Systemprogrammierung

Prozessverwaltung: Einlastung

Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

Ergänzende Materialien

Gliederung

Programmfaden

Einlastungseinheit (engl. *unit of dispatching*), Ausführungsstrang, **Aktivitätsträger**

Einlastung der CPU folgt mehr oder weniger zeitnah zur Ablaufplanung von Programmfäden, ist ihr nachgeschaltet

- ein Abfertiger (engl. *dispatcher*) führt die eingeplanten Fäden der CPU zur Verarbeitung zu
  - *Mechanismus* zur Prozessverarbeitung: CPU umschalten
- dazu nimmt er Aufträge vom Planer (engl. *scheduler*) entgegen
  - *Strategie* zur Prozessverarbeitung: Aufträge an die CPU sortieren

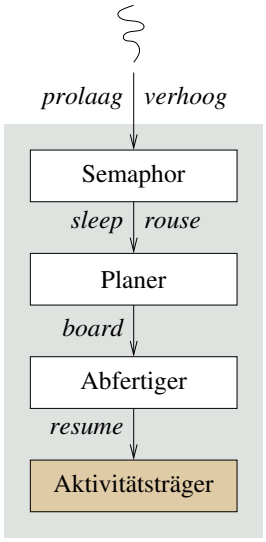
Umschalten der CPU bedeutet, zwischen zwei Aktivitätsträgern desselben oder verschiedener Programme zu wechseln

1. CPU-Stoß endet: der laufende Aktivitätsträger wird weggeschaltet
2. CPU-Stoß beginnt: ein laufbereiter Aktivitätsträger wird zugeschaltet

- Aktivitätsträger lassen sich adäquat durch **Koroutinen** repräsentieren

Gesamtzusammenhang

Funktionale Hierarchie typischer Komponenten einer Prozessverwaltung



prolaag bedingte Prozessblockade

verhoog bedingte Prozessdeblockade
- zur Erinnerung: SP1, VI Prozesse, S. 22

sleep unbedingte Prozessblockade und -auswahl
- nächsten laufbereiten Prozess auswählen

rouse bedingte Prozessdeblockade und ggf. -auswahl
- schlafende Prozesse aufwecken

board Prozesseinlastung
- ausgewählten Prozess der CPU zuteilen
- Prozesszeiger aktualisieren

resume Koroutinenwechsel
- Prozessorstatus austauschen

Gliederung

Routinenartige Komponente eines Programms

Koroutine $\mapsto$ **gleichberechtigtes Unterprogramm**

- entwickelt um 1958 [6, S. 226], genutzt als Architekturmerkmal eines Fließbandübersetzers (engl. *pipeline compiler*)
- darin wurden zentrale Komponenten des Übersetzers konzeptionell als Datenflussfließbänder zwischen Koroutinen aufgefasst
- Koroutinen repräsentierten dabei *first-class* Prozessoren wie z.B. Lexer, Parser und Codegenerator

Ko{existierende, operierende}-Routine: [3, S. 396]

- *coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines*
- *subroutines all at the same level, each acting as if it were the master program when in fact there is no master program*

Autonomer Kontrollfluss eines Programms

Kontrollfaden (engl. *thread of control*, TOC)

Koroutinen konkretisieren Prozesse (implementieren Prozessinkarnationen), sie repräsentieren die Aktivitätsträger von Programmen

1. Ausführung beginnt immer an der letzten „Unterbrechungsstelle“
   - d.h., an der zuletzt die Kontrolle über den Prozessor abgegeben wurde
   - Kontrollabgabe geschieht dabei grundsätzlich kooperativ (freiwillig)
2. zw. aufeinanderfolgenden Ausführungen ist ihr Zustand invariant
   - lokale Varbiablen (ggf. auch aktuelle Parameter) behalten ihre Werte
   - bei Abgabe der Prozessorkontrolle terminiert die Koroutine nicht

Koroutine $\equiv$ „zustandsbehaftete Prozedur“

Aktivierungskontext bleibt während Phasen der Inaktivität erhalten:

deaktivieren $\mapsto$ Koroutinenkontext „einfrieren“ (sichern)

aktivieren $\mapsto$ Koroutinenkontext „auftauen“ (wieder herstellen)

Programmiersprachliches Mittel zur Prozessorweitergabe

Konzept zum Multiplexen des Prozessors zwischen Prozessinkarnationen

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. [5, S. 49]*

Routine kein Kontrollflusswechsel bei Aktivierung/Deaktivierung

- asymmetrisch, ungleichberechtigte Rollen

Koroutine Kontrollflusswechsel bei Aktivierung/Deaktivierung

- symmetrisch, gleichberechtigte Rollen

Routine vs. Koroutine

Merkmal	Routine	Koroutine
Aktivierung	• mehrmals	• einmal: initial
Unterbrechung	• niemals	• mehrmals
Fortsetzung	• niemals	• mehrmals
Beendigung	• einmal: je Aktivierung	• niemals

„Unsterblichkeit“

- da Koroutinen eine Aufrufhierarchie fehlt, können sie sich auch nicht von selbst beendigen
  - Routinen werden durch Unterprogrammaufrufe aktiviert, Rücksprung aus dem Unterprogramm beendigt somit auch ihre Ausführung
  - Koroutinen werden nicht durch Unterprogrammaufrufe aktiviert
- um sich zu beendigen müssen sie einer anderen Koroutine dazu einen diesbezüglichen Auftrag erteilen

Routine vs. Koroutine: Spezialisierung/Generalisierung

Routine spezifischer als Koroutine

- ein einziger Einstiegspunkt
  - immer am Anfang
- ggf. mehrere Ausstiegspunkte
  - kehrt aber nur einmal zurück
- begrenzte Lebensdauer
  - Ausstieg ⇝ Beendigung

Koroutine generischer als Routine

- ggf. mehrere Einstiegspunkte
  - dem letzten Ausstieg folgend
- ggf. mehrere Ausstiegspunkte
  - kehrt ggf. mehrmals zurück
- unbegrenzte Lebensdauer
  - Ausstieg ⇝ Unterbrechung

Routinen können durch Koroutinen implementiert werden

call ↦ *resume* der aufgerufenen Routine an ihrer Einsprungadresse
- Rücksprungkontext einfrieren, Aktivierungsblock aufsetzen

return ↦ *resume* der aufrufenden Routine an ihrer Rücksprungadresse
- Aktivierungsblock zurücksetzen, Rücksprungkontext auftauen

Routine vs. Koroutine: Laufzeitstapel

Mitbenutzung desselben Laufzeitstapels durch mehrere Koroutinen ist der von Routinen sehr ähnlich — und legt die Analogie auf S. 10 nahe:

- Unterbrechung und Fortsetzung von Koroutinen sind Spezialfälle des Ansprungs von Unterroutinen (engl. *jump to subroutine*, JSR)
- Prozessoren (Soft-/Hardware) stellen Elementaroperationen dafür zur Verfügung ⇝ *Buchführung über Fortsetzungspunkte*

PDP-11/40

> *Another special case of the JSR instruction is* **JSR PC,@(SP)+** *which exchanges the top element of the processor stack and the contents of the program counter. Use of this instruction allows two routines to swap program control and resume operation when recalled where they left off. Such routines are called “co-routines.” [4, S. 4-58/59]*

Buchführung über Fortsetzungspunkte

Fortsetzung (engl. *continuation*) einer Programmausführung

Fortsetzungspunkt ist die Programmstelle, an der die Wiederaufnahme (engl. *resumption*) der Programmausführung möglich ist

- eine Adresse im Textsegment, an der ein Kontrollfluss (freiwillig, erzwungenermaßen) unterbrochen wurde
- die Stelle, an der der CPU-Stoß der einen Koroutine endet und der CPU-Stoß einer anderen Koroutine beginnt

Koroutinen zu implementieren bedeutet, **Programmfortsetzungen** zu verbuchen und **Aktivierungskontexte** zu wechseln:

- Fortsetzungsadressen sind dynamisch festzulegen und zu speichern
  - z.B. wie im Falle der Rücksprungadresse einer Prozedur
- ggf. sind weitere Laufzeitzustände zu sichern/wieder herzustellen
  - z.B. die Inhalte der von einer Koroutine benutzten Arbeitsregister

- auf den **Programmzähler** an ausgewiesenen Stellen zugreifen können

Elementaroperation: *resume*

Unterbrechung und Fortsetzung von Koroutinenausführungen

1. Prozedurale Abstraktion von der *resume*-Implementierung

- die Implementierung von *resume* als Prozedur auslegen:
  - Eingabe $\mapsto$ Adresse der fortzusetzenden Koroutine
  - Ausgabe $\mapsto$ Adresse der unterbrochenen Koroutine
- ein Koroutinenwechsel verläuft damit über einen Prozeduraufruf

2. Herleitung der Fortsetzungsadresse einer Koroutine

- bei jedem Prozeduraufruf wird die Rücksprungadresse hinterlegt:
  - (a) bei CISC indirekt über den Stapelzeiger (engl. *stack pointer*)
  - (b) bei RISC direkt in einem Verweisregister (engl. *link register*)
- die Rücksprungadresse von *resume* ist damit eine Fortsetzungsadresse

3. Fortsetzung einer Koroutine

- die Fortsetzungadresse in das **Programmzählerregister** übertragen

Koroutinenwechsel aus Benutzersicht

Ebene 5: C

```
#include "luxe/coroutine.h"

coroutine_t next, last;

last = cor_resume(next);
```

`next` Koroutinen-Fortsetzungsadresse
- wohin *resume* geht

`last` Koroutinen-Fortsetzungsadresse
- von woher *resume* kommt

Ebene 4: ASM (x86)

```
pushl next        # pass continuation address of next coroutine
call  cor_resume  # perform coroutine switch
movl  %eax, last  # save continuation address of last coroutine
```

Funktion des Maschinenbefehls `call cor_resume`:

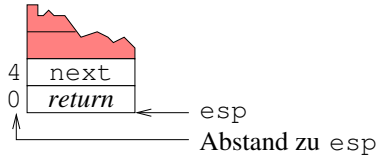
1. Aufruf der Prozedur zum Koroutinenwechsel
2. Generierung der Fortsetzungsadresse der laufenden Koroutine
   - repräsentiert durch die Rücksprungadresse der Prozedur `cor_resume`
   - diese verweist auf den `call` folgenden Maschinenbefehl `movl...`

Koroutinenwechsel aus Systemsicht

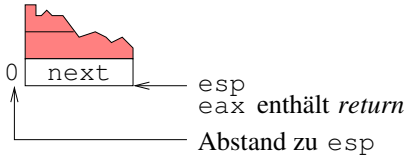
Ebene 4: ASM (x86)

```
cor_resume:
  popl %eax     # remove return address from top of stack
  jmp  *(%esp)  # continue at address given by actual parameter
```

Stapelaufbau nach Einsprung



Stapelaufbau vor Weitersprung



Rücksprung aus Prozedur `cor_resume`

- geht nicht zurück zur Prozedur, die den aktuellen Aufruf getätigt hat
- sondern zu einer Prozedur, die `cor_resume` irgendwann früher aufrief

Anlauf einer Koroutine: Inkarnation

Koroutinen fehlt die Aufrufhierarchie:

- sie werden nicht aufgerufen, um mit der Ausführung zu beginnen
- stattdessen wird ihre Ausführung immer nur fortgesetzt

Problem: Einrichtung der initialen Fortsetzungsadresse

Optionen:

(a) statische **Anfangsadresse** einer Prozedur

(b) dynamische **Verzweigungsadresse** eines Ausführungsstrangs

zu (a)
- eine Prozedurdeklaration ist Referenz für die Anfangsadresse
- die **Einrichtungsfunktion** führt zur Koroutineninkarnation
- Bereitstellung einer weiteren Elementaroperation: *invoke*

zu (b)
- eine Prozedurinkarnation gabelt sich in zwei Ausführungen
- Ausgabe der **Gabelungsfunktion** ist die Fortsetzungsadresse
- Bereitstellung einer weiteren Elementaroperation: *launch*

Elementaroperation: *invoke*, Koroutineninkarnation anlegen

Ergänzung zur Option (a) zur Einrichtung einer Fortsetzungsadresse

Ebene $_4$: ASM (x86)

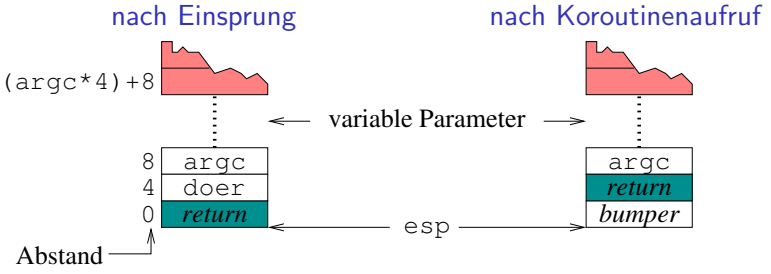
```
cor_invoke:
  popl  %eax           # remove return address to caller
  movl  (%esp), %ecx   # grab coroutine start address
  movl  %eax, (%esp)   # return address becomes first parameter
  call  *%ecx          # coroutine invocation: should not return!
  pushl %eax           # assume return code, pass to interceptor
bumper:
  call  *cor_vector    # unexpected return: switch to interceptor
  jmp   bumper         # catch get lost coroutine...
```

Deklaration einer Koroutinenprozedur

```
void dingus(coroutine_t doer, unsigned int argc, ...)
```

- die Fortsetzungsadresse des „Schöpfers“ wird in `doer` empfangen
- die Anzahl variabler aktueller Parameter wird `argc` aufnehmen

Elementaroperation: *invoke*, Laufzeitstapel



- der erste aktuelle Aufruferparameter (`doer`) trägt die Startadresse der Koroutine (d.h., ihre initiale Fortsetzungsadresse)
- die Rücksprungadresse des Aufrufers (d.h., seine Fortsetzungsadresse) wird der Koroutine als erster aktueller Parameter übergeben
- die Rücksprungsadresse der Koroutine verweist auf einen „Prellbock“ (engl. *bumper*), um ihren eventuellen Rücksprung abzufangen

Elementaroperation: *invoke*, Verwendungsmuster

Ebene $_5$: C

```
int niam(coroutine_t doer, unsigned int argc, int foo, char *bar) {
    doer = cor_resume(doer);  /* continue creator of coroutine */
    ...
    return foo;               /* causes lost coroutine: catch it... */
}
```

```
void alert (int foo) {        /* come here for unexpected return */
    exit(foo);                /* commit suicide... */
}
```

```
main() {
    coroutine_t last;

    cor_vector = alert;       /* interceptor for lost coroutine */

    last = cor_invoke(niam, 2, 42, "4711");
    ...
}
```

Koroutine `niam()` wurde bewusst als „*function returning int*“ deklariert, obwohl sie kein Ergebnis liefern dürfte. Die Warnung des Kompilieres weist in dem Fall darauf hin, dass Koroutinen mangels Aufrufgeschichte nicht zurückkehren sollten. Hier wird der Wert von `foo` (also 42) schließlich an `alert()` übergeben.

Elementaroperation: *launch*, Prozedurinkarnation gabeln

Ergänzung zur Option (b) zur Einrichtung einer Fortsetzungsadresse

Ebene $_{5/4}$: C/ASM (x86)

```
#include "luxe/coroutine.h"

coroutine_t cor_launch (coroutine_t *this) {
    coroutine_t back;

    asm volatile ("movl (%%esp),%0" : "=r" (back));

    *this = back;  /* setup continuation address */
    return 0;      /* indicate creator return */
}
```

- Schalter `-fomit-frame-pointer` vorausgesetzt
- Abstand von `esp` zur Rücksprungadresse ist Null

- `gcc -Os -fomit-frame-pointer -S` erzeugt:

```
cor_launch:
  movl (%esp),%edx
  movl 4(%esp), %eax
  movl %edx, (%eax)
  xorl %eax, %eax
  ret
```

Elementaroperation: *launch*, Verwendungsmuster

Ebene $_5$: C

```
main() {
    coroutine_t niam, last;

    if (last = cor_launch(&niam)) {  /* coroutine comes here */
        for (;;) {                   /* never return! */
            ...
            last = cor_resume(last); /* suspend execution */
        }
    } else {                         /* creator comes here */
        last = cor_resume(niam);     /* activate new coroutine */
        ...
    }
    exit(0);
}
```

- die Operation ist `fork(1)` nicht unähnlich in der Verwendung
- allerdings: `cor_launch()` dupliziert nur den Programmzählerwert

Elementaroperation: *launch*, Rückkehrverhalten

launch kehrt (mindestens) zweimal zurück

1. Rückkehr zum aufrufenden Ausführungsstrang
   - normale Beendigung der Prozedurinkarnation von *launch*
   - Rückgabewert von *launch* ist Null, generiert vom Aufrufer selbst
2. Anlauf der gegabelten Koroutineninkarnation
   - ausgelöst durch das erste *resume* zur erzeugten Koroutine
   - Rückgabewert von *launch* ist der Rückgabewert von *resume*
     - dieser ist immer ungleich Null, nämlich eine Fortsetzungsadresse
3. und ggf. mehr: Wiederanlauf der gegabelten Koroutineninkarnation

Koroutinenabspaltung meint Duplikation des Programmzählerwertes:

- *launch* weist die eigene Rücksprungadresse dem Ausgabeparameter zu
- diese kann beliebig oft als Eingabewert für *resume* verwendet werden
- jedes derart parametrierte *resume* führt zur Rückkehr aus *launch*

Deklaration der Elementaroperationen

Ebene $_5$: C, `coroutine.h`

```
#include "luxe/annunciator.h"

typedef void (*coroutine_t)();

extern annunciator_t cor_vector;

extern coroutine_t cor_resume (coroutine_t, ...);
extern coroutine_t cor_invoke (coroutine_t, unsigned int, ...);
extern coroutine_t cor_launch (coroutine_t *);
```

```
typedef void (*annunciator_t)(int, ...);
```

Bei initialer Aktivierung einer als Prozedur deklarierter Koroutine mittels resume, kann dieser eine variable Anzahl aktueller Parameter (Ellipsis ...) übertragen werden:

- *dies setzt aber einen gemeinsamen Stapel für die Koroutinen voraus!*

Laufzeitzustand (Kontext) inkarnierter Koroutinen

Prozeduren ähnlich bestimmt sich der Zustand einer Koroutineninkarnation durch deren **Koroutinendefinition**:

unbedingt enthalten ist ein Platzhalter für den Programmzähler

bedingt ist dieser Minimalzustand um weitere Elemente anzureichern

- lokale Daten (allg. Programmvariablen)
- gehalten in Prozessorregistern oder im Arbeitsspeicher
- Invarianz solcher Daten ist bislang nicht sichergestellt !!!

Forderung nach **Invarianz** auch lokaler Daten bedingt die Sicherung und Wiederherstellung des erweiterten Zustands einer Koroutine

- das erfordert die Einrichtung von Stapelspeicher für diese Koroutinen
- benutzt durch ein „erweitertes *resume*“ für einen **Kontextwechsel**

Minimale Koroutinenerweiterung (s. Anhang)

(a) Koroutinen einen eigenen **Stapelzeiger** (engl. *stack pointer*) geben
- je nach Stoßart adressiert der SP unterschiedliche Zustandsdaten:
  - CPU-Stoß: den der Koroutine zugeordnete lokale Datenraum
  - E/A-Stoß: dann zusätzlich noch die gesicherten Arbeitsregister und PC
- nur während eines E/A-Stoßes bleibt der Koroutinenzustand invariant

(b) **Arbeitsregister** beim Koroutinenwechsel sichern und wiederherstellen
- je nach der *resume* zugeordneten Abstraktionsebene bedeutet dies:
  - Ebene 3: alle Arbeitsregister sichern/wiederherstellen
  - Ebene 4: nur die nichtflüchtigen Register sichern/wiederherstellen
  - Ebene 5: nur der Teil davon, den der Aufrufkontext von *resume* belegt
- *resume* folgt damit zwei grundsätzlich verschiedenen Konzepten:
  - i Aufruf, sichern, SP umschalten, wiederherstellen, Rücksprung (Eb. 3/4)
  - ii sichern, Aufruf, SP umschalten, Rücksprung, wiederherstellen (Eb. 5)
- Betriebssysteme realisieren Optionen (i), Kompilierer ggf. Option (ii)

- die Erweiterung ist Implementierungsgrundlage von **Programmfäden**

Gliederung

Koroutinen „mechanisieren“ Programmfäden

Technisches Detail zum Multiplexen der CPU zwischen Prozessen

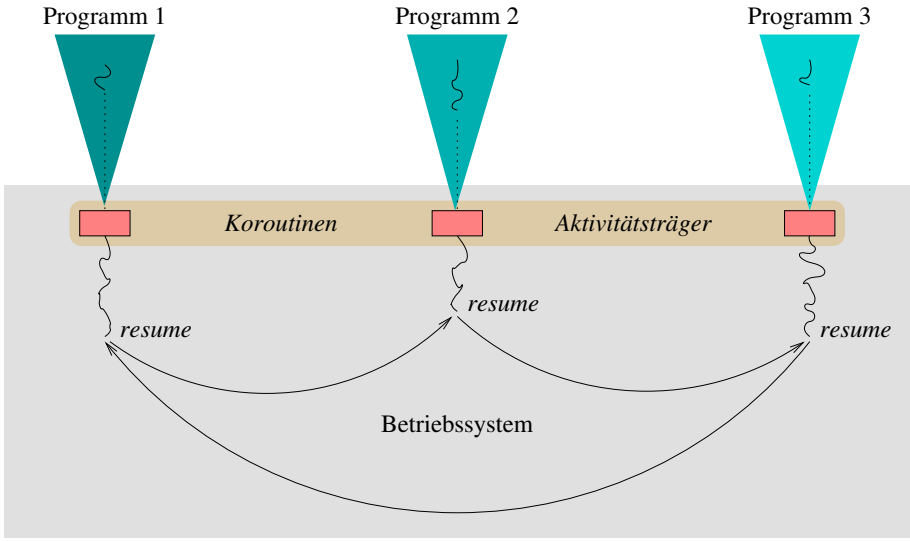
Mehrprogrammbetrieb basiert auf Koroutinen des Betriebssystems
- pro auszuführendes Programm gibt es (wenigestens) eine Koroutine
  - ggf. für jeden Programmfaden $\rightsquigarrow$ leichtgewichtiger Prozess
- ist eine Koroutine aktiv, so ist das ihr zugeordnete Programm aktiv
  - der durch die Koroutine implementierte Programmfaden ist aktiv
- ein anderes Programm ausführen $\mapsto$ Koroutine wechseln

Koroutinen sind (autonome) **Aktivitätsträger** des Betriebssystems
- ihr Aktivierungskontext ist globale Variable des Betriebssystems
- für jede Prozessinkarnation gibt es eine solche Betriebssystemvariable

- ein Betriebssystem ist Inbegriff für das nicht-sequentielle Programm

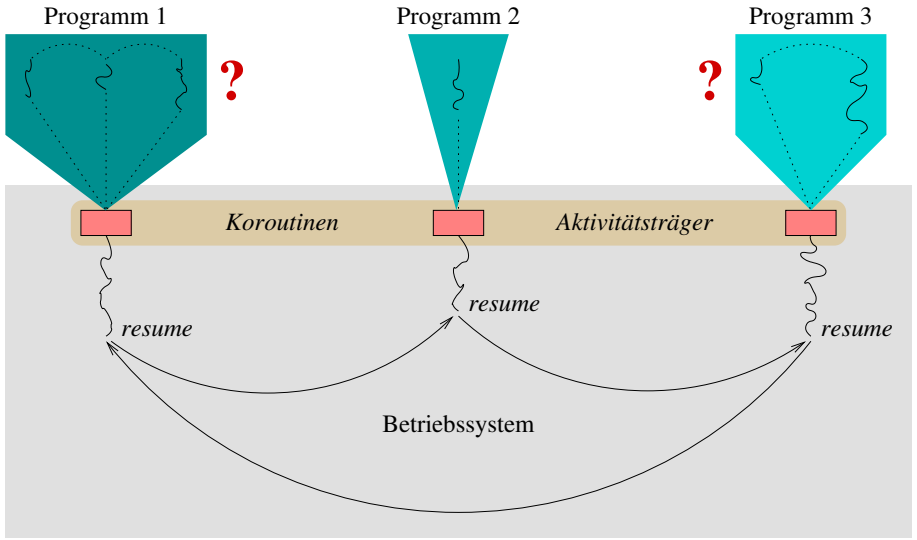
Verarbeitung sequentieller Programme

Koroutine als abstrakter Prozessor — Bestandteil des Betriebssystems



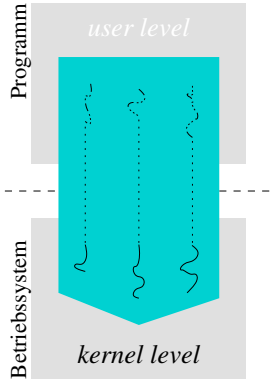
Verarbeitung nicht-sequentieller Programme

Multiplexen eines abstrakten Prozessors



Fäden der Kernebene

Klassische Variante von Mehrprozessbetrieb



Prozessinkarnationen als Ebene $_3$-Konzept basieren auf **Kernfäden** (engl. *kernel-level threads*)

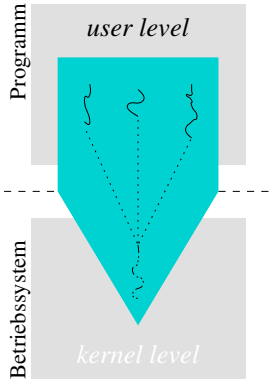
- egal, ob die Maschinenprogramme ein- oder mehrfädig ausgelegt sind
  - jeder Anwendungsfaden ist Kernfaden
  - nicht jeder Kernfaden ist Anwendungsfaden
- Kernfäden $\neq$ Prozessinkarnationen der Ebene $_3$
  - Maschinenprogramme verwenden Fäden
  - im Programmiermodell des BS manifestiert
- Fäden realisiert durch Ebene $_2$-Programme

Einplanung und Einlastung der (leichtgewichtigen) Anwendungsprozesse sind Funktionen des Betriebssystem(kern)s

- Erzeugung, Koordination, Zerstörung von Fäden $\mapsto$ Systemaufrufe

Fäden der Benutzerebene

Ergänzung oder Alternative zu Kernfäden



Prozessinkarnationen als Ebene $_{2/3}$-Konzept sind **Benutzerfäden** (engl. *user-level threads*)

- virtuelle Prozessoren bewirken die Ausführung der (mehrfädigen) Maschinenprogramme
  - Benutzerfäden = Koroutinen $\mapsto$ Ebene $_2$
  - 1 Kernfaden f. ≥ 2 Benutzerfäden $\mapsto$ Ebene $_3$
- der Kern stellt ggf. Planeransteuerungen (engl. *scheduler activations* [1]) bereit
  - zur Propagation von Einplanungsereignissen
- Fäden realisiert durch Ebene $_{2/3}$-Programme

Einplanung und Einlastung der (federgewichtigen) Anwendungsprozesse sind keine Funktionen des Betriebssystem(kern)s

- Erzeugung, Koordination, Zerstörung von Fäden $\mapsto$ Prozeduraufrufe

Prozesskontrollblock (engl. *process control block*, PCB)

Datenstruktur zur Verwaltung von Prozessinstanzen

Kopf eines Datenstrukturgeflechts zur Beschreibung und Verwaltung einer Prozessinkarnation und Steuerung eines Prozesses

- oft auch als **Prozessdeskriptor** (PD) bezeichnet
  - UNIX Jargon: *proc structure* (von „`struct proc`“)
- ein abstrakter Datentyp (ADT) des Betriebssystem(kern)s

Softwarebetriebsmittel zur Verwaltung von Programmausführungen

- jeder Faden wird durch ein Exemplar vom Typ „PD“ repräsentiert
  - Kernfaden Variable des Betriebssystems
  - Benutzerfaden Variable des Anwendungsprogramms
- die Exemplaranzahl ist statisch (Systemkonstante) oder dynamisch

Objekt, das mit einer **Prozessidentifikation** (PID) assoziiert ist

- eine für die gesamte Lebensdauer des Prozesses gültige Bindung

Aspekte der Prozessauslegung

Verwaltungseinheit einer Prozessinkarnation

Dreh- und Angelpunkt, der **prozessbezogene Betriebsmittel** bündelt

- Speicher- und, ggf., Adressraumbelegung *1
  - Text-, Daten-, Stapelsegmente (*code, data, stack*)
- Dateideskriptoren und -köpfe (*inode*) *2
  - {Zwischenspeicher,Puffer}deskriptoren, Datenblöcke
- Datei, die das vom Prozess ausgeführte Programm repräsentiert *3

Datenstruktur, die Prozess- und Prozessorzustände beschreibt

- Laufzeitkontext des zugeordneten Programmfadens/Aktivitätsträgers
- gegenwärtiger Abfertigungszustand (*Scheduling*-Informationen) *4
- anstehende Ereignisse bzw. erwartete Ereignisse *5
- Benutzerzuordnung und -rechte *6

Aspekte der Prozessauslegung: Optionale Merkmale

∗ Auslegung des PD ist höchst abhängig von Betriebsart und -zweck:

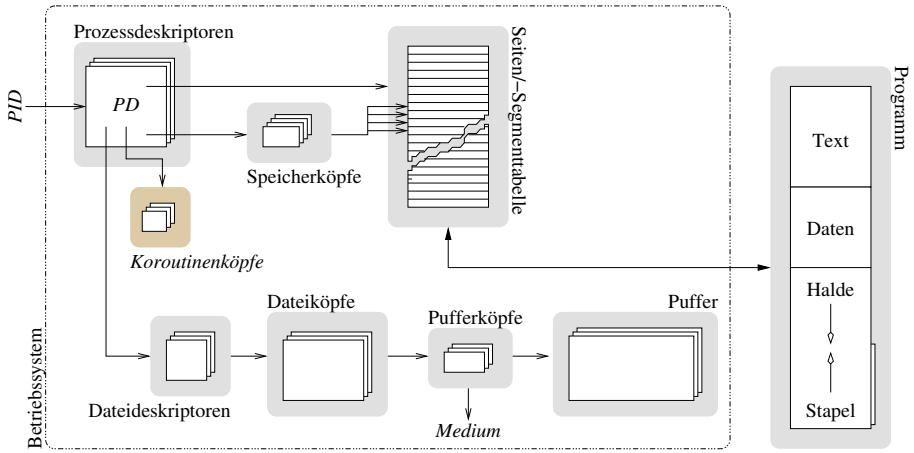
1. Adressraumdeskriptoren sind nur notwendig in Anwendungsfällen, die eine Adressraumisolation erfordern
2. für ein Sensor-/Aktorsystem haben Dateideskriptoren/-köpfe wenig Bedeutung
3. in ROM-basierten Systemen durchlaufen die Prozesse oft immer nur ein und dasselbe Programm
4. bei statischer Prozesseinplanung ist die Buchführung von Abfertigungszuständen verzichtbar
5. Ereignisverwaltung fällt nur an bei ereignisgesteuerten und/oder verdrängend arbeitenden Systemen
6. in Einbenutzersystemen ist es wenig sinnvoll, prozessbezogene Benutzerrechte verwalten zu wollen

Problemspezifische Datenstruktur

- Festlegung auf eine Ausprägung grenzt Einsatzgebiete unnötig aus

Generische Datenstruktur

Logische Sicht eines Geflechts abstrakter Datentypen



einfädiger Prozess $\mapsto$ 1 Koroutinenkopf

mehrfädiger Prozess $\mapsto$ $N > 1$ Koroutinenköpfe

Instanzvariable des Typs „Prozessdeskriptor“

Buchführung über den aktuell laufenden Prozess

Zeiger auf den Kontrollblock des laufenden Prozesses: **Prozesszeiger**

- für jeden Prozessor(kern)[1] ist solch ein Zeiger bereitzustellen
- innerhalb des Betriebssystems ist somit jederzeit bekannt, welcher Prozess, Faden, welche Koroutine die CPU „besitzt“
- wichtige Funktion der Einlastung ist es, den Zeiger zu aktualisieren

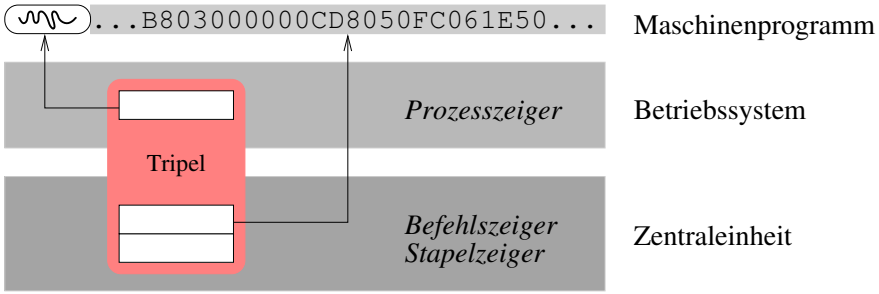
Laufgefahr: Verdrängend arbeitende Prozesseinplanung

- Einlastung meint zweierlei: Prozesszeiger- *und* Fadenumschaltung
- Verdrängung des dazwischen laufenden Fadens ist kritisch (S. 38)
- der Programmabschnitt dazu ist ein typischer kritischer Abschnitt
- dieser Abschnitt ist in Abhängigkeit von der Betriebsart zu schützen
  - (a) Verdrängung zeitweise unterbinden (pessimistischer Ansatz, leicht)
  - (b) gleichzeitige Prozesse tolerieren (optimistischer Ansatz, schwer)

[1] Bei mehr-/vielkerniger (engl. *multi-/many-core*) CPU ist ein Zeiger nicht genug.

Tripel aus Prozess-, Stapel- und Befehlszeiger

Logische Sicht:

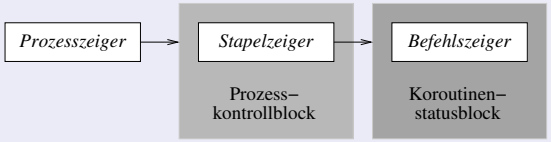


Physische Sicht

Prozesswechsel impliziert, ein Zeigerpaar umzusetzen:

(a) Prozesszeiger

(b) Stapelzeiger



Abfertigung eines Programmfadens

```
void pd_board(thread_t *next) {
    cothread_t last = cot_resume(next->cot);
    pd_being()->cot = last; /* save SP for last thread */
    pd_check(next);         /* define thread pointer */
}
```

```
_pd_board:
    pushl %ebx
    subl  $24, %esp
    movl  32(%esp), %ebx
    movl  (%ebx), %eax
    movl  %eax, (%esp)
    call  _cot_resume
    movl  _life, %edx
    movl  %eax, (%edx)
    movl  %ebx, _life
    addl  $24, %esp
    popl  %ebx
    ret
```

being liefert den Zeiger auf den aktuellen Prozess

check aktualisiert den Prozesszeiger

resume wechselt Koroutinen erw. Zustands (S. 25, i)

Laufgefahr: Überlappung zwischen *resume* und *check*

Annahme: Verdrängung des laufenden, Einlastung eines anderen Fadens

- der durch *being* identifizierte 1. Faden führte *resume* durch
- Koroutine des 2. Fadens läuft, *being* verweist noch auf 1. Faden
- jetzt erfolgt die Verdrängung: Koroutine des 3. Fadens läuft
- $SP_{\texttt{last}}$ (2. Faden) wird ggf. in PD_{being} (1. Faden) gesichert ☹

Gliederung

Resümee

- **Koroutinen** konkretisieren Prozesse, realisieren Prozessinstanzen
  - die Gewichtsklasse von Prozessen spielt eine untergeordnete Rolle
    - feder-, leicht-, schwergewichtige Prozesse basieren auf Koroutinen
  - ihr Aktivierungskontext überdauert Phasen der Inaktivität
    - gesichert („eingefroren“) im jeder Koroutine eigenen Stapelspeicher
- **Programmfäden** (engl. *threads*) sind durch Koroutinen repräsentiert
  - unterschieden in zwei Fadenarten, je nach Ebene der Abstraktion:
    Kernfaden implementiert durch Ebene $_2$-Programme
    Benutzerfaden implementiert durch Ebene $_{2/3}$-Programme
  - Einlastung eines Fadens führt einen Koroutinenwechsel nach sich
- der **Prozessdeskriptor** ist Objekt der Buchführung über Prozesse
  - Datenstruktur zur Verwaltung von Prozess- und Prozessorzuständen
    - insbesondere des Aktivierungskontextes der Koroutine eines Prozesses
  - Softwarebetriebsmittel zur Beschreibung einer Programmausführung
- jeder Prozessor(kern) verfügt über einen eigenen **Prozesszeiger**

Literaturverzeichnis

[1] Anderson, T. E. ; Bershad, B. N. ; Lazowska, E. D. ; Levy, H. M.:
Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.
In: *Proceedings of the Thirteenth ACM Symposium on Operating System Principles (SOSP 1991), October 13–16, 1991, Pacific Grove, California, USA*, ACM Press, 1991, S. 95–109

[2] Baker, T. P.:
Stack-Based Scheduling of Realtime Processes.
In: *Real-Time Systems* 3 (1991), Nr. 1, S. 67–99

[3] Conway, M. E.:
Design of a Separable Transition-Diagram Compiler.
In: *Communications of the ACM* 6 (1963), Jul., Nr. 7, S. 396–408

[4] Digital Equipment Corporation (Hrsg.):
PDP-11/40 Processor Handbook.
Maynard, MA, USA: Digital Equipment Corporation, 1972.
(D-09-30)

Literaturverzeichnis (Forts.)

[5] Herrtwich, R. G. ; Hommel, G. :
Kooperation und Konkurrenz — Nebenläufige, verteilte und echtzeitabhängige Programmsysteme.
Springer-Verlag, 1989. –
ISBN 3–540–51701–4

[6] Knuth, D. E.:
The Art of Computer Programming. Bd. 1: *Fundamental Algorithms*.
Reading, MA, USA : Addison-Wesley, 1973

[7] Tauchen, M. O. ; Prokopetz, J. ; Humpe, I. ; Humpe, A. :
CODO... düse im Sauseschritt.
DÖF, 1983

Gliederung

Wiederaufnahme des Falls `coroutine.h`

Frage Ist damit die Implementierung einer Prozessinkarnation gegeben?

Antwort Im Prinzip ja, aber...

- die gemeinsame Benutzung desselben Laufzeitstapels durch mehrere Prozessinkarnationen ist nur bedingt möglich
  1. die Prozesse dürfen nicht blockieren
  2. die Einplanung muss die Stapelmitbenutzung beachten [2]
- ein Prozess, der blockieren kann, benötigt einen eigenen Laufzeitstapel zur Sicherung seines Kontextes
  - die Fortsetzungsadresse einer Koroutine
  - je nach Prozess ggf. den kompletten Prozessorzustand
- Aktivierung der diesbezüglichen Prozessinkarnation bedingt den Wechsel des Laufzeitstapels — und ggf. mehr...

Koroutinen Ausführungsdomänen zuweisen

1. mit eigenem Laufzeitstapel im gemeinsamen Adressraum versehen
2. erweiterten Zustand in Phasen von Inaktivität invariant halten

Koroutine ↦ Laufzeitstapel

Optionales Merkmal einer Prozessverwaltung

```
cod_resume:
  movl %esp, %eax    # return SP
  movl 4(%esp), %esp # switch SP
  ret
```

```
cod_launch:
  movl (%esp), %eax   # grab PC
  movl 4(%esp), %ecx  # grab param.
  movl (%ecx), %edx   # grab new SP
  leal -8(%edx), %edx # update and
                      # skip param.
  movl %eax, (%edx)   # copy PC
  movl %edx, (%ecx)   # send new SP
  xorl %eax, %eax     # return 0
  ret
```

resume ⇝ Stapelumschaltung

- liefert alten Stapelzeiger
  - *ref* Fortsetzungsadresse
- definiert neuen Stapelzeiger
- der Rücksprung (`ret`) holt die Rücksprungadresse vom neuen Stapel
- Fortsetzung an der einem *resume*/*launch*-Aufruf folgenden Adresse

launch „vererbt“ ihren Aktivierungsblock (den PC) an die neue Koroutine

- als wenn die neue Koroutine, die Funktion selbst aufgerufen hätte
- genauer: als wenn sie ein *resume* bereits ausgeführt hätte

Koroutine ↦ Koroutinenprozedur

Ebene $5_{/4}$: C bzw. x86

```
extern void cod_bumper();

codomain_t cod_invoke (codomain_t this, coroutine_t code, unsigned int argc, ...) {
    word_t *from, *to;
    coroutine_t sink = code;          /* we need this because of (#) below */

    asm volatile (
        "std\n\t"                     /* decrement from and to */
        "rep movsd"                   /* copy parameter list */
        : "=D" (to), "=S" (from)
        :  "D" ((word_t*)this - 1),  "S" ((word_t*)&argc + argc),
           "c" (argc + 1)
        : "memory");

    *(--to) = (word_t)cod_bumper;     /* stop coroutine upon return */

    asm volatile (
        "pushl $1f\n\t"               /* generate own resumption address (#) */
        "movl %%esp, %0\n\t"          /* pass own coroutine domain pointer */
        "movl %1, %%esp\n\t"          /* switch coroutine domain */
        "jmp *%2\n"                   /* activate new coroutine (#) */
        "1:"                          /* come upon resumption... */
        : "=g" (to[1])
        : "g" (to), "r" (sink)
        : "memory");

    return (codomain_t)to;
}
```

```
#include "luxe/codomain.h"

void cod_bumper() {
    (*cod_vector)(-1);
}
```


Bereitstellung des Laufzeitstapels

Platzhalter der Koroutinendomäne

```
#include "luxe/codomain.h"

#define STACKSIZE 42*42

char codo[STACKSIZE];

codomain_t niam = COD_PURIFY(codo, sizeof(codo));
```

`codo` [7] ist Platzhalter für den Laufzeitstapel der Koroutineninkarnation

- ein linear zusammenhängender Bereich von `STACKSIZE` Bytes

`niam` ist Platzhalter für den Stapelzeiger der Koroutineninkarnation

- *purify* richtet den initialen Stapelzeigerwert aus
- die Operation hängt vom Stapelkonzept des Prozessor ab

Deklaration der Elementaroperationen von `codomain`

Ebene 5: C, `codomain.h`

```
#include "luxe/coroutine.h"
#include "luxe/machine/codomain.h"

typedef coroutine_t* codomain_t;

extern annunciator_t cod_vector;

extern codomain_t cod_resume (codomain_t);
extern codomain_t cod_invoke (codomain_t, coroutine_t, unsigned int, ...);
extern codomain_t cod_launch (codomain_t *);
```

- Verwendungsmuster der Operationen entsprechend Beispiel S. 21
- Vorbedingung zum *launch*-Aufruf: eingerichteter Laufzeitstapel (S. 47)

Ebene $5_{/2}$: C (x86), `machine/codomain.h`

```
#define COD_PURIFY(codo, size) (codomain_t)(codo + size)
```

- x86-Prozessoren lassen den Stapel „von oben nach unten“ wachsen
- der Stapelzeiger verweist auf das zuletzt abgelegte Element im Stapel

Erweiterter Koroutinenzustand: Prozessorstatus

Sicherung und Wiederherstellung

Unterbrechung der Koroutinenausführung $\mapsto$ **Programmunterbrechung**

- eine eigenständige, inaktive Koroutine ist davon abhängig, dass ihr Prozessorstatus invariant bleibt (S. 25)
- Koroutinenwechsel und -einrichtung sind fallspezifisch zu erweitern

Koroutinenwechsel $\rightsquigarrow$ Delta zu `cod_resume()`

- Prozessorstatus der abgebenden Koroutine auf den Stapel ablegen
- Prozessorstatus der annehmenden Koroutine vom Stapel nehmen

Koroutineneinrichtung $\rightsquigarrow$ Delta zu `cod_launch()`

- Prozessorstatus der erzeugenden Koroutine auf den Stapel der sich in Einrichtung befindlichen, neuen Koroutine ablegen

Prozessorstatus invariant halten und vererben

```
cot_resume:
  __PUSH                  # save
  movl %esp, %eax         # return SP
  movl __N+4(%esp), %esp  # switch SP
  __PULL                  # restore
  ret
```

`__N` Umfang in Bytes
`__PUSH` Sicherung
`__PULL` Wiederherstellung
`__DUMP` Übertragung/Vererbung

```
cot_launch:
  movl (%esp), %eax          # grab return address from stack
  movl 4(%esp), %ecx         # grab parameter: domain pointer reference
  movl (%ecx), %edx          # extract initial stack pointer and
  leal -(__N+8)(%edx), %edx  # update it by size of processor status
                             # skip parameter placeholder: leave void
  movl %eax, __N(%edx)       # copy return address to new stack
  __DUMP(%edx)               # copy processor status to new stack
  movl %edx, (%ecx)          # send pointer thereon back to caller
  xorl %eax, %eax            # return 0
  ret
```


Prozessorstatus verwalten

Sichern

```
#define __PUSH \
  pushl %ebp;  \
  pushl %edi;  \
  pushl %esi;  \
  pushl %ebx
```

Wiederherstellen

```
#define __PULL \
  popl %ebx;   \
  popl %esi;   \
  popl %edi;   \
  popl %ebp
```

Abladen

```
#define __DUMP(codo)   \
  movl %ebp, 12(codo); \
  movl %edi, 8(codo);  \
  movl %esi, 4(codo);  \
  movl %ebx, 0(codo)
```

Nichtflüchtige Arbeitsregister (x86): `ebp`, `edi`, `esi`, `ebx`

- Ann.: Prozedur *resume* wird von Programmen der Ebene $_5$ aufgerufen
- der Kompilierer dieser Ebene gibt **Prozedurkonventionen** vor
- im gegebenen Fall (Ebene der Programmiersprache C, `gcc(1)`):
  - flüchtige Register sind innerhalb von Prozeduren frei verwendbar
  - ihre Inhalte brauchen nicht gesichert und wiederhergestellt zu werden
  - nach Rücksprung aus der Prozedur sind diese Inhalte undefiniert

- nach Konvention gelten Arbeitsregister `eax`, `ecx` und `edx` als flüchtig

Deklaration der Elementaroperationen von `cothread`

Ebene $_{5/4}$: C, `cothread.h`

```
#include "luxe/codomain.h"
#include "luxe/machine/things.h"

struct cothread {
    word_t psw[NPSW]; /* processor status word save area */
    coroutine_t cor;  /* coroutine resumption address */
};

typedef struct cothread* cothread_t;

extern annunciator_t cot_vector;

extern cothread_t cot_resume (cothread_t);
extern cothread_t cot_invoke (cothread_t, coroutine_t, unsigned int, ...);
extern cothread_t cot_launch (cothread_t *);
```

- Verwendungsmuster und Vorbedingung entsprechend `codomain.h`
- `NPSW` gibt die Anzahl der zu sichernden Arbeitsregister vor (S. 25, i)
  - `__N` (S. 50) ergibt sich dann aus `NPSW * sizeof(word_t)`

Selbstversuch • Implementierung von `cot_invoke()`; Hinweis: `cod_invoke()`

Koroutine *considered harmful*? Ja und nein!

Prozessinkarnationen sind auf unterster, technischer Ebene Koroutinen...

- so ist das Koroutinenkonzept in Betriebssystemen unerlässlich

...eine echte Systemprogrammiersprache hätte Koroutinen im Angebot

- weder C noch C++ kennen vergleichbare Sprachkonstrukte
  - `setjmp()` und `longjmp()` sind Bibliotheksfunktionen
  - damit kann man mit einigem Geschick Koroutinen nachbilden
- von Java ganz zu schweigen: Fäden von Java sind keine Koroutinen
  - darüberhinaus sind diese Fäden für Betriebssystembelange ungeeignet
  - die JVM nimmt diesbezüglich zuviel Entwurfsentscheidungen vorweg

Behauptung: Echte Systemprogrammiersprachen gibt es nicht mehr

- daher sind Koroutinen händisch in Assembliersprache bereitzustellen
- gleichwohl bleiben sie ein Programmiersprachenkonzept der Ebene 5