Betriebssystemtechnik

Fadenimplementierung: Minimale Basis

Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

22. Mai 2012

Gliederung

Gleichzeitige Prozesse

Prozesse, die sich zeitlich überlappen — bedingt durch:

1. asynchrone Programmunterbrechungen
   - verschachtelte Ausführung *desselben* Programmfadens
2. Multiplexverfahren zum mehrfädigen Betrieb eines Prozessor(kern)s
   - pseudo- oder quasiparallele Ausführung *verschiedener* Programmfäden
3. Multiprozessor- beziehungsweise Mehr- oder Vielkernbetrieb
   - echt parallele Ausführung *verschiedener* Programmfäden

Programmfaden (kurz: Faden, engl. *thread*)

- ein **Ausführungskontext** eines (ein-/mehrfädigen) Programms
- der **Aktivitätsträger** eines Programms, **Ausführungsstrang**
- ein einzelner **sequentieller Kontrollfluss** in einem Prozess
- die **Einheit** der Prozesskoordinierung, -einplanung und -einlastung

Autonomer Kontrollfluss in einem Programm

Programmfäden werden durch **Koroutinen** konkretisiert, um Inkarnationen von (gleichzeitigen) Prozessen zu schaffen

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

Koroutinen sind beständig „zustandsbehaftete Prozeduren“

Inaktivität einer Koroutine bedeutet, einen *Fixpunkt* vom **Laufzeitzustand** ihres Prozesses zum Wiederanlauf gezogen zu haben:

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

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

Fixpunkt des Laufzeitzustands eines Fadens

Unterbrechung und Fortsetzung von Koroutinen sind Spezialfälle des Ansprungs von Unterroutinen (engl. *jump to subroutine*, JSR)

PDP-11/40 als Beispiel für eine elementare Unterstützung von Koroutinen in Hardware

> *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." [**?**, S. 4-58/59]*

Zusammensetzung und Umfang vom **Fixpunkt** richtet sich nach den Eigenschaften der „virtuellen Maschine“ [**?**] eines Fadens

Programmzähler	• *kooperative* gleichzeitige Prozesse *im Stapel*
+ Stapelzeiger	• *kooperative* gleichzeitige Prozesse
+ Arbeitsregister	• gleichzeitige Prozesse

Gliederung

Buchführung über Fortsetzungspunkte

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

Beachte: Programmfamilie, schrittweiser Systementwurf

- die Sicherung weiterer Laufzeitzustände wird vorerst zurückgestellt

Varianten zur Fixpunktbildung von Programmfortsetzungen

Sicherung und Wiederherstellung des Programmzählers

Art der Auslegung der Operationen zur Einrichtung, Unterbrechung sowie Fortsetzung von Koroutinen ist Ansatz für eine **Variantenbildung**

Unterprogrammanordnung

- automatische Bildung der Programmfortsetzung: Rücksprungadresse
- Basis für „fliegengewichtige Fäden“, konventioneller Ansatz
  - wie beispielsweise in SP [**?**] und BS [**?**] gezeigt
  - zu BST, vgl. Anhang S. 35*ff.*

Reihenanordnung: Einbettbare (engl. *in-line*) Unterprogramme

- Bildung der Programmfortsetzung von Hand: Sprungmarke
- Basis für „strohgewichtige Fäden“, unkonventioneller Ansatz
  - exemplarisch hier (S. 12*ff.*) vorgestellt

Funktionale/Prozedurale Abstraktion

Modulschnittstelle: coroutine.h

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

typedef void (*coroutine_t)();

extern annunciator_t cor_vector;

extern coroutine_t cor_launch (coroutine_t *);
extern coroutine_t cor_invoke (coroutine_t, size_t, ...);

extern coroutine_t cor_resume (coroutine_t, ...);
extern void        cor_regain (coroutine_t, coroutine_t *, ...);
```

„Prellbock“ zum Abfangen von Koroutinen: annunciator.h

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

Funktionale/Prozedurale Abstraktion (Forts.)

Initialisierung einer als Prozedur deklarierten Koroutine

Ellipsis (. . .)

- Option zur Angabe zusätzlicher, variabler Parameter
- Deklaration weiterer **formaler Parameter** einer als Koroutinenstellvertreter vorgesehenen Prozedur
  - `void dingus(coroutine_t, size_t, ...);`
- Definition (dazu passender) **aktueller Parameter** beim initialen Koroutinenwechsel
  - *resume*, *regain*, *invoke*

Beachte: Bedingt kostenloser Nebeneffekt

- bei einem von den beteiligten Koroutinen **gemeinsam mitbenutzten Laufzeitstapel**, aber nur im Falle von Stapelmaschinen (z.B. x86)
- nicht zwingend bei Registermaschinen (z.B. PowerPC)
- und sofern das Laufzeitmodell des Übersetzers dazu passt!!!

Gliederung

Reihenanordnung: *resume*

Unterbrechung und Fortsetzung von Koroutinenausführungen

1. Funktionale Abstraktion von der *resume*-Implementierung

- die Implementierung von *resume* als **einbettbare Funktion**:

  Parameter $\mapsto$ Adresse der fortzusetzenden Koroutine

  Wert $\mapsto$ Adresse der unterbrochenen Koroutine

- Koroutinenwechsel verlaufen über **scheinbare Funktionsaufrufe**

2. Herleitung der Fortsetzungsadresse einer Koroutine

- Programmiertechniken der Assembliersprachenebene ausnutzen
  - zur Koroutinenfortsetzung eine **Sprungmarke** (engl. *label*) deklarieren
  - den Wert der Sprungmarke durch **Direktwertadressierung** sichern
- die Sprungmarke in *resume* wird damit zur Fortsetzungsadresse

3. Fortsetzung einer Koroutine

- die Fortsetzungadresse in das **Programmzählerregister** übertragen

Koroutinenwechsel „von außen“ betrachtet

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

Anweisungsfolgen (auf Assembliersprachenebene) innerhalb von *resume*:

1. Adresse der Sprungmarke als Rückgabewert der Funktion aufbereiten
   - der Programmvariablen `last` einen Wert zuweisen
2. Programmverzweigung hin zur nächsten Koroutine
   - den Wert der Programmvariablen `next` als Sprungziel nehmen
3. Deklaration der Sprungmarke als Abschluss der Funktion

Beachte: Festlegung der Fortsetzungsadresse *vor* Laufzeit

- durch Zusammenspiel von Assemblierer und Binder/bindendem Lader

Koroutinenwechsel „von innen“ betrachtet: Funktion

C/ASM (x86)

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

INLINE coroutine_t cor_resume(coroutine_t nxt, ...) {
    coroutine_t slf;

    __asm__ __volatile__(
        "movl $1f, %0\n\t" /* setup coroutine continuation */
        "jmp *%1\n"        /* resume next coroutine */
        "1:"               /* suspended coroutine resumes here */
        : "=&a" (slf)      /* output: "earlyclobber" operand */
        : "r" (nxt));      /* input: continuation address */

    return slf;
}
```

- zur Definition von `INLINE`, siehe Anhang S. 34

Einbettung von *resume*: Übersetztes Programm von S. 13

ASM (x86)

```
  movl next, %edx
#APP
  movl $1f, %eax
  jmp  *%edx
1:
#NO_APP
  movl %eax, last
```

Kommentar (zum expandierten Programmtext)

```
# prolog: supply of input parameter
= begin of inlined text: inserted by gcc
# back up "resume label" value
# resume execution of next coroutine
# "resume label" of this coroutine
= end of inlined text: inserted by gcc
# epilog: save "resume label" value
```

`$1f`
- Direktwertadressierung
- Anweisung zur Erzeugung des Wertes der Sprungmarke
- Ziel ist die nächste Sprungmarke 1, vorwärts (engl. *forward*)

`*`
- indirekte Adressierung
- Dereferenzierung des Zeigers zur fortzusetzenden Koroutine

`1:`
- Deklaration der temporären (internen) Sprungmarke Nummer 1

Reihenanordnung: *regain*, Alternative zu *resume*

Prinzip: Sicherung der Fortsetzungsadresse der laufenden Koroutine durch sie selbst — und nicht der vorherigen durch die nächste (vgl. S. 13):

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

coroutine_t next, self;

cor_regain(next, &self);
```

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

`self` Koroutinen-Fortsetzungsadresse
- wohin *regain* zurückkehrt

Beachte: Koroutinenwechsel der konventionellen Art

- in ihrer eigentlichen programmiersprachlichen Bedeutung liefert eine **Prozedur** kein Berechnungsergebnis, im Gegensatz zur **Funktion**
- *regain* weist die Fortsetzungsadresse der Koroutinenvariablen `self` zu, bevor die Programmausführung bei `next` wieder aufgenommen wird

Koroutinenwechsel „von innen“ betrachtet: Prozedur

C/ASM (x86)

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

INLINE void cor_regain(coroutine_t nxt, coroutine_t *slf, ...) {
    __asm__ __volatile__(
        "movl $1f, %0\n\t" /* setup coroutine continuation */
        "jmp *%1\n"        /* resume next coroutine */
        "1:"               /* suspended coroutine resumes here */
        : "=g" (*slf)      /* output: own continuation address */
        : "r" (nxt)        /* input: continuation address */
        : "memory");       /* clobbers memory */
}
```

Beachte (den „feinen“ Unterschied zur Funktion *resume*, S. 14)

- die beiden Operanden der `movl`-Anweisung zur Erzeugung des Wertes der Sprungmarke liegen im Arbeitsspeicher

Einbettung von *regain*: Übersetztes Programm von S. 16

ASM (x86)

```
  movl next, %eax
#APP
  movl $1f, self
  jmp  *%eax
1:
#NO_APP
```

- der einbettbaren Variante von *resume* sehr ähnlich (vgl. S. 15)
- die Sicherung der eigenen Fortsetzungsadresse geschieht jedoch vor dem Wegsprung
- damit sichert die abgebende Koroutine ihren Wiederaufsetzungspunkt immer selbst

Beachte

- erzwungene Sicherung der Fortsetzungsadresse in den Arbeitsspeicher
  - impliziert durch `&self` zur Parameterübergabe resp. `coroutine_t*`
- Arbeitsregister als Platzhalter scheiden hier aus
  - im Gegensatz zu *resume*, das die zu sichernde Fortsetzungsadresse als Funktionsrückgabewert liefert
  - wodurch sich für *resume* die (mögliche) effizientere Einbettung ergibt
- *regain* erscheint nur auf dem ersten Blick die bessere Variante...

resume S. 13 $\Longleftrightarrow$ *regain* S. 16

Vergleich der Assemblierprotokolle (engl. *assembly listing*)

Maschinenbefehle		*resume*		*regain*	
		#	# Bytes	#	# Bytes
Prolog	**optional**	1	6	1	5
Koroutinenwechsel	**obligatorisch**	2	7	2	12
Epilog	**optional**	1	5	0	0

Beachte

- Prolog und Epilog sind optional, da ihre jeweilige Ausprägung von den Fähigkeiten des Kompilierers (gcc) abhängt
- darüberhinaus sind sie durch den Programmtext bestimmt, in dem die Funktion *resume* bzw. Prozedur *regain* einbettbar ist
- im günstigsten Fall profitiert *resume* durch die Registeradressierung bei der Erzeugung des Wertes der Sprungmarke

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*
  - obligatorische Unterprogrammanordnung: **Aktivierungsblock**

zu (b)
- ein Programmfaden gabelt sich in zwei Ausführungsstränge
- Ausgabe der **Gabelungsfunktion** ist die Fortsetzungsadresse
- Bereitstellung einer weiteren Elementaroperation: *launch*

Unterprogrammanordnung: *invoke*, Koroutineninkarnation

Ergänzung zur Option (a) zur Einrichtung einer Fortsetzungsadresse (S. 20)

ASM (x86): `coroutine_t cor_invoke(coroutine_t, size_t, ...)`

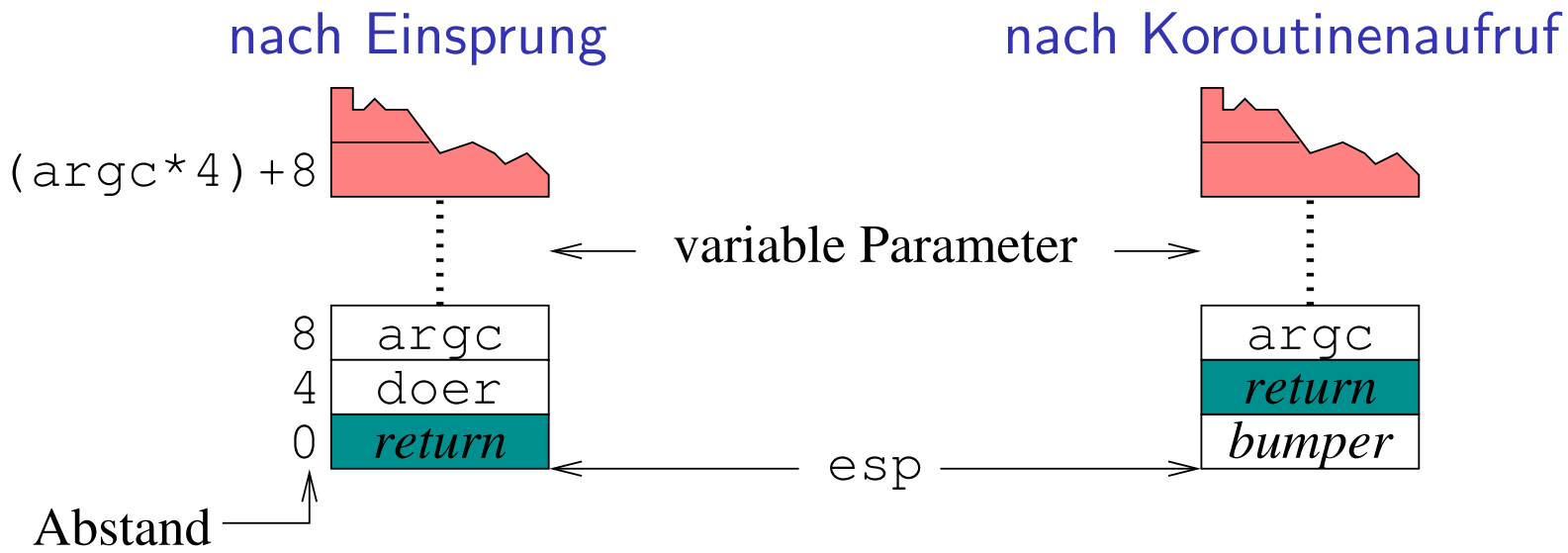
```
cor_invoke:
  popl  %eax            # remove return address to caller
  movl  (%esp), %ecx    # grab coroutine start address
  movl  %eax, (%esp)    # return address becomes first parameter
  call  *%ecx           # coroutine invocation: should not return!
  pushl %eax            # assume return 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, size_t argc, ...)`

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

Unterprogrammanordnung: *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

Verwendungsmuster: *invoke* einer rückkehrenden Koroutine

C

```
int niam(coroutine_t doer, size_t 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.

Reihenanordnung: *launch*, Prozedurinkarnation gabeln

Ergänzung zur Option (b) zur Einrichtung einer Fortsetzungsadresse

C/ASM (x86)

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

INLINE coroutine_t cor_launch(coroutine_t *leak) {
    coroutine_t aux;

    __asm__ __volatile__(
        "movl $1f, %0\n\t"   /* setup coroutine start address */
        "xorl %1, %1\n"      /* setup return value for creator */
        "1:"                 /* created coroutine starts here */
        : "=m" (*leak), "=r" (aux)
        :
        : "cc", "memory");

    return aux;
}
```

Verwendungsmuster: *launch* im Ensemble mit *resume*

```
coroutine_t niam, last;

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

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

Einbettung von *launch*: Übersetztes Programm von S. 25

Ausschnitt der Fallunterscheidung betreffs *launch*

ASM (x86)

```
#APP
  movl  $1f, niam
  xorl  %eax, %eax
1:
#NO_APP
  testl %eax, %eax
  movl  %eax, last
  je    .L2
```

Kommentar (zum expandierten Programmtext)

```
= begin of inlined text: inserted by gcc
# back up "launch label" value
# launch yields null for caller (creator)
# "launch label" where offspring starts off
= end of inlined text: inserted by gcc
# check for launch outcome: who came by "1:"?
# back up "resume label" value
# branch: true, caller; false, offspring
```

Erzeugung der Koroutine mittels *launch* schafft, im Gegensatz zu *invoke*, keinen neuen **Geltungsbereich** (engl. *scope*)

- ein und derselbe **Aktivierungsblock** (engl. *activation record*) ist für erzeugende und erzeugte Koroutine weiterhin gültig
- Rücksprung (*return*) einer der beteiligten Koroutinen zerstört den Aktivierungsblock für die anderen desselben Geltungsbereichs

Einbettung von *launch*, im Ensemble mit *resume*

Übersetztes Programm von S. 25, komplett: Welche „Werte“ enthält %eax wo?

ASM (x86)

```
main:
#APP last = cor_launch(&niam)
   movl  $1f, niam
   xorl  %eax, %eax
1:
#NO_APP
   testl %eax, %eax
   movl  %eax, last
   je    .L2
.L5:
   movl  %eax, %edx
#APP /* last = */ cor_resume(last)
   movl  $1f, %eax
   jmp   *%edx
1:
#NO_APP
   jmp   .L5
.L2:
   movl  niam, %eax
   movl  %eax, %edx
#APP /* niam = */ cor_resume(niam)
   movl  $1f, %eax
   jmp   *%edx
1:
#NO_APP
   movl  %eax, niam
   ret
```

Rückkehrverhalten: *launch*

launch kehrt (mindestens) zweimal zurück

1. Rückkehr zum aufrufenden Ausführungsstrang
   - normale Beendigung der Funktionsausführung 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 Sprungmarke dem Ausgabeparameter zu
- diese kann beliebig oft als Eingabewert für *resume* verwendet werden
- jedes derart parametrierte *resume* führt zur Rückkehr aus *launch*

Gliederung

Koroutinenkontrollflüsse „*Considered Harmful*“ ?

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. Prozesse dürfen nicht blockieren (*run to completion*)
  2. Prozesse dürfen sich nur stapelweise überlappen
- 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. . .

Beachte: Anwendungsanforderungen und Einplanungsstrategie

1. kooperative, durchlaufende Anwendungsprozesse
2. Prozesseinplanung, die eine Stapelmitbenutzung ermöglicht [**?**]

Resümee

- **Koroutinen** konkretisieren Prozesse, realisieren Prozessinkarnationen
  - die Gewichtsklasse von Prozessen spielt eine untergeordnete Rolle
    - feder-, leicht-, schwergewichtige Prozesse basieren auf Koroutinen
  - ihr Aktivierungskontext überdauert Phasen der Inaktivität
- minimale Basis der Fadenfamilie ist der **Koroutinenkontrollfluss**
  - die Grundlage für kooperative gleichzeitige Prozesse im Stapel
  - der Aktivierungskontext umfasst nur den Programmzähler
- einfachste **Elementaroperationen** für Koroutinenanlauf und -wechsel

launch	Gabelungsfunktion, spaltet eine Koroutine im Programm ab
invoke	Einrichtungsfunktion, startet eine Prozedur als Koroutine
resume	Umschaltfunktion, Sicherung des PC nach dem Wechsel
regain	Umschaltprozedur, Sicherung des PC vor dem Wechsel

- das Modell für **stapelorienterte Prozesseinplanung** und -einlastung
  - kooperative, durchlaufende (engl. *run to completion*) Prozesse
  - Überlappung gleichzeitiger Prozesse wie kaskadierte Unterbrechungen

Literaturverzeichnis

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

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

[3] Lohmann, D. ; Schröder-Preikschat, W. :
Betriebssysteme.
http://www4.informatik.uni-erlangen.de/Lehre/WS08/V_BS, 2008 *ff.*

[4] Schröder-Preikschat, W. ; Kleinöder, J. :
Systemprogrammierung.
http://www4.informatik.uni-erlangen.de/Lehre/WS08/V_SP, 2008 *ff.*

[5] Tanenbaum, A. S.:
Multilevel Machines.
In: *Structured Computer Organization*.
Prentice-Hall, Inc., 1979. –
ISBN 0–130–95990–1, Kapitel 7, S. 344–386

Gliederung

Definition einbettbarer Unterprogramme in C

```
/*
 * Language support for inline functions in C. The macro provides a portable
 * solution for C compilers being GNU or C99 standards compliant.
 *
 * GNU C:
 * inline           stand-alone object code is always emitted
 * static inline    stand-alone object code may be emitted if required
 * extern inline    stand-alone object code is never emitted
 *
 * C99:
 * inline           stand-alone object code is never emitted
 * static inline    same as GNU C
 * extern inline    stand-alone object code is always emitted
 */

#ifndef INLINE
#  if __GNUC__
#    define INLINE extern inline
#  else
#    define INLINE inline
#  endif
#endif
```

Unterprogrammanordnung: *resume*

Unterbrechung und Fortsetzung von Koroutinenausführungen

1. Funktionale Abstraktion von der *resume*-Implementierung

- die Implementierung der Funktion *resume* als **Unterprogramm**:

  Parameter $\mapsto$ Adresse der fortzusetzenden Koroutine

  Wert $\mapsto$ Adresse der unterbrochenen Koroutine

- Koroutinenwechsel verlaufen über **wirkliche Funktionsaufrufe**

2. Herleitung der Fortsetzungsadresse einer Koroutine

- jeder Unterprogrammaufruf hinterlegt eine **Rücksprungadresse**:
  (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 „von oben“ betrachtet

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

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
```

Bedeutung des Maschinenbefehls `call cor_resume`:

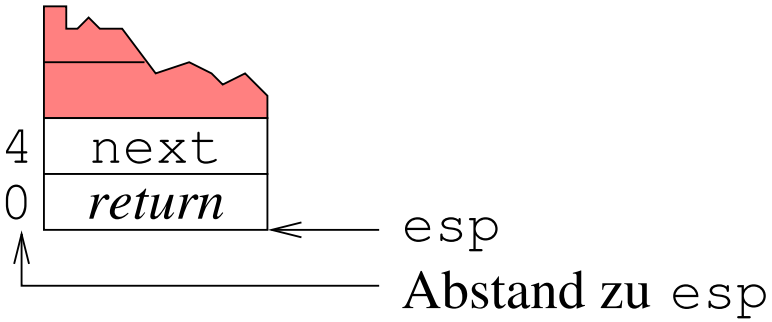
1. Aufruf des Unterprogramms zum Koroutinenwechsel
2. Generierung der Fortsetzungsadresse der laufenden Koroutine
   - repräsentiert durch die Rücksprungadresse der Funktion `cor_resume`
   - diese verweist auf den `call` folgenden Maschinenbefehl `movl...`

Koroutinenwechsel „von unten“ betrachtet

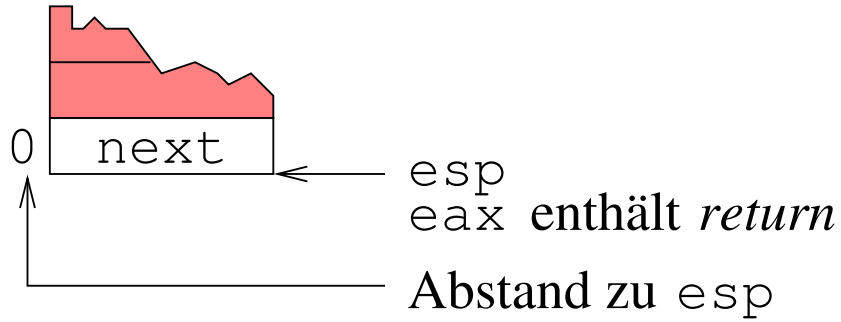
ASM (x86)

```
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 Unterprogramm cor_resume

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

Unterprogrammanordnung: *regain*, Alternative zu *resume*

Prinzip: Sicherung der Fortsetzungsadresse der laufenden Koroutine durch sie selbst — und nicht der vorherigen durch die nächste (vgl. S. 36):

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

coroutine_t next, self;

cor_regain(next, &self);
```

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

`self` Koroutinen-Fortsetzungsadresse
- wohin *regain* zurückkehrt

ASM (x86)

```
cor_regain:
  movl 8(%esp), %ecx  # grab pointer to backup variable
  popl %eax           # grab return address from top of stack
  movl %eax, (%ecx)   # backup return address
  jmp  *(%esp)        # continue at address given by 1. parameter
```

Unterprogrammanordnung: *launch*, Kontrollfluss gabeln

Ergänzung zur Option (b) zur Einrichtung einer Fortsetzungsadresse (S. 20)

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
```