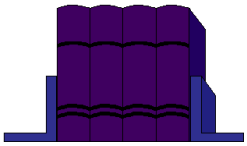
Nebenläufigkeit

Bisher wurde ein Berechnungsvorgang als zeitliche Folge einzelner Berechnungsschritte modelliert (*sequentieller* Prozeß). In realen Systemen können sich Prozesse zeitlich überlappen und interagieren - sie sind *nebenläufig*.

Wir befassen uns mit wichtigen Aspekten nebenläufiger Systeme:

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



A. Kemper, A. Eickler:
Datenbanksysteme -
Eine Einführung, 3. Auflage
Oldenbourg 1999

R.G. Herrtwich, G. Hommel
Nebenläufige Programme,
Springer 1994

J. Magee, J. Kramer
Concurrency -
State models & Java programs,
Wiley 1999

1

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)	

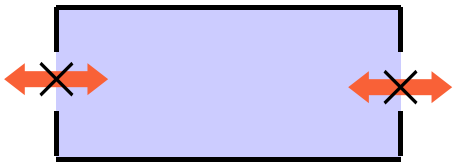
Wo ist die Zinsgutschrift geblieben??

2

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)
}
```

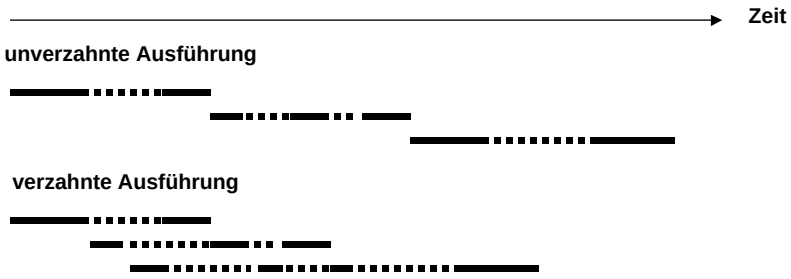
Verzahnte Ausführung der zwei Prozesse Drehkreuz1 und Drehkreuz2 mit Zugriff auf gemeinsamen Counter kann inkorrekte Besucherzahl ergeben!

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

3

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

4

Mehrbenutzerbetrieb von Datenbanksystemen

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

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

Da Fehler während einer Transaktion auftreten können, muß eine Transaktionsverwaltung dafür sorgen, daß unvollständige Transaktionen ggf. zurückgenommen werden können.

Befehle für Transaktionsverwaltung:

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

5

Eigenschaften von Transaktionen

ACID-Paradigma steht für 4 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 Datenbank, auch bei Hardware- und Software-Fehlern.

6

Problembereiche bei Mehrbenutzerbetrieb von Datenbanksystemen

Synchronisation mehrerer nebenläufiger Transaktionen:

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

Wir können in P3 nur einige Themen anschneiden, Vertiefung in weiterführenden Lehrveranstaltungen.

7

Synchronisation bei Mehrbenutzerbetrieb

Synchronisationsproblem = verzahnte sequentielle Ausführung nebenläufiger Transaktionen, so daß deren Wirkung der intendierten unverzahnten ("seriellen") Hintereinanderausführung der Transaktionen entspricht.

Konfliktursache im DB-Kontext ist read und write von zwei 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.

8

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

Serialisierung 1

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

Serialisierung 2

Der Effekt dieser Verzahnung entspricht keiner der 2 möglichen Serialisierungen T1 vor T2 oder T2 vor T1: Die Historie ist nicht serialisierbar

9

Sperrsynchronisation

Viele Datenbank-Scheduler verwenden Sperranweisungen zur Erzeugung konfliktfreier Abläufe:

- **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	-	-

10

Zwei-Phasen-Sperrprotokoll

(Englisch: two-phase locking, 2PL)

Protokoll gewährleistet die Serialisierbarkeit von Transaktionen.
Für jede individuelle Transaktion muß gelten:

1. **Jedes von einer Transaktion betroffene Objekt muß vorher entsprechend gesperrt werden.**
2. **Eine Transaktion fordert eine Sperre, die sie besitzt, nicht erneut an.**
3. **Eine Transaktion muß solange warten, bis es eine erforderliche Sperre entsprechend der Verträglichkeitstabelle erhalten kann.**
4. **Jede Transaktion durchläuft 2 Phasen:**
   - **in Wachstumsphase werden Sperren angefordert, aber nicht freigegeben**
   - **in Schrumpfungsphase werden Sperren freigegeben, aber nicht angefordert**
5. **Bei EOT (Transaktionsende) muß eine Transaktion alle ihre Sperren zurückgeben.**

Verschärfung zum "Strengen 2PL-Protokoll" zur Vermeidung nicht-rücksetzbarer Abläufe:

Keine Schrumpfungsphase, alle Sperren werden bei EOT freigegeben.

11

Beispiel für 2PL-Verzahnung

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

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

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	

12

Verklemmungen (Deadlocks)

Sperrbasierte Synchronisationsmethoden können (unvermeidbar) zu Verklemmungen führen:

Gegenseitiges Warten auf Freigabe von Sperren

Transaktionen leicht modifiziert:

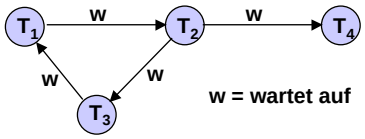
T1: Modifikation von A und B (z.B. Umbuchung)

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

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 zur Erkennung und Vermeidung von Verklemmungen

1. Wartegraph hat Zyklen



w = wartet auf

Nach Erkennen eines Zyklus muß Verklemmung durch Zurücksetzen einer geeigneten Transaktion beseitigt werden.

2. Preclaiming - Vorabforderung aller Sperren
Beginn einer Transaktion erst, nachdem die für diese Transaktion insgesamt erforderlichen Sperren erfolgt sind.
Problem: Vorab die erforderlichen Sperren erkennen

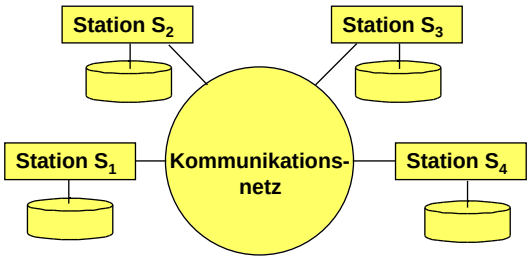
3. Zeitstempel
Transaktionen werden durch Zeitstempel priorisiert. Zurücksetzen statt Warten, wenn T_1 Sperre fordert, T_2 aber Sperre erst freigeben muß:

- **Strategie Wound-wait: Abbruch von T_2, falls T_2 jünger als T_1, sonst warten**
- **Strategie Wait-die: Abbruch von T_1, wenn T_1 jünger als T_2, sonst warten**

Verteilte Datenbanksysteme

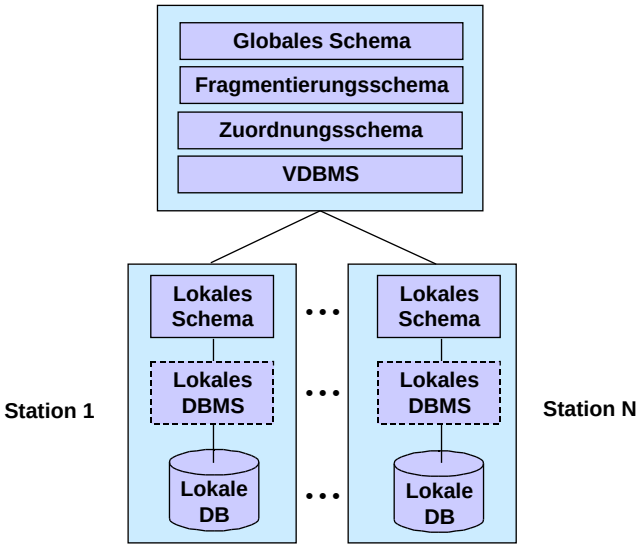
Die zunehmende Vernetzung von Datenbanken führt zum Konzept der "verteilten Datenbank".

Eine verteilte Datenbank ist eine Sammlung von Informationseinheiten, die auf mehrere über ein Kommunikationsnetz miteinander verbundene Rechner verteilt sind.



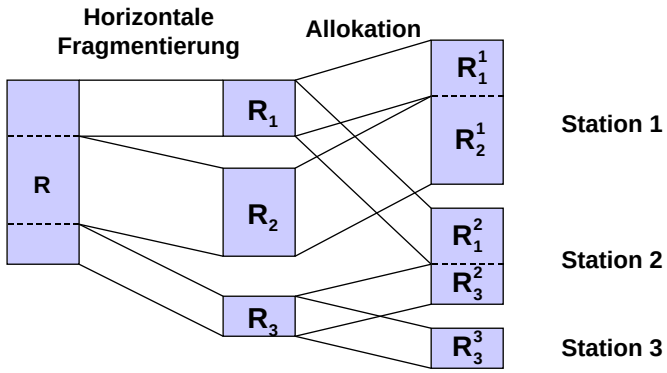
15

Schematischer Aufbau eines verteilten Datenbanksystems



16

Fragmentierung und Allokation einer Relation



Horizontale Fragmentierung:

Zerlegung einer Relation in disjunkte Tupelmengen

Vertikale Fragmentierung:

Zerlegung einer Relation durch Projektionen

Allokation:

Zuordnung von Fragmenten auf Stationen (ggf. mit Replikation)

17

Horizontale Fragmentierung

Definition von *Zerlegungsprädikaten* p_i ergibt Tupelmengen entsprechend SQL-Anweisung:

R_i := select * from R where p_i

Definition von *Selektionsprädikat*

$q = q_1 \wedge \ldots \wedge q_n$

$q_i \in \{p_1, \neg p_1, \ldots, p_m, \neg p_m\}$

ergibt Tupelmenge entsprechend SQL-Anweisung:

R_q := select * from R where q

18

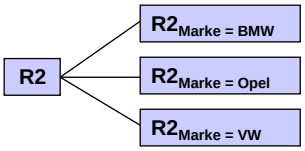
Abgeleitete horizontale Fragmentierung

Der Zugriff auf verteilte Datenbanken kann effizienter gemacht werden, wenn die horizontale Fragmentierung einer Relation R1 von der horizontalen Fragmentierung einer Relation R2 abgeleitet wird.

Beispiel: Autohandel

R1 = { [Verkäufer KfzNr] } R2 = { [KfzNr Marke Preisgruppe] }

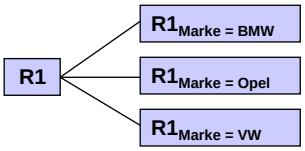
R2 sei entsprechend der Marke fragmentiert:



Anfrage:

```
select Verkäufer, Preisgruppe
from R1, R2
where Marke = Opel
```

Verbesserung der Effizienz durch abgeleitete Fragmentierung von R1 bezüglich R2 mithilfe Semi-Join:



19

Übersicht über Join-Operationen

L		
A	B	C
a1	b1	c1
a2	b2	c2

x

R		
C	D	E
c1	d1	e1
c3	d2	e2

natürlicher Join:

A	B	C	D	E
a1	b1	c1	d1	e1

linker äußerer Join:

A	B	C	D	E
a1	b1	c1	d1	e1
a2	b2	c2	-	-

rechter äußerer Join:

A	B	C	D	E
a1	b1	c1	d1	e1
-	-	c3	d2	e2

äußerer Join:

A	B	C	D	E
a1	b1	c1	d1	e1
a2	b2	c2	-	-
-	-	c3	d2	e2

Semi-Join von L mit R:

A	B	C
a1	b1	c1

Semi-Join von R mit L:

C	D	E
c1	d1	e1

20

Vertikale Fragmentierung

Zerlegung einer Relation durch Projektionen, um Zugriffe auf Teilmengen der Attribute entsprechend verschiedener Anwendungen zu unterstützen.

Beispiel:

Professoren = {[PersNr Name Raum Fakultät Rang Gehalt Steuerklasse]}

1. Fragment: ProfVerw = {[PersNr Name Rang Gehalt Steuerklasse]}

2. Fragment: Profs = {[PersNr Name Raum Fakultät]}

Um *Rekonstruierbarkeit* zu gewährleisten, erhält jedes (vertikale) Fragment den Primärschlüssel der Originalrelation (oder ein Surrogat).

***Redundanz* (d.h. Überlappung von Fragmenten)**

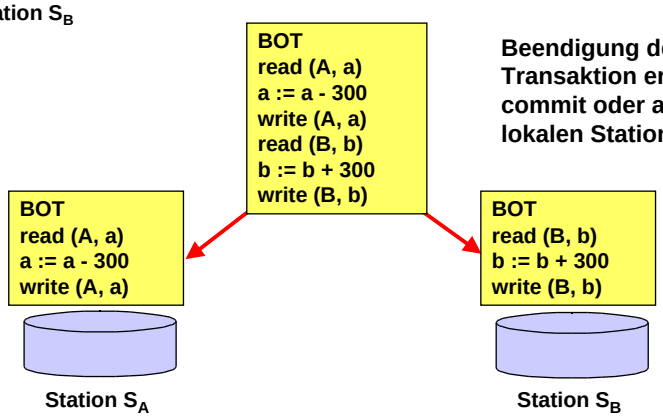
- **ist hinsichtlich des Primärschlüssels notwendig,**
- **kann aus Anwendungssicht nützlich sein,**
- **führt zu Mehraufwand bei Änderungen.**

21

Transaktionskontrolle in verteilten Datenbanksystemen

Verteilte Datenbankverwaltungssysteme (VDBMS) verwalten Transaktionen, die sich über mehrere Stationen (Rechner) erstrecken.

Beispiel: Umbuchung eines Betrags von Konto A auf Station S_A nach Konto B auf Station S_B



Beendigung der globalen Transaktion erfordert commit oder abort *für alle* lokalen Stationen

22

Zweiphasen-Commit-Protokoll

2PC-Protokoll zur EOT-Behandlung in verteilten Datenbanksystemen

K Koordinator

A_1 ... A_n Agenten (Stationen des verteilten Datenbanksystems)

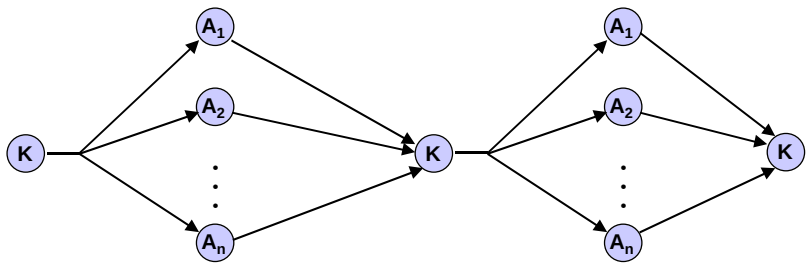
1. K sendet PREPARE-Nachricht an alle Agenten, um abzufragen, ob diese ihre Transaktion festschreiben können
2. Jeder Agent A_i antwortet mit einer der zwei Nachrichten:
   - READY, falls A_i lokale Transaktion festschreiben kann
   - FAILED, falls A_i lokale Transaktion nicht festschreiben kann
3. Wenn K READY von allen Agenten empfangen hat, sendet K COMMIT an alle Agenten und fordert damit zum Festschreiben auf. Falls nicht alle READY innerhalb Timeout-Frist empfangen wurden oder mindestens ein FAILED empfangen wurde, sendet K ABORT an alle Agenten und fordert damit zum Abbruch auf.
4. Agenten quittieren den Erhalt von Nachricht 3 mit ACK (Acknowledgement, Bestätigung) und führen entsprechende EOT-Behandlung durch.

23

Nachrichtenaustausch beim 2PC-Protokoll

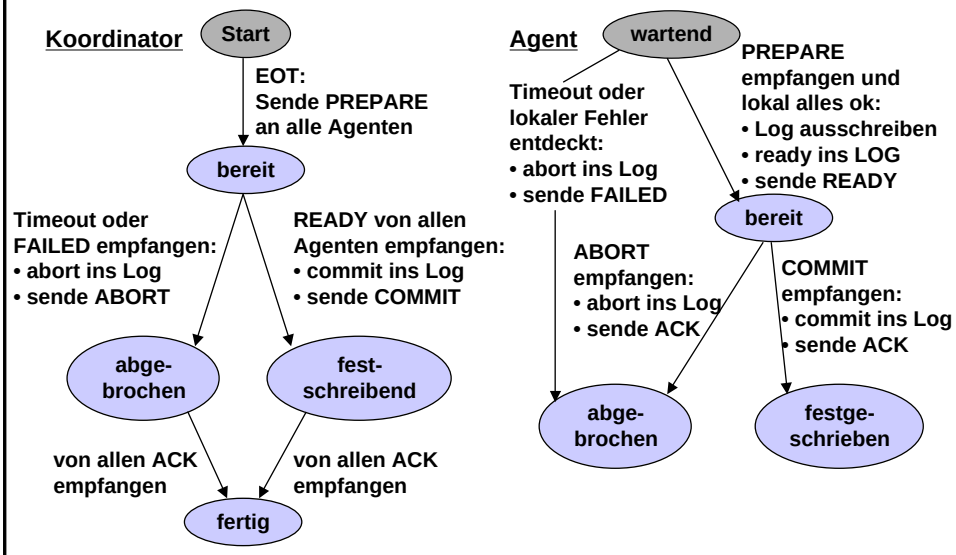
K Koordinator

A_1 ... A_n Agenten (Stationen des verteilten Datenbanksystems)



24

Zustandsübergänge beim 2PC-Protokoll



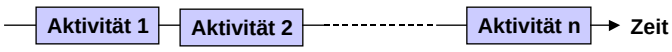
Prozesse in der Informatik

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 Startzeitpunkt 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**

27

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**

28

Nichtdeterminismus und Determiniertheit

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

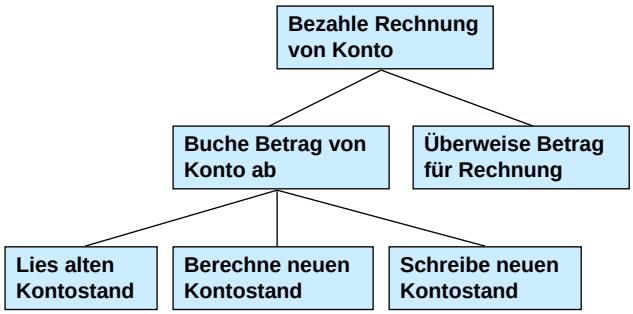
Man ist jedoch i.A. daran interessiert, daß nebenläufige Prozesse ein *determiniertes* Ergebnis haben, egal wie verzahnt sie ausgeführt werden.

Auch *nichtdeterminierte* 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

29

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.

30

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**

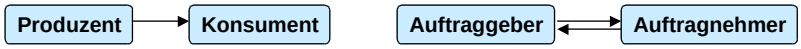
31

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).



Produtenten/Konsumenten-System (producer/consumer system)

Auftraggeber/Auftragnehmer-System (client/server system)

2. Grundform der Abhängigkeit: Konkurrenz

Prozesse behindern sich durch Nutzung begrenzter Ressourcen.

Kampf um Betriebsmittel

32

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

33

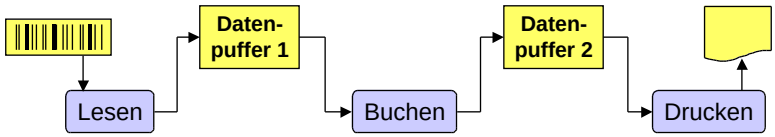
Einseitige Synchronisation

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

34

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

35

Programmiertechnische Lösungen für Synchronisationsaufgaben

Wir haben bei Problemen mit Nebenläufigkeit den Bedarf von Synchronisation kennengelernt (z.B. bei nebenläufigen Kontozugriffen)

Wie löst man Synchronisationsaufgaben programmiertechnisch?

Zwei Klassen von Lösungen:

1. **Schloßvariable => Aktives Warten (busy-wait)**
2. **Semaphore und höhere Abstraktionsformen => Prozeßverwaltung (process control)**

Einseitige Synchronisation: Aktivität1 → Aktivität2

Prozeß 1:	Prozeß 2:
flag1 = false;	...
...	while (! flag1)
<Aktivität1>	{ };
flag1 = true;	<Aktivität2>
...	...

Prozeß 1:	Prozeß 2:
...	...
<Aktivität1>	P(semaphore)
V(semaphore);	<Aktivität2>
...	...

mehr zu Sema-phoren später!

36

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;}
}
```

Funktioniert nicht, weil Lesen und Schreiben der Schloßvariablen nicht ununterbrechbar sind!

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

37

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 die 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 meldet, entscheidet Prioritätsvariable* favorit *über Vorrang*

→ *kritischer Abschnitt*

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

38

Semaphore

Semaphor ist Zähler mit Prozeßverwaltungskompetenz: Statt aktivem 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();
...                                ...
}                                  }
```

39

Implementierung nebenläufiger Prozesse

Vorgestellte Synchronisationsmethoden verwenden meist Ausdrucksmöglichkeiten klassischer Programmiersprachen auf niedriger Abstraktionsebene:

- **Semaphore**
- **kritische Abschnitte**
- **Monitore**

Problematisch bei komplexen Synchronisierungsaufgaben!

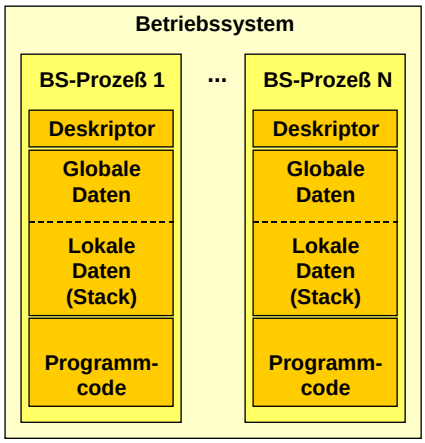
Moderne Programmiersprachen (wie Java) bieten vorgefertigte Möglichkeiten, nebenläufige Prozesse und Synchronisationsverfahren auf höherer Abstraktionsebene zu definieren.

40

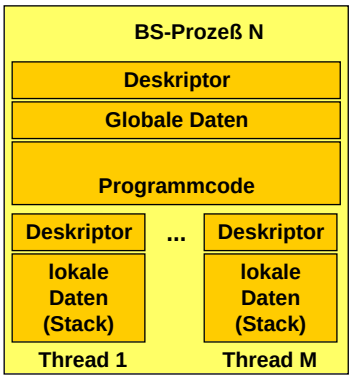
Schwergewichtige und leichtgewichtige Prozesse

Schwergewichtige Prozesse eines Betriebssystems (BS):

Aufträge mit Ressourcenbedarf

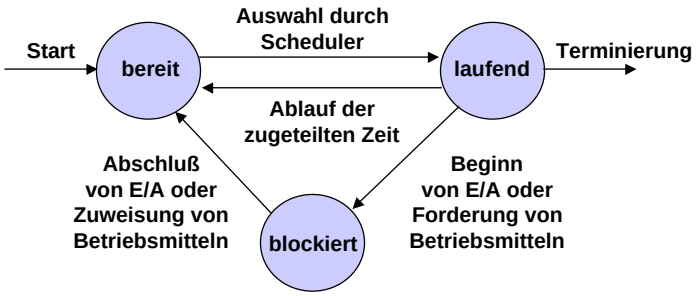


Leichtgewichtige Prozesse (Threads) als Teile eines BS-Prozesses

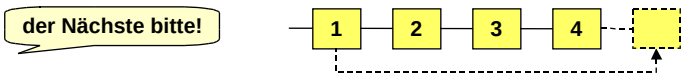


41

Prozessorzuteilung durch Scheduler

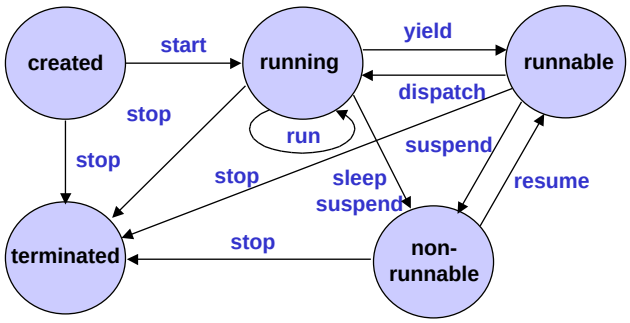


- Status "bereit" kann mehrere Warteschlangen mit verschiedener Priorität besitzen
- Einordnung von Prozessen nach dem "Round-Robin"-Verfahren



42

Lebenszyklus von Java-Threads



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

43

Synchronisation in Java

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

Prozeßoperationen *wait* und *notify* ermöglichen Prozeßverwaltung.

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!

44

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

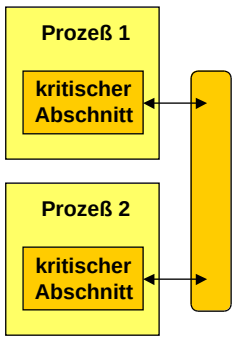
passives Warten, bis Semaphor positiven Wert hat, dann dekrementieren und Passage

nach Inkrementieren nächsten wartenden Prozess aktivieren

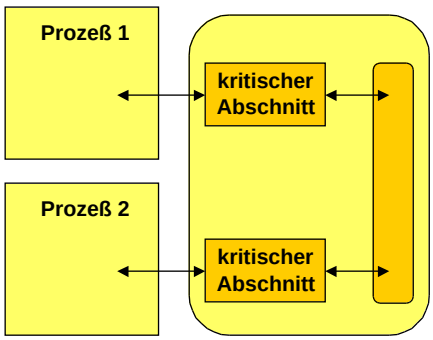
45

Semaphore versus Monitore

Kapselung der kritischen Abschnitte kann größere Klarheit schaffen



gemeinsame Variable (Semaphore)



gemeinsame abstrakte Datenstruktur (Monitor)

46

Implementierung des Produzenten-Konsumenten-Problems (1)

```
class Produkt {
    private int Ware;
    private boolean verfügbar = false;
    public synchronized int verbraucht() {
        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 buchführen

47

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

Testprogramm

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

Ausdruck bei Testlauf zeigt abwechselnde Produktion und Konsumption:

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

48

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] = 0;
        ++count;
        in = (in + 1)%size;
        notify();
    }
    public synchronized Object get()
            throws InterruptException {
        while (count == 0) wait();
        Object o = buf[out];
        buf[out] = null;
        --count;
        out = (out + 1)%size;
        notify();
        return (o);
    }
}
```

9

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)*

50

Prozeßkommunikation

Bisher haben Prozesse über gemeinsam zugreifbare Variablen 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)

51

Relevante Eigenschaften für Synchronisierung

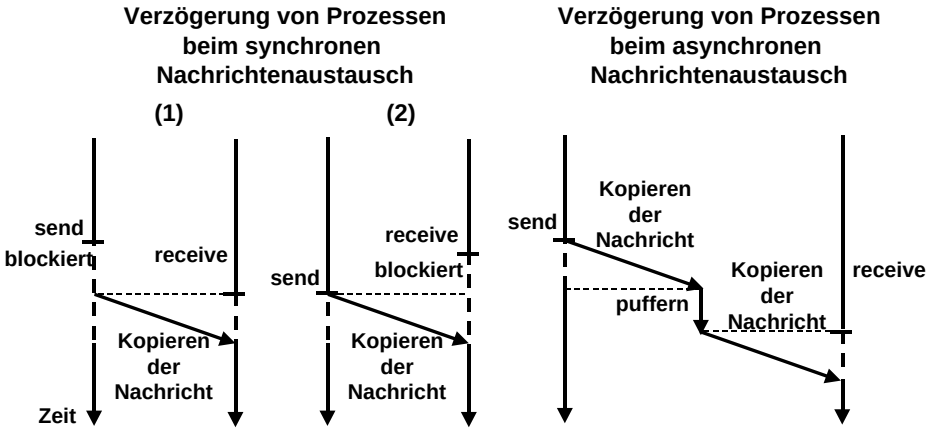
Senden von Nachrichten	
blockierend:	***Prozeß wartet nach Sendeoperation auf Empfangsbestätigung***
nichtblockierend:	***Prozeß läuft nach Sendeoperation weiter***

Empfangen von Nachrichten	
blockierend (üblich):	***Prozeß wartet auf Empfang einer Nachricht***
nichtblockierend:	***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***

52

Synchroner und asynchroner Nachrichtenaustausch



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

53

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
```

54

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***

55

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 *mithilfe 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*

56

Prozeßkommunikation mit Occam

Occam ist verbreitete Prozeßkommunikationssprache für Mehrprozessorsysteme (insbesondere Transputer), basiert auf Hoare´s CSP (Communicating Sequential Processes).

```
CHAN Kanal :
PAR                          → nebenläufige Anweisungen folgen
   Kanal ! 'a'               → sende Wert 'a' über Kanal
   VAR c :
   Kanal ? c                 → empfange Wert fürVariable c über Kanal
```

```
CHAN Kanal1, Kanal2 :
VAR count, neg :
ALT                                  → nichtdeterministische Auswahl
   (count < 100) & Kanal1 ? ANY      → irgendetwas empfangen?
      count := count + 1
   (count > 0) & Kanal2 ? ANY
      SEQ                            → sequentielle Anweisungen folgen
         count := count - 1
         neg := TRUE
```

57

Replikatoren in Occam

Replikatoren beschreiben Wiederholungen zur Laufzeit oder Reihungen von Anweisungen. Grundmuster:

index = [basiswert FOR maxzahl]

Beispiele:

```
VAR x [n], s :          → Feldvereinbarung für x
SEQ
   s := 0
   SEQ i = [0 FOR n-1]  → Konventionelle FOR-Schleife
   s := s + x [i]
```

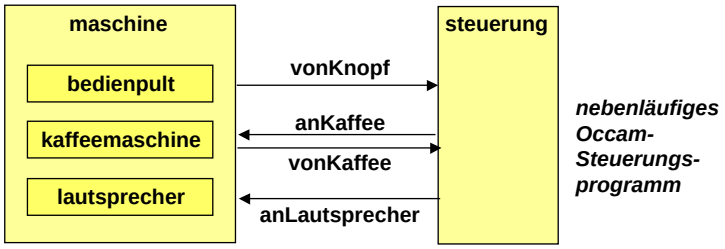
```
CHAN k [n] :            → Vereinbarung von n Kanälen
VAR s :
SEQ
   s := 0
   WHILE TRUE
      VAR x :
      ALT i = [0 FOR n-1]
         k [i] ? x      → Aufaddieren von Werten, die
         s := s + x       nebenläufig über n Kanäle
                          eintreffen
```

58

Steuerung einer Kaffeemaschine mit einem Occam-Programm

Die Kaffeemaschine kann

- **auf Knopfdruck Kaffee machen**
- **fertigen Kaffee in eine Tasse gießen**
- **auf Knopfdruck die Uhrzeit ansagen**
- **zur Weckzeit Morgengruß sagen und Kaffee machen**



Kommunikationskanäle

Steuerungsprogramm (1)

```
PROC steuerung =
  VAR inArbeit :
  SEQ
    inArbeit := FALSE
    WHILE TRUE
      ALT
        vonKnopf ? druck
          IF
            (druck = kaffee) AND NOT inArbeit
              PAR
                anKaffee ! kaffeeFiltern
                inArbeit := TRUE
            (druck = Zeit)
              anLautsprecher ! sagZeit; NOW
        ...
        vonKaffee ? ANY
          SEQ
            anKaffee ! tasseFüllen
            anLautsprecher ! sagSpruch; "Der Kaffee ist fertig!"
            inArbeit := FALSE
        ...
```

1. Alternative von neben-läufigen Ereignissen

2. Alternative von neben-läufigen Ereignissen

Steuerungsprogramm (2)

```
...
WAIT NOW AFTER weckzeit
   SEQ
      weckzeit := weckzeit + einTag
      anLautsprecher ! sagSpruch; "Guten Morgen!"
      IF
         NOT inArbeit
            PAR
                  anKaffee ! kaffeeFiltern
                  inArbeit := TRUE
...
```

3. Alternative von neben-läufigen Ereignissen

WAIT-Anweisung: Warte, bis Bedingung wahr ist

WAIT <bedingung>

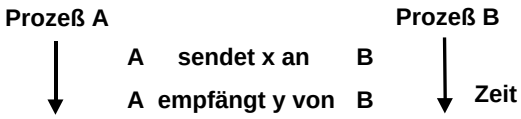
Zeitbedingung: **T1 AFTER T2**

Aktuelle Zeit: **NOW**

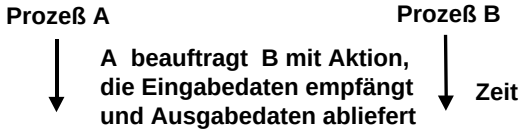
61

Abstraktion vom Nachrichtenaustausch

Bisher datenorientierter Nachrichtenaustausch mit typischem Muster:



Aktionsorientierter Nachrichtenaustausch bietet Abstraktions-möglichkeit durch Zusammenfassen der Aktivitäten von B als Aktion:



62

Fernaufruf von Prozeduren

Die Beziehung zwischen Auftraggeber und Auftragnehmer läßt sich durch einen Prozedurfernaufruf (remote procedure call, RPC) in vertrauter Weise modellieren.

Unterschied zum Prozeduraufruf:

- **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
```

63

Rendezvous

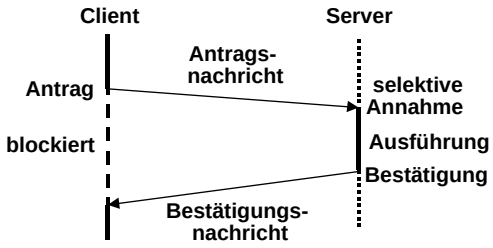
Rendezvous = Prozedurfernaufruf mit größerer Autonomie des aufgerufenen Prozesses (bestimmt selbst über Ausführung des Auftrags)

Sprachgebrauch:

Client **beantragt (request) einen Dienst**

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.

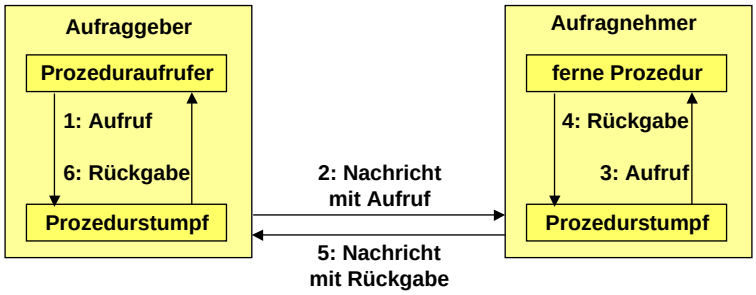


64

Implementierung von Prozedurfernaufrufen

Prozedurstumpf repräsentiert ferne Prozedur im Adreßraum des Auftraggebers und sorgt für Nachrichtenaustausch mit Auftragnehmer.

Prozedurstumpf auf Auftragnehmerseite sorgt für Prozeduraufruf und Nachrichtenaustausch mit Auftraggeber.

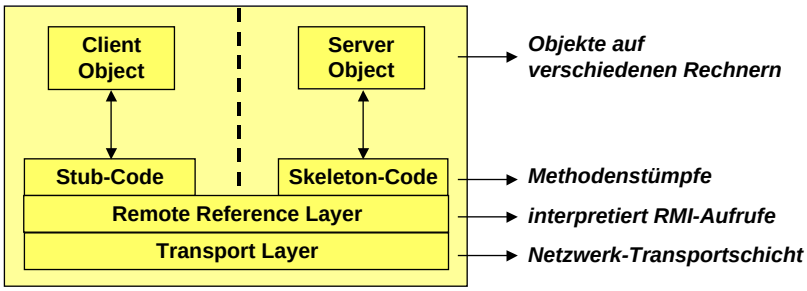


65

Methodenfernaufruf in Java

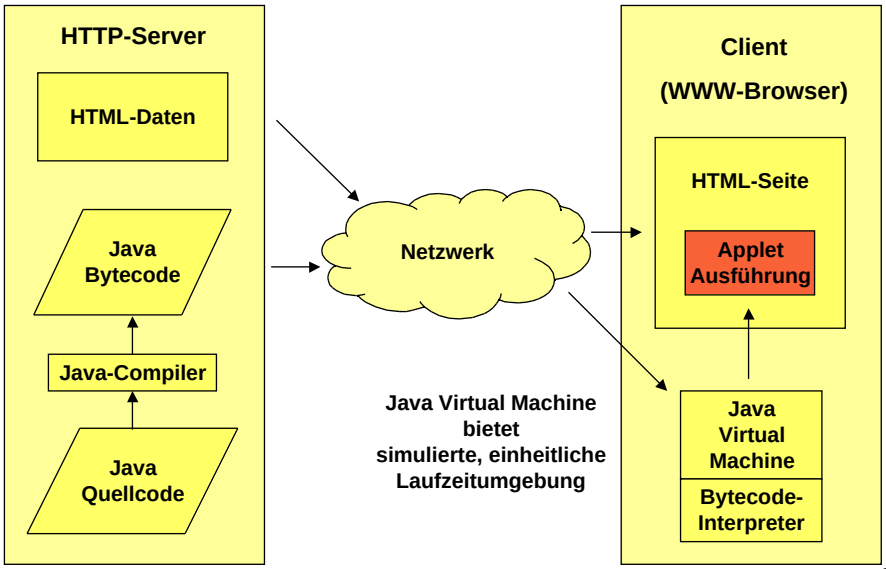
Java-Objekt auf Rechner A (Client) kann Methoden eines entfernten Objektes auf Rechner B (Server) durch Remote Method Invocation (RMI) aufrufen.

Architektur des RMI-Systems:



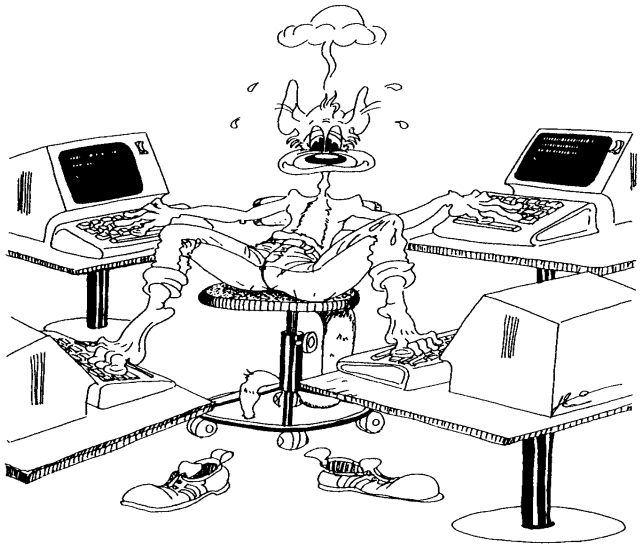
66

Fernausführung von Java-Applets im WWW



67

Nebenläufigkeitsprobleme erfordern abstrakte Modellierung

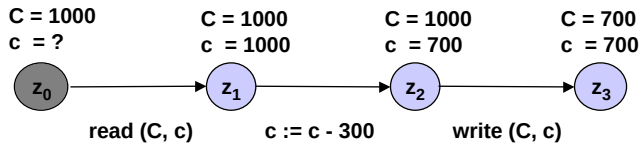


68

Modellierung von Prozessen durch endliche Automaten

Prozeßzustände	**Zustände eines Automaten**
Aktivitäten	**Eingaben des Automaten, bewirken Zustandsübergänge**

Beispiel:



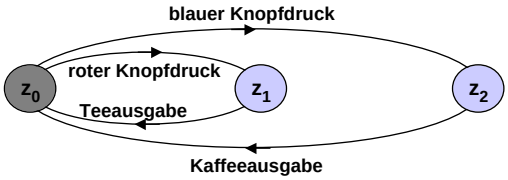
$Z = \{ z_0 \ldots z_N \}$	**Menge der Zustände**
$A = \{ a_0 \ldots a_M \}$	**Menge der Aktivitäten**
$E = \{ (z_i\ a_j\ z_k) \}$	**Zustandsübergänge des endlichen Automaten**

Deterministische Wahl

Eine deterministische Wahl besteht, wenn ein Prozeß von einem Zustand aus durch unterschiedliche Aktivitäten in verschiedene Folgezustände übergehen kann.

Beispiel:

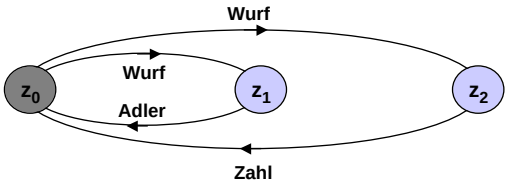
Getränkeautomat hat roten Knopf für Tee und blauen Knopf für Kaffee



Nichtdeterministische Wahl

Ein Prozeß ist nichtdeterministisch, wenn er bei gleicher Aktivität in verschiedene Zustände übergehen kann.

<u>Beispiel</u>: Münzwurf

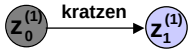


Parallele Ausführung nebenläufiger Prozesse

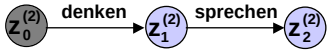
Beschreibung der möglichen Verzahnungen zweier nebenläufiger Prozesse durch einen Produktautomaten.

<u>Beispiel</u>:

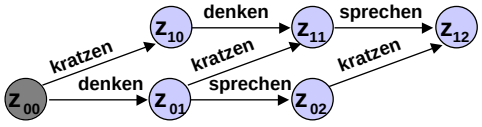
Jucken



Vortragen



Jucken || Vortragen



<u>Definition Produktautomat</u>:

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

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

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

Nebenläufige Prozesse mit gemeinsamen Aktivitäten

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

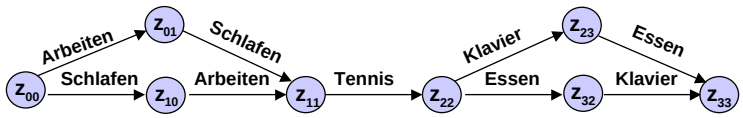
Beispiel:

gemeinsame Aktivität "Tennis"

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

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

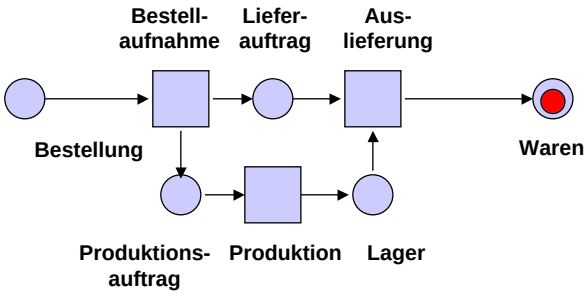
Felix || Marietta $= \{(z_{00}\ \text{Schlafen}\ z_{10})(z_{01}\ \text{Schlafen}\ z_{11})(z_{00}\ \text{Arbeiten}\ z_{01})(z_{10}\ \text{Arbeiten}\ z_{11})$
$(z_{11}\ \text{Tennis}\ z_{22})$
$(z_{22}\ \text{Essen}\ z_{32})(z_{23}\ \text{Essen}\ z_{33})(z_{22}\ \text{Klavier}\ z_{23})(z_{32}\ \text{Klavier}\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

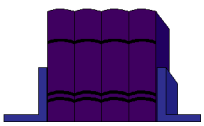


Petri-Netze

Anschauliche Modellierungsmethode für nebenläufige Prozesse und ihre Synchronisation.

Beispiel:
Materialverwaltung

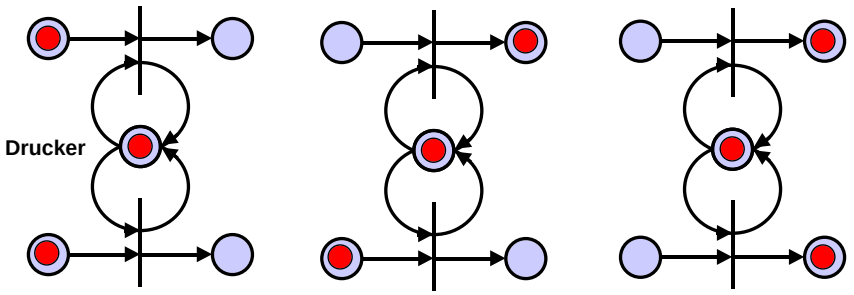




W. Reisig
Petri-Netze - Eine Einführung (2. Auflage)
Pringer, 1986

Benutzung eines Betriebsmittels durch zwei Prozesse

Zwei Prozesse wollen einen Drucker benutzen ...



beide Transitionen feuerbereit

nur eine Transition kann feuern

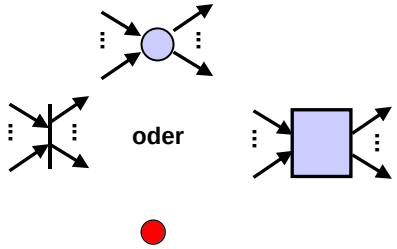
die zweite Transition kann danach feuern

75

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**

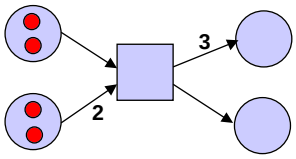
76

Kapazität und Gewichtung

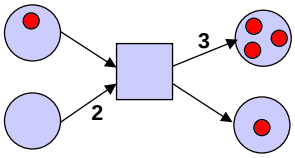
Die *Kapazität* einer Stelle ist die Zahl der maximal aufnehmbaren Marken dieser Stelle. Ohne Angabe ist die Kapazität ∞.

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

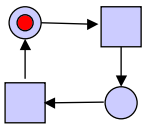


nachher

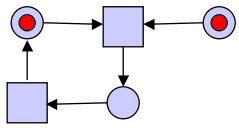
77

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*.

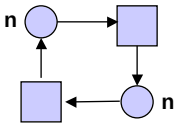


lebendig

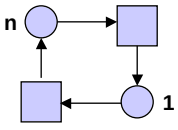


todesgefährdet

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



sicher

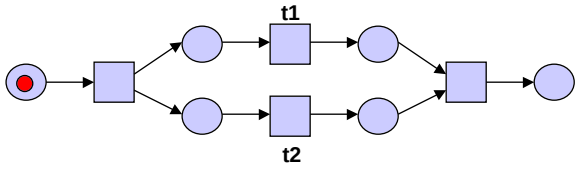


unsicher

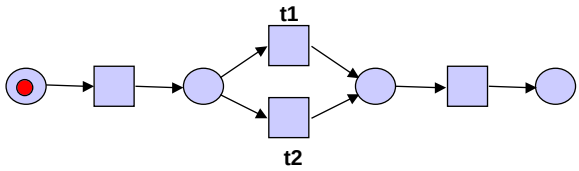
78

Netzmuster (1)

Nichtdeterministische Reihenfolge von t1 und t2



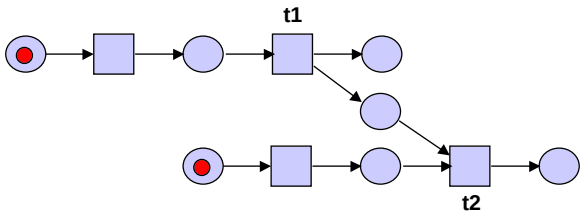
Nichtdeterministische Auswahl von t1 und t2



79

Netzmuster (2)

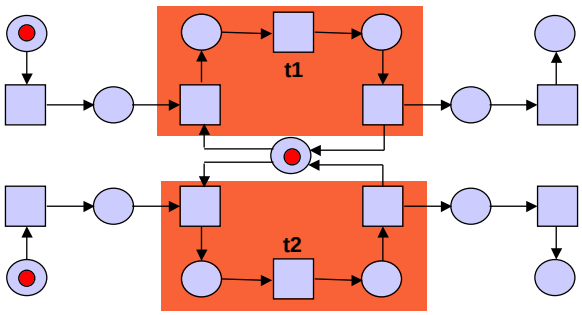
Einseitige Synchronisierung t1→t2



80

Netzmuster (3)

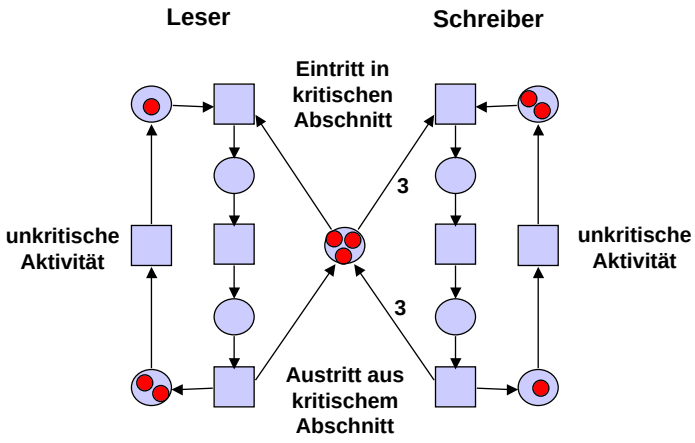
Gegenseitiger Ausschluß t1 ↔ t2



kritische Abschnitte

81

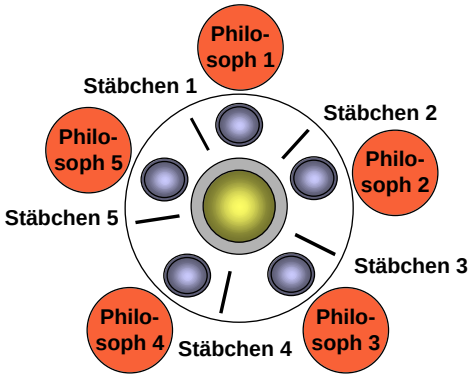
Leser und Schreiber



Markenbelegung für 3 Schreiber und 3 Leser

82

Die fünf speisenden Philosophen

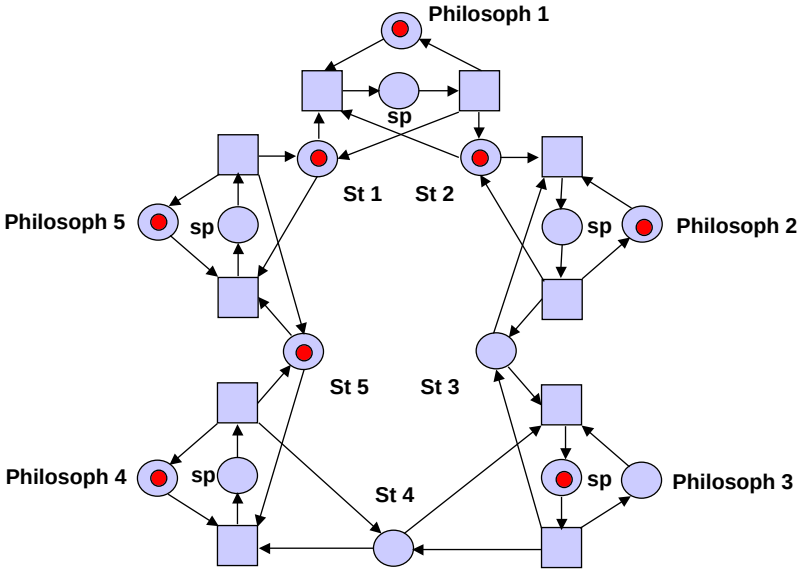


- Jeder Philosoph will entweder denken oder aus der großen Schale essen.
- Zum Essen braucht er zwei Stäbchen, sein eigenes (rechts) und das seines linken Nachbarn.

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

83

Petri-Netz für 5 speisende Philosophen

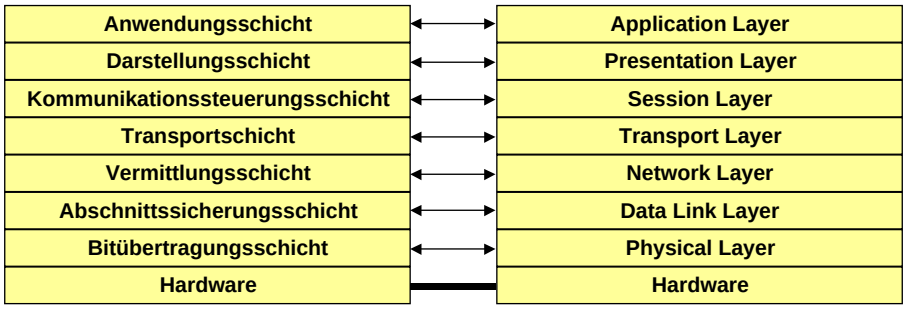


84

Schichten des ISO-Referenzmodells

Die Internationale Organization for Standardization (ISO) hat 1985 ein Referenzmodell für "Offene Systemkommunikation" definiert ("offen" = offengelegt zur Zusammenarbeit mit anderen Systemen).

Prozeßkommunikation für Anwendungen bedient sich in der Regel standardisierter Dienste von niedrigeren Schichten.



85

Erläuterungen zum ISO-Referenzmodell

Bitübertragungsschicht beschreibt physikalische Übertragung von Signalen über konkretes Kommunikationsmedium (Material der Leitung, Spannungen, Stecker etc.)

Abschnittssicherungsschicht führt Fehlerkontrolle und Fehlerkorrektur an Datenabschnitten durch (z.B. durch Paritätsbits).

Vermittlungsschicht stellt Verbindung von Sender zu Empfängern her. Sorgt für Weiterleitung über einzelne Stationen. Vermeidet Überlastung von Netzteilen.

Transportschicht verwendet Kommunikationsdienste der unteren Schichten wie End-zu-End-Verbindung. Sorgt für kostenoptimale Nutzung einer Verbindung.

Kommunikationssteuerungsschicht synchronisiert Zusammenarbeit zweier Partner. Strukturierung der Kommunikation durch Wiederaufsetzpunkte für Störungen.

Darstellungsschicht sorgt für Umkodierungen, falls Kommunikationspartner verschiedene Datencodierungen verwenden.

Anwendungsschicht führt alle verbleibenden Aufgaben einer Kommunikation durch (Aufgaben des Betriebssystems und der Anwendungsprogramme jedes Partners).

86

Echtzeitbetrieb

DIN 44300:

"Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind derart, daß die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zufälligen zeitlichen Verteilung oder zu vorbestimmten Zeitpunkten auftreten."

Zu den Korrektheitsbedingungen von (nebenläufigen) Programmen kommen zusätzliche Zeitbedingungen.

Harte Zeitbedingung: Nichterfüllung verursacht katastrophalen Schaden

Weiche Zeitbedingung: Verletzung verursacht zunehmend Kosten

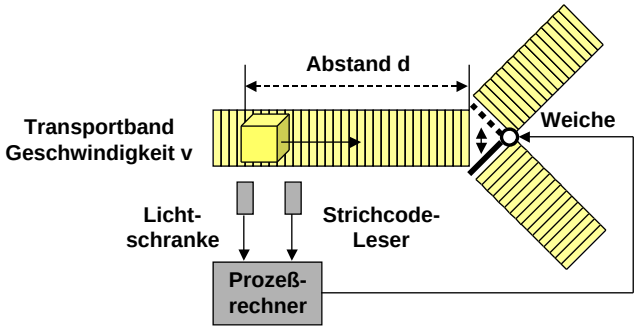
Korrektheit bezüglich Zeitbedingungen kann nur bei einer Spezifikation der zu betrachtenden Situationen untersucht und ggf. garantiert werden:

Fallen mehr als N Daten pro Zeiteinheit an?

Ist Rechnerausfall, Stromausfall, ... möglich?

87

Szenario für Echtzeitverarbeitung: Paketsortieranlage



- **Jedes Paket hat Strichcode entsprechend Bestimmungsort**
- **Bei Eintreffen eines Pakets schickt Lichtschranke SIgnal an Rechner**
- **Bei Signal startet Rechner Strichcodeleser und bestimmt Weichenstellung**
- **Rechner sendet Stellsignal an Weiche**

Weichenstellung muß innerhalb Zeitdauer d/v berechnet werden!

88

Kommunikation mit technischen Prozessen

Wie kommunizieren Rechnerprozesse (RP) mit allgemeinen technischen Prozessen (TP)?

Beispiele:

Transportvorgänge	Meßgeräte
chemische Prozesse	Ein/Ausgabegeräte
Abfüllvorgänge	Zeituhr
Produktionsprozesse	

Moderne technische Prozesse besitzen häufig Kommunikationshardware zur Realisierung der unteren Schichten des ISO-Referenzmodells.

Kommunikation auf Hardware-Ebene:

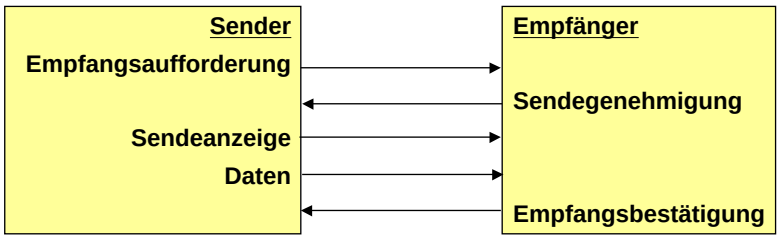
TP → RP:
- Unterbrechungssteuerung (Interrupt Control)
- Aktives Warten auf Signal (Busy Wait)
- Erfassen eines Meßwertes über A/D-Wandler

RP → TP:
- Steuersignale
- Setzen von Parametern über D/A-Wandler

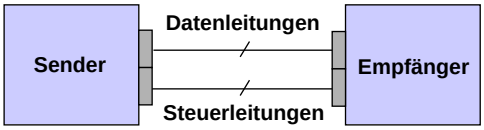
89

"Handshake"-Verfahren

Kommunikationsprotokoll auf der Transportebene



Beispiel einer physikalischen Realisierung:



90

Ein-/Ausgabe von Zeichen

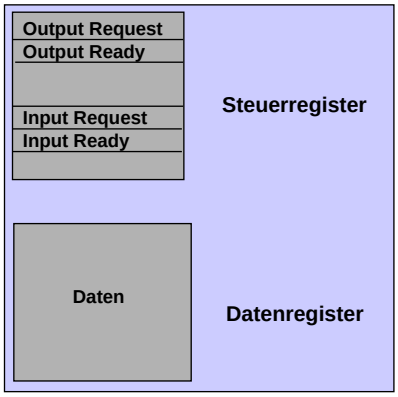
Beispiel für die Kommunikation zwischen einem Rechenprozeß (RP) und einem technischen Prozeß (TP)

Rechenprozeß

Output
Wiederhole:
- **Schreibe Daten**
- **Setze Output Request**
- **Warte bis Output Ready**

Input
Wiederhole:
- **Setze Input Request**
- **Warte bis Input Ready**
- **Lies Daten**

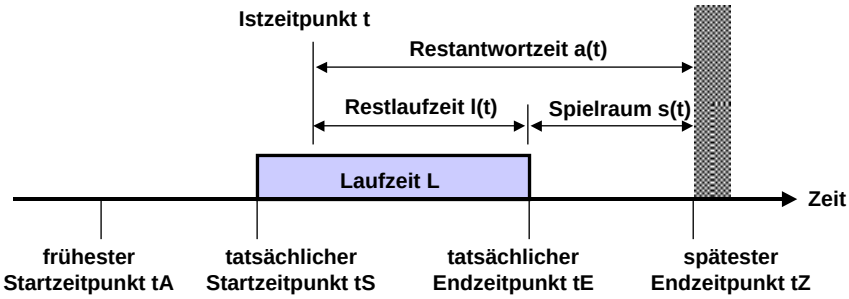
Technischer Prozeß



91

Zeitschranken von Prozessen

Wir abstrahieren von den technischen Randbedingungen eines Echtzeit-Rechnerprozesses durch Vorgabe von Zeitschranken.



Zeitgerechte Planung = Ablaufplanung, so daß alle Prozesse vor ihrem spätesten Endzeitpunkt beendet sind

Optimaler Planungsalgorithmus findet stets zeitgerechte Planung, falls eine existiert

92

Planungsverfahren für Prozesse eines Echtzeitsystems

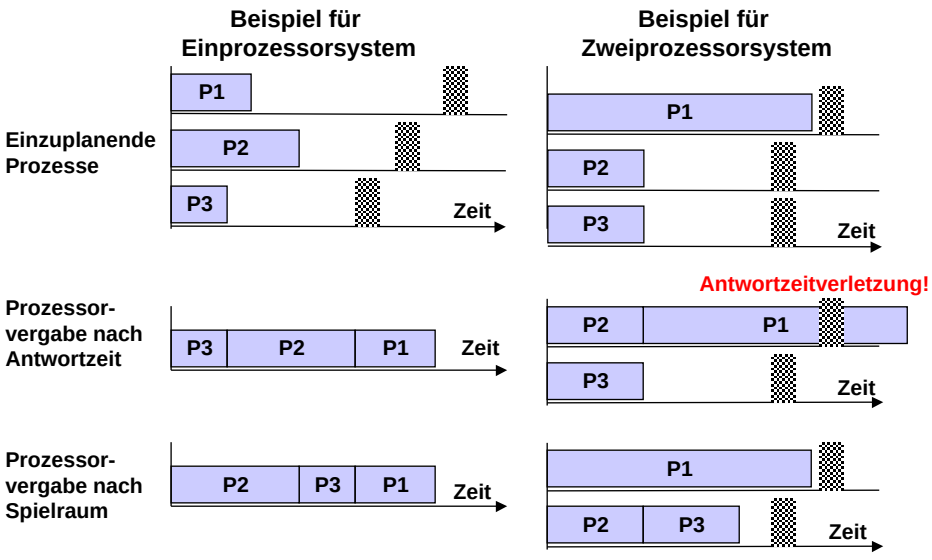
1. **Einplanung nach Vorab-Prioritäten**
   Umsetzung von Zeitschranken in feste Prioritäten ist nicht immer möglich und führt häufig zu Leistungsverlust des Systems
2. **Einplanung nach Antwortzeiten**
   Prozeß mit kürzester Antwortzeit a(t) erhält Prozessor
3. **Einplanung nach Spielraum**
   Prozeß mit geringstem Spielraum s(t) erhält Prozessor

Verfahren 2 und 3 sind für Einprozessorsysteme nachweislich optimal, für Mehrprozessorsysteme nur bei gleichen frühesten Startzeiten aller Prozesse.

Sind zusätzliche Bedingungen zu berücksichtigen (Reihenfolge, Ressourcen, Synchronisation zwischen Prozessen etc.), können Lösungen durch Constraint-Systeme gefunden werden (vertiefte Behandlung in höheren Semestern).

93

Planungsbeispiele



94