Algorithmen und Programmierung IV: Nichtsequentielle Programmierung

Robert Tolksdorf
Basiert auf ALP IV, SS 2003
Klaus-Peter Löhr
Freie Universität Berlin

Überblick

Überblick

- Strukturtreue Implementierung
- Mehrprozessorbetrieb
- Mehrprozeßbetrieb

Implementierung von Prozessen und Synchronisationskonstrukten (zur Auffrischung nachlesen: 2.3)

Strukturtreue Implementierung

Prozeß ←?→ Prozessor

Daten ←?→ Speicher

- Def.:
  Strukturtreue Implementierung:

1. *Mehrprozessorsystem (multiprocessor)*
   mit privaten und gemeinsamen Speichern für private und gemeinsame Variable
2. *Parallele*,
   d.h. echt simultane, Ausführung aller Prozesse

Beispiele für *nicht* strukturtreue Implementierung

- Mehrprozessorsystem *ohne private* Speicher

→ alle Daten im zentralen Arbeitsspeicher

- Weniger Prozessoren als Prozesse, z.B. Einprozessorsystem

→ quasi-simultane, verzahnte Ausführung im Mehrprozessbetrieb *(multitasking, multiprogramming)*

- Mehrrechnersystem *ohne gemeinsamen* Speicher *(multicomputer)*

→ nachrichtenbasierte Prozessinteraktion

5.1 Mehrprozessorbetrieb

- ... zunächst betrachtet unter der Voraussetzung:
  - jeder Prozess läuft auf einem eigenen Prozessor;
  - gemeinsame Daten und Synchronisationsobjekte liegen in gemeinsamem Speicher.

5.1.1 Sperrsynchronisation

- Fragestellung:
  Wie kann die Sperroperation lock (3.1.5) implementiert werden?

```
MONITOR Lock {// setzt bereits Sperrmechanismus voraus !
  private boolean lock = false;
  public void lock() when !lock { lock = true; }
  public void unlock() { lock = false; }
}
```

Sperrsynchronisation

- *Idee:*
  - Wiederholt lock prüfen – solange lock gesetzt ist;
  - sobald lock nicht gesetzt ist, selbst setzen.
- *Aber:*
  naive Umsetzung dieser Idee führt nicht zum Erfolg.
- Hardware bietet Unterstützung: unteilbare Instruktionen
- *Beispiel:*
  Instruktion TAS *(test-and-set)* für unteilbares Lesen/Schreiben im Arbeitsspeicher

Sperrsynchronisation

- Damit

```
class Lock {
  private boolean lock = false;
  public void lock() { while(TAS(lock)) ; }
  public void unlock() { lock = false; }
}
```

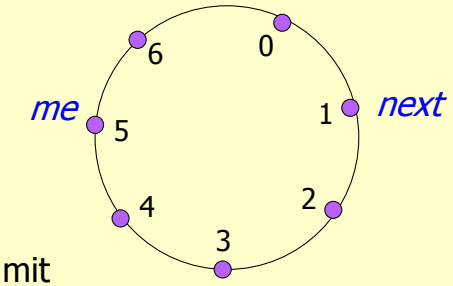
- Andere unteilbare Operationen ermöglichen ähnliches – z.B.
  - SWAP
  - INC
  - …

Sperrsynchronisation

- Terminologie:
  - Ein so benutzte Sperrvariable heißt *spin lock*.
  - Die hier praktizierte Art des Wartens heißt aktives Warten *(busy waiting).*
- Aktives Warten ist normalerweise *verpönt* – weil es Ressourcen verbraucht (Prozessor *und* Speicherzugriffe!).
- *Spin locks* werden daher nur auf unterster Ebene für extrem kurzes Sperren eingesetzt, z.B. für die Implementierung von Semaphoren oder Monitoren – nicht explizit im Anwendungs-Code.

5.1.2 Sperren ohne spezielle Instruktionen

- Nur mit atomarem Lesen bzw. Schreiben natürlicher Zahlen (Dekker, Dijkstra, Habermann, Peterson, ...)
- n Prozesse
- Gemeinsame Variable:
  - *next* = 0,1,...,n-1, anfangs 0
  - next ist der Prozess, der als nächster den Lock hat
  - Solang ein Prozess nicht der mit der Nummer next ist, muss er warten

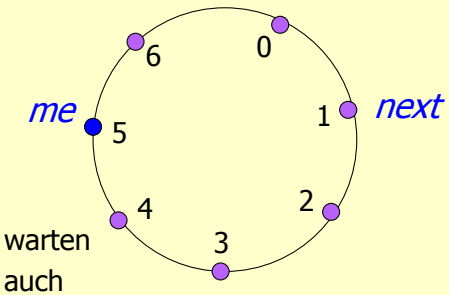


- Private Variable:
  - *me* = Prozeßnummer
  - Zustand = *idle, eager, critical,* anfangs *idle*
  - Zustandsübergänge immer idle > eager > critical > idle

Sperren ohne spezielle Instruktionen

lock:
repeat
  set *eager* // will die Sperre
  // warte solange zwischen next und mir kein Prozess
  // die Sperre hat oder haben will
  repeat set *p* to *next*,
    while *p* is *idle* **do** increment *p*
  until *p* equals *me*
  set *critical*
until nobody else is *critical*;
set *next* to *me*.

- wenn 2-4 idle sind kann 5 versuchen die Sperre zu bekommen
- Wenn danach 6 das versucht, wird 6 warten
- Wenn danach 4 das versucht, wird 4 auch versuchen die Sperre zu bekommen

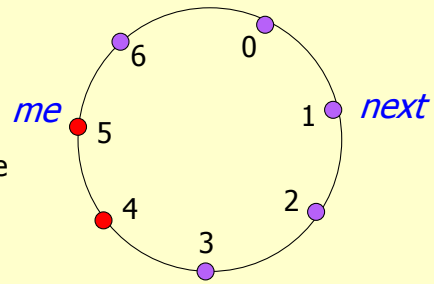


Sperren ohne spezielle Instruktionen

lock:
repeat
  set *eager* // will die Sperre
  // warte solange zwischen next und mir kein Prozess
  // die Sperre hat oder haben will
  repeat set *p* to *next*,
    while *p* is *idle* **do** increment *p*
  until *p* equals *me*
  set *critical*
until nobody else is *critical*;
set *next* to *me*.

nobody else ...:
set *p* to *me* ⊕ 1,
  while *p* is not *critical* **do** set *p* to *p* ⊕ 1,
p=me?

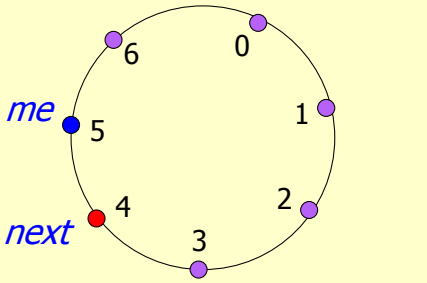
- Wenn 5 keinen anderen gesehen hatte wird er critical
- 4 wird auch critical
- 4 und 5 stellen fest, dass ein anderer Prozess auch critical ist



Sperren ohne spezielle Instruktionen

lock:
repeat
  set *eager* // will die Sperre
  // warte solange zwischen next und mir kein Prozess
  // die Sperre hat oder haben will
  repeat set *p* to *next*,
    while *p* is *idle* **do** increment *p*
  until *p* equals *me*
  set *critical*
until nobody else is *critical*;
set *next* to *me*.

- 4 und 5 werden wieder eager
- 5 sieht nun 4 und wartet
- 4 wird critical
- next wird 4 und hat die Sperre
- nur 4 hat die Sperre

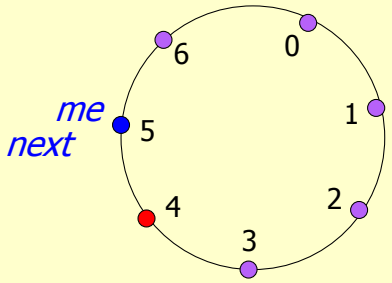


Sperren ohne spezielle Instruktionen

unlock:
set *next* to *me* ⊕ 1,
set *idle*

- 4 gibt Sperre auf
- next wird zu 5
- 5 hat Sperre

- Zusammengefasst
  - Prozesse sind geordnet
  - versuchen Sperre zu bekommen, wenn das keiner „vor" ihnen tut
  - Falls das nebenläufig passiert entsteht Konflikt
  - Dieser wird aufgehoben durch Ordnung beim nochmaligen Versuch



5.1.3 Ereignissynchronisation

```
MONITOR M {
public R op(A a) WHEN C { S }
…
}
```

```
{
   do {
        monitor.lock();
        if(!C) {monitor.unlock(); continue;}
        else break;
   } while(true);
   S
   monitor.unlock();
}
```

mit Sperrvariable monitor

Ereignissynchronisation

- *Problematisch:*
  geschachteltes aktives Warten, sogar für längere Zeiträume

- Spezialfall:
  S leer und C unteilbar,
  - z.B. „Warten bis c != 0"
  - **while**(c == 0);
  - (Java: c sollte **volatile** sein)

5.2 Mehrprozeßbetrieb (multitasking, multiprogramming)

- *Voraussetzung:*
  (zunächst) nur 1 Prozessor

- *Idee:*
  Prozessor ist *abwechselnd* mit der Ausführung der beteiligten Prozesse befasst
- („quasi-simultane" Ausführung, *processor multiplexing*)

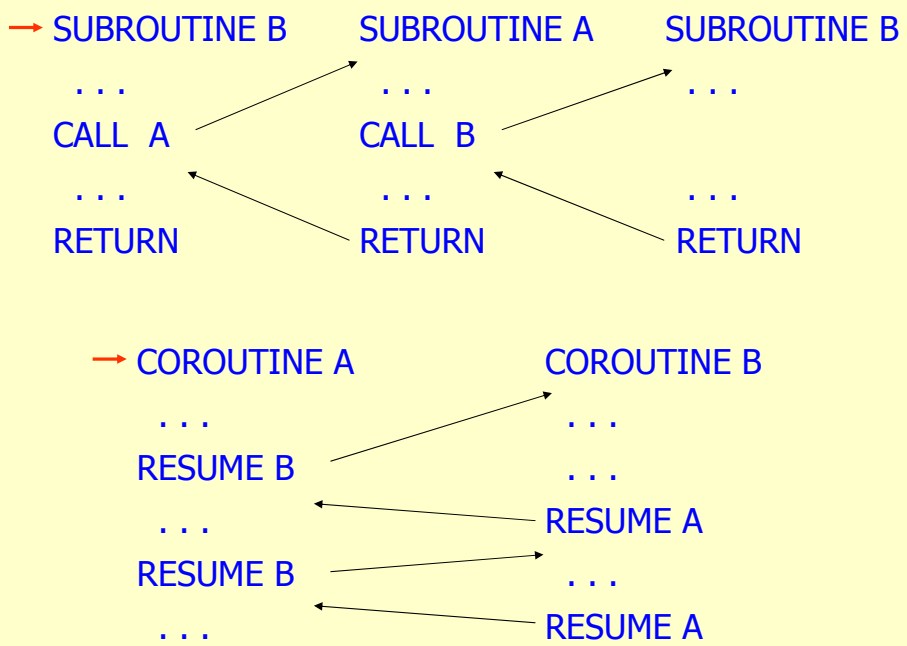
5.2.1 Koroutinen

- Subroutinen, Prozeduren, Operationen, Methoden, ... :

- Interaktion mit **Aufruf/Rücksprung** *(call/return)*, d.h. „Übergang" zwischen Prozeduren bedeutet
  - entweder **Aufruf**:
    - A-Inkarnation erzeugt B-Inkarnation,
    - übergibt Parameter und
    - springt zum Startpunkt von B

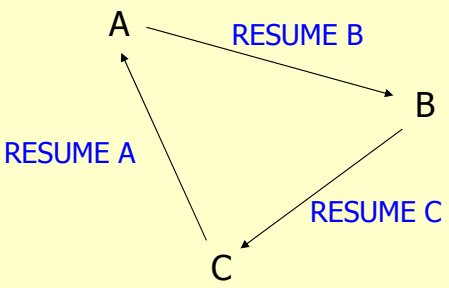
  - oder **Rücksprung**:
    - Löschen der B-Inkarnation,
    - Rückgabe von Parametern,
    - Rücksprung zur Aufrufstelle in A

Koroutinen



Koroutinen

- Beliebig viele Koroutinen können beteiligt sein,
- z.B. 3 :



Koroutine (coroutine)

- Interaktion mit *exchange jump* RESUME statt *call/return:*
- A: RESUME B bewirkt
- Sprung von A-Inkarnation nach B-Inkarnation an denjenigen Punkt, an dem die B-Inkarnation das letzte Mal (mit RESUME) verlassen wurde
- Jede Inkarnation „kennt" ihren eigenen letzten RESUME-Punkt.

Koroutinen

- **Erzeugen/Löschen** von Koroutinen-Inkarnationen:
- verschiedene Varianten möglich, z.B.
  - erstes RESUME A erzeugt (einzige!) A-Inkarnation;
  - Ende von A wirkt wie RESUME X, wobei X diejenige Koroutine ist, die A letztmalig am Startpunkt aktiviert hat;
  - erneutes RESUME A aktiviert A wiederum am Startpunkt.

Koroutinen

- *Programmiersprachen* mit Koroutinen :
  - Simula (1967, Dahl/Myhrhaug/Nygaard)
  - Modula (1980, Wirth)
  - u.a.
- *Anwendungen*:
  - Übersetzerbau
  - Simulation diskreter Ereignisse
  - → Mehrprozessbetrieb / Threading

Koroutinen in Modula-2

- Modula-2
  - Imperative Programmiersprache
  - Pascal plus Module (Schnittstelle und Implementierung)
- Modula-2 hat Standardbibliotheken, die Module
- Im Modul SYSTEM sind Koroutinen vorhanden
- Basierend auf den Koroutinen können in Modula-2 Ablaufsteuerung und Synchronisationsmechanismen implementiert werden
- siehe auch www.modula2.org
  - http://www.modula2.org/reference/isomodules/isomodule.php?file=PROCESSE.DEF
  - http://www.modula2.org/reference/isomodules/isomodule.php?file=SEMAPHOR.DEF

Koroutinen in Modula-2

- Erzeugen einer neuen Koroutine
  - PROCEDURE NEWPROCESS(proc: PROC; addr: ADDRESS; size: CARDINAL; VAR new: PROCESS);
    - Modula-2 Typen:
      - PROC: eine Prozedur (z.B. über Bezeichner referenziert)
      - ADDRESS: Adresstyp (Speicheradresse)
      - CARDINAL: Positive Ganzzahl
      - PROCESS: Prozess (eventuell kompatibel zu ADDRESS)
    - Parameterart VAR: Call-by-reference
    - Erzeugt neue Koroutine
    - Genauer: Richtet einen Bereich der Größe size im Speicher ab addr ein in dem die Prozedur proc lokale Variablen und Aufrufstrukturen ablegen kann
    - Liefert Bezeichnung der Koroutine als Adresse in new zurück

Koroutinen in Modula-2

- Kontrolltransfer zu einer Koroutine
  - PROCEDURE TRANSFER(VAR source, destination: PROCESS);
  - Vermerkt „Absprungkoroutine", also ausführenden Prozess in source
  - Übergibt Kontrolle an die durch destination bezeichnete Koroutine
  - destination muss vorher durch NEWPROCESS oder durch Verwendung als source Parameter in einem TRANSFER belegt worden sein

Beispiel 1

```
MODULE Beispiel1;
FROM SYSTEM IMPORT ADR;
FROM Process IMPORT PROCESS, NEWPROCESS, TRANSFER;
FROM InOut IMPORT WriteInt, WriteLn;

VAR Koroutine1, Koroutine2, Main : PROCESS;
      Speicher1, Speicher2 : ARRAY[0..1000] OF INTEGER;

PROCEDURE Prozedur1;
  [...]
END Prozedur1;

PROCEDURE Prozedur2;
  [...]
END Prozedur2;

BEGIN
  NEWPROCESS(Prozedur1,ADR(Speicher1),SIZE(Speicher1),Koroutine1);
  NEWPROCESS(Prozedur2,ADR(Speicher2),SIZE(Speicher2),Koroutine2);
  TRANSFER(Main,Koroutine1);
END Beispiel1.
```


Beispiel 1

```
PROCEDURE Prozedur1;
VAR i : INTEGER;
BEGIN
  i := -1;
  LOOP
    i := i + 1;
    IF i>100 THEN
      TRANSFER(Koroutine1,Main);
    END;
    WriteInt(i,5); WriteLn;
    TRANSFER(Koroutine1,Koroutine2);
  END;
END Prozedur1;

PROCEDURE Prozedur2;
VAR i : INTEGER;
BEGIN
  i := 1001;
  LOOP
    i := i - 1;
    WriteInt(i,5); WriteLn;
    TRANSFER(Koroutine2,Koroutine1);
  END;
END Prozedur2;
```

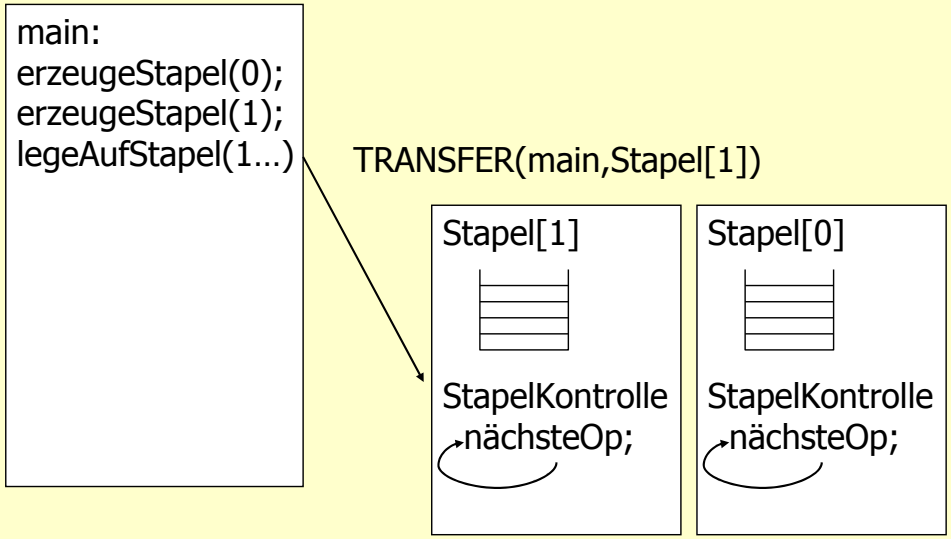

Beispiel 2

- Module ist nicht objektorientiert
- Objekt = Zustand + Verhalten
- Idee mit Koroutinen in Modula-2:
  - Für Zustand wird ein Speicherbereich eingerichtet
  - Es gibt einen Prozess, der Operationen darauf ausführt
  - Operationen suchen geeigneten Prozess aus
- Beispiel:
  - Stapelobjekte mit üblichen Operationen
  - Für jeden Stapel wird Prozess eingerichtet
  - Operationen führen einen TRANSFER zum gewünschten Stapelprozess aus

Beispiel 2

```
DEFINITION MODULE StapelCoroutinen;

EXPORT QUALIFIED erzeugeStapel,legeAufStapel,
   entnehmeVonStapel, leerStapel,vollStapel,tElement,
   StapelAnzahl,tStapelNr;

CONST StapelAnzahl = 5;

TYPE   tElement = CHAR; (* oder ein anderer Typ *)
       tStapelNr = [1..StapelAnzahl];

PROCEDURE erzeugeStapel(s:tStapelNr);
PROCEDURE legeAufStapel(s:tStapelNr;el:tElement);
PROCEDURE entnehmeVOnStapel(s:tStapelNr;VAR el :tElement);
PROCEDURE leerStapel(s:tStapelNr):BOOLEAN;
PROCEDURE vollStapel(s:tStapelNr):BOOLEAN;
END StapelCoroutinen.
```

Drei Prozesse



Beispiel 2

```
IMPLEMENTATION MODULE StapelCoroutinen;

FROM SYSTEM        IMPORT ADDRESS,NEWPROCESS,
                          TRANSFER,PROCESS;
FROM Storage       IMPORT ALLOCATE;
FROM InOut         IMPORT WriteString;

VAR
   main: PROCESS;
   Stapel: ARRAY[1..StapelAnzahl] OF PROCESS;

TYPE OpCode = ( pop,push,voll,leer );

VAR
   Operation:        OpCode;
   istLeer,istVoll:  BOOLEAN;
   ELEMENT:          tElement;
   StapelNr:         tStapelNr;
```


Beispiel 2

```
PROCEDURE erzeugeStapel(s:tStapelNr);
VAR Adr : ADDRESS;
BEGIN
   ALLOCATE(Adr,254);
   StapelNr := s;
   NEWPROCESS(StapelKontrolle,Adr,254,Stapel[s]);
   TRANSFER(main,Stapel[s])
END erzeugeStapel;
```


Beispiel 2

```
PROCEDURE StapelKontrolle;
CONST StapelLaenge = 10;
VAR id:[1..StapelAnzahl];
Stapelplatz : ARRAY[1..StapelLaenge] OF tElement;
top : [O..StapelLaenge];
BEGIN
   id := StapelNr;
   top := 0;
   TRANSFER(Stapel[id],main);
   LOOP
      …
```

Beispiel 2

```
PROCEDURE legeAufStapel(s:tStapelNr; el : tElement);
BEGIN
   Operation := push;
   ELEMENT := el;
   TRANSFER(main,Stapel[s])
END legeAufStapel;

PROCEDURE entnehmeVonStapel(s:tStapelNr;VAR el:tElement);
BEGIN
   Operation := pop;
   TRANSFER(main,Stapel[s]);
   el := ELEMENT
END entnehmeVonStapel;
```


Beispiel 2

```
PROCEDURE StapelKontrolle;
CONST  StapelLaenge = 10;
VAR    id:[1..StapelAnzahl];
       Stapelplatz : ARRAY[1..StapelLaenge] OF tElement;
       top : [O..StapelLaenge];
BEGIN
   id := StapelNr;
   top := 0;
   TRANSFER(Stapel[id],main);
   LOOP
       CASE Operation OF
         push:  IF top < StapelLaenge THEN
                        INC(top);
                        Stapelplatz[top] := ELEMENT
                END;
         pop:   IF top >0 THEN
                        ELEMENT := Stapelplatz[top];
                        DEC(top)
                END;
```


Beispiel 2

```
         voll:      istVoll := (top = StapelLaenge);
         leer:      istLeer := (top = 0)
       ELSE
         WriteString(' Unerlaubte Operation ')
       END;
       TRANSFER(Stapel[id],main)
   END;
END StapelKontrolle;

BEGIN
END StapelCoroutinen.
```


Beispiel 2

```
PROCEDURE leerStapel(s:tStapelNr):BOOLEAN;
BEGIN
   Operation := leer;
   TRANSFER(main,Stapel[s]);
   RETURN istLeer
END leerStapel;

PROCEDURE vollStapel(s:tStapelNr):BOOLEAN;
BEGIN
   Operation := voll;
   TRANSFER(main,Stapel[s]);
   RETURN istVoll
END vollStapel;
```

5.2.2 Prozesse/Threads als Koroutinen

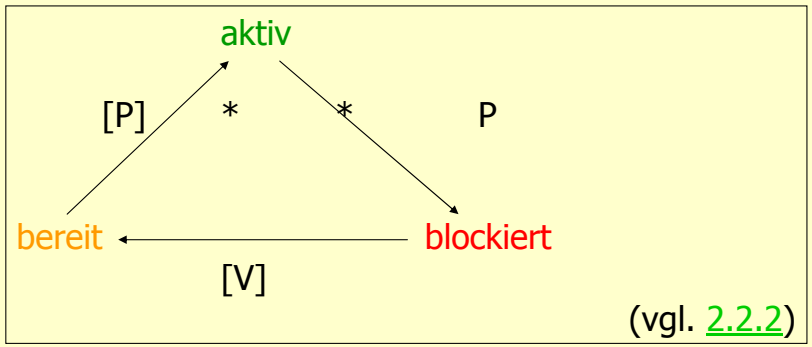
- Ansatz:
  - *Prozess* wird durch Koroutine simuliert;
  - *Blockieren* nicht als aktives Warten, sondern als *exchange jump* zu anderer Koroutine:
    „Prozesswechsel", „Prozessumschaltung"

Prozesse/Threads als Koroutinen

- 3 mögliche (Makro-)Zustände eines Prozesses:
  - **aktiv**:
    läuft, „ist im Besitz des Prozessors"
  - **blockiert**:
    (wartend) wartet auf Ereignis
  - **bereit**:
    ist weder aktiv noch blockiert,
    d.h. ist lauffähig, aber nicht im Besitz des Prozessors

Prozesse/Threads als Koroutinen

- Zustandsübergänge z.B. bei Verwendung von Semaphoren:



[ ]: Operation wird durch anderen Prozeß ausgeführt
*t: Entscheidung über Prozessorvergabe erforderlich

Prozesse/Threads als Koroutinen

```
class Semaphore { // assuming coroutine support
// active process
static private COROUTINE current;

// ready processes
static private Queue<COROUTINE> readyList
   = new LinkedQueue<COROUTINE>();
private int count;

// blocked processes
private Queue<COROUTINE> waitingList
   = new LinkedQueue<COROUTINE>();

public Semaphore(int init) {count = init;}
```

Prozesse/Threads als Koroutinen

```
public void P() {
   count--;
   if(count < 0) {
        waitingList.add(current);
        current = readyList.remove();
                        // leere Bereitliste ? Verklemmung !
        RESUME current;
   }
}

public void V() {
   count++;
   if(count <= 0)
        readyList.add(waitingList.remove());
                        // leere Warteliste ? Nicht möglich !
}
```


Bemerkungen

1. Prozesswechsel nur bei *Blockade* (d.h. jede Anweisungsfolge ohne Blockade ist *unteilbar* – insbesondere auch P und V !)
2. Daher *spin locks* weder erforderlich *noch korrekt (!)* (keine echte Parallelität)
3. Sowohl Prozessorvergabe als auch Semaphore sind *FIFO*
4. Bei Prioritätsteuerung eventuell Prozesswechsel auch in V (wenn höherrangiger Prozess geweckt wird)

Bemerkung/2

- Nichtblockierender Prozeß monopolisiert den Prozessor!
  (Weil Prozesswechsel nur bei *Blockade*)
- Abhilfe:
  - Operation **yield**(), die einen Prozesswechsel erlaubt
  - Automatisch *erzwungener* Prozesswechsel in regelmäßigen Abständen, mittels *Zeitgeber-Unterbrechung* *(time slicing → round-robin scheduling)*

5.3 Java Threading

- Zur Erinnerung: 2.3
  - Threads und Synchronisation werden
    - in verschiedenen Java-Systemen
    - verschieden realisiert !
  - Meist – nicht immer – werden
    - sowohl die Prioritäten berücksichtigt (4.1)
    - als auch *time slicing* praktiziert
- Wenn kein *time slicing*, dann Thread.yield() verwenden, um absichtlich den Prozessor an einen anderen bereiten Thread abzugeben (sofern vorhanden)

Zusammenfassung

Überblick

- Strukturtreue Implementierung
- Mehrprozessorbetrieb
  - Sperrsynchronisation
  - Sperren ohne spezielle Instruktionen
  - Ereignissynchronisation
- Mehrprozeßbetrieb
  - Koroutinen
  - Prozesse/Threads als Koroutinen
  - Java Threading