

3.4 Zustände in nebenläufigen Systemen

Kap. 1.+2.:	- keine Zustände - refentielle Transparenz	⇒	- einfaches Substitutionsmodell - Ausdrücke sind „zeitlos“
Kap. 3.:	- lokale Zustände - veränderbare Datenstrukturen	⇒	- Umgebungsmodell - Ausdruckswert <u>zeitabhängig</u>

Beispiel

Bankkonto: >(abheben 50)

 80

 >(abheben 50)

 30

- ⇒ Ausdruckswert abhängig von: - Form des Ausdrucks
 - Vorgeschichte, d.h. alle vorherigen Auswertungen

Die Welt ist nebenläufig:

Unabhängige Objekte, die alleine agieren bzw. sich synchronisieren. Programme sollen reale Welt möglichst einfach abstrahieren!

- ⇒ Anwendungsprogramme möglichst nebenläufig agieren (auch wenn diese sequentiell abgearbeitet werden !)

Nebenläufige Anwendung immer wichtiger:

- Internet
- Verteilte Informationssysteme
- Simulationen

(Programmiertechniken: herkömmliche Sprachen + prozessorientierte Konstrukte / Bibliotheken)

Nebenläufigkeit: neue Probleme bei Zuständen wegen besonderer Relevanz der Zeit

Beispiel: *Abheben:*

(define (abheben betrag)

 (if (>= kontostand betrag)

 (begin (set! kontostand (- kontostand betrag))

 Kontostand))

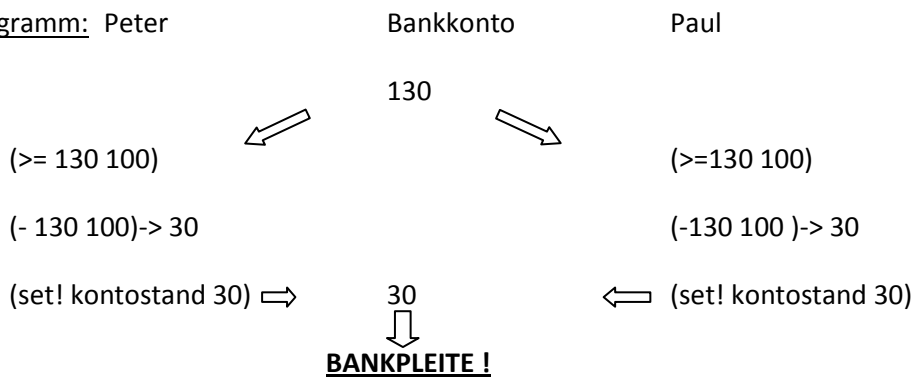
Annahme:

- Peter und Paul haben ein gemeinsames Konto
- Kontostand 130
- Peter und Paul heben an unterschiedlichen Automaten 100 ab

Möglichkeiten bei „gleichzeitiger“ Abhebung:

1. Beide dürfen abheben ($\geq \dots$) wahr
Peter hebt 100 ab: kontostand: 30
Paul hebt 100 ab: kontostand: -70 !
2. Beide dürfen abheben:
Peter hebt 100 ab: kontostand 30
Paul hebt 100 ab: kontostand 30!

Zeitdiagramm: Peter



Notwendig: Kontrolle kritischer Programmbereiche

Semaphore: Signale(z.B. Werte 0,1) mit Operationen P und V („passeren“ /“vrijgeven“) (Signale bei Eisenbahnen)

(P s) Falls $s = 1$, setze $s = 0$

Falls $s = 0$, stoppe Ausführung, warte bis $s = 1$

(V s) setze $s = 1$ (falls Prozess aufs warten, aktiviere einen)

Wichtig: P und V: unteilbare Operationen (Betriebssystem garantiert, dass nur ein Prozess P oder V ausgeführt wird)

Verhinderung der Bankpleite:

1. Erzeuge Semaphore S
2. (define (konstr-abheben betrag)

(begin (P s)

(abheben betrag) <- kritischer bereich

(V s)))

Nachteil: nur ein Prozess kann gleichzeitig abheben

Besser: für jedes konto ein eigener Semaphor:

```
(define (konstr-konto kontostand)
```

```
  (local ((define s (konstr-semaphor))
```

```
          (define (abheben betrag)
```

```
            (P s)
```

```
            (V s))
```

⇒ neues Problem: Verklemmung (Deadlock)

Beispiel: Überweisung zwischen Konten k1 und k2:

Zustände von k1 + k2 verändert => Operation kontrollieren

Annahme: Si Semaphor für Konto ki (i = 1, 2)

```
(define (ueberweisen k1 k2 s1 s2)
```

```
  (begin (P s1)
```

```
         (P s2)
```

```
         <Überweisung tätigen>
```

```
         (V s2)
```

```
         (V s1)))
```

Mögliche Situation: Peter und Paul haben verschiedene Konten, wollen „gleichzeitig“ zum anderen überweisen

Zeit: (ueberweisen peter-k paul-k peter-s paul-s) (ueberweisen paul-k peter-k paul-s peter-s)

(P peter-s)

(P paul-s)

(P paul-s) -> warten

(P peter-s) -> warten!

Beide warten, keiner kann fortfahren => Verklemmung

Vermeidung: sorgfältige Planung -> „Verklemmungsvermeidungsstrategien“

Trotzdem schwer zu überschauende Zeitabhängigkeiten

Fazit: Zustandsänderung =>Zeitaspekte relevant

=> Schwierige Kontrolle in realen Systemen

Konsequenz für Programmentwurf:

- zustandsfrei (referentiell transparent) wo immer es sinnvoll möglich
- Zustände lokal halten, alle Änderungen kontrollieren (-> Nachrichtenweitergabe)
- Änderungsoperationen synchronisieren, mögliche Verklemmungen ausschließen

3.5 Zusammenfassung

- kein universelles Programmierparadigma (für alle Probleme gut geeignet)
- verschiedene Paradigmen:
 - funktional: Datenabhängigkeiten funktional
 - relational (Beschränkungen): Datenabhängigkeiten können bei verschiedenen Berechnungen variieren
 - imperativ / zustandsorientiert: Geschichte der Berechnungen relevant
 - nebenläufig: unabhängige Akteure
- es gib (noch) keine universelle (für jedes Problem gleich gut geeignete) Sprache
- mächtige Sprachen mit guten Abstaktionsmöglichkeiten (z.B. Scheme) anpassbar an verschiedene Programmier-Stile (vgl. Bankkonten, Beschränkungsnetze) oder verschiedene Problembereiche (vgl. Schaltkreissimulation)
- wähle je nach Problem die passende Modellierung und Sprache

Tendenz:

- Für große, interaktive Systeme: zustandsorientiert, objektorientert, nebenläufig
- Für algorithmische Aspekte (Korrektheit!): funktional, Datenorientiertheit
- Für Datenbanken: relational