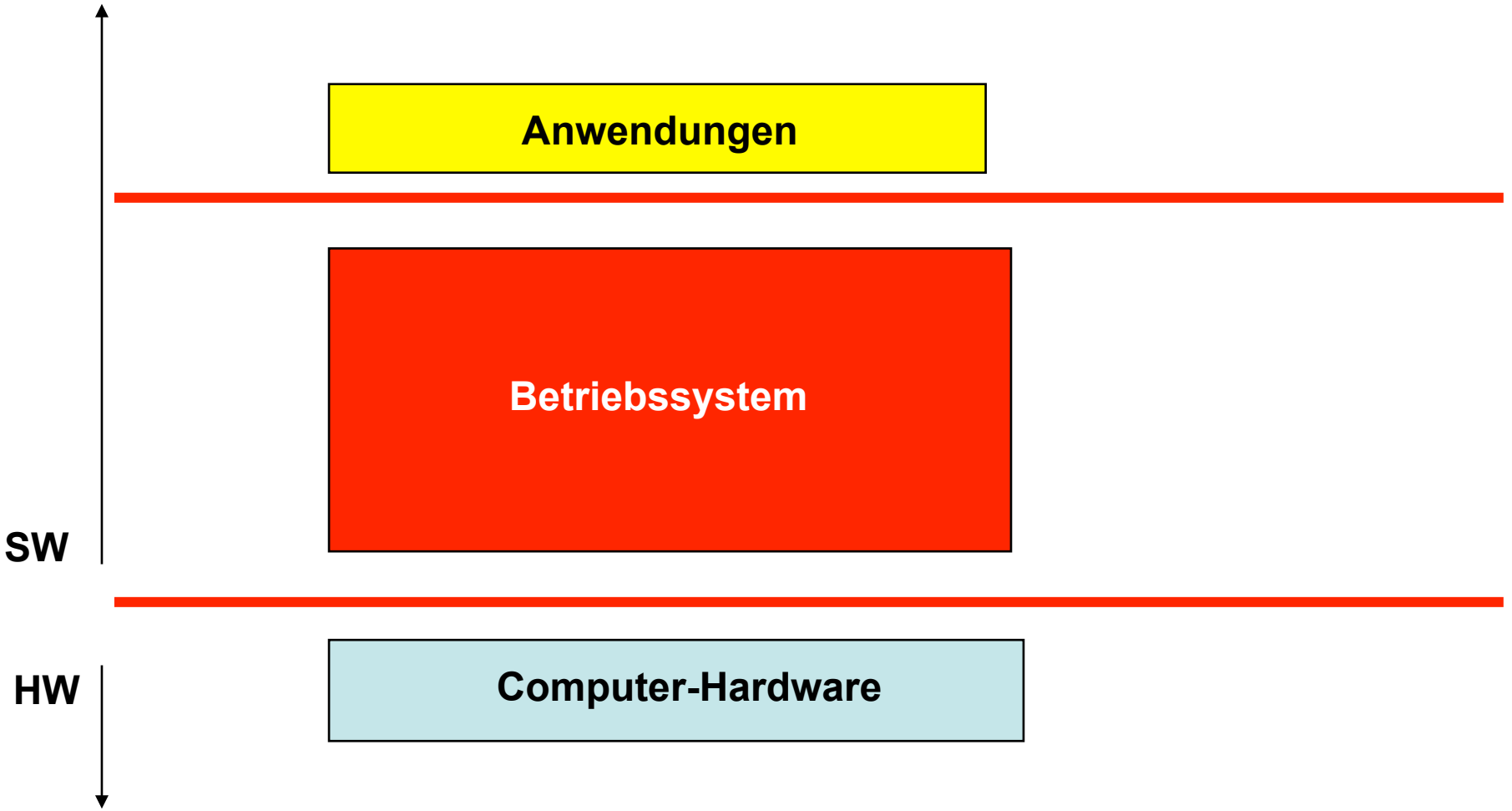
Strukturierung von Aktivitäten und Nebenläufigkeit

Betriebssysteme
WS 2010/2011

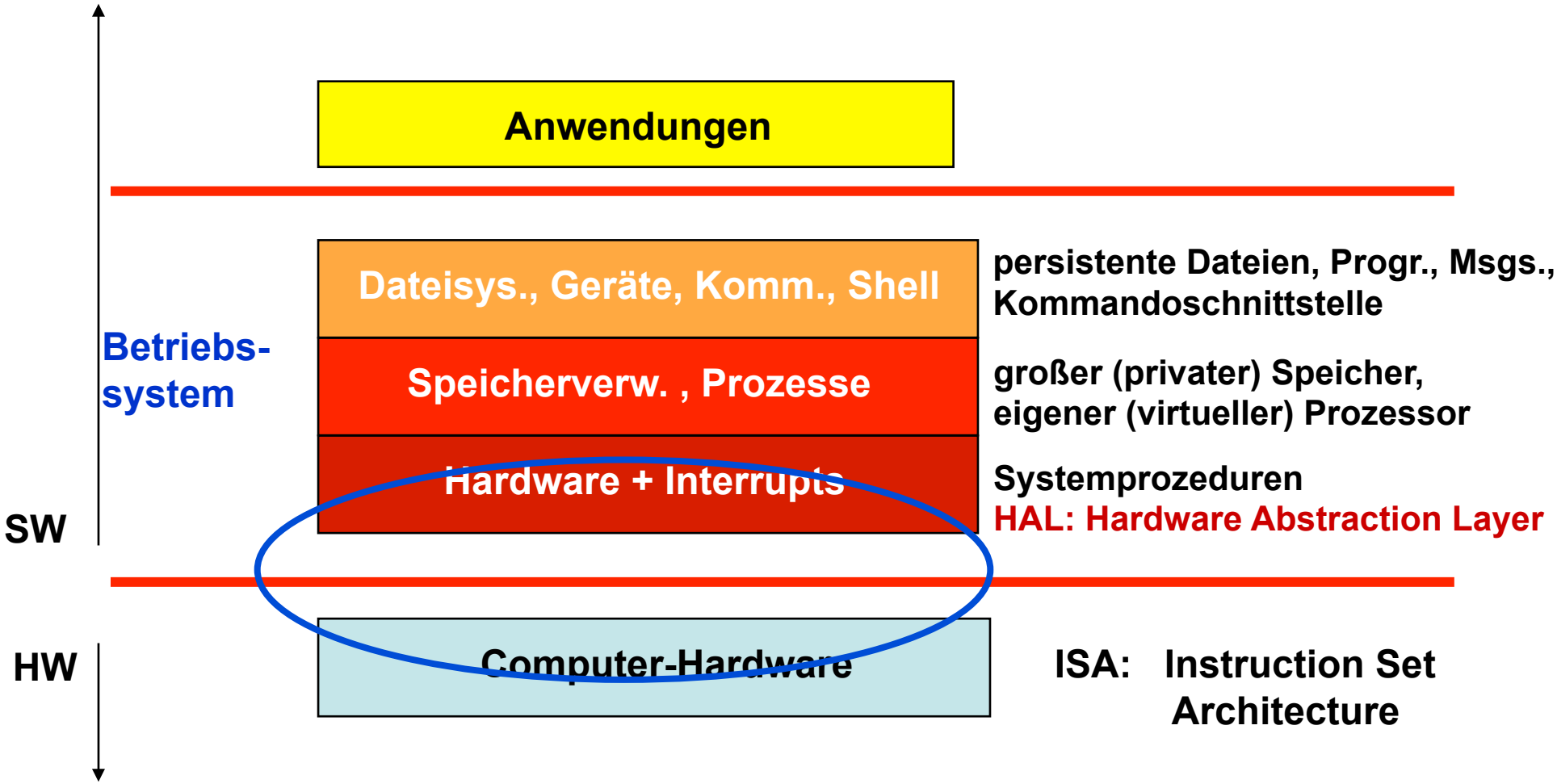
Jörg Kaiser
IVS – EOS

Otto-von-Guericke-Universität Magdeburg

Schichtenmodell



Schichtenmodell



Strukturierung von Aktivitäten und Nebenläufigkeit

Ein Programm "in Ausführung":

Benötigte Resourcen:

- Prozessor
- Speicher

Beschreibung eines Zustands während der Programmausführung:

- Befehlszähler
- Prozessorregister
- Speicherinhalt

Bekannte Programmierungkonzepte:

- Prozedur, Routine, Unterprogramm, Funktion, ..

Strukturierung von Aktivitäten und Nebenläufigkeit

Progammierkonzept: Prozedur, Routine, Funktion, ..

Bestandteile:

- Programmcode
- Daten
- Dyn. Ausführungsumgebung
  - Parameter
  - Ausführungsstatus

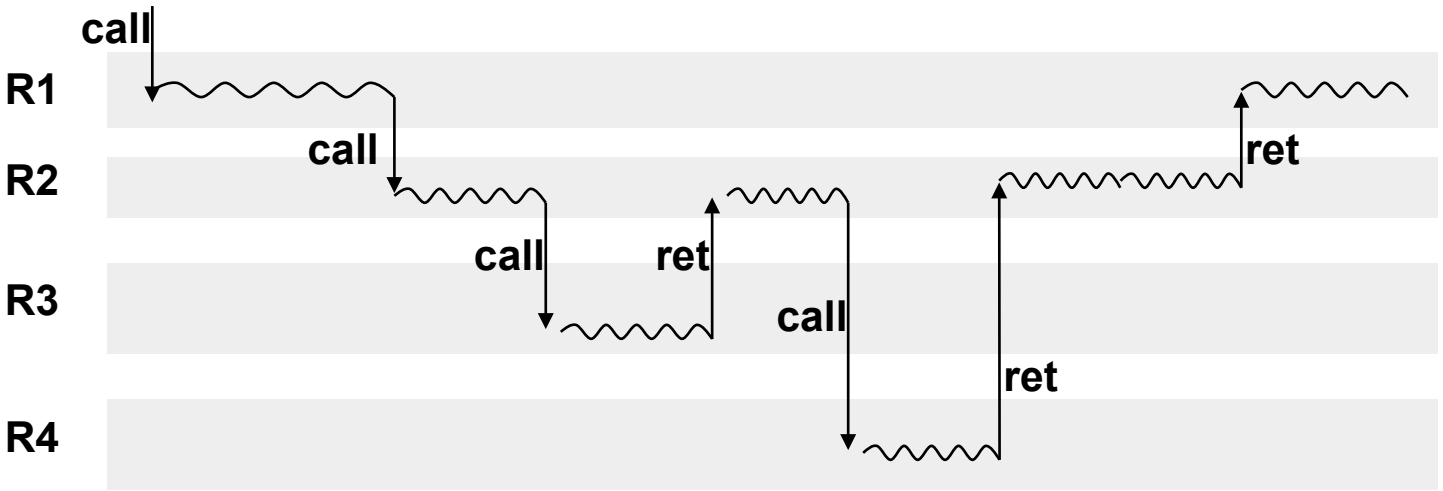
} wird auf dem Stack angelegt und verwaltet

Aufruf: Programmierter, expliziter Aufruf aus dem Programm, Rückkehr zur Aufrufstelle nach Abarbeitung.

Systemunterstützung: Spezielle Instruktionen, Compiler

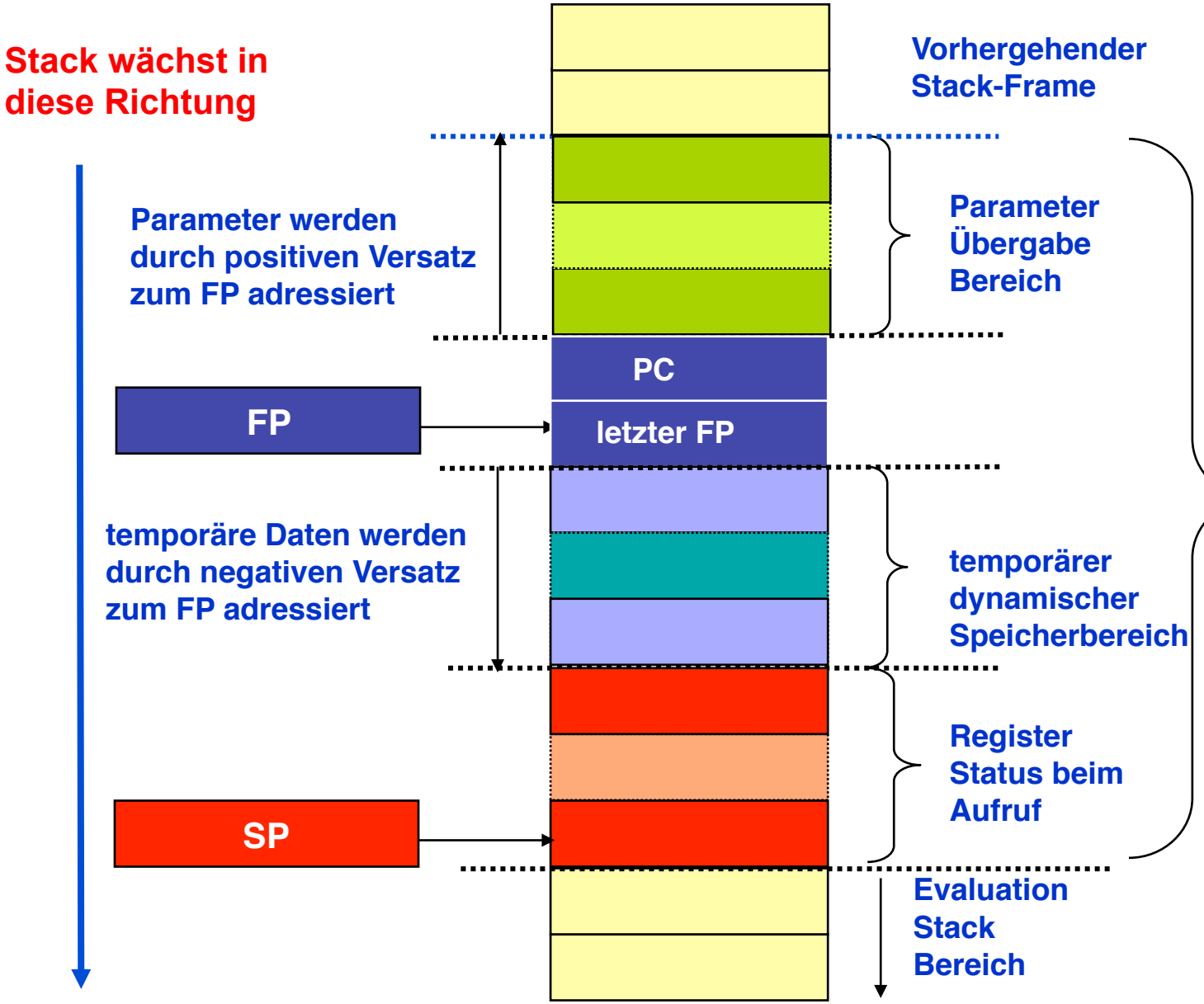
Strukturierung von Aktivitäten und Nebenläufigkeit

Aufrufhierarchie bei Routinen



feste, streng geschachtelte, hierarchische (asymmetrische) Aufruffolge.

Aktivierungs-Block



SP: Stack-Pointer
FP: Frame-Pointer

- activation record
- prozedure- context
- stack-frame

LINK

Link and Allocate
(M68000 Family)

LINK

Operation: SP – 4 → SP; An → (SP); SP → An; SP + d_n → SP

Assembler Syntax: LINK An, # < displacement >

Attributes: Size = (Word, Long*)

*MC68020, MC68030, MC68040 and CPU32 only.

Description: Pushes the contents of the specified address register onto the stack. Then loads the updated stack pointer into the address register. Finally, adds the displacement value to the stack pointer. For word-size operation, the displacement is the sign-extended word following the operation word. For long size operation, the displacement is the long word following the operation word. The address register occupies one long word on the stack. The user should specify a negative displacement in order to allocate stack area.

Condition Codes:

Not affected.

Instruction Format:

WORD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	0	REGISTER		
WORD DISPLACEMENT															

Instruction Format:

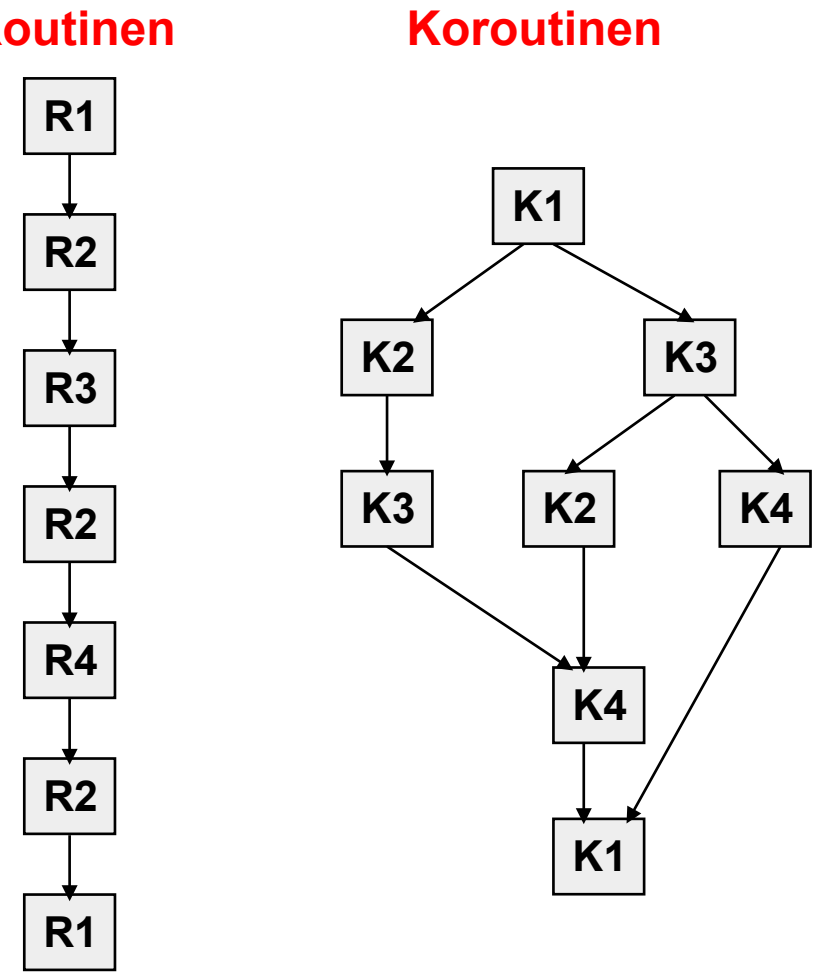
LONG

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	0	0	1	REGISTER		
HIGH-ORDER DISPLACEMENT															
LOW-ORDER DISPLACEMENT															

```
main:
 pushl %ebp
 movl %esp,%ebp
 pushl $0
 call exit
 addl $4,%esp
 movl  %ebp,%esp
 popl %ebp
 ret
```

Das Konzept der Ko-Routine

- Mittel zur Strukturierung und expliziten Kontrolle nebenläufiger Aktivitäten.
- KR repräsentieren gleichberechtigte, autonome Kontrollflüsse.
- Gleichberechtigte (symmetrische) Beziehungen.
- Anstelle eines "Return" bei Routinen wird beim Wechsel der Koroutinen jedesmal explizit angegeben, wohin die Kontrolle transferiert werden soll.
- Abfolge der Aufrufe kann sich ändern.



Das Konzept der Ko-Routine

Primitive zur Steuerung von Koroutinen:

create:	**Erzeugung, nicht Aktivierung, einer neuen Koroutine**
resume:	**Supendierung der laufenden Koroutine und Übertragen der Kontrolle auf eine andere Koroutine. Wiederaufnahme einer suspendierten Koroutine an der Stelle, an der die Kontrolle abgegeben wurde.**

Grundregeln:

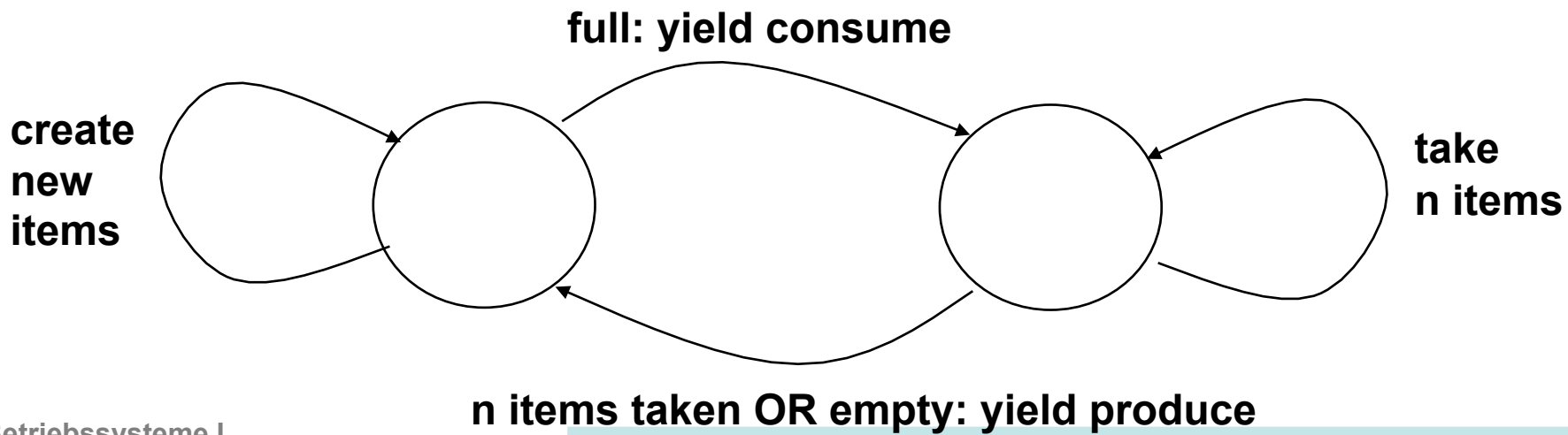
- **die Ausführung einer Koroutine muss dort fortgesetzt werden, wo die Koroutine suspendiert wurde.**
- **jede Koroutine muss zu Ende kommen.**

Anwendungsbeispiel für Co-Routinen

var q := new queue

```
coroutine produce
   loop
      while q is not full
         create some new items
         add the items to q
      yield to consume
```

```
coroutine consume
   loop
      while q is not empty
         remove some items from q
         use the items
      yield to produce
```



Eigenschaften der Koroutinen

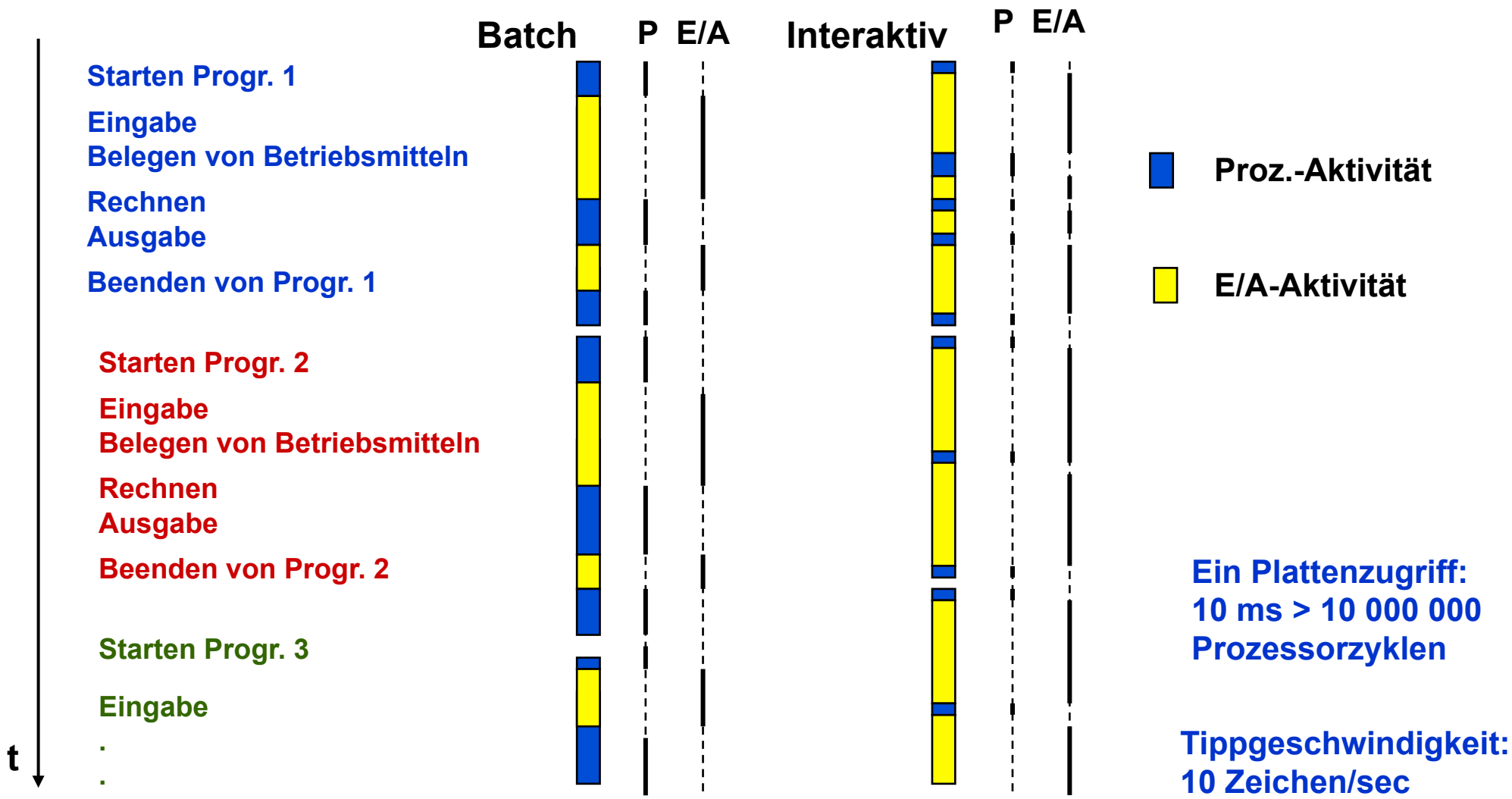
1. die Ausführung beginnt immer an der letzten ”Unterbrechungsstelle“, d. h., an der zuletzt die Kontrolle über den Prozessor abgegeben wurde.

2. Die Kontrollabgabe geschieht dabei grundsätzlich kooperativ (freiwillig).

3. Der Zustand ist invariant zwischen zwei aufeinanderfolgenden Ausführungen.

4. Eine Koroutine muss ihren Zustand bei Abgabe der Kontrolle speichern. Sie kann als ”zustandsbehaftete Prozedur“ aufgefasst werden.

"One program at a time"



"One program at a time"

Fragen:

1. Woher weiß das Programm, wann eine E/A Operation stattfindet oder beendet ist?

2. Was macht man mit der Wartezeit?

Effizienz

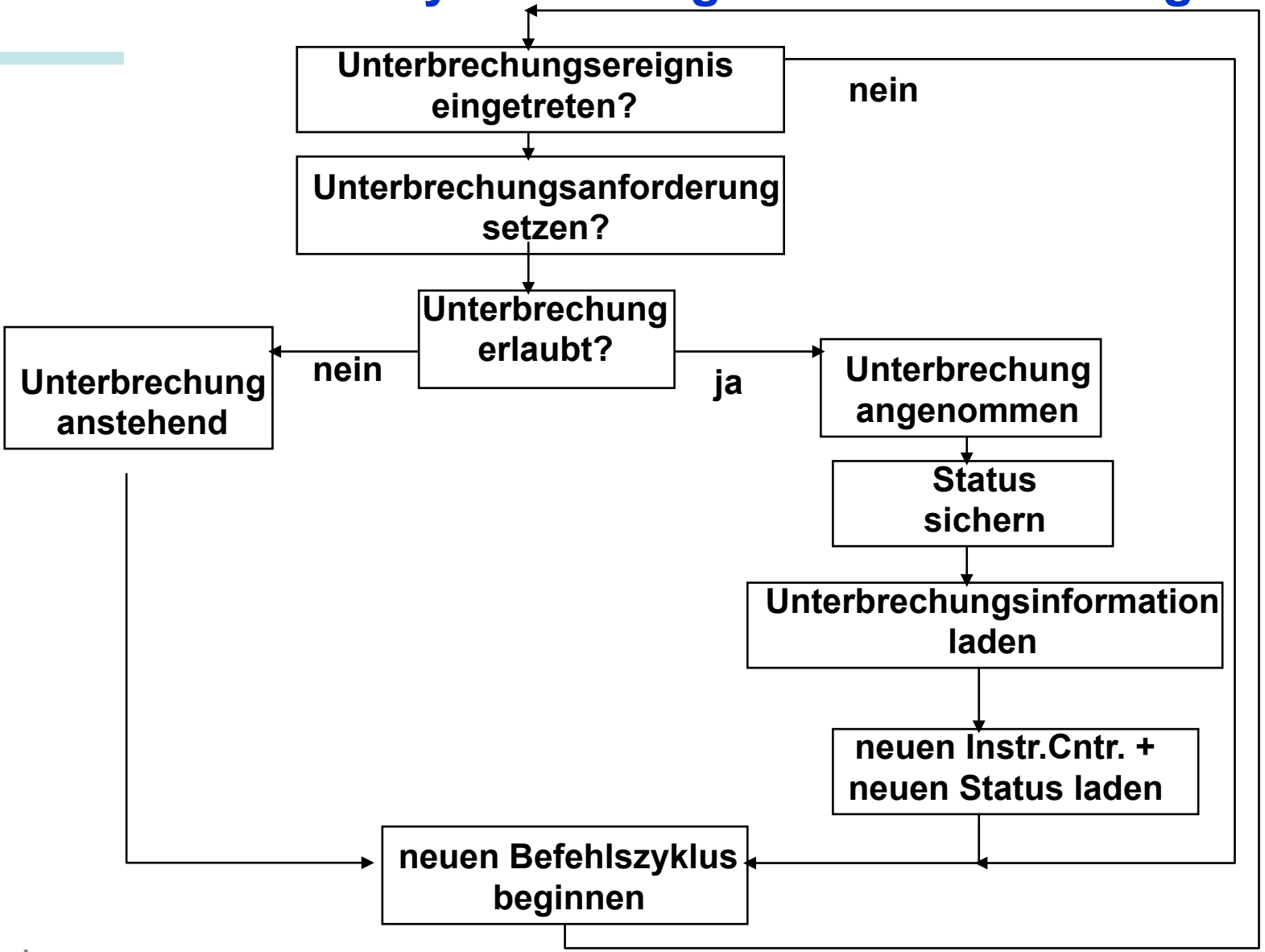
Synchronisation

Ad hoc Lösung zu 1:
Programmierte Ein/Ausgabe, periodische Abfrage von Kontrollregistern und Gerätezustand. ➡ Aktives Warten, busy waiting.

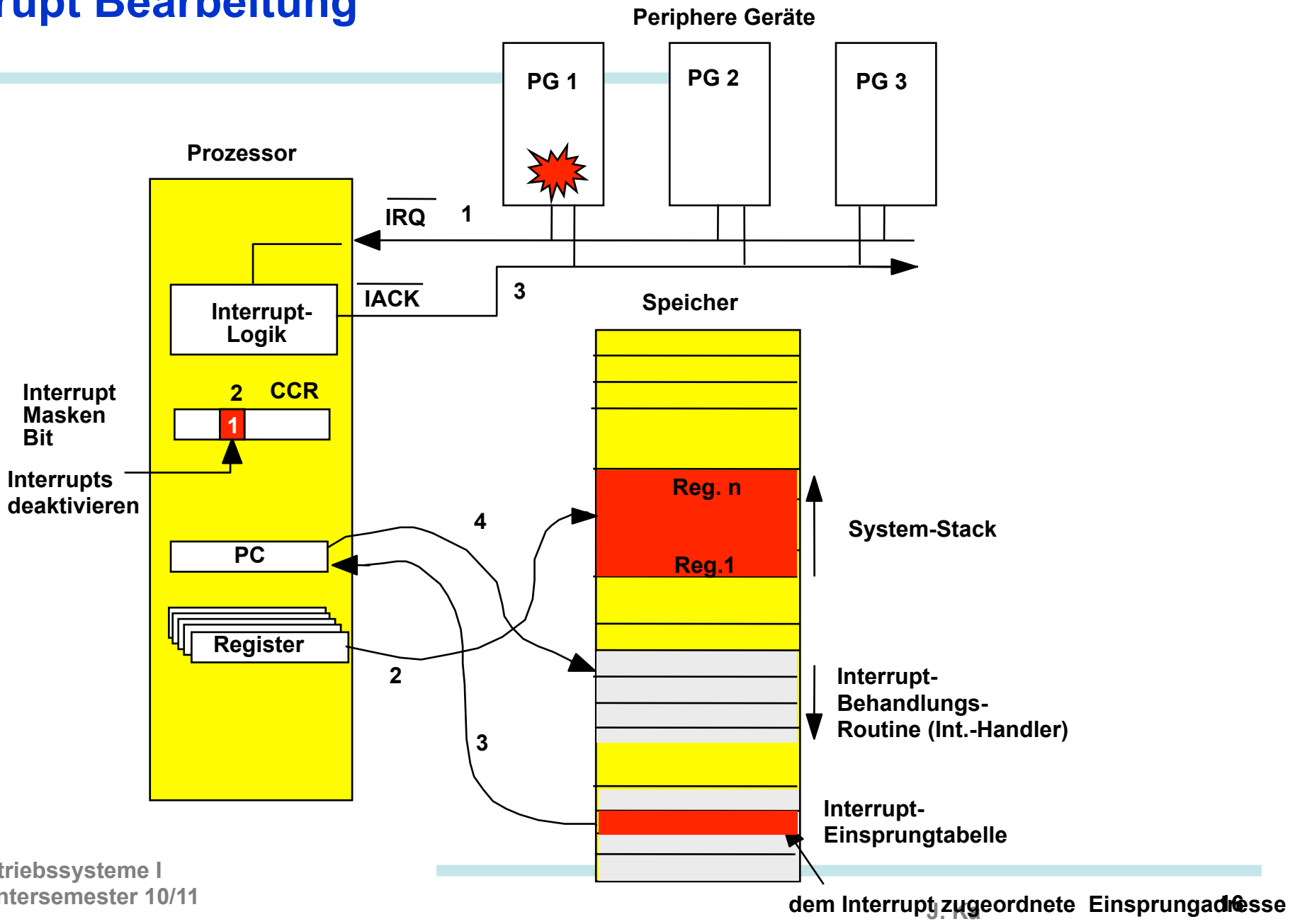
Ad hoc Lösung zu 2:
Explizite Programmierung der Ausführung mehrerer Aufgaben in einem sequentiellen Programmablauf.

Koordination mit Geräten: Systembezogene Unterbrechungen

Hardware-Unterstützung durch die CPU



Interrupt Bearbeitung



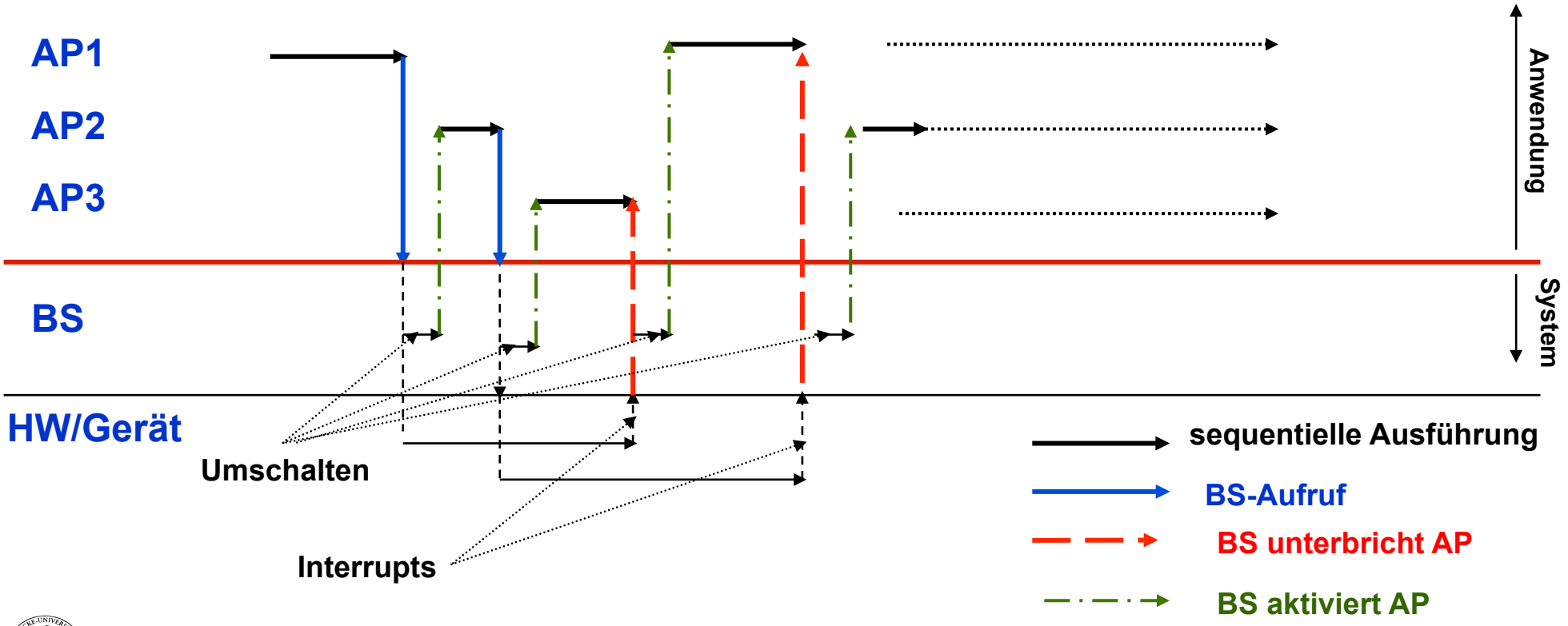
Strukturierung von Aktivitäten und Nebenläufigkeit

Eigenschaften von Unterbrechungen:

- **Unterbrechungen führen auf der Hardwareebene einen Kontrolltransfer durch.**
- **Unterbrechungen werden zwingend behandelt.**
- **Unterbrechungen führen häufig zu einem Wechsel der Schutzebene z.B. von der Anwendungsebene in die Systemebene.**

Strukturierung von Aktivitäten und Nebenläufigkeit

Unterbrechungsbearbeitung bietet die Möglichkeit zu einer für das Anwendungs-programm (AP) völlig transparenten Nebenläufigkeit.

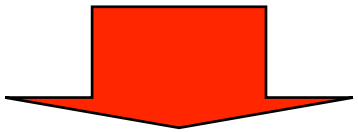


Strukturierung von Aktivitäten und Nebenläufigkeit

Gesucht: Mechanismus zum sicheren Aufruf von Funktionen/Diensten des Betriebssystems.

Problem: Trennung von Anwenderadressraum und Systemadressraum.
Häufig Unterstützung durch die Hardware: Supervisor/User Status, MMU

Anforderungen: Nur erlaubte Einsprungstellen in die Systemroutinen.
Transparenz, wo bestimmte Routinen tatsächlich liegen.



Programmbezogene Unterbrechung, Trap: synchrone, reproduzierbare Unterbrechung

- spezielle Instruktion der CPU oder Ausnahmebedingung der Hardware.
- Systemaufruf.
- Adreßraumverletzung.
- Unbekannter Befehl.
- Falsche Adressierungsart.
- Fehlerhafte Rechenoperation.

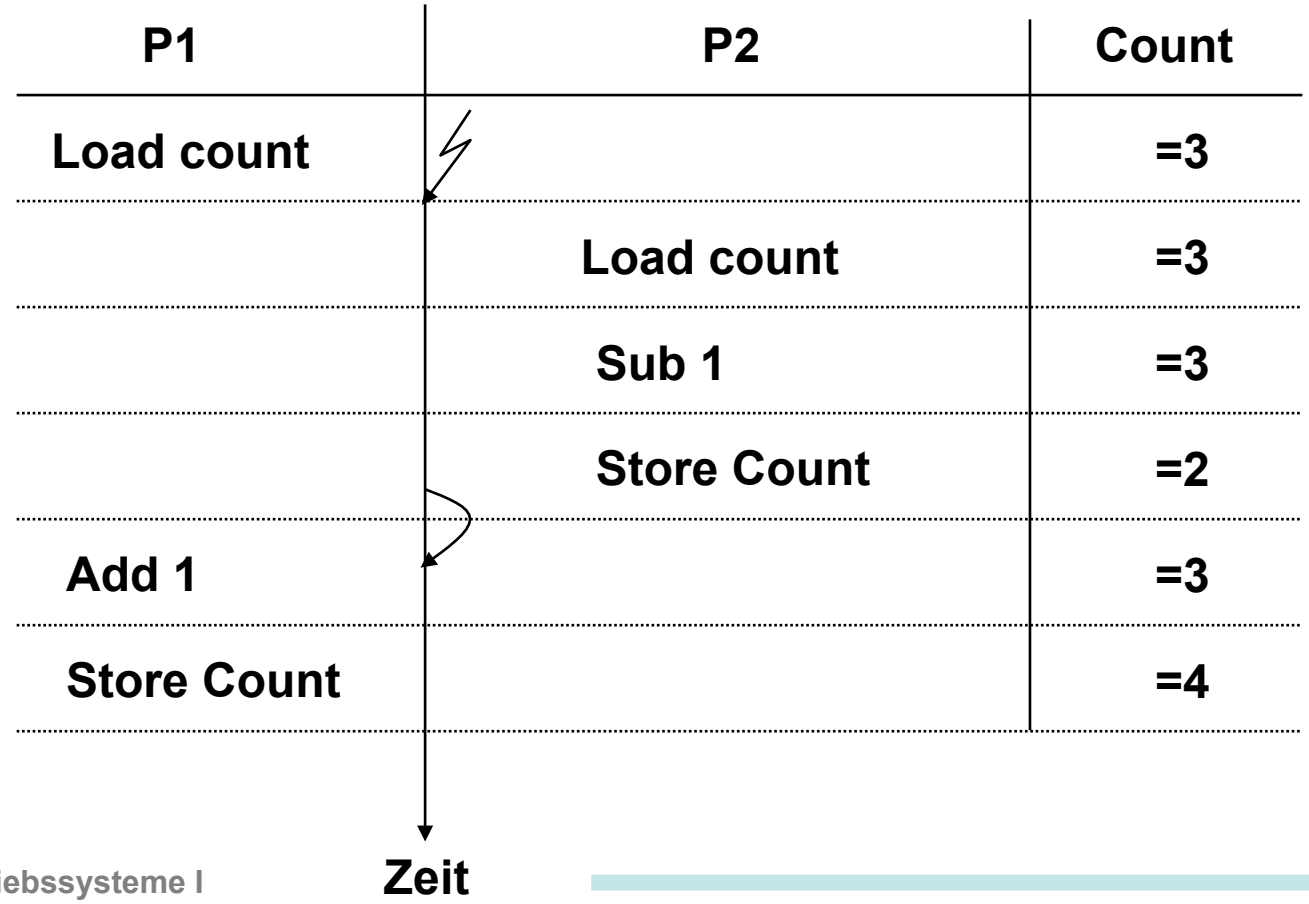
Strukturierung von Aktivitäten und Nebenläufigkeit

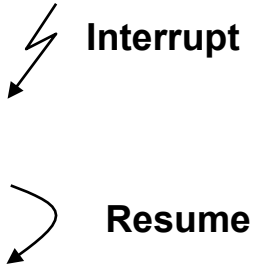
Vergleich Interrupt und Trap:

	Interrupt	Systemaufruf (Trap)	Ausnahme (Trap)
Quelle	extern	intern	intern
Synchronität	async.	sync.	sync.
Vorhersagbarkeit	nein	ja	nur bedingt
Reproduzierbarkeit	nein	ja	ja

Der Preis der Nebenläufigkeit: Konsistenz

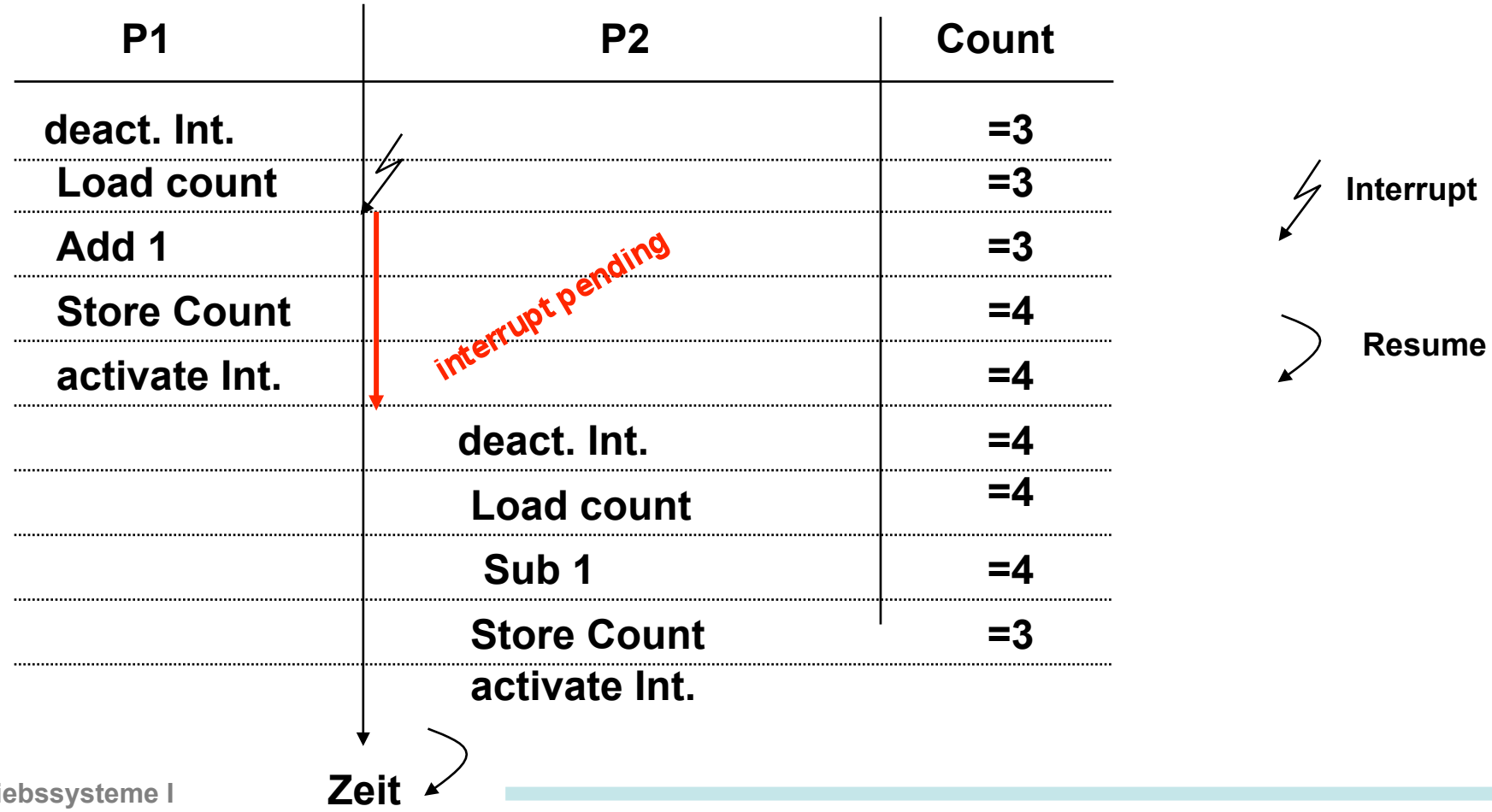
Beispiel: Gemeinsame Nutzung einer Zählvariablen.
Problem: Transparenz der nebenläufigen Aktivitäten.





Der Preis der Nebenläufigkeit: Konsistenz

Lösungsansatz 1: Abschalten der Interrupts



Probleme

Während der Zeitdauer der Interruptabschaltung können wichtige Interrupts nicht behandelt werden.

Deaktivierung betrifft auch völlig unabhängige Programme, die keine gemeinsame Variable nutzen.

Der Preis der Nebenläufigkeit: Konsistenz

Lösungsansatz 2: Unteilbare, atomare Operationen

- Unteilbare read-modify-write Buszyklen,
- Test and Set (TAS), Compare and Swap (CAS),
- Atomare Befehlssequenzen (z.B. ATOMIC (C167 Microcontroller))

Zur Erinnerung: Diese Lösungen werden von den unteren Schichten der Hardware im Zusammenspiel mit dem BS verwendet. Weitere Lösungen zu den Problemen der Nebenläufigkeit werden später behandelt.

Test and Set

Befehl: Test and Set:

```
boolean testset (int i)
{
        if (i==0)
        {
                i= 1;
                return true;
        }
        else
        {
                return false;
        }
}
```

Zusammenfassung

Programme werden in einer individuellen Umgebung ausgeführt.

Die Ausführungsumgebung wird durch einen Aktivierungsblock repräsentiert.

Ein Programm kann mehrfach ausgeführt werden.

Routinen bilden eine asymmetrische, synchrone Aufrufbeziehung.

Koroutinen repräsentieren gleichberechtigte, autonome Kontrollflüsse.

Koroutinen bilden eine symmetrische, synchrone Aufrufbeziehung.

Unterbrechungsbearbeitung ermöglicht eine transparente Ausführung mehrerer Programme.

Kontrolltransfer wird nicht mehr kooperativ organisiert, sondern übergeordnet.

Nebenläufigkeit erfordert Maßnahmen zur Konsistenzerhaltung.