Wettbewerbsrisiken

- *race hazard** (auch *race condition*)
  - fehlerhafte Stelle in einem System, an der eine Berechnung eine unerwartet kritische Abhängigkeit vom relativen Zeitverlauf von Ereignissen zeigt
  - Softwaresysteme sind mit dem Problem konfrontiert, sobald die Ausführung von Programmen überlappend, nebenläufig oder parallel möglich ist
    - während ein Prozess den Platzhalter einer Variablen ausliest (beschreibt), beschreibt (liest) ein anderer Prozess denselben (aus)
    - typischer Fall: die Behandlung asynchroner Programmunterbrechungen
  - Zugriffe auf gemeinsame Variablen hinterlassen ggf. undefinierte Zustände

*Dem Begriff liegt die Vorstellung zu Grunde, dass sich zwei (oder mehr) Signale in einem Wettlauf zueinander befinden, um als erstes die Ausgabe (ein Berechnungsergebnis) zu veranlassen.

Nebenläufiges Zählen (1)

```
unsigned int wheel = 0;

void attribute ((interrupt))
   tip () {

      wheel++;

   }

int main () {

   for (;;)
     printf("%10u", wheel++);

   }
```

```
tip:
   pushl %eax
   movl wheel, %eax
   incl %eax
   movl %eax, wheel
   pushl %eax
   iret

main:
   ...

.L2:
   movl wheel, %edx
   incl %edx
   movl %edx, wheel
   ...
   jmp .L2
```

Welche `wheel` Werte werden ausgegeben?

Nebenläufiges Zahlen (2)

main()		**tip()**		**wheel**
Befehl	**%edx**	**Befehl**	**%eax**	
movl wheel, %edx	42			42
		movl wheel,%eax	42	42
		incl %eax	43	42
		movl %eax,whee	43	43
incl %edx	43			43
movl %edx, whee	43			43

- Zweimal Zählanweisung, nur einmal wheel inkrementiert
- Problem hier: Zählen ist **nicht** atomar.

Atomares Zählen (1)

- Zählen als Elementaroperation (Elop) eines abstrakten Prozessors auffassen
  - eine Elop ist per Definition eine atomare Operation
  - die Unteilbarkeit sichert der die Elop implementierende Prozessor
- als Lösung (f¨ur Monoprozessorsysteme) bieten sich folgende Ansätze an:
  - Komplexbefehle der CPU verwenden (nur bei CISC)

    statt
    ```
    movl wheel, %edx
    incl %edx
    movl %edx, wheel
    ```

    einfach
    ```
    incl wheel
    ```
  - temporäres Abschalten von Interrupts (disable/enable interrupts)

Atomares Zählen (2)

- Komplexbefehl*

```
inline int incr (int* ip) {
   asm ("incl %0" : : "g" (*ip));
   return *ip;
   }
```

- Interrupt Aus/Ein*

```
inline int incr (int* ip) {
   asm ("pushf");
   asm ("cli");
   *ip += 1;
   asm ("popf");
   return *ip;
```

`cli` schaltet ab, `popf` stellt alten Status (`pushf`) her

```
unsigned int wheel = 0;

void attribute ((interrupt)) tip () {
      incr(&wheel);
   }

int main () {
      for (;;) printf("%10u", incr(&wheel));
   }
```

*Beispiele für x86 und gcc Inline Assembler

Betriebssystemmaschine

- Betriebssysteme (sollen) helfen, die semantische Lücke weiter zu verringern
  - sie befinden sich zwischen zwei Stühlen (Ebene 2 und Ebene 4+)
  - eine besonders ”exponierte Lage“, konfrontiert mit viel Konfliktstoff
- ihre logische Struktur ist zuweilen recht komplex und auch sehr vielschichtig
  - sie orientiert sich an der (historisch gewachsenen) Gesamtfunktionalität
  - differenzierte Sichten entstehen mit unterschiedlichen Betriebsarten
- welche Schichten in welcher Weise vorliegen, ist vom realen Problem abhängig

Logische vs. physische Struktur

- Schichten sind logische und nicht physische (d. h., wirkliche) Strukturelemente
  - ihre Funktionen sind realisiert als Makros, Prozeduren, Module oder Prozesse

    Makro zur Übersetzungszeit an der Aufrufstelle expandierte Befehlsfolgen

    Prozedur an mehreren Stellen aufrufbares Unterprogamm als Unikat

    Modul Prozedur(en) mit gekapseltem (gemeinsamen) Datenbestand

    Prozess ein autonomer Kontrollfluss, ggf. mit eigenem Adressraum
  - entsprechend sind die Funktionsaufrufe technisch unterschiedlich ausgelegt
- im funktionalen Sinn ist jede Repräsentation einer Schicht gleich gut

Betriebssystemarchitektur

- Die ”Erscheinungsform“ der Betriebssystemmaschine wird mitbestimmt durch das Abkapselungsinstrument zur technischen Repräsentation der Systemfunktionen:

  prozedurorientiert: Makros und/oder Prozeduren; Systemaufruf bedeutet Makroexpansion bzw. Prozeduraufruf.

  modulorientiert: Module; Systemaufruf bedeutet Schutzdomänenwechsel (gestützt durch Sprachen und/oder Hardware).

  prozessorientiert: Prozesse; Systemaufruf bedeutet Prozedur-Fernaufruf (*remote procedure call, RPC*) bzw. Prozesswechsel.

- Mischformen sind gebräuchlich. Keine Form ist der anderen zwingend überlegen.

Zusammenfassung

- zwischen Problemstellung und Rechnersystem klafft eine semantische Lücke
  - der semantische Abstand variiert mit dem Problem und der Hardware
  - auch das Wissensniveau der jeweiligen ”Experten“ ist ein Einflussfaktor
- die Hardware/Software-Hierarchie hilft, die semantische Lücke zu schließen
  - ein Rechner setzt sich demnach aus fünf grundsätzlichen Ebenen zusammen
  - eine Ebene ist repräsentiert als virtuelle Maschine bzw. reale Maschine
- die Betriebssystemmaschine ist eine besondere Art einer virtuellen Maschine
  - diese Ebene 3 bewerkstelligt die partielle Interpretation der Systemaufrufe
  - ihre Architektur variiert mit der jeweils zu unterstützenden Problemdomäne

Pro'zess <m.; -es, -e> Vorgang, Ablauf [lat.]

- Fadenverwaltung ~inkarnation, ~deskriptor
  - Aktivitätsträger, Koroutine
- ~abfertigung (process dispatching)
  - CPU-Stoß (CPU burst) vs. E/A-Stoß (I/O burst)
- ~einplanung (CPU/process scheduling)
  - Klassifikation, Einplanungsebenen und -strategien
- Aspekte der Nebenläufigkeit (race hazards/conditions)

Prozessinkarnation

- die Verwaltungseinheit zur Beschreibung/Repräsentation eines Prozesses
  - der Typ einer Datenstruktur ”**Prozessdeskriptor**“ (PD)
    - **Prozesskontrollblock** (process control block, PCB), . . .
    - UNIX Jargon: **proc structure** (von ”struct proc“)
  - Kopf eines (komplexen) Datenstrukturgeflechts (+ Objektkomposition)
- das (Software-) Betriebsmittel zur Ausführung eines Programms
  - eine Instanz vom Typ ”PD“
- die Identität für ein sich in Ausführung befindliches Programm
  - das mit einer **Prozessidentifikation** (PID) assoziierte Objekt

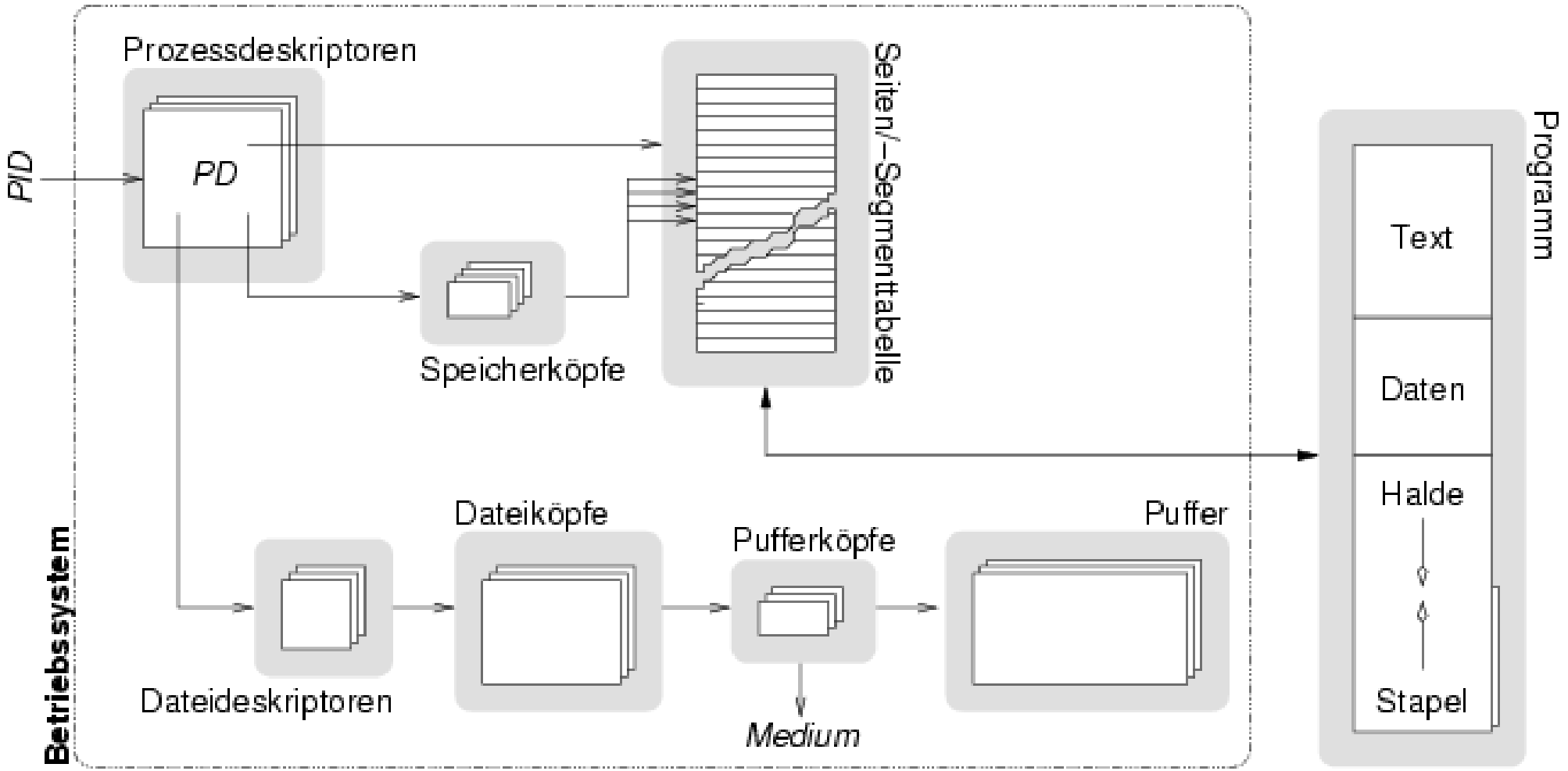
Prozessdeskriptor

- Dreh- und Angelpunkt, der alle prozessbezogenen Betriebsmittel bündelt
  - Speicher- und, ggf., Adressraumbelegung
    - Text-, Daten-, Stapelsegmente (code, data, stack)
  - Dateideskriptoren und -köpfe (inode)
    - {Zwischenspeicher,Puffer}deskriptoren, Datenblöcke
  - Datei, die das vom Prozess ausgeführte Programm repräsentiert
- zentrales Objekt, das Prozess- und Prozessorzustände beschreibt
  - Laufzeitkontext des zugeordneten Programmfadens/Aktivitätsträgers
  - gegenwärtiger Abfertigungszustand (Scheduling-Informationen)
  - anstehende Ereignisse bzw. erwartete Ereignisse
  - Benutzerzuordnung und -rechte

Variationen des Prozessdeskriptors

- Aufbau & Struktur des PD ist höchst abhängig von Betriebsart/-zweck:
  - **Adressraumdeskriptoren** sind nur notwendig im Falle von Systemen, die eine Adressraumisolation erfordern.
  - Für ein Sensor-/Aktorsystem haben **Dateideskriptoren/-köpfe** wenig Bedeutung.
  - In ROM-basierten Systemen durchlaufen die Prozesse oft immer nur ein und dasselbe **Programm**.
  - In Einbenutzersystemen ist es wenig sinnvoll, prozessbezogene **Benutzerrechte** verwalten zu wollen.
  - Bei statischer Ablaufplanung ist die Buchführung von **Abfertigungszuständen** verzichtbar.
  - Ebenso fällt **Ereignisverwaltung** nur bei ereignisgesteuerten und/oder präemptiven Systemen an.
  - . . .
- Festlegung auf genau eine Ausprägung grenzt Einsatzgebiete unnötig aus

Generische Datenstruktur „Prozess“



Koroutine (1)

- ein autonomer Kontrollfluss innerhalb eines Programms (z. B. Betriebssystem)
  - **Programm(kontroll)faden** (*thread of control, TOC*)
- mit zwei wesentlichen Unterschieden zu herkömmlichen Routinen/Prozeduren:
  1. die Ausführung beginnt immer an der letzten ”Unterbrechungsstelle“
     - d. h., an der zuletzt die Kontrolle über den Prozessor abgegeben wurde
     - die Kontrollabgabe geschieht dabei grundsätzlich kooperativ (freiwillig)
  2. der Zustand ist invariant zwischen zwei aufeinanderfolgenden Ausführungen
- eine Koroutine kann als ”zustandsbehaftete Prozedur“ aufgefasst werden

Koroutine (2)

- Koroutinen sind Prozeduren ähnlich, es fehlt jedoch die Aufrufhierarchie:
  - Beim Verlassen einer Koroutine geht anders als beim Verlassen einer Prozedur die Kontrolle nicht automatisch an die aufrufende Routine zurück. Stattdessen wird mit einer `resume`-Anweisung beim Verlassen einer Koroutine explizit bestimmt, welche andere Koroutine als nächste ausgeführt wird.
- ein programmiersprachliches Mittel zur Prozessorweitergabe an Prozesse

Routine vs. Koroutine

- hinter beiden Konzepten verbergen sich unterschiedliche Ablaufmodelle:

  asymmetrische Aktivierung im Falle von Routinen

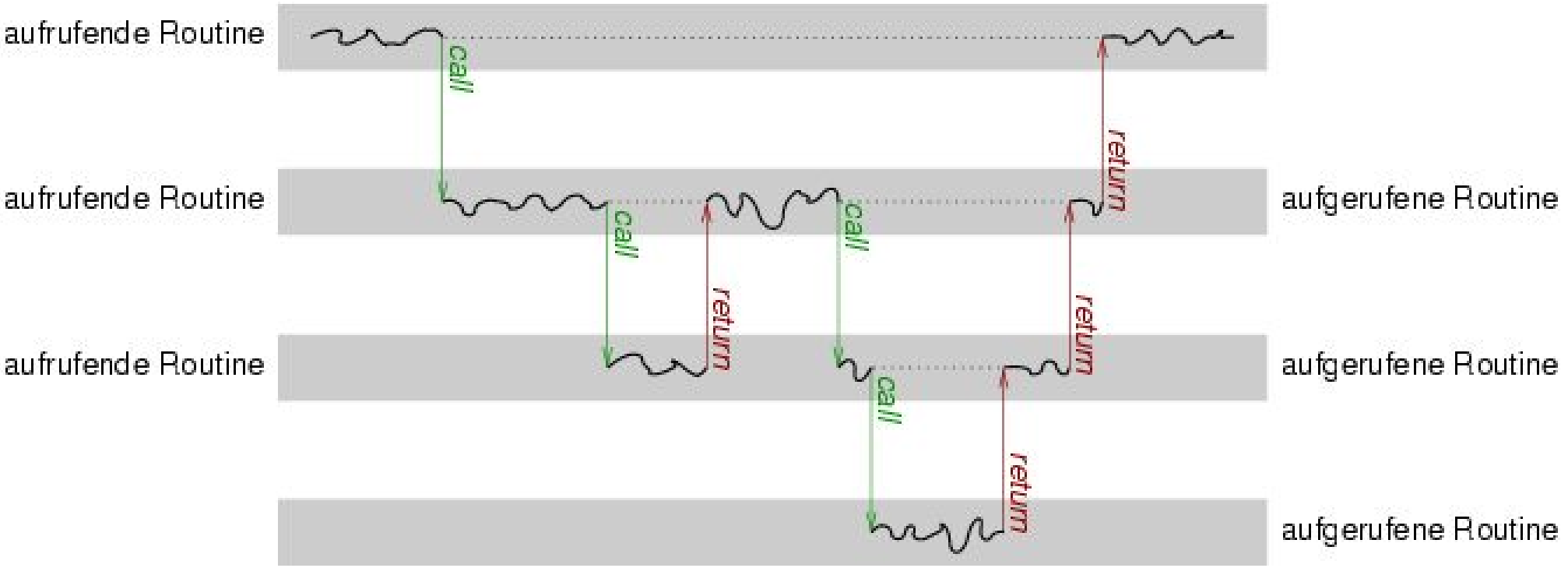
  - die Beziehung zwischen den Routinen ist nicht gleichberechtigt
  - über die Routinen ist eine Aufrufhierarchie definiert

  symmetrische Aktivierung im Falle von Koroutinen

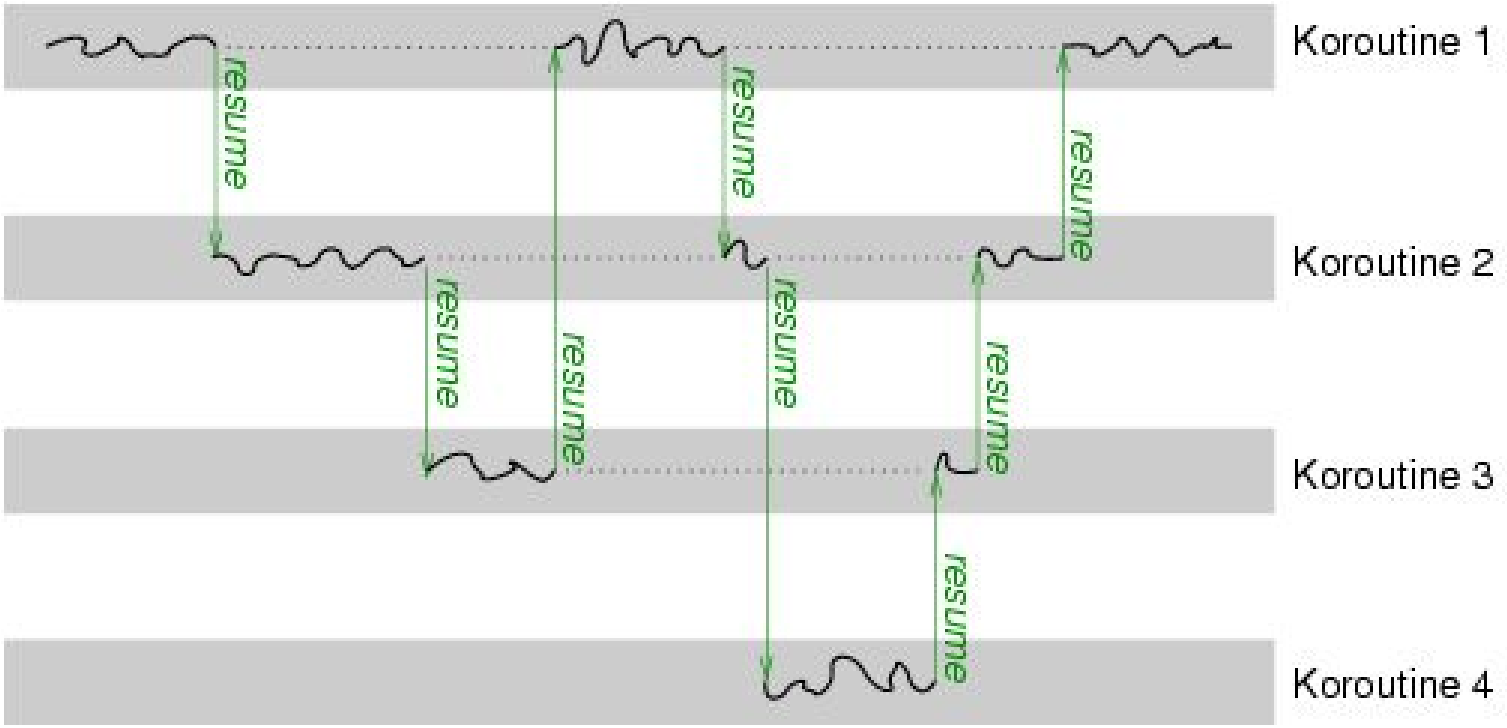
  - die Beziehung zwischen den Koroutinen ist gleichberechtigt
  - über die Koroutinen ist keine Aufrufhierarchie definiert

- Routinen werden aufgerufen, um sie zu aktivieren, im Gegensatz zu Koroutinen, die erzeugt, aktiviert und zerstört werden und sich selbst nur deaktivieren

Aufrufhierarchie Routine



Nebenläufigkeit Koroutine



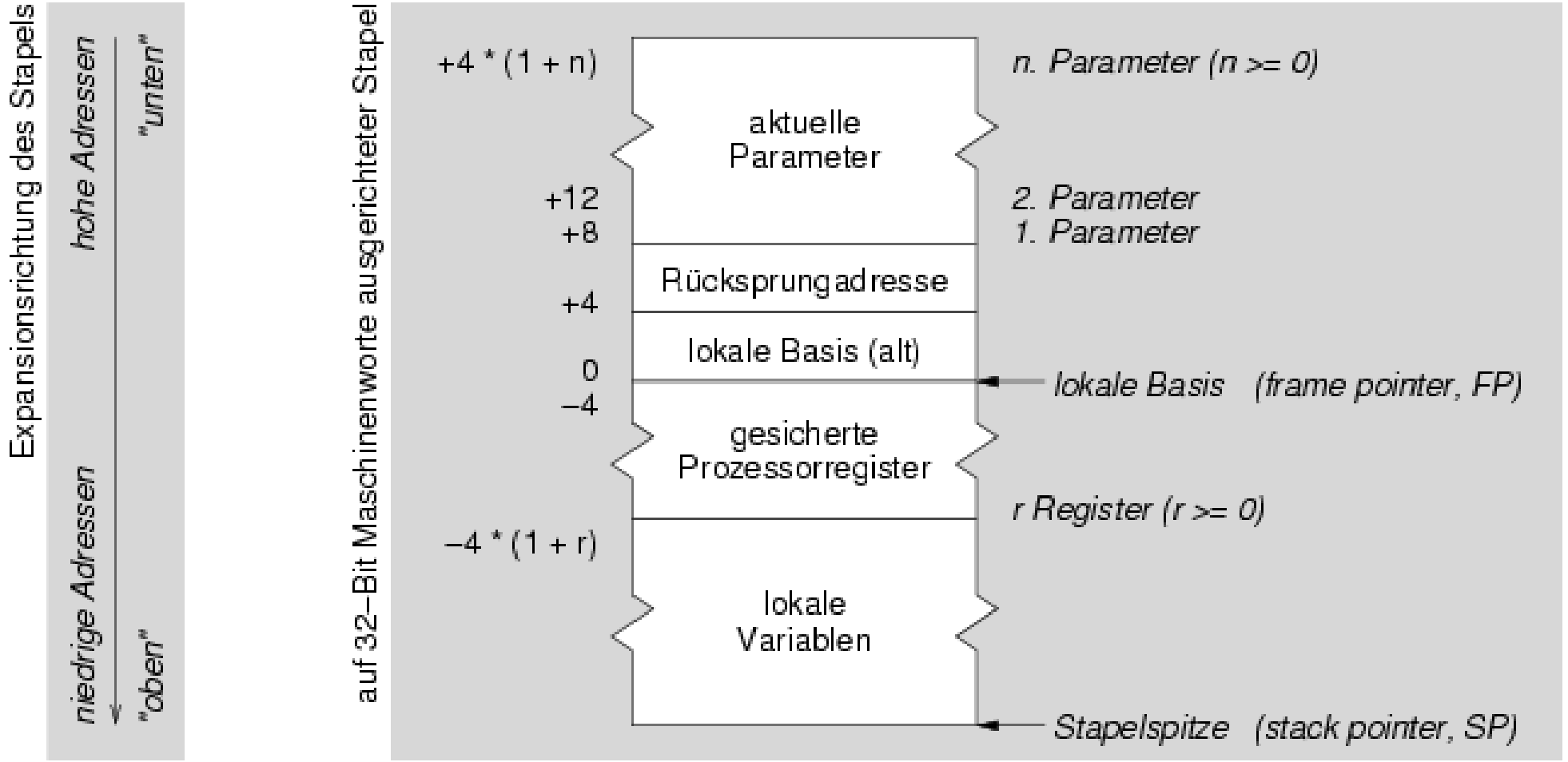
Gemeinsamkeiten {Kor,R}outine

- {Kor,R}outinen sind zu reaktivieren, um weiter ausgeführt werden zu können:
  - Routine beim Rücksprung aus der aufgerufenen Instanz
  - Koroutine beim Suspendieren der die Kontrolle abgebenden Instanz
- jeder ”Aufruf“ hinterlässt seinen ”Fußabdruck“ im Aktivierungsblock
  - die Rückkehradresse zur aufrufenden {Kor,R}outine
  - die von der {Kor,R}outine belegten Prozessorregister
- der Aufbau des Aktivierungsblocks ist abhängig vom Prozessor bzw. Kompilierer

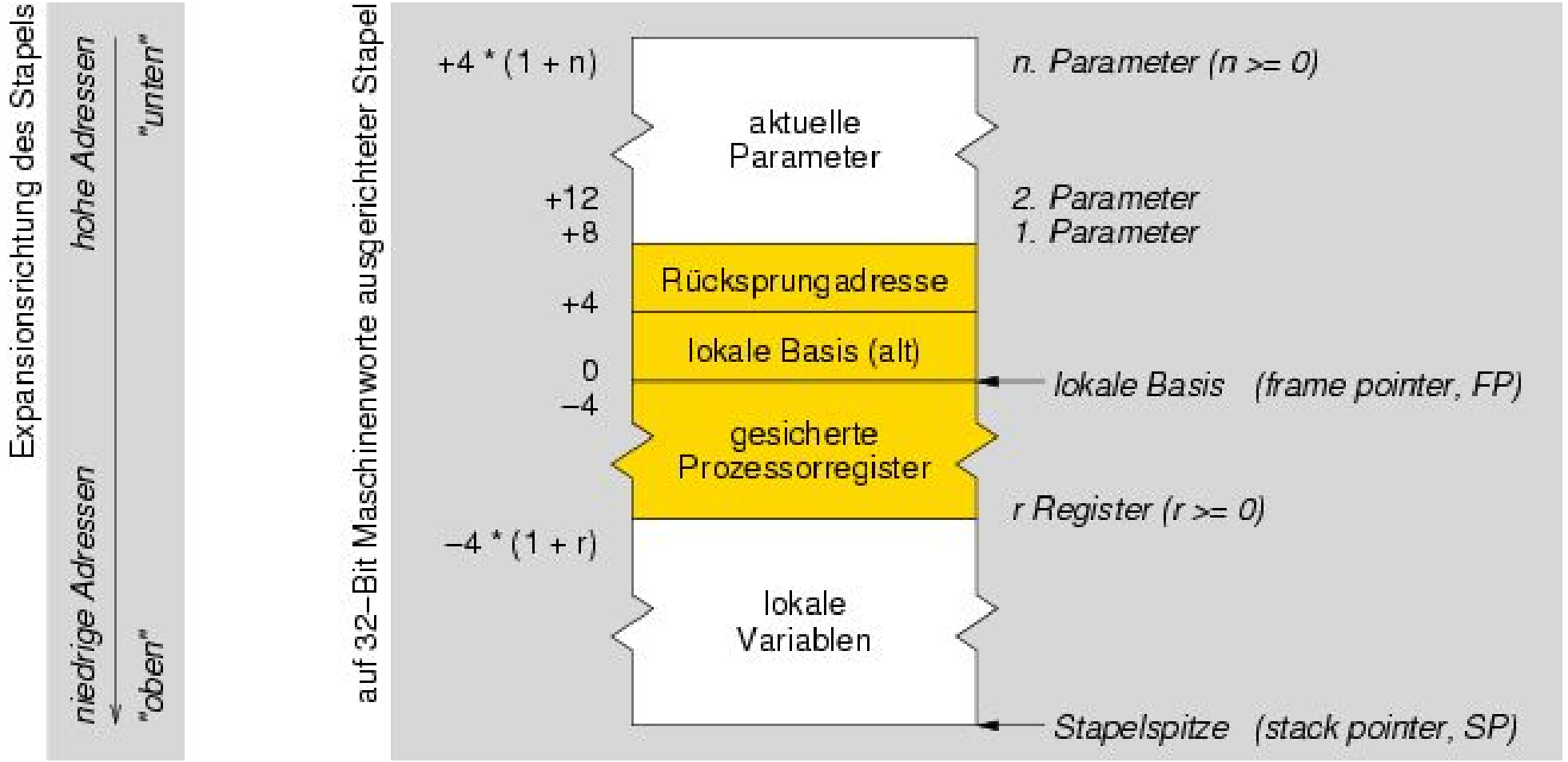
Unterschiede {Kor,R}outine

- eine Koroutine besitzt *eigene* Betriebsmittel zur Aktivierungsblockverwaltung
  - Art und Anzahl der Betriebsmittel ist sehr problemspezifisch
    - CPU: CISC (Stapel) vs. RISC (Register und/oder Stapel) [Ebene 2]
    - Kompilierer: Laufzeitmodell der Programmiersprache [Ebene 5]
    - Anwendungsprogramm: die zu bew'ltigende Aufgabe [Ebene 6]
  - Verfügbarkeit eigener Betriebsmittel begründet (relative) Unabhängigkeit
- eine Routine muss sich diese Betriebsmittel mit anderen Routinen teilen

Aktivierungsblock (*activation record*)



Aktivierungsblock (2)



Laufzeitkontext

- der Aktivierungsblock enthält den Laufzeitstatus der aufrufenden Routine:
  - die Programmadresse, an der die Routine ihre Ausführung nach erfolgtem Rücksprung weiter fortsetzen wird
    - die Adresse des dem Aufruf nachfolgenden Maschinenbefehls
  - die Inhalte der Arbeitsregister, die von der aufgerufenen Routine im weiteren Verlauf verwendet werden
    - die Inhalte der nicht-flüchtigen Register; beim x86: EBP, ESI, EDI, EBX
- dieser Status ist invariant in Bezug auf die Ausführung aufgerufener Routinen
  - die Anzahl gesicherter nicht-flüchtiger Register ist jedoch variabel
  - abhängig von der aufgerufenen Routine und der virtuellen Maschine

Autonomer Kontrollfluss

- im Gegensatz zur Routine bedeutet die Aktivierung einer Koroutine:
  - Laufzeitstatussicherung des aktiven Kontrollflusse
  - Laufzeitstatuswiederherstellung eines inaktiven Kontrollflusses
- d. h., Laufzeitstatus-Sicherung/Wiederherstellung verläuft in zwei Dimensionen:

  vertikal bei Einsprung in und Rücksprung aus einer Routine ohne dabei den aktiven Kontrollfluss zu wechseln

  horizontal bei Deaktivierung und Aktivierung eines Kontrollflusses
- Koroutinen repräsentieren gleichberechtigte, autonome Kontrollflüsse

Virtuelle Maschine / Laufzeitstatus von Koroutinen

Ebene 5 abstrakter Prozessor ”Kompilierer“

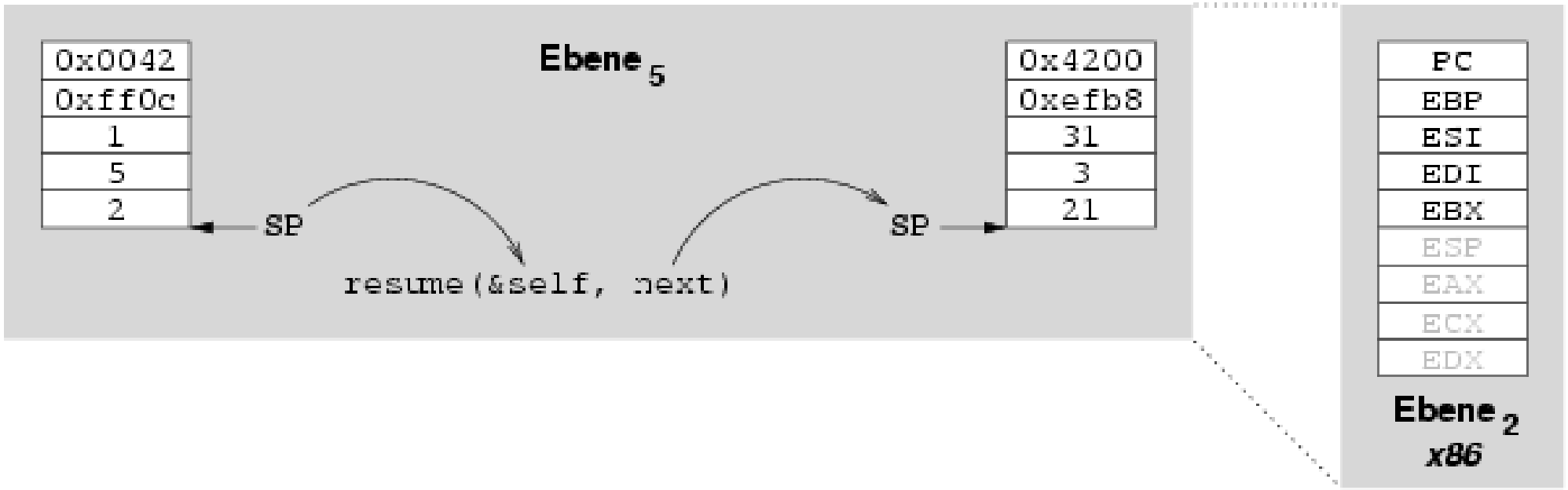
- der ”Prozessor“ unterscheidet zwischen zwei Arten von Registern
  1. **nicht-flüchtige Register** ; invariante Inhalte über Prozedurgrenzen hinweg
  2. **flüchtige Register** ; sonst
- die nicht-flüchtigen Register speichern den Laufzeitstatus einer Koroutine
- ein Koroutinenwechsel muss die Inhalte ”einiger“ CPU-Register austauschen

Ebene 4 abstrakter Prozessor ”Assemblierer“

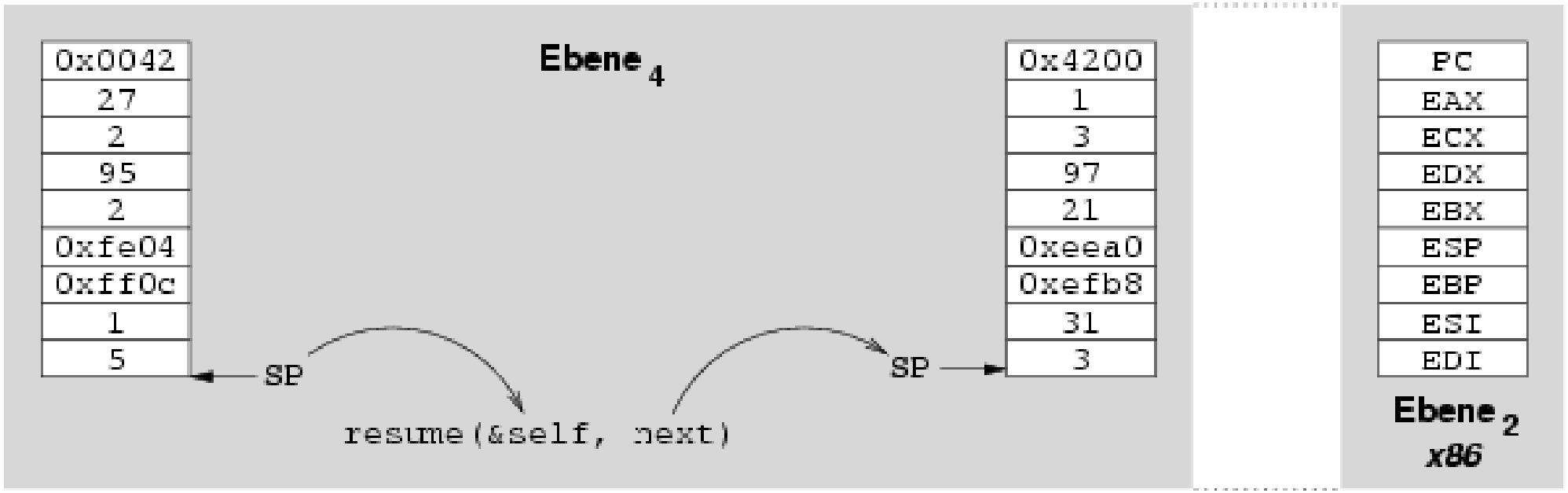
- alle (Ebene 2/CPU) Register speichern den Laufzeitstatus einer Koroutine
- ein Koroutinenwechsel muss die Inhalte ”aller“ CPU-Register austauschen

- Koroutinenwechsel auf Ebene 5 (z. B. in C) sind effzienter als auf Ebene 4

Kontrollflusswechsel (1)



Kontrollflusswechsel (2)



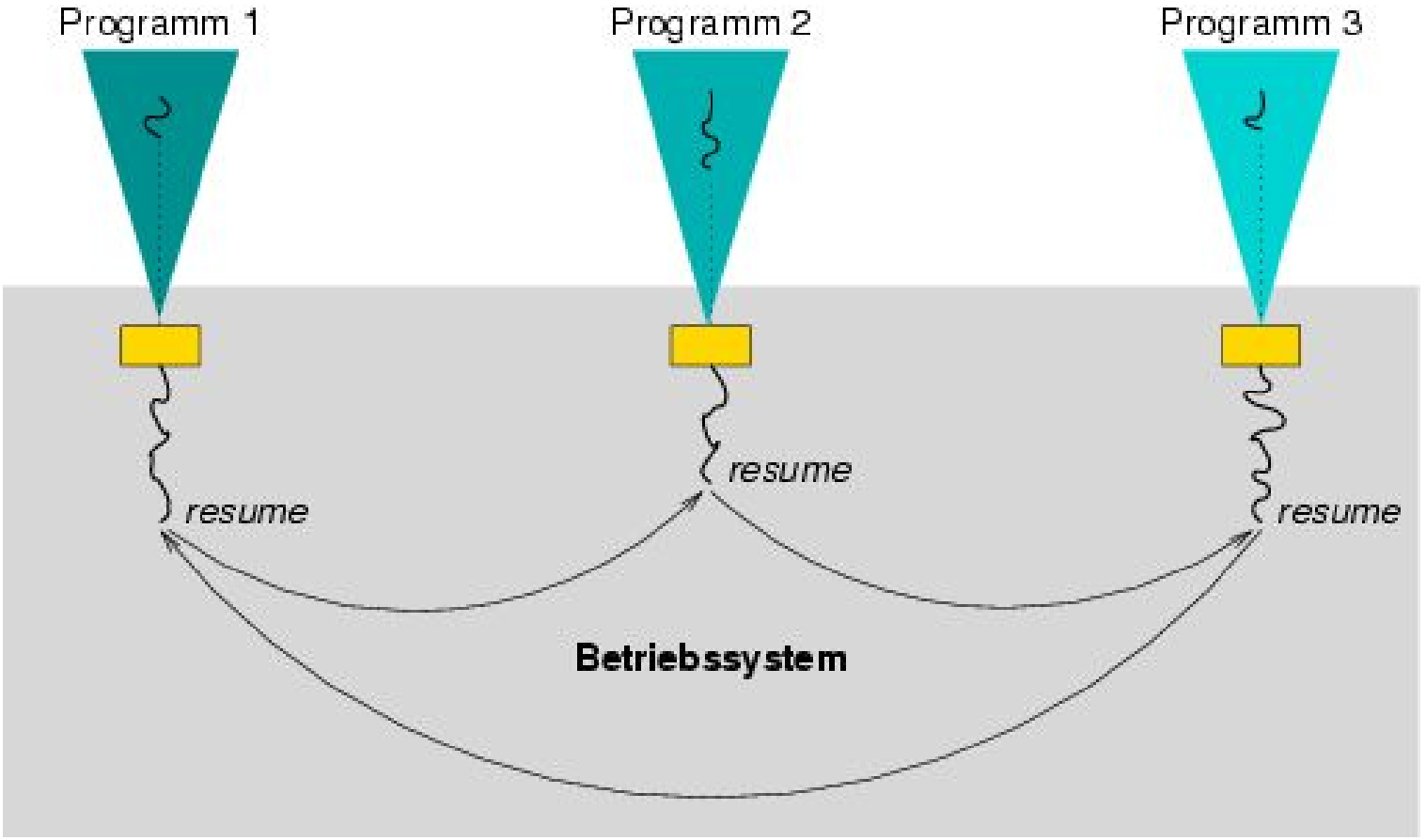
Koroutine – Operationsprinzip

- die Koroutine ”mechanisiert“ den autonomen Kontrollfluss eines Prozesses
  - mit ihr wird die Prozessorzuteilung an ausführbereite Prozesse möglich
  - sie bildet die Grundlage zum Multiplexen der CPU zwischen Prozessen
- das nicht-sequentielle Operationsprinzip von Koroutinen ist streng kooperativ
  - eine Koroutine gibt das Betriebsmittel ”CPU“ immer nur freiwillig ab
  - die Abgabe ist programmiert, sie muss darüberhinaus erreichbar sein
- Betriebssysteme schützen die CPU vor ”unkooperativen Koroutinen“
  - den Koroutinen kann das Betriebsmittel ”CPU“ entzogen werden
  - einer Koroutine wird es dadurch unmöglich, die CPU zu monopolisieren

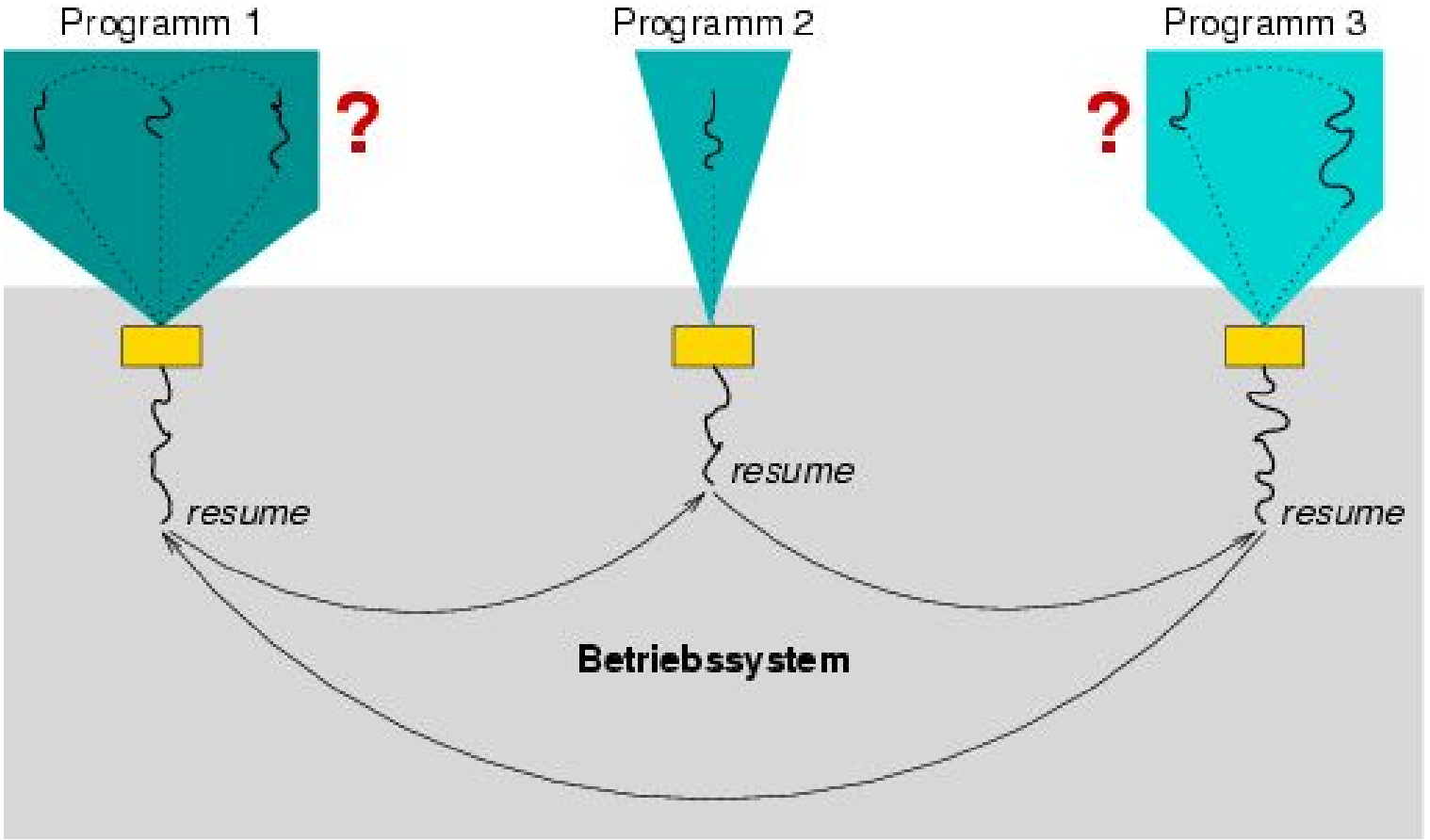
Koroutine und Mehrprogrammbetrieb

- die Koroutine als ”abstrakter Prozessor“ zur Ausführung eines Programms:
  - für jedes auszuführende Programm wird genau eine Koroutine bereitgestellt
  - ist eine Koroutine aktiv, läuft auch immer das ihr zugeordnete Programm
  - um ein anderes Programm auszuführen, ist die Koroutine zu wechseln
- die Koroutinen sind als ”Aktivitätsträger“ Bestandteil des Betriebssystems
  - für den Laufzeitstatus gibt es pro Koroutine eine Kontextvariable
  - jedem Benutzerprogramm ist eine solche Variable zugeordnet
  - die Variablen sind gültig nur innerhalb des Programms ”Betriebssystem“
- ein Betriebssystem ist Inbegriff für das nicht-sequentielle Programm

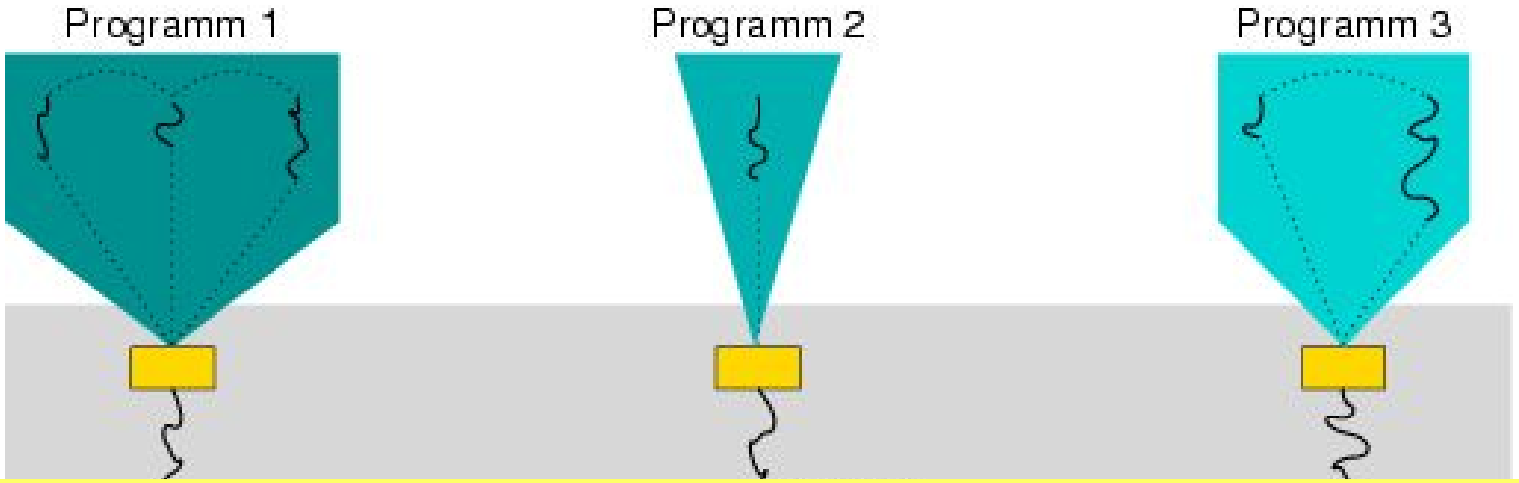
Verarbeitung sequentieller Programme



Verarbeitung (nicht)-sequentieller Programme



Verarbeitung nicht-sequentieller Programme



- Programme 1 und 3 sind nicht-sequentielle Programme. Die Verarbeitung aller Kontrollflüsse derartiger Programme durch das Betriebssystem ist nur möglich, wenn im Betriebssystem f¨ur jeden Kontrollfluss ein Repräsentant in Form einer Koroutine zur Verf¨ugung steht. Genauso, wie das nicht-sequentielle Programm”Betriebssystem“ seine Kontrollflüsse selbst verwalten muss, wenn die CPU dazu unfähig ist, müssen Programme 1 und 3 ihre Kontrollflüsse selbst verwalten, wenn das Betriebssystem dazu unfähig ist.

Koroutinen == Kooperation

- den Gültigkeitsbereich von Koroutinen legt das sie umgebende Programm fest
  - programmübergreifende Koroutinenwechsel lassen sich nicht ausdrücken
  - programmübergreifende Koroutinenauswahl ist nicht praktizierbar
- die Einplanung (scheduling) von Koroutinen ist nicht betriebsmittelorientiert
  - im Vordergrund stehen vielmehr organisatorische/strukturelle Fragen
  - nicht-sequentielle Programme von Koroutinen bilden Kooperativen
  - Koroutinen sind Mittel zum Zweck, Kontrollflüsse zu repräsentieren
- lokale Kooperation ist um ”globales Denken und Handeln“ zu ergänzen