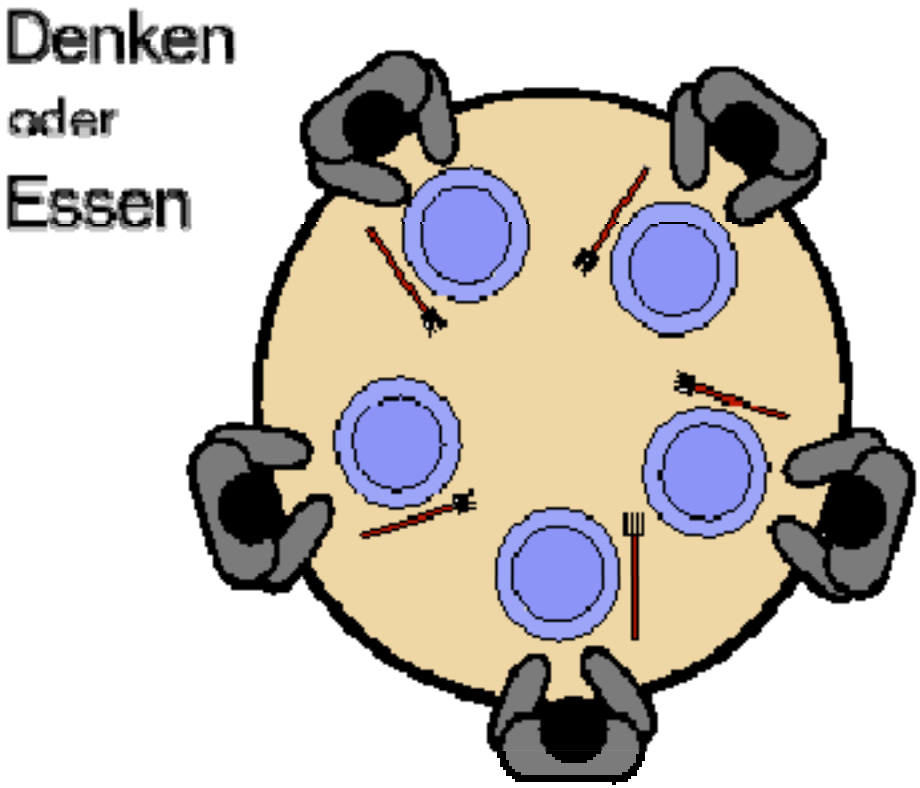
kritische Abschnitte

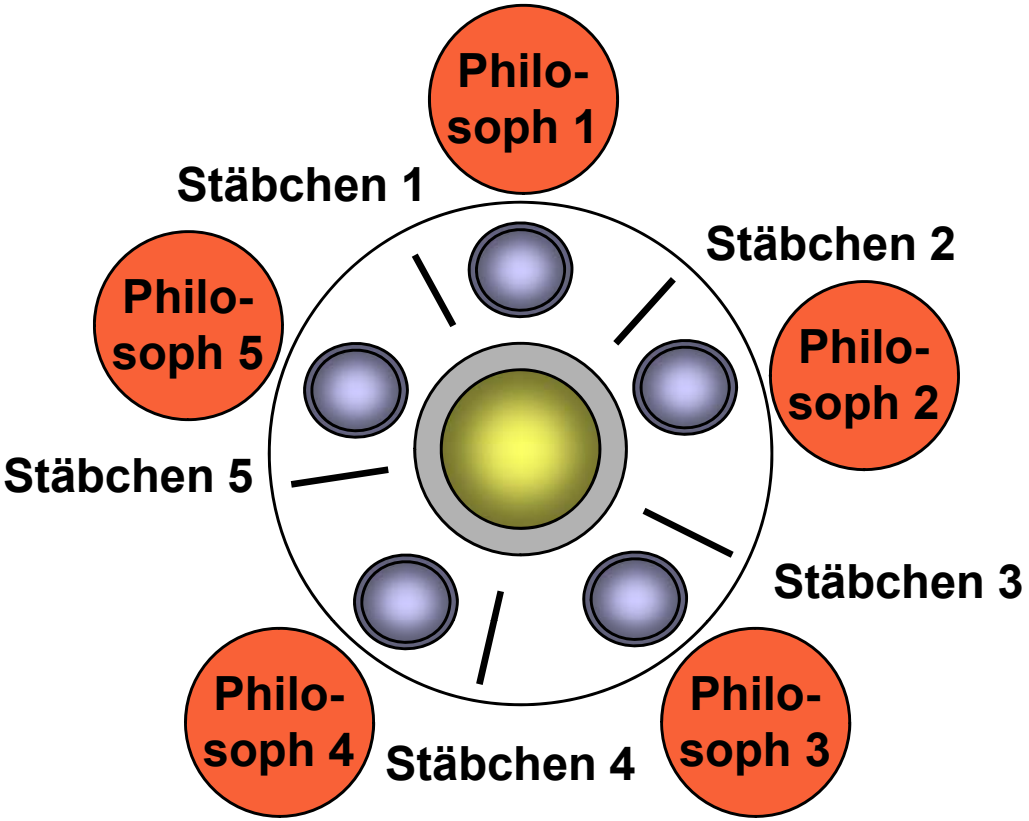
Leser und Schreiber



Markenbelegung für 3 Schreiber und 3 Leser

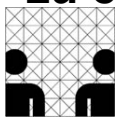
Beispiel: Die fünf speisenden Philosophen



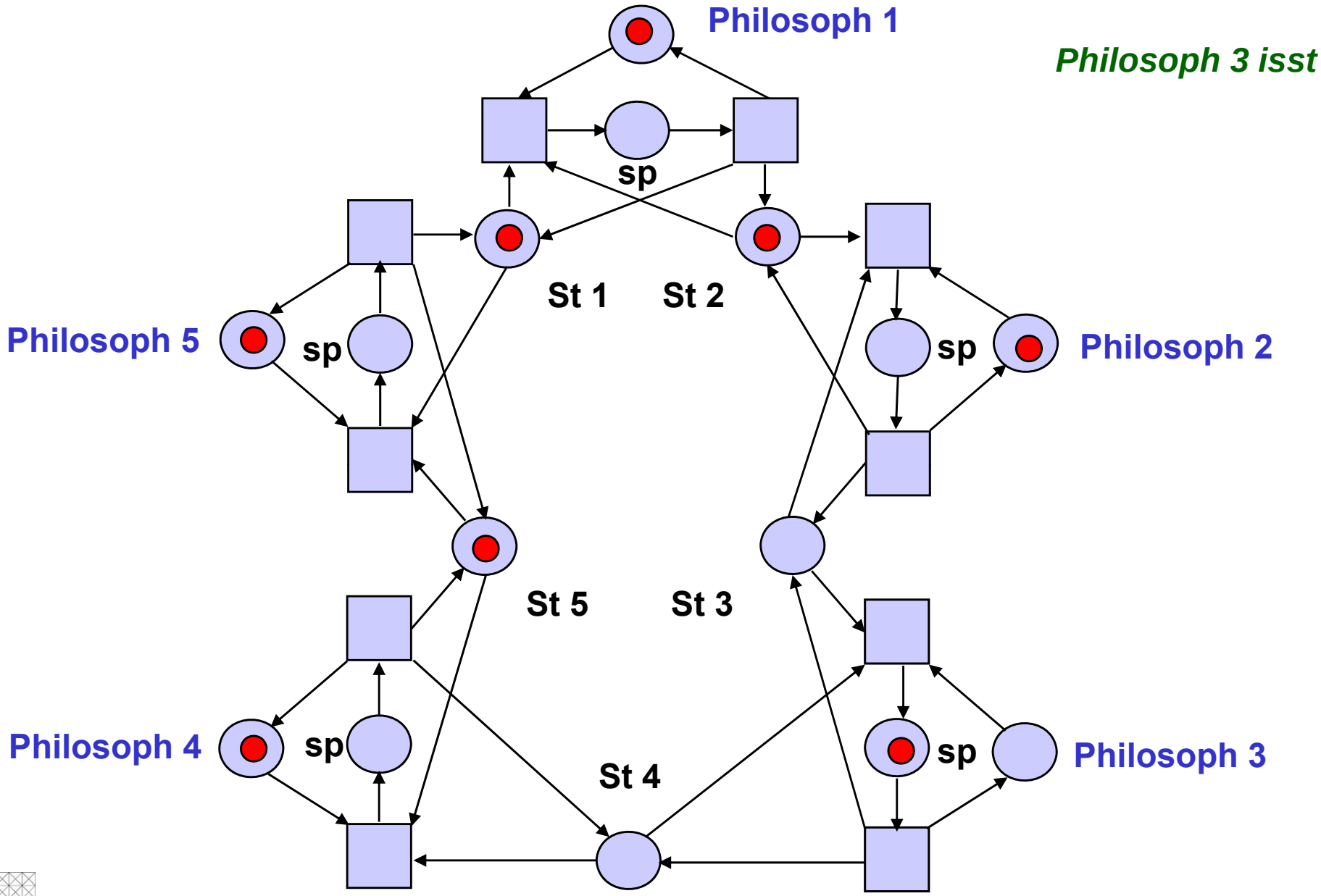


- Jeder Philosoph will entweder denken oder aus der großen Schale essen.
- Zum Essen braucht er zwei Stäbchen, sein eigenes (rechts) und das seines linken Nachbarn.

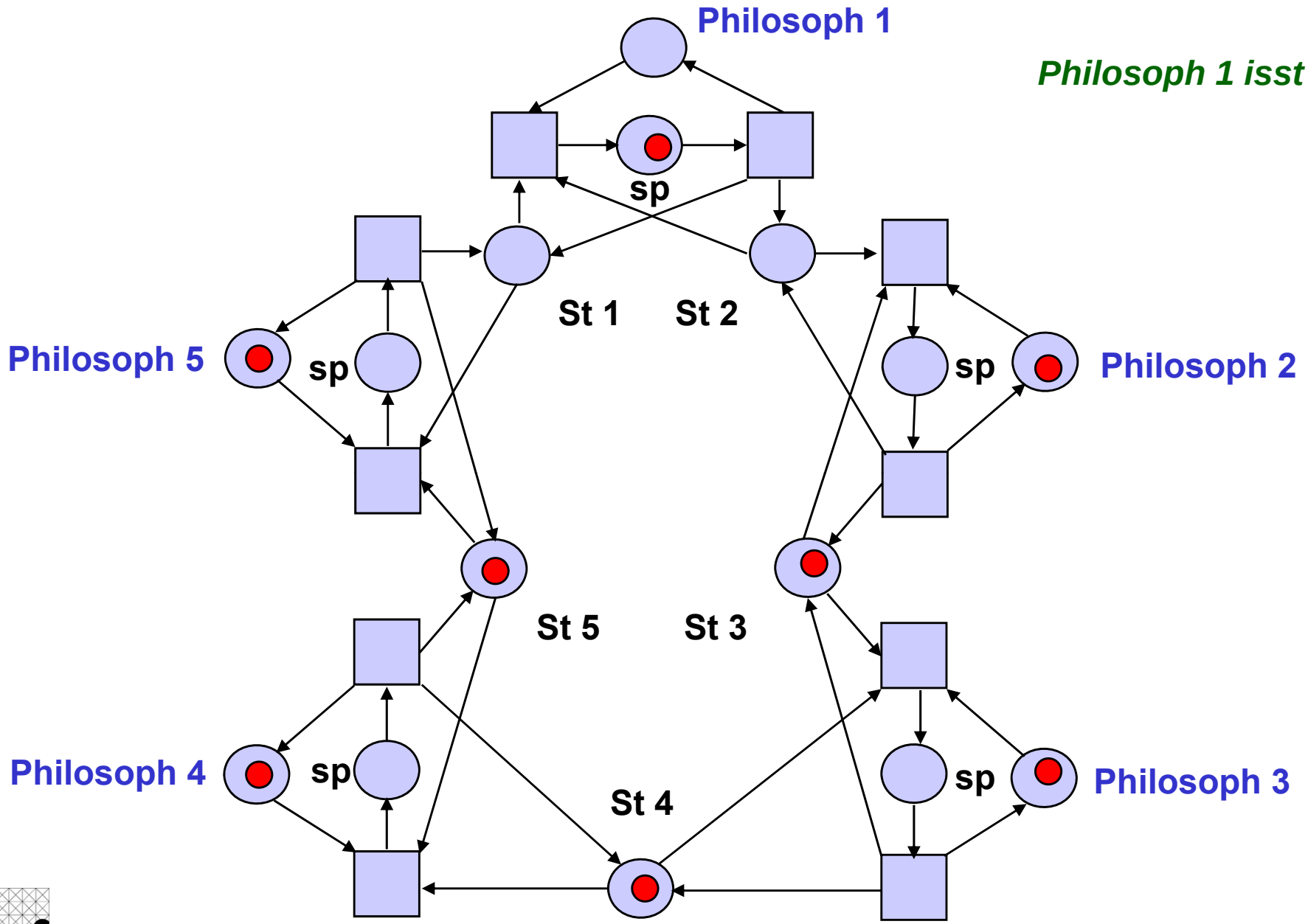
Wie können sich die Philosophen synchronisieren, so dass jeder einen fairen Anteil zu essen bekommt?



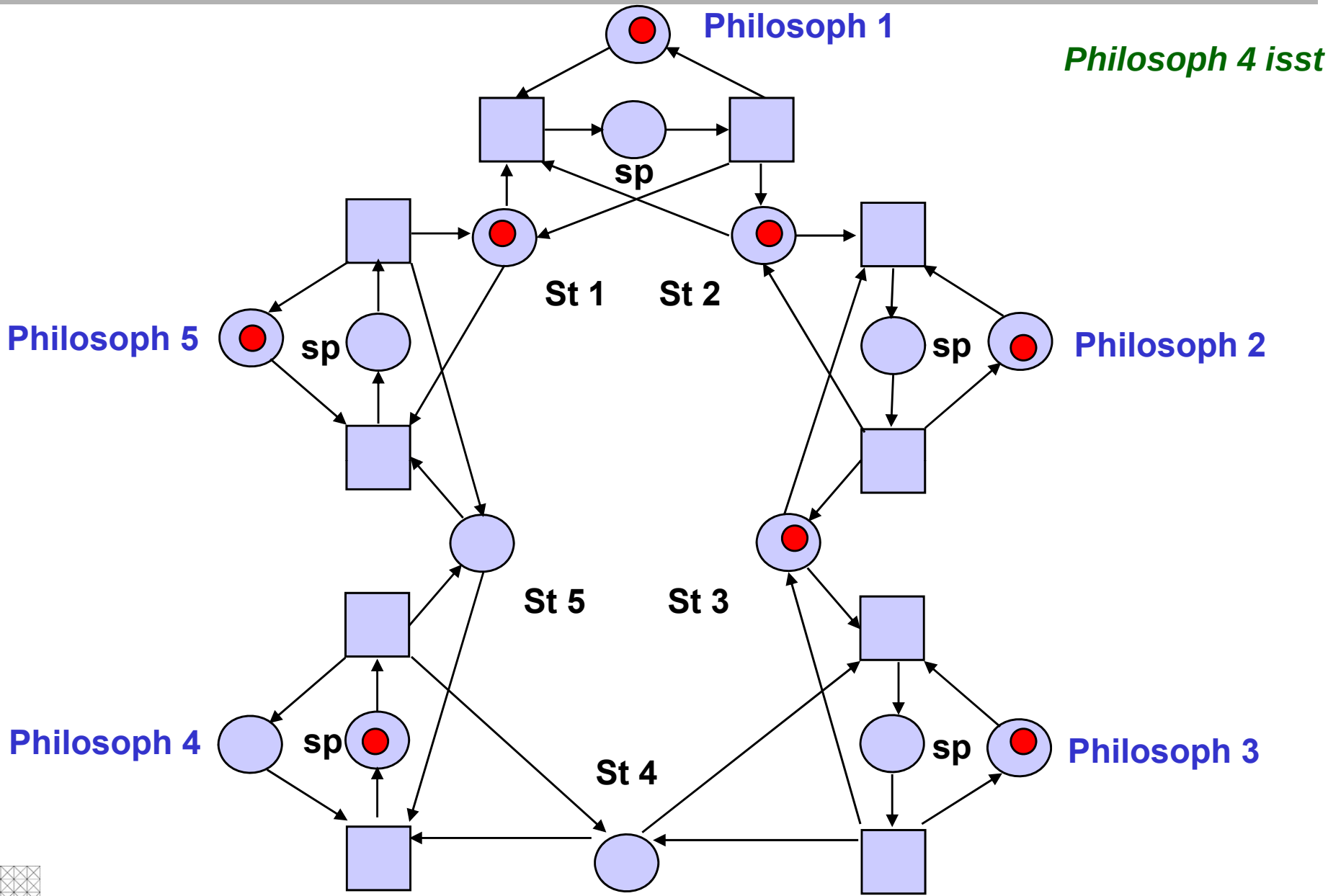
Petri-Netz für 5 speisende Philosophen (a)



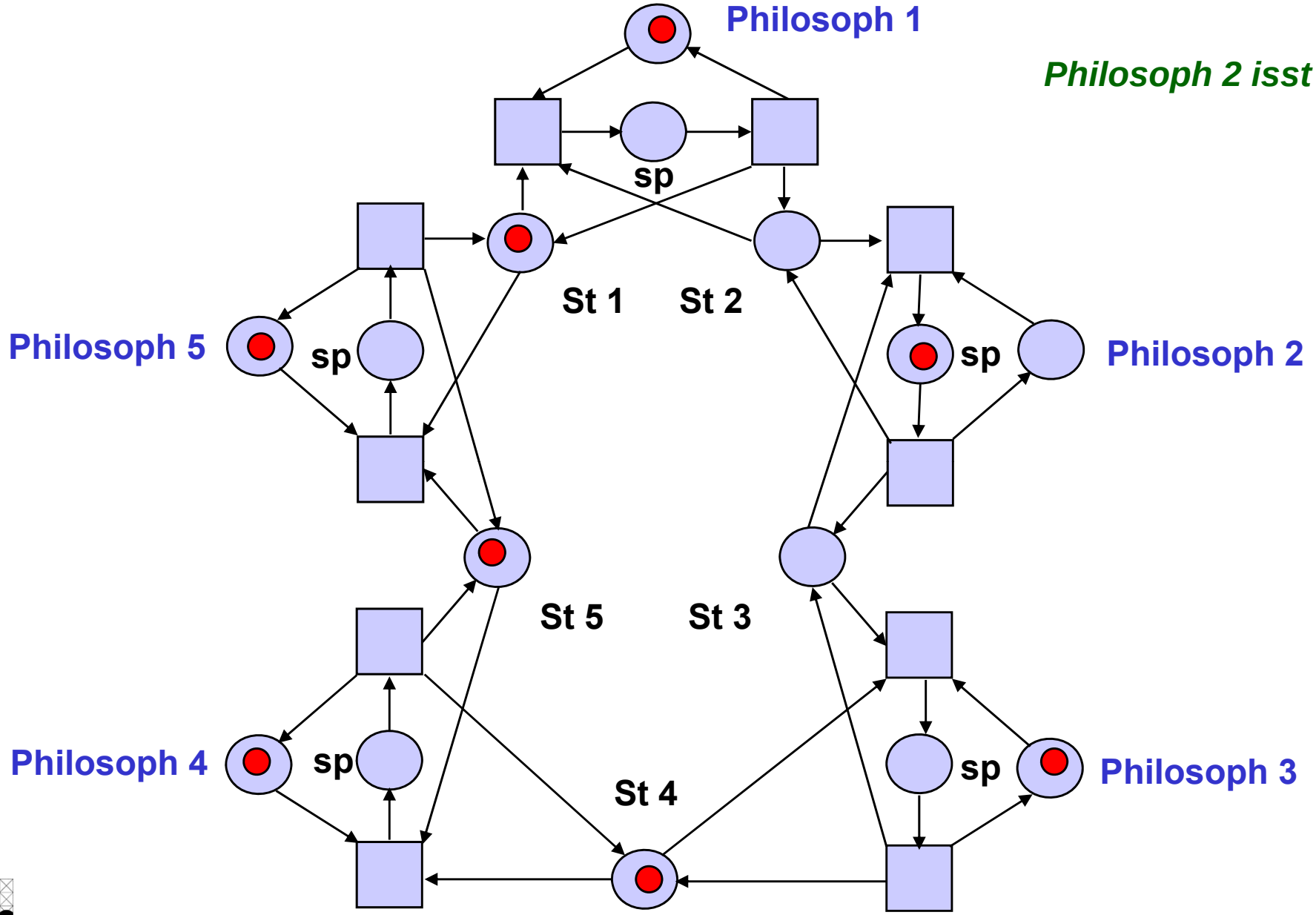
Petri-Netz für 5 speisende Philosophen (b)



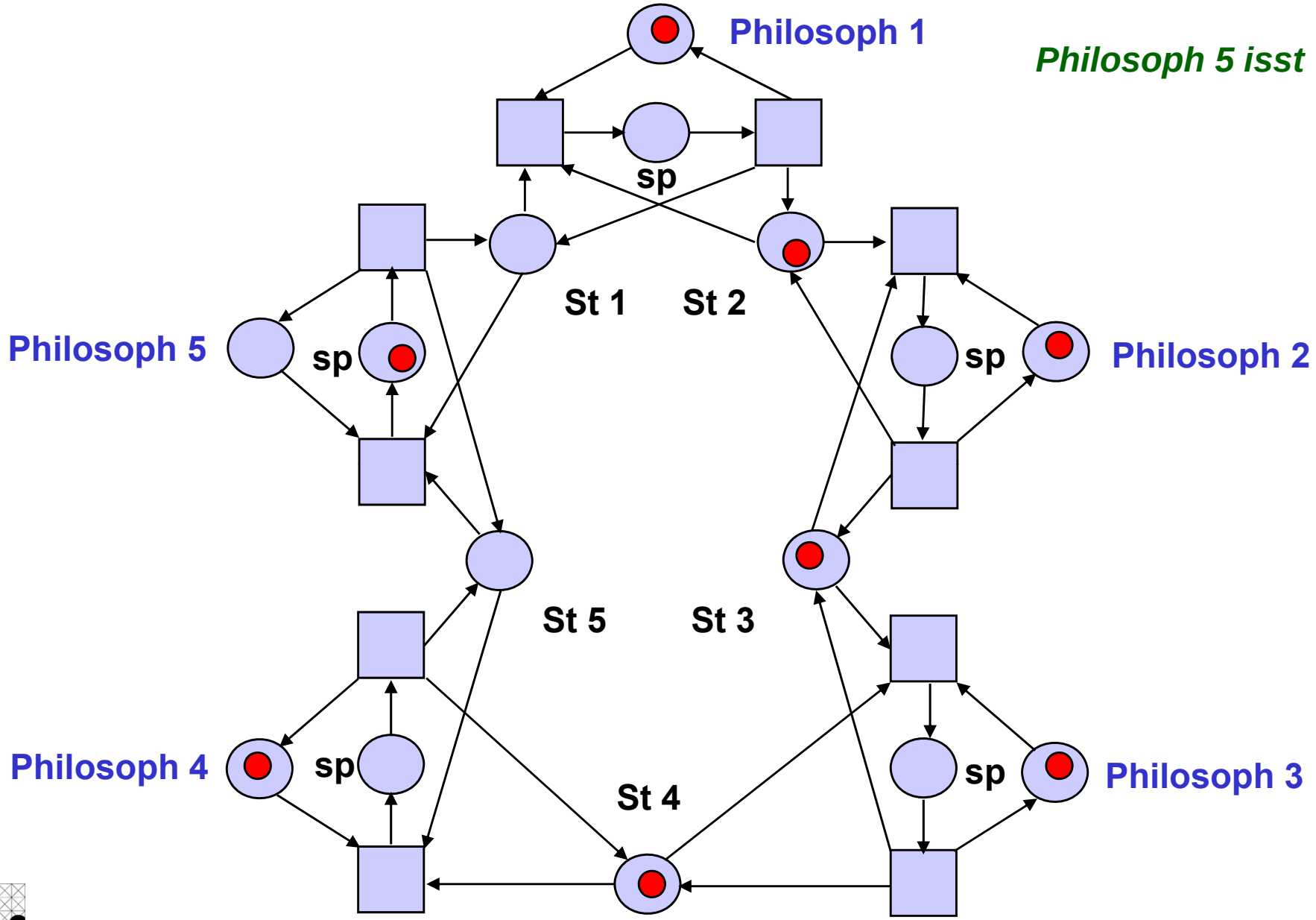
Petri-Netz für 5 speisende Philosophen (c)



Petri-Netz für 5 speisende Philosophen (d)



Petri-Netz für 5 speisende Philosophen (e)



Petri-Netz für 5 speisende Philosophen (f)