

# Übersicht und Wiederholung

Prof. Dr. David Sabel

LFE Theoretische Informatik



# 1 Einleitung

---

1.1 Warum nebenläufige Programmierung?

1.2 Begriffe der nebenläufigen Programmierung

## 2 Synchronisation

---

### 2.1 Die Interleaving-Annahme

Fairness-Annahme, atomare Aktionen, bekannte Prozesse

### 2.2 Das Mutual-Exclusion Problem

Mutual-Exklusion, Deadlockfreiheit, Starvationfreiheit

### 2.3 Mutual-Exclusion Algorithmen für zwei Prozesse

Dekker, Peterson, Kessels

### 2.4 Mutual-Exclusion Algorithmen für $n$ Prozesse

Lamports Algorithmus, Bakery-Algorithmus

### 2.5 Drei Komplexitätsresultate zum Mutual-Exclusion Problem

### 2.6 Stärkere Speicheroperationen

Nebenläufige Objekte (z.B. Test-and-set-Bit, RMW-Objekt, CAS-Objekt, Swap-Objekt,...) Algorithmen (Ticket-Algorithmus, MCS Algorithmus)

### 2.7 Konsensus und die Herlihy-Hierarchie

Prozessmodell mit Abstürzen, Konsensus-Problem, Konsensus-Zahl

# 3 Programmierprimitiven

---

## 3.1 Nebenläufigkeit in Java

## 3.2 Erweiterungen des Prozessmodells

Prozesse sind inaktiv, bereit, laufend, beendet, oder blockiert

## 3.3 Semaphore

Mutual-Exclusion mittels Semaphore, Varianten von Semaphore

## 3.4 Semaphore in Java

## 3.5 Anwendungsbeispiele für Semaphore

Erzeuger-Verbraucher Probleme, speisende Philosophen, Sleeping-Barber, Cigarette Smokers, Barrieren, Readers & Writers

## 3.6 Monitore

Condition Variablen, Arten von Monitoren, Condition Expressions

## 3.7 Einige Anwendungsbeispiele mit Monitoren

Readers & Writers, speisende Philosophen, Sleeping Barber, Barrieren

## 3.8 Monitore in Java

## 3.9 Kanäle Definition, Anwendungsbeispiele, Kanäle in Go

## 3.10 Tuple Spaces: Das Linda Modell

# 4 Zugriff auf mehrere Ressourcen

---

## 4.1 Deadlocks bei mehreren Ressourcen

4 notwendige Bedingungen

## 4.2 Deadlock-Verhinderung

2-Phasen Sperr-Protokoll (mit Timestamping), Total-Order Theorem

## 4,3 Deadlock-Vermeidung

Bankiers-Algorithmus

## 4.4 Transactional Memory

Basisprimitive, Atomare Blöcke, abort, retry, orElse, Eigenschaften von TM Systemen, Korrektheitskriterien (z.B. Sequentialisierbarkeit), TL2-Algorithmus

### 5.1 I/O in Haskell

### 5.2 Concurrent Haskell

MVars mit Operationen, forkIO, ...

### 5.3 Software Transactional Memory in Haskell

>>=, return, retry, orElse

### 6.1 Der Lambda-Kalkül

### 6.2 Ein Message-Passing-Modell: Der $\pi$ -Kalkül

synchron / asynchron, Turing-mächtig, monadisch / polyadisch, Summen,  
Bisimulation

### 6.3 CHF-Kalkül

Im Folgenden: Auswahl wichtiger Themen / Folien



## Interleaving-Annahme

Ausführung eines nebenläufigen Programms:

*Sequenz der atomaren Berechnungsschritte der Prozesse, die **beliebig durchmischt** sein können.*

## Fairness-Annahme

Jeder Prozess für den ein Berechnungsschritt möglich ist, führt in der Gesamt-Auswertungssequenz diesen Schritt nach endlich vielen Berechnungsschritten durch.

## Code-Struktur jedes Prozesses

```
loop forever
  restlicher Code
  Initialisierungscode
  Kritischer Abschnitt
  Abschlusscode
end loop
```

## Annahmen

- Programmvariablen, des Initialisierungscode u. Abschlusscodes werden durch den Code im Kritischen Abschnitt und den restlichen Code nicht verändert.
- Keine Fehler bei Ausführung des Initialisierungscode, des Codes im kritischen Abschnitt und im Abschlusscode
- Code im kritischen Abschnitt und im Abschlusscode: Nur endlich viele Ausführungsschritte

## Lösung des Mutual-Exclusion-Problems

Fülle Initialisierungs- und Abschlusscode, so dass die folgenden Anforderungen erfüllt sind:

- **Wechselseitiger Ausschluss:** Es sind niemals zwei oder mehr Prozesse zugleich in ihrem kritischen Abschnitt.
- **Deadlock-Freiheit:** Wenn **ein Prozess** seinen kritischen Abschnitt betreten möchte, dann betritt **irgendein** Prozess schließlich den kritischen Abschnitt.

## Starvation-Freiheit

Wenn **ein Prozess** seinen kritischen Abschnitt betreten möchte, dann muss **er** ihn nach endlich vielen Berechnungsschritten betreten.

# Algorithmus von Peterson

---

Initial:  $wantp = False$ ,  $wantq = False$ ,  $turn = 0$

Prozess P:

```
loop forever
(P1)  restlicher Code
(P2)   $wantp := True$ ;
(P3)   $turn := 1$ ;
(P4)   $await wantq = False \text{ or } turn = 2$ 
(P5)  Kritischer Abschnitt
(P6)   $wantp := False$ ;
end loop
```

Prozess Q:

```
loop forever
(Q1)  restlicher Code
(Q2)   $wantq := True$ ;
(Q3)   $turn := 2$ ;
(Q4)   $await wantp = False \text{ or } turn = 1$ 
(Q5)  Kritischer Abschnitt
(Q6)   $wantq := False$ ;
end loop
```

# Komplexitätsresultate bei atomarem Lesen & Schreiben

Untere Schranke für den Platz:

## Theorem

*Jeder Deadlock-freie Mutual-Exclusion Algorithmus für  $n$  Prozesse benötigt mindestens  $n$  gemeinsam genutzte Speicherplätze.*

Obere Schranke für den Platz:

## Theorem

*Es gibt einen Deadlock-freien Mutual-Exclusion Algorithmus für  $n$  Prozesse der  $n$  gemeinsame Bits verwendet.*

Laufzeit lässt sich nicht beschränken:

## Theorem

*Es gibt keinen (Deadlock-freien) Mutual-Exclusion Algorithmus für 2 (oder auch  $n$ ) Prozesse, der eine obere Schranke hat für die Anzahl an Speicherzugriffen (des gemeinsamen Speichers), die ein Prozess ausführen muss, bevor er den kritischen Abschnitt betreten darf.*

# Stärkere Speicheroperationen

## Notation

```
function xyz( $p_1 : \mathbf{Typ}_1, \dots, p_n : \mathbf{Typ}_n \dots$ ) returns : Typ  
    "atomare Ausführung des Rumpfs"  
end function
```

## test-and-set( $r, v$ )

```
function test-and-set( $r : \mathbf{Register}, v : \mathbf{Wert}$ ) returns : Wert  
    temp :=  $r$ ;  
     $r := v$ ;  
    return(temp);  
end function
```

## swap

```
function swap( $r : \mathbf{Register}, l : \mathbf{Lokales Register}$ )  
    temp :=  $r$ ;  
     $r := l$ ;  
     $l := temp$ ;  
end function
```

## Stärkere Speicheroperationen (2)

### fetch-and-add

```
function fetch-and-add(r : Register, v : Wert) returns : Wert
  temp := r;
  r := temp + v;
  return(temp);
end function
```

### read-modify-write

```
function read-modify-write(r : Register, f : Funktion)
  returns : Wert
  temp := r;
  r := f(temp);
  return(temp);
end function
```

# Stärkere Speicheroperationen (3)

## compare-and-swap

```
function compare-and-swap(r : Register, old : Wert, new : Wert)
  returns : Wert
  if r = old then
    r := new;
    return(True);
  else
    return(False);
end function
```

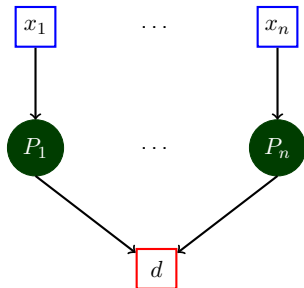
## move

```
function move(r1 : Register, r2 : Register)
  temp := r2;
  r1 := temp;
end function
```



# Das Konsensus Problem

- $n$  Prozesse, die auch abstürzen können
- Prozess  $i$  erhält einen **Eingabewert**  $x_i \in \{0, 1\}$
- Programmier die Prozesse, so dass alle (nicht-abstürzenden) Prozesse sich für einen gemeinsamen **Entscheidungswert**  $d \in \{0, 1\}$  entscheiden
- **Übereinstimmung**: Alle nicht-abgestürzten Prozesse entscheiden sich für den gleichen Wert  $d$ .
- **Gültigkeit**:  $d \in \{x_1, \dots, x_n\}$ , d.h.  $d$  ist einer der Eingabewerte.



# Lösung mit dreiwertigem RMW-Objekt

Objekte und Initialisierung:

$x$ : RMW-Objekt mit den möglichen Werten  $\perp, 0, 1$ , initial  $\perp$

$x_i$ : Eingabewert von Prozess  $i$

$d_i$ : Entscheidungswert, den Prozess  $i$  trifft.

## Programm des $i$ . Prozesses

- (1)  $d_i := \text{read-modify-write}(x, f_i);$
- (2) **if**  $d_i = \perp$  **then**  
     $d_i := x_i;$

## Funktion $f_i$ des $i$ . Prozesses

```
function  $f_i(v)$   
  if  $v = \perp$  then return  $x_i$   
  else return  $v$   
end function
```

erster Prozess setzt sein  $x_i$  als neuen Wert,  
alle anderen nicht-abstürzenden Prozesse lesen diesen Wert  
 $\Rightarrow$  alle  $d_i$ -Werte identisch

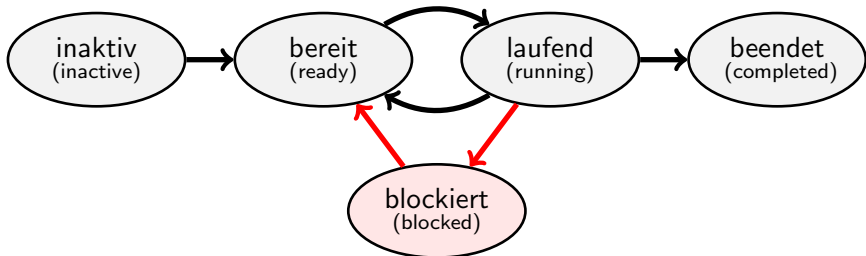
## Definition

Für ein nebenläufiges Objekt vom Typ  $o$  ist die **Konsensus-Zahl**  $\mathbb{CN}(o)$  die größte Zahl an Prozessen  $n$  für die man das Konsensus-Problem für  $n$  Prozesse lösen kann, indem man beliebig viele Objekte vom Typ  $o$  und beliebig viele atomare Register (mit *read* und *write*) verwendet. Ist die Anzahl unbeschränkt, so sei  $\mathbb{CN}(o) = \infty$ .

$\mathbb{CN}(o)$	Objekt $o$
1	atomares Register mit <i>read</i> und <i>write</i>
2	test-and-set Objekt, fetch-and-increment Objekt, fetch-and-add Objekt, swap-Objekt, read-modify-write Bit
$\Theta(\sqrt{m})$	swap <sup><math>m</math></sup> -Objekt
$2m - 2$	$m$ -Register mit $m$ -facher Zuweisung ( $m > 1$ )
$\infty$	(drei-wertiges) RMW-Objekt, Compare-and-swap-Objekt, Sticky-Bit

# Erweitertes Prozessmodell

Prozesse  $P$  haben einen Zustand  $P.state$ :



## Attribute (i.a.):

- $V$  = Nicht-negative Ganzzahl
- $M$  = Menge von Prozessen

Schreibweise für Semaphor  $S$ :  $S.V$  und  $S.M$

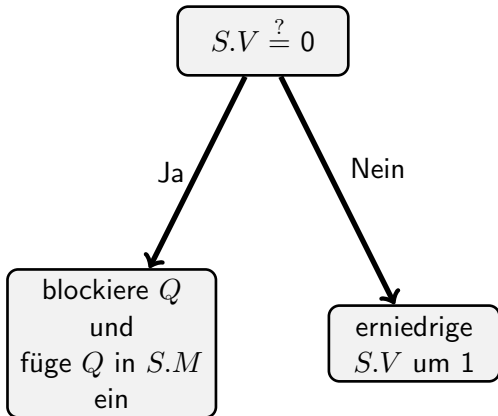
## Operationen:

- `newSem( $k$ )`: Erzeugt neuen Semaphor mit  $S.V = k$  und  $S.M = \emptyset$
- `wait( $S$ )`
- `signal( $S$ )`

# wait(*S*)

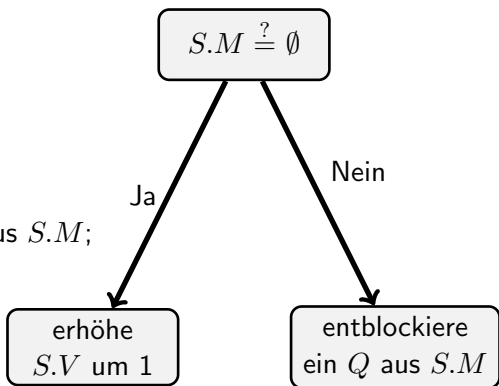
Sei *Q* der aufrufende Prozess:

```
procedure wait(S)  
  if  $S.V > 0$  then  
     $S.V := S.V - 1$ ;  
  else  
     $S.M := S.M \cup \{Q\}$ ;  
     $Q.state := blocked$ ;
```



# signal(*S*)

```
procedure signal(S)  
  if  $S.M = \emptyset$  then  
     $S.V := S.V + 1$ ;  
  else  
    wähle ein Element  $Q$  aus  $S.M$ ;  
     $S.M := S.M \setminus \{Q\}$ ;  
     $Q.state := ready$ ;
```



# Erzeuger / Verbraucher

---

- Erzeuger: Produziert Daten
- Verbraucher: Konsumiert Daten
- Beispiel: Tastatur / Betriebssystem usw.

Erzeuger / Verbraucher mit infinite Buffer:

- Lesen / Schreiben auf den Puffer **sicher** (atomar)
- Verbraucher braucht Schutz für den Fall, dass der Puffer leer ist

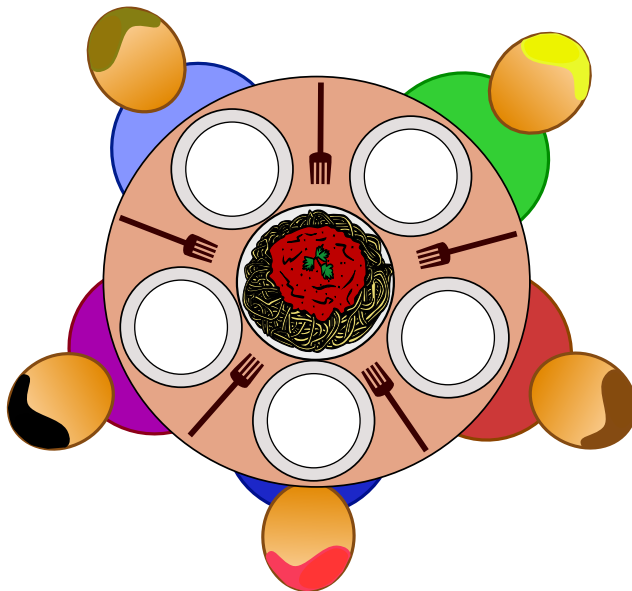
Erzeuger / Verbraucher mit bounded Buffer:

- Lesen / Schreiben auf den Puffer **sicher** (atomar)
- Verbraucher braucht Schutz für den Fall, dass der Puffer leer ist
- Erzeuger braucht Schutz für den Fall, dass der Puffer voll ist



# Speisende Philosophen

---



## Situation

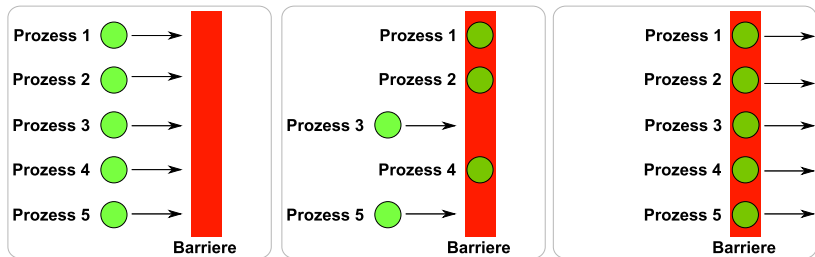
- Philosoph denkt oder isst Spaghetti, abwechselnd
- Philosoph braucht beide Gabeln zum Essen
- Philosoph nimmt Gabeln **nacheinander**

## Anforderungen:

- Kein Deadlock: Irgendein Philosoph kann nach endlicher Zeit immer essen
- Kein Verhungern: Jeder Philosoph isst nach endlicher Zeit

- Manche Algorithmen erfordern “Phasen”
- D.h.: Die Prozesse führen Berechnungen durch, aber an einem Schritt warten alle Prozesse aufeinander
- Erst wenn alle an dieser Stelle angekommen sind, dürfen die Prozesse weiter rechnen
- Ähnlich war es beim Mergesort-Beispiel, dort wartet allerdings nur ein Prozess auf zwei weitere

# Allgemeines Schema



## Gruppierung der Prozesse in

- **Readers:** Prozesse, die auf eine gemeinsame Ressource **lesend** zugreifen
- **Writers:** Prozesse, die auf die gemeinsame Ressource **schreibend** zugreifen

## Erlaubt / Nicht erlaubt

- Mehrere lesende Prozesse gleichzeitig, aber
- Nur ein Prozess schreibt gleichzeitig

## Problem:

- Löse den Zugriff so, dass viele gleichzeitig lesen, aber nie mehrere gleichzeitig schreiben.

## Verschiedene Lösungen:

- Priorität für Readers
- Priorität für Writers

```
monitor Konto {  
  int Saldo;  
  int Kontonummer;  
  int KundenId  
  
  abheben(int x) {  
    Saldo := Saldo - x;  
  }  
  
  zubuchen(int x) {  
    Saldo := Saldo + x;  
  }  
}
```

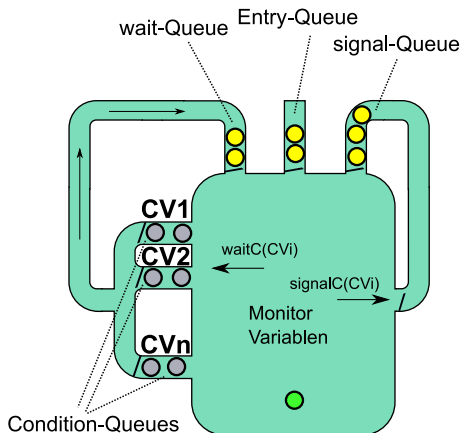
- Kapselung von Daten und Methoden
- **Kein** direkter Zugriff auf Attribute
- Zugriff nur über die Methoden
- Nur **ein** Prozess kann zu einer Zeit **im** Monitor sein
- D.h. nur eine Methode von einem Prozess zu einer Zeit am Ausführen
- Andere Prozesse werden blockiert

# Monitore mit Condition Variables

- FIFO-Queue (meistens) mit Operationen
- Name der Condition Variables wird meistens so gewählt, dass er die wahr werdene Bedingung erläutert, aber
- Operationen für Condition Variable cond:  
waitC(cond) und signalC(cond)

Semaphore Sem	Monitore (Condition Variable cond)
wait(Sem) kann zum Blockieren führen, muss aber nicht	waitC(cond) blockiert den Prozess <b>stets</b>
signal(Sem) hat stets einen Effekt: Entblockieren eines Prozesses oder Erhöhen von Sem.V	signalC(cond) kann effektivlos sein: Entweder Prozess in cond wird entblockiert, oder effektivlos, wenn cond leer ist

# Monitor-Modellierung mit drei Queues für die Condition Variable



## Prioritäten

- 1  $E = W = S$
- 2  $E = W < S$  Wait and Notify
- 3  $E = S < W$  Signal and Wait
- 4  $E < W = S$
- 5  $E < W < S$  Signal and Continue
- 6  $E < S < W$  Klassische Definition
- 7  $E > W = S$  nicht sinnvoll
- 8  $E = S > W$  nicht sinnvoll
- 9  $S > E > W$  nicht sinnvoll
- 10  $E = W > S$  nicht sinnvoll
- 11  $W > E > S$  nicht sinnvoll
- 12  $E > S > W$  nicht sinnvoll
- 13  $E > W > S$  nicht sinnvoll

- **Wartende Prozesse: Verwaltung durch Monitor.lock**
- **Wartende Prozesse: Verwaltung durch Condition**
- **Prozess im Monitor**



```
class MonitoredClass {  
    ... Attribute ...  
    synchronized method1 {...}  
  
    synchronized method2 {...}
```

## Statt Condition Variables

- Operationen `wait`, `notify`, `notifyAll`
- Nur eine Queue pro Objekt
- `wait()`: Thread wartet an der Queue des Objekts
- `notify()`: Ein wartender Thread wird entblockiert, aber:  
Aufrufender Prozess behält Lock!
- `notifyAll()`: Alle wartende Threads werden entblockiert,  
aber:  
Aufrufender Prozess behält Lock!
- Wartende Threads haben gleiche Priorität wie neue!
- Entspricht  $W = E < S$

- $ch \leftarrow w$ 
  - entspricht: “sende  $w$  über den Kanal  $ch$ ”
  - dabei ist  $w$  ein Wert vom passenden Typ
  - wir schreiben auch  $ch \leftarrow x$ , für eine Programmvariable  $x$   
Semantik: Sende den Wert der Variablen  $x$  über Kanal  $ch$
  - in Go:  $ch \leftarrow w$
- $ch \Rightarrow x$ 
  - entspricht “empfange über den Kanal  $ch$  und setze Variable  $x$  auf den empfangenen Wert”
  - Hier: Nur Variablen erlaubt!
  - In Go:  $x := \leftarrow ch$

# Tuple Spaces: Operationen

---

$\text{out}(N, v_1, \dots, v_n)$ :

- Einfügen eines Tupels in den Tuple Space
- $N$ : Name (String)
- $v_i$ : Programmvariable oder Wert
- Wenn  $v_i$  Programmvariable, dann wird der aktuelle Wert von  $v_i$  eingefügt

$\text{in}(N, x_1, \dots, x_n)$ :

- Entfernen eines Tupels aus dem Tuple Space
- $N$ : Name (String),  $x_i$ : Programmvariable
- Im Tuple Space muss ein "Matching Tuple" vorhanden sein (= gleiche Länge, gleiche Typen, gleiche Werte)
- Falls kein passendes Tuple vorhanden: Prozess **blockiert**
- Ansonsten: Variablen werden durch das Entfernen an die Werte gebunden

$\text{read}(N, x_1, \dots, x_n)$ : Wie in aber ohne Entfernen des Tupels

# Mutual-Exclusion mit Tuple Spaces

---

Initial: Tupel ("MUTEX") im Tuple Space

Prozess i:

```
loop forever
(1) Restlicher Code;
(2) in("MUTEX");
(3) Kritischer Abschnitt;
(4) out("MUTEX");
end loop
```

# Wann tritt globaler Deadlock auf?

---

**Vier notwendige Bedingungen** (alle gleichzeitig erfüllt):

- ➊ **Wechselseitiger Ausschluss (Mutual-Exclusion):** Nur ein Prozess kann gleichzeitig auf eine Ressource zugreifen,
- ➋ **Halten und Warten (Hold and Wait):** Ein Prozess kann eine Ressource anfordern (auf eine Ressource warten), während er eine andere Ressource bereits belegt hat.
- ➌ **Keine Bevorzugung (No Preemption):** Jede Ressource kann nur durch den Prozess freigegeben (entsperrt) werden, der sie belegt hat.
- ➍ **Zirkuläres Warten:** Es gibt zyklische Abhängigkeit zwischen wartenden Prozessen: Jeder wartende Prozess möchte Zugriff auf die Ressource, die der nächste Prozesse im Zyklus belegt hat.

# Deadlock-Verhinderung: Verhindern von Hold and Wait

---

- Möglichkeit: Prozess fordert zu Beginn alle Ressourcen an, die er benötigt.
- Philosophen: Exklusiver Zugriff auf alle Gabeln
- 1. Problem: Evtl. zu sequentiell
- 2. Problem: Oft nicht klar, welche Ressourcen jeder Prozess braucht
- Variation dieser Lösung: **2-Phasen Sperrprotokoll**

## 2-Phasen Sperrprotokoll

---

Die Prozesse arbeiten in zwei Phasen

Jeder Prozess führt dabei aus:

- **1. Phase:** Der Prozess versucht alle benötigten Ressourcen zu belegen.  
Ist **eine** benötigte Ressource **nicht frei**, so gibt der Prozess **alle** belegten Ressourcen zurück und der Prozess startet von neuem mit Phase 1.
- **2. Phase:** Der Prozess hat alle benötigten Ressourcen  
Nachdem er fertig mit seiner Berechnung ist, gibt er alle Ressourcen wieder frei.

- Prozesse erhalten **eindeutigen Zeitstempel**, wenn sie beginnen.

## Zwei-Phasen Sperrprotokoll mit Timestamping

Jeder Prozess geht dabei so vor:

- 1. Phase: Der Prozess versucht alle benötigten Ressourcen auf einmal zu sperren.  
Ist eine benötigte Ressource belegt mit **kleinerem Zeitstempel**, dann gibt Prozess **alle** Ressourcen frei und startet von neuem.  
Ist **eigener** Zeitstempel **kleiner**, dann wartet der Prozess auf die restlichen Ressourcen.
- 2. Phase: Wenn der Prozess erfolgreich in diese Phase gekommen ist, hat er alle benötigten Ressourcen. Er benutzt sie und gibt sie anschließend wieder frei.

Beachte: Neue Zeitstempel werden nur vergeben, nach erfolgreichem Durchlauf durch beide Phasen.



# Deadlock-Verhinderung: Keine Zirkularität zulassen, Total-Order Theorem

---

## **Total-Order Theorem**

Sind alle gemeinsamen Ressourcen durch eine totale Ordnung geordnet und jeder Prozess belegt seine benötigten Ressourcen in aufsteigender Reihenfolge bezüglich der totalen Ordnung, dann ist ein Deadlock unmöglich.

# Deadlock-Vermeidung: Bankier-Algorithmus

```
function testeZustand( $\mathcal{P}$ ,  $\vec{A}$ ):  
  if  $\mathcal{P} = \emptyset$  then  
    return "sicher"  
  else  
    if  $\exists P \in \mathcal{P}$  mit  $\vec{M}_P - \vec{C}_P \leq \vec{A}$  then  
       $\vec{A} := \vec{A} + \vec{C}_P$ ;  
       $\mathcal{P} := \mathcal{P} \setminus \{P\}$ ;  
      testeZustand( $\mathcal{P}$ ,  $\vec{A}$ )  
    else  
      return "unsicher"
```

atomic-Blöcke:

```
atomic {  
    Code der Transaktion  
}
```

Der `retry`-Befehl

- Ermöglicht es Transaktionen zu koordinieren
- `retry`: Transaktion wird abgebrochen (Roll-back) und erneut gestartet

Der `orElse`-Befehl

- Gibt Alternativen vor, wenn Transaktionen abbrechen
- $T_1$  `orElse`  $T_2$ .

# Korrektheitskriterien für STM-Systeme

---

Historie= Folge von Ereignissen, wobei Ereignis:

- Aufrufe & Rückgaben v.  $READ(x)$ ,  $WRITE(x,v)$ ,  $COMMIT$ ,  $ABORT$
- Spezialwert  $A_T$  = Transaktion  $T$  ist abgebrochen.

## Sequentialisierbarkeit

- Historie der einzelnen Read/Write-Zugriffe der **erfolgreichen** Transaktionen
- Historie des nebenläufigen Ablaufs ist **äquivalent** zur Historie eines sequentiellen Ablaufs aller erfolgreichen Transaktionen.

## Äquivalenz:

Zwei Historien sind äquivalent, wenn die Ereignisefolge pro Transaktion dieselbe ist (d.h. gleiche Reihenfolge innerhalb einer Transaktion und gleiche Rückgaben).

## Sequentieller Ablauf:

Die Ereignisse verschiedener Transaktionen treten nicht verzahnt sondern sequentiell nacheinander auf.

- `forkIO :: IO () -> IO ThreadId`
- Terminierung des main-Threads, beendet alle Threads
- `newEmptyMVar :: IO (MVar a)`  
erzeugt leere MVar
- `takeMVar :: MVar a -> IO a`
  - liest Wert aus MVar, danach ist die MVar leer
  - falls MVar vorher leer: Thread wartet
  - Bei mehreren Threads: FIFO-Warteschlange
- `putMVar :: MVar a -> a -> IO ()`
  - speichert Wert in der MVar, wenn diese leer ist
  - Falls belegt: Thread wartet
  - Bei mehreren Threads: FIFO-Warteschlange

- `atomically :: STM a -> IO a` überführt eine STM-Aktion in eine IO-Operation
- `data TVar a = ...`
- `newTVar :: a -> STM (TVar a)`  
Erzeugt eine neue TVar mit Inhalt
- `readTVar :: TVar a -> STM a`  
Liest den den momentanen Wert einer TVar
- `writeTVar :: TVar a -> a -> STM ()`  
Schreibt einen neuen Wert in die TVar
- `newTVarIO :: TVar a -> a -> IO (TVar a)`  
Erzeugen einer TVar in der IO Monade
- `retry :: STM a`
- `>>=, >>, do`
- `orElse :: STM a -> STM a -> STM a`

## Syntax

- $\mathcal{N}$  abzählbar unendliche Menge von **Namen** (ähnlich zu Variablen)
- Syntax für  $\pi$ -Kalkül-**Prozesse** ( $x \in \mathcal{N}$ )

$P$	$::=$	$\pi.P$	(Aktion)
		$P_1 \mid P_2$	(Parallele Komposition)
		$!P$	(Replikation)
		$\mathbf{0}$	(Inaktiver Prozess)
		$\nu x.P$	(Restriktion)

- Syntax für **Aktionspräfixe** wobei  $x, y \in \mathcal{N}$

$\pi$	$::=$	$x(y)$	Input
		$\bar{x}y$	Output

## Reduktionsregeln

(INTERACT)  $x(y).P \mid \bar{x}v.Q \rightarrow P[v/y] \mid Q$

(PAR)  $P \mid Q \rightarrow P' \mid Q$ , falls  $P \rightarrow P'$

(NEW)  $\nu x.P \rightarrow \nu x.P'$ , falls  $P \rightarrow P'$

(STRUCTCONGR)  $P \rightarrow P'$ , falls  $Q \rightarrow Q'$ ,  $P \equiv Q$  und  $P' \equiv Q'$



## Weiteres zum $\pi$ -Kalkül

---

- Turing-mächtig (da Lambda-Kalkül simulierbar)
- asynchron vs. synchron
- monadisch vs. polyadisch
- Rekursion vs. Replikation