Betriebssystemtechnik

Fadenimplementierung: Minimale Erweiterungen

Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

5. Juni 2012

Gliederung

1 Rekapitulation

2 Koroutinendomäne
- Schnittstelle
- Stapelspeicher
- Elementaroperationen

3 Koroutinenkontext
- Schnittstelle
- Prozessorstatus
- Elementaroperationen

4 Zusammenfassung

5 Anhang

Laufzeitzustand (Kontext) inkarnierter Koroutinen

Unterprogrammen („Routinen“) 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

Beachte
- 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 Koroutinenerweiterungen: Optionen

(a) Koroutinen einen eigenen **Stapelzeiger** (engl. *stack pointer*) geben
- je nach Arbeitsphase adressiert der SP unterschiedliche Zustandsdaten:
  - laufend ⇒ den der Koroutine zugeordnete lokale Datenraum
  - blockiert ⇒ zusätzlich noch die gesicherten Arbeitsregister und PC
- nur während der Blockadephase bleibt der Koroutinenzustand invariant

(b) **Prozessorstatus** beim Koroutinenwechsel sichern und wiederherstellen
- je nach der jeweils betrachteten Abstraktionsebene [3] bedeutet dies:
  - Ebene $_3$ • neben PC und SP, alle Arbeitsregister sichern/wiederherstellen
  - Ebene $_4$ • nur die *nichtflüchtigen Register* sichern/wiederherstellen
  - Ebene $_5$ • nur der Teil davon, den der Aufrufkontext von *resume* belegt
- der Wechsel 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)

Beachte: Programmfamilie, inkrementeller Maschinenentwurf
- Optionen (a) und (b) resultieren in eigene Familienmitglieder

Stroh-, fliegen- und bantamgewichtige Prozesse

Strohgewicht
- Koroutine mit eigenem **Programmzähler** (Reinform)
  - Koroutinenwechsel meint Programmzählerwechsel
- Basis für *kooperative* gleichzeitige Prozesse *im Stapel*
- darf *nicht blockieren*, aber andere *bedingt überlappen*

Fliegengewicht
- Koroutine mit eigenem Herrschaftsbereich: **Domäne**
  - Koroutinenwechsel meint zus. Stapelzeigerwechsel
- Erweiterung für *kooperative* gleichzeitige Prozesse
- darf *blockieren* und andere *beliebig überlappen*

Bantamgewicht
- Koroutine mit eigenem **Kontext** (Prozessorstatus)
  - Koroutinenwechsel meint zus. Arbeitsregisterwechsel
- Erweiterung für gleichzeitige Prozesse
- darf *blockieren* und andere *beliebig überlappen*

Beachte: Gemeinsamkeit
- Mitbenutzung desselben (phys., log., virt.) Adressraums

Gliederung

1 Rekapitulation
2 Koroutinendomäne
- Schnittstelle
- Stapelspeicher
- Elementaroperationen
3 Koroutinenkontext
- Schnittstelle
- Prozessorstatus
- Elementaroperationen
4 Zusammenfassung
5 Anhang

Funktionale/Prozedurale Abstraktion

Modulschnittstelle: `codomain.h`, Differenz zu `coroutine.h`[a]

[a] Abgesehen von der verschiedenen Typung (`codomain_t` anstatt `coroutine_t`) und Namensgebung (`cod` anstatt `cor`), zur Deklaration eines neuen „abstrakten Datentyps“.

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

typedef coroutine_t* codomain_t;      /* coroutine's "stack pointer" */

extern codomain_t cod_purify (const char *, size_t, size_t);
extern codomain_t cod_invoke (codomain_t, coroutine_t, size_t, ...);
```

purify
- generiert den passenden initialen Wert für den Stapelzeiger

invoke
- erhält diesen als Parameter, um die Domäne einzurichten

Beachte: Wechsel der Koroutinendomäne
- sichert den Programmzähler (`coroutine_t`) auf den Stapelspeicher
- bringt Umschalten des Stapelzeigers (`coroutine_t *`) mit sich

Der *Stack* — „Das unbekannte Wesen“

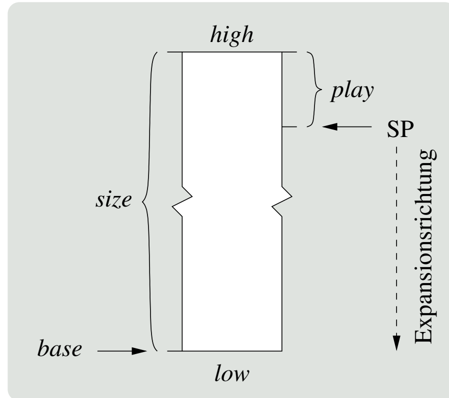
Stapelspeicher ist nichts weiter als ein **Segment** im Arbeitsspeicher von dynamischer aber maximaler Ausdehnung
- der **Stapelzeiger** adressiert eine „Entität“ in diesem Segment

Beachte: Prozessorabhängigkeiten des Stapelzeigers
- worauf zeigt der Stapelzeiger genau?
  - i auf das zuletzt abgelegte, also oberste Element?
  - ii oder auf das nächste freie Element, wie z.B. M6800 [1]/M6809 [2]?
- ist seine Ausrichtung[a] freigestellt, empfohlen oder gar zwingend?
  - i löst die CPU bei nicht ausgerichtetem Stapelzeiger einen *Trap* aus?
  - ii oder arbeitet sie dann nur langsamer, wie z.B. x86 (ab $x = 802$)?
- wie verhält es sich mit der Wortbreite des Stapelzeigerregisters?
  - i ist diese N bei einem 2^N Bytes großen physikalischen Adressraum?
  - ii oder gar kleiner als N, z.B. $N/2$ mit $N = 16$ wie beim 8051 [4]?
- nicht zuletzt, welche Expansionsrichtung des Stapels ist vorgegeben?

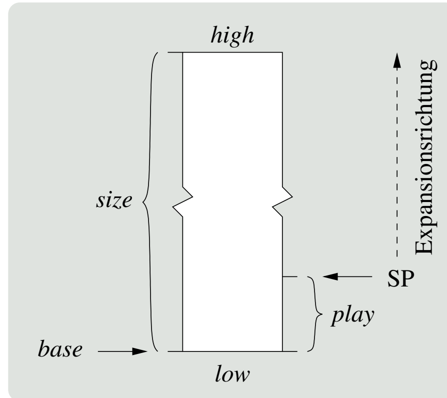
[a] engl. *alignment*

Stapelspeichervarianten: Expansionsrichtungen

abwärts wachsender Stapel



aufwärts wachsender Stapel (8051 [4])



Beachte: evtl. Spielraum (engl. *play*) ist dem Kompilierer geschuldet

- beim 8051 beginnt der Stapel gar erst ab Adresse 0x8 zu wachsen!!!

Aufbereitung des initialen Stapelzeigers: *purify*

C/x86

```
INLINE codomain_t cod_purify(const char *base, size_t size, size_t play) {
    return (codomain_t)
        (((unsigned)base + size - play) & ~(sizeof(codomain_t) - 1));
}
```

base	Anfangsadresse
size	Länge (in Bytes)
play	Spielraum (in Bytes)

x86	abwärts wachsender Stapel; Ausrichtung geboten
gcc	ggf. Füllwerk (engl. *padding*)

C/8051 [4]: aufwärts wachsend, byteweise ausgerichtet

```
INLINE codomain_t cod_purify(const char *base, size_t size, size_t play) {
    return (codomain_t)((unsigned)base + play);
}
```


Anlauf einer Koroutinendomäne: Inkarnation

Gemeinsamkeiten mit der Koroutine in Reinform (vgl. Kap. IV-1):

- unabhängig von der Domäne fehlt Koroutinen die Aufrufhierarchie
  - sie werden nicht aufgerufen, um mit der Ausführung zu beginnen
  - stattdessen wird ihre Ausführung immer nur fortgesetzt
- explizit muss eine **initiale Fortsetzungsadresse** eingerichtet werden
  (a) statische Anfangsadresse einer Prozedur: *invoke*
  (b) dynamische Verzweigungsadresse eines Ausführungsstrangs: *launch*

Unterschiede in funktionaler Hinsicht

- zur Inkarnation einer Koroutinendomäne ist ein Laufzeitstapel zu initialisieren, bevor die zugehörige Koroutine ablaufen kann
  - *invoke*: Kopieren des Aktivierungsblocks auf den fabrikneuen Stapel
  - *launch*: Reihenanordnung: Definition/Sicherung der Startadresse
  - Unterprogrammanordnung: Stapelplatz für Aktivierungsblock schaffen und Kopieren der Rücksprungadresse nach dahin
- allgemein: die **Vererbung** eines Aufrufkontextes an ein Koroutine

Anlauf einer Koroutinendomäne: *invoke*

C/x86: Aktivierungsblock kopieren und Koroutine aktivieren

```
#include <stdint.h>
#include "luxe/codomain.h"

extern void cod_bumper();

codomain_t cod_invoke(codomain_t this, coroutine_t code, size_t argc, ...) {
    uint32_t *tos;

    __asm__ __volatile__(           /* copy activation record: */
        "std\n\t"                       /* while copying, decrement source and destination addresses */
        "rep movsd"                     /* copy parameter list starting with argc */
        : "=D" (tos) : "D" ((uint32_t *)this - 1), "S" ((uint32_t *)&argc + argc), "c" (argc + 1)
        : "memory");

    *(--tos) = (uint32_t)cod_bumper;    /* make sure coroutine gets caught upon return */

    __asm__ __volatile__(           /* activate coroutine: */
        "pushl $1f\n\t"                 /* define own continuation address */
        "movl  %%esp, %0\n\t"           /* pass own coroutine domain pointer as actual parameter */
        "movl  %1, %%esp\n\t"           /* switch coroutine domain */
        "jmp   *%2\n"                   /* switch coroutine */
        "1:"                            /* resuming label: come back here */
        : "=g" (((uint32_t *)tos)[1]) : "g" (tos), "r" (code)
        : "esp", "memory");

    return (codomain_t)tos;
}
```

Anlauf einer Koroutinendomäne: *invoke* (Forts.)

„Vertraue, aber prüfe nach“ (russ. Sprichwort): `gcc -O6 -static -S cod_invoke.c`

ASM (x86)

```
cod_invoke:
  subl   $8, %esp                  # allocate space for two "callee saved" registers
  movl   %edi, 4(%esp)             # save 1st "callee saved" register
  movl   20(%esp), %ecx            # read "argc" actual parameter
  movl   12(%esp), %edi            # read "this" actual parameter
  movl   %esi, (%esp)              # save 2nd "callee saved" register
  leal   20(%esp,%ecx,4), %esi     # $1: compute "(unit32_t *)&argc + argc"
  addl   $1, %ecx                  # $2: compute "argc + 1"
  subl   $4, %edi                  # $3: compute "(unit32_t *)this - 1"
#APP                               = begin of inlined text
  std                              # tell CPU to decrement source and destination addresses
  rep movsd                        # perform memory copy of $2 bytes: "push" actication record
#NO_APP                            = end of inlined text
  movl   $cod_bumper, -4(%edi)     # stop coroutine upon (unexpected/forbidden) return
  movl   16(%esp), %edx            # read "code" actual parameter
  leal   -4(%edi), %eax            # $4: compute initial stack pointer for new coroutine
#APP                               = begin of inlined text
  pushl  $1f                       # leave resuming address on new coroutine's stack
  movl   %esp, (%edi)              # backup stack pointer of current coroutine
  movl   %eax, %esp                # switch coroutine domain: stack pointer becomes $4
  jmp    *%edx                     # actually invoke new coroutine
1:                                 # resuming label: come here upon resume/regain
#NO_APP                            = end of inlined text
  movl   (%esp), %esi              # restore 2nd "callee saved" register
  movl   4(%esp), %edi             # restore 1st "callee saved" register
  addl   $8, %esp                  # free space for two "callee saved" registers
  ret                              # return to callee: %eax holds value passed back by resume/regain
```


Anlauf einer Koroutinendomäne: *launch*

Reihenanordnung

```
INLINE codomain_t cod_launch(codomain_t *next) {
    *next = *next - 1;  /* play for resume addr. */
    return cod_adjust(*next);
}
```

```
INLINE codomain_t cod_adjust(codomain_t next) {
    codomain_t null;

    __asm__ __volatile__(
        "movl $1f, (%0)\n\t"  /* resuming address */
        "xorl %1, %1\n"       /* caller returns! */
        "1:"                  /* resuming label */
        : "=g" (next), "=a" (null) : "0" (next)
        : "cc", "memory");

    return null;
}
```

. . . expandierter Programmtext dazu

```
  movl next, %edx    # read domain pointer
  subl $4, %edx      # give play for resuming addr.
  movl %edx, next    # save value as domain pointer
#APP
  movl $1f, (%edx)   # save resuming address
  xorl %eax, %eax    # indicate return by caller
1:                   # resuming label: return here
#NO_APP
```

Unterprogrammanordnung

```
codomain_t cod_launch(codomain_t *next) {
    register codomain_t codo __TOS;

     *next = *next - 2;  /* play for act. record */
    **next = *codo;      /* copy return address */

    return 0;
}
```

```
#define __TOS __asm__("esp")
```

gcc -O6 -static -S

```
_cod_launch:
  movl 4(%esp), %edx   # grab actual parameter
  movl (%edx), %ecx    # grab initial SP
  leal -8(%ecx), %eax  # play for activation record
  movl %eax, (%edx)    # save stack address as SP
  movl (%esp), %eax    # grab return address
  movl %eax, -8(%ecx)  # inherit (copy) to coroutine
  xorl %eax, %eax      # indicate return by caller
  ret
```

. . . Aufrufumgebung dazu

```
  movl $_next, (%esp)  # pass "next" actual parameter
  call _cod_launch     # launch "next" coroutine
```


Wechsel einer Koroutinendomäne: *resume*

Reihenanordnung

```
INLINE codomain_t cod_resume(codomain_t next) {
    codomain_t self;

    __asm__ __volatile__(
        "pushl $1f\n\t"       /* save res. addr. */
        "movl  %%esp, %0\n\t" /* grab stack ptr. */
        "movl  %1, %%esp\n\t" /* switch stack */
        "ret\n"               /* next coroutine */
        "1:"                  /* resuming label */
        : "=&a" (self)        /* same as launch! */
        : "g" (next)
        : "%esp", "memory");

    return self;
}
```

. . . expandierter Programmtext dazu

```
#APP                  = begin of inlined text
  pushl $1f           # save resuming address
  movl  %esp, %eax    # grab own stack pointer
  movl  next, %esp    # switch stack
  ret                 # return to next coroutine
1:                    # resuming label: return here
#NO_APP               = end of inlined text
```

Unterprogrammanordnung

```
codomain_t cod_resume(codomain_t next) {
    return cod_switch(next);
}
```

```
INLINE codomain_t cod_switch(codomain_t next) {
    codomain_t self;

    __asm__ __volatile__(
        "movl %%esp, %0\n\t" /* grab stack ptr. */
        "movl %1, %%esp"     /* switch stack */
        : "=&g" (self) : "g" (next) : "%esp");

    return self;
}
```

gcc -O6 -static -S

```
_cod_resume:
  movl %esp, %eax     # grab own stack pointer
  movl 4(%esp), %esp  # switch stack
  ret                 # return to next coroutine
```

. . . Aufrufumgebung dazu

```
  movl  _next, %eax    # read coroutine domain pointer
  movl  %eax, (%esp)   # pass "next" actual parameter
  call  _cod_resume    # resume "next" coroutine
```


Wechsel einer Koroutinendomäne: *regain*

Reihenanordnung

```
INLINE
void cod_regain(codomain_t next, codomain_t *self) {
    __asm__ __volatile__(
        "pushl $1f\n\t"       /* save res. addr. */
        "movl  %%esp, %0\n\t" /* save stack ptr. */
        "movl  %1, %%esp\n\t" /* switch stack */
        "ret\n"               /* next coroutine */
        "1:"                  /* resuming label */
        : "=g" (*self)
        : "g" (next)
        : "%esp", "memory");
}
```

. . . expandierter Programmtext dazu

```
#APP                  = begin of inlined text
  pushl $1f           # save resuming address
  movl  %esp, last    # save stack pointer
  movl  next, %esp    # switch stack
  ret                 # return to next coroutine
1:                    # resuming label: return here
#NO_APP               = end of inlined text
```

Funktion oder **Prozedur**?

- eine Frage von Laufzeitaufwand und Lenkbarkeit (Übersetzung)

Unterprogrammanordnung

```
void cod_regain(codomain_t next, codomain_t *self) {
    *self = cod_switch(next);
}
```

. . . expandierter Programmtext dazu !?

```
cod_regain:
#APP                    = begin of inlined text
  movl %esp, %eax       # grab stack pointer
  movl 4(%esp), %esp    # switch (!) stack
#NO_APP                 = end of inlined text
  movl 8(%esp), %edx    # grab (self) actual param.
  movl %eax, (%edx)     # save (?) stack pointer
  ret                   # resume (next) coroutine
```

Konsequenz: Einbettung „von Hand“

```
_cod_regain: cod_regain:
  movl 8(%esp), %eax    # grab "self" actual param.
  movl %esp, (%eax)     # save stack pointer
  movl 4(%esp), %esp    # switch stack
  ret                   # resume "next" coroutine
```

. . . Aufrufumgebung dazu

```
  movl $_last, 4(%esp) # pass "self" actual param.
  movl _next, %eax      # read coroutine domain ptr.
  movl %eax, (%esp)     # pass "next" actual param.
  call _cod_regain      # resume "next" coroutine
```

Fliegengewichtige Prozesse in Aktion. . .

```
#include <stdio.h>
#include <stdlib.h>
#include "luxe/codomain.h"
#include "luxe/fame/inline.h"

#ifdef __fame_inline
#include "luxe/inline/cod_resume.c"
#include "luxe/machine/inline/cod_purify.c"
#include "luxe/machine/inline/cod_launch.c"
#endif

#define STACKSIZE 4*1024

codomain_t last, next;

int beep = 0;

main() {
    char *codo = malloc(STACKSIZE);
    int loop;

    printf("main(): codo 0x%x, size %d\n", codo, STACKSIZE);

    if (codo) {
        next = cod_purify(codo, STACKSIZE, 0);

        printf("main(): codo purified, next 0x%x\n", next);

        if (last = cod_launch(&next)) {
            for (;;) {
                beep++;
                last = cod_resume(last);
            }
        } else {
            for (loop = 1; loop < 43; loop++)
                next = cod_resume(next);

            printf("main(): beep %d\n", beep);
        }

        printf("main(): free(0x%x)\n", codo);
        free(codo);
    }

    printf("main(): exit...\n");
    exit(0);
}
```


Gliederung

1. Rekapitulation
2. Koroutinendomäne
   - Schnittstelle
   - Stapelspeicher
   - Elementaroperationen
3. Koroutinenkontext
   - Schnittstelle
   - Prozessorstatus
   - Elementaroperationen
4. Zusammenfassung
5. Anhang

Funktionale/Prozedurale Abstraktion

Modulschnittstelle: `coaction.h`, Differenz zu `codomain.h`[a]

[a] Abgesehen von der verschiedenen Typung (`coaction_t` anstatt `codomain_t`) und Namensgebung (`coa` anstatt `cod`), zur Deklaration eines neuen „abstrakten Datentyps“.

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

typedef struct corecord* coaction_t;  /* coroutine's "status pointer" */
```

struct corecord
- Datensatz einer Koroutine $\models$ **Prozessorstatus**
- Verbund aus Programmzähler und Arbeitsregister
  - nichtflüchtige („*callee saved*“) Register, ohne SP

Beachte: Wechsel des Koroutinenkontextes
- sichert den Prozessorstatus (`struct corecord`) im Stapelspeicher
- wechselt die Koroutinendomäne durch Umschalten des Stapelzeigers

Invarianter Datensatz eines Koroutinenkontextes

Reihenfolge der Zusammensetzung hängt ab von der Art und Weise der Einbettung der Routinen zum Kontextwechsel in den Programmtext

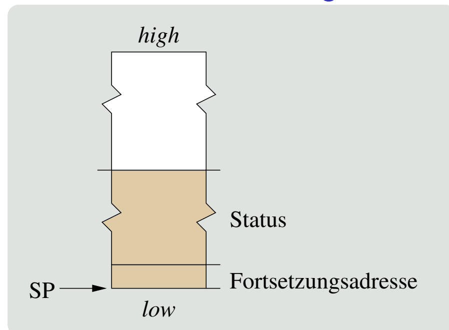
```
struct corecord {
    union {
        struct {
            coroutine_t code;    /* resuming address */
            costatus_t  cost;    /* "callee saved" registers */
        } __fame_status_inline;
        struct {
            costatus_t  cost;    /* "callee saved" registers */
            coroutine_t code;    /* resuming address */
        } __fame_status_exline;
    };
};
```

Beachte: Optionen der Reihenfolge der Kontextsicherung

Reihenanordnung	Status *vor* Fortsetzungsadresse
Unterprogrammanordnung	Status *nach* Fortsetzungsadresse

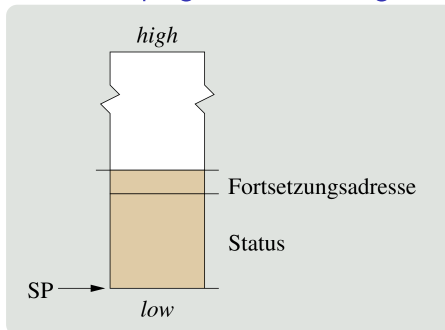
Eingebetteter *vs.* nicht eingebetteter Kontextwechsel

Reihenanordnung



- Koroutine sichert ihren eigenen Prozessorstatus und stellt diesen auch **selbst** wieder her

Unterprogrammanordnung



- Koroutine sichert ihren eigenen Prozessorstatus und stellt den einer anderen wieder her

Beachte: Möglicher variabler Status bei Reihenanordnung

- allen Koroutinen gemeinsam ist nur die Fortsetzungsadresse!!!

Abhängige *vs.* unabhängige Programmfäden

Fadeneigenschaften als Konsequenz aus der Art der (programmier-) technischen Umsetzung des Kontextwechsels

Reihenanordnung

1. Kontextsicherung der laufenden Koroutine
2. Stapelumschaltung
3. Kontrollflussumschaltung
4. Kontextwiederherstellung der laufenden Koroutine

Unterprogrammanordnung

1. Kontextsicherung der laufenden Koroutine
2. Stapelumschaltung
3. Kontextwiederherstellung der nächsten Koroutine
4. Kontrollflussumschaltung

Unabhängige Fäden

- Koroutinen müssen nicht vom selben Kontexttyp sein

Abhängige Fäden

- Koroutinen müssen vom selben Kontexttyp sein

Invariant zu haltende Inhalte von Arbeitsregistern

Nichtflüchtige Register: „*callee saved*“ (x86)

```
#include <stdint.h>

typedef struct costatus {
    uint32_t save[4];    /* 0=edi, 1=esi, 2=ebp, 3=ebx */
} costatus_t;
```

Verantwortung trägt das aufgerufene Unterprogramm, das nichtflüchtige Register vor Nutzung sichern und vor Rückkehr wieder herstellen muss

- betroffen sind nur die im Unterprogramm verwendeten Register
- die betroffenen Register sind i.d.R. nur dem Kompilierer bekannt
- ihre Anzahl hängt ab von:
  (a) den jeweiligen Eigenschaften des Unterprogramms und
  (b) der Binärschnittstelle (engl. *application binary interface*, ABI)

Beachte: Fehlendes „Programmwissen“ auf Betriebssystemebene

- alle nichtflüchtigen Register sind zu sichern und wiederherzustellen

Wechsel des Koroutinenkontextes: 1. *resume* und 2. *regain*

Reihenanordnung: Einbettung durch gcc

```
INLINE coaction_t coa_resume(coaction_t next) {
    codomain_t codo;

    abi_push();
    codo = cod_resume((codomain_t)next);
    abi_pull();

    return (coaction_t)codo;
}
```

```
INLINE void coa_regain(coaction_t next, coaction_t *self) {
    abi_push();
    cod_regain((codomain_t)next, (codomain_t *)self);
    abi_pull();
}
```

```
INLINE uint32_t *abi_push() {
    register uint32_t *tos __TOS;

    __asm__ __volatile__(
        "pushl %%ebx\n\t"
        "pushl %%ebp\n\t"
        "pushl %%esi\n\t"
        "pushl %%edi"
        : : : "%esp", "memory");

    return tos;
}
```

```
INLINE void abi_pull() {
    __asm__ __volatile__(
        "popl %%edi\n\t"
        "popl %%esi\n\t"
        "popl %%ebp\n\t"
        "popl %%ebx"
        : : : "%esp", "memory");
}
```

1. expandiert

```
  movl next, %edx
#APP
  pushl %ebx
  pushl %ebp
  pushl %esi
  pushl %edi
  pushl $1f
  movl  %esp, %eax
  movl  %edx, %esp
  ret
1:
  popl  %edi
  popl  %esi
  popl  %ebp
  popl  %ebx
#NO_APP
```

2. expandiert

```
  movl  last, %eax
#APP
  pushl %ebx
  pushl %ebp
  pushl %esi
  pushl %edi
  pushl $1f
  movl  %esp, next
  movl  %eax, %esp
  ret
1:
  popl  %edi
  popl  %esi
  popl  %ebp
  popl  %ebx
#NO_APP
```

Kaum besser „von Hand“

Wechsel des Koroutinenkontextes: 1. *resume* und 2. *regain*

Unterprogrammanordnung: Einbettung durch gcc, `-fomit-frame-pointer`

```
coaction_t coa_resume(coaction_t next) {
    codomain_t codo;

    abi_push();
    codo = cod_switch((codomain_t)next);
    abi_pull();

    return (coaction_t)codo;
}
```

```
void coa_regain(coaction_t next, coaction_t *self) {
    abi_push();
    *self = (coaction_t *)cod_switch((codomain_t)next);
    abi_pull();
}
```

1. expandiert

```
coa_resume:
#APP
  pushl %ebx
  pushl %ebp
  pushl %esi
  pushl %edi
  movl %esp, %eax
  movl 4(%esp), %esp
  popl %edi
  popl %esi
  popl %ebp
  popl %ebx
#NO_APP
  ret
```

2. expandiert

```
coa_regain:
#APP
  pushl %ebx
  pushl %ebp
  pushl %esi
  pushl %edi
  movl  %esp, %eax
  movl  4(%esp), %esp
#NO_APP
  movl  8(%esp), %edx
  movl  %eax, (%edx)
#APP
  popl  %edi
  popl  %esi
  popl  %ebp
  popl  %ebx
#NO_APP
  ret
```

Stapelzeiger als lokale Basis beim Parameterzugriff

- Kontextsicherung auf den Stapelspeicher verschiebt den Stapelzeiger und somit den Versatz zu den aktuellen Parametern
- gcc „sieht" die Veränderung und korrigiert den Versatz nicht, daher:
  - (a) das Unterprogramm ohne `-fomit-frame-pointer` übersetzen oder ☹
  - (b) die Einbettung von *push*, *switch* und *pull* „von Hand" machen ☺

Wechsel des Koroutinenkontextes: 1. *resume* und 2. *regain*

Unterprogrammanordnung: Einbettung „von Hand"

1. Funktion

```
_coa_resume: coa_resume:
  pushl %ebx              # save coroutine context
  pushl %ebp              # "
  pushl %esi              # "
  pushl %edi              # "
  movl  %esp, %eax        # grab stack pointer: result
  movl  20(%esp), %esp    # switch coroutine domain
  popl  %edi              # restore coroutine context
  popl  %esi              # "
  popl  %ebp              # "
  popl  %ebx              # "
  ret                     # switch coroutine
```

... Aufrufumgebung dazu

```
  movl  _next, %eax   # read coroutine action pointer
  movl  %eax, (%esp)  # pass "next" actual parameter
  call  _coa_resume   # resume "next" coroutine
```

2. Prozedur

```
_coa_regain: coa_regain:
  pushl %ebx              # save coroutine context
  pushl %ebp              # "
  pushl %esi              # "
  pushl %edi              # "
  movl  24(%esp), %eax    # grab addr. of backup var.
  movl  %esp, (%eax)      # save stack pointer
  movl  20(%esp), %esp    # switch coroutine domain
  popl  %edi              # restore coroutine context
  popl  %esi              # "
  popl  %ebp              # "
  popl  %ebx              # "
  ret                     # switch coroutine
```

... Aufrufumgebung dazu

```
  movl $_last, 4(%esp) # pass "self" actual param.
  movl _next, %eax     # read coroutine action ptr.
  movl %eax, (%esp)    # pass "next" actual param.
  call _coa_regain     # resume "next" coroutine
```

Beachte: Variante ohne `-fomit-frame-pointer` (für S. 25, links, C)

- der Mehraufwand ist in Relation zum Prozessorstatusumfang zu sehen

Anlauf mit eigenem Koroutinenkontext

Initialisierung des Stapelspeichers mit einem Koroutinenkontext derart, dass der Anlauf zu den Varianten (S. 21) der Kontextsicherung passt

- ist die Operation zum initialen Wechsel eingebettet, muss der Stapel nur mit einer Fortsetzungsadresse vorbelegt sein
  - *resume* & *regain* stellen nur den eigenen Prozessorstaus wieder her !!!
- anderenfalls muss auf der Rücksprungadresse zusätzlich noch der Prozessorstatus gestapelt vorliegen
  - *resume* & *regain* stellen einen fremden Prozessorstaus wieder her !!!

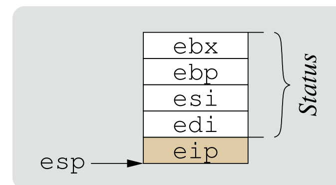
Unterschiede zeichnen sich auch in den beiden Funktionen zur Einrichtung von Koroutinen mit eigenen Kontexten ab

launch
- lässt neue Koroutine in einen definierten Kontext zurückkehren
- daher: Kontext der laufenden, lancierenden Koroutine vererben

invoke
- verlässt laufende Koroutine hinein in undefinierten Kontext
- daher: Kontext der laufenden Koroutine invariant halten

Stapelaufbau inaktiver Koroutinen

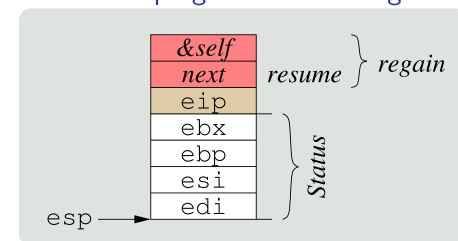
Reihenanordnung



Schritte in *resume*/*regain*:

1. nichtflüchtige Register sichern
2. Fortsetzungsadresse sichern

Unterprogrammanordnung



Schritte in *resume*/*regain*:

1. Rücksprungadresse sichern
2. nichtflüchtige Register sichern

Randbedingungen zum Koroutinenanlauf: *launch*, nicht eingebettet!

- zusätzlichen Platzhalter für den aktuellen Parameter vorsehen
- gehört mit zum Aufrufkontext, auch wenn Inhalt bedeutungslos ist

Anlauf mit eigenem Koroutinenkontext: *launch*

```
INLINE coaction_t coa_launch (coaction_t *next) {
    coaction_t null;

    *next = *next - 1;      /* make room for coroutine action record */
    abi_dump((*next)->__fame_status_inline.cost.save);

    __asm__ __volatile__(
        "movl $1f, %0\n\t" /* define resuming address */
        "xorl %1, %1\n\t"  /* indicate return from caller (launcher) */
        "jmp  2f\n"        /* skip context recovering */
        "1:"               /* resuming label: launched coroutine starts here */
        : "=m" ((*next)->__fame_status_inline.code), "=a" (null)
        : : "cc", "memory");

    abi_pull();             /* recover inherited context */

    __asm__ __volatile__(
        "\n2:");            /* exit label for launching coroutine */

    return null;
}
```

dump
- sichert nichtflüchtige Register in den Stapel der Koroutine
- ausgeführt von der lancierenden Koroutine: Kontextvererbung

pull
- stellt die nichtflüchtigen Register vom Stapel wieder her
- ausgeführt von der anlaufenden Koroutine: Kontextübernahme

Anlauf mit eigenem Koroutinenkontext: *invoke*

Verwendung der Startadresse einer (nicht eingebetteten) Prozedur als initiale Fortsetzungsadresse für eine Koroutine

- analog zu einer aufgerufenen Prozedur, richtet sich die anlaufende Koroutine ihren **lokalen Kontext** selbst ein
  - die notwendige Kontextübertragung wie im Falle von *launch* entfällt ☺
- jedoch muss der Prozedur bei Koroutinenanlauf ein **Aufrufkontext** gegeben werden, um eine zurückkehrende Koroutine abzufangen
  - die zu erzeugende Koroutine muss dazu geeignet „aufgerufen“ werden
  - dabei ist die Integrität des lokalen Kontextes der Aufruferin gefährdet:
    1. die Aufgerufene sichert nur die benutzten nichtflüchtigen Register
    2. sie übergibt an eine dritte Koroutine, die alle Register „verschmutzt“
    3. die Dritte übergibt zurück an die Erste, die erzeugende Koroutine
    4. diese kehrt aus *invoke* zurück, mit ggf. ungültigem lokalen Kontext ☹

Beachte: Kontextsicherung in *invoke*

Zum Anlauf einer Koroutinenprozedur muss die aufrufende Koroutine für die Invarianz ihrer nichtflüchtigen Register sorgen.

Anlauf mit eigenem Koroutinenkontext: *invoke* (Forts.)

```
coaction_t coa_invoke (coaction_t this, coroutine_t code, size_t argc, ...) {
    uint32_t *tos;
```

...kopieren des Aktivierungsblocks, einrichten eines Aufrukontextes zum Abfangen einer zurückkehrenden Koroutine: vgl. S 12

Sicherung & Wiederherstellung restlicher nichtflüchtiger Register, Koroutinenaktivierung

```
    __asm__ __volatile__(
        "pushl $2f\n\t"        /* save resuming address for non-inlined resume/regain */
        "pushl %%ebx\n\t"      /* save not yet saved non-volatile registers */
        "pushl %%ebp\n\t"      /* " */
        "subl  $4, %%esp\n\t"  /* %esi: has already been "caller saved"; spare area */
        "pushl $1f\n\t"        /* %edi: dito.; used to save resuming address for inlined resume/regain */
        "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:\n\t"               /* resuming label: come back through inlined resume/regain */
        "addl  $4, %%esp\n\t"  /* remove spare area (%esi) */
        "popl  %%ebp\n\t"      /* restore saved non-volatile registers */
        "popl  %%ebx\n\t"      /* " */
        "addl  $4, %%esp\n"    /* remove resuming address for non-inlined resume/regain */
        "2:"                   /* resuming label: come back through non-inlined resume/regain */
        : "=g" (((uint32_t *)tos)[1])
        : "g" (tos), "r" (code)
        : "esp", "memory");

    return (coaction_t)tos;
}
```

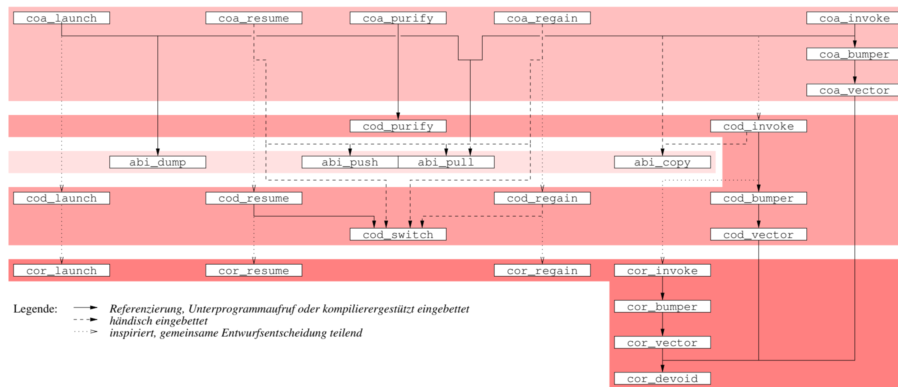
- Beachte: `esi` und `edi` wurden bereits „*callee saved*“ gesichert

Gliederung

1. Rekapitulation
2. Koroutinendomäne
   - Schnittstelle
   - Stapelspeicher
   - Elementaroperationen
3. Koroutinenkontext
   - Schnittstelle
   - Prozessorstatus
   - Elementaroperationen
4. **Zusammenfassung**
5. Anhang

Funktionale Hierarchie: Koroutinenfamilie

Ausgangspunkt ist die Unterprogrammanordnung der Schnittstellenoperationen



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

Echte Systemprogrammiersprachen gibt es nicht mehr. . .

- daher sind Koroutinen händisch in Assembliersprache bereitzustellen
- gleichwohl bleiben sie ein Programmiersprachenkonzept der Ebene $_5$

Resümee

- **Koroutinen** konkretisieren Prozesse, realisieren Prozessinkarnationen
  - implementieren stroh-, fliegen- oder bantamgewichtige Prozesse
  - residieren alle im selben (phys., log., virt.) Adressraum
- **minimale Basis** der Fadenfamilie ist der **Koroutinenkontrollfluss**
  - Grundlage für kooperative gleichzeitige Prozesse im Stapel
  - der Aktivierungskontext umfasst nur den Programmzähler
- **minimale Erweiterungen** ergänzen diese um weitere Eigenschaften:

  Koroutinendomäne
  - Basis für kooperative gleichzeitige Prozesse
  - Kontext erweitert um den Stapelzeiger
  - Stapelspeicher einrichten und verwalten

  Koroutinenaktion
  - Basis für gleichzeitige Prozesse
  - Kontext erweitert um nichtflüchtige Register
  - abhängige und unabhängige Fäden
- weitere Varianten durch die **Art der Einbettung** der Fadenprimitiven:

  Reihenanordnung
  - einbettbare (engl. *in-line*) Routinen

  Unterprogrammanordnung
  - gewöhnliche Unterprogrammaufrufe

Literaturverzeichnis

[1] MOTOROLA SEMICONDUCTOR PRODUCTS INC. (Hrsg.):
M6800 Programming Reference Manual.
Phoenix, Arizona, USA: Motorola Semiconductor Products Inc., Nov. 1976

[2] MOTOROLA SEMICONDUCTOR PRODUCTS INC. (Hrsg.):
MC6809-MC6809E 8-Bit Microprocessor Programming Manual.
Phoenix, Arizona, USA: Motorola Semiconductor Products Inc., März 1981

[3] SCHRÖDER-PREIKSCHAT, W. ; KLEINÖDER, J. :
Systemprogrammierung.
http://www4.informatik.uni-erlangen.de/Lehre/WS08/V_SP, 2008 *ff.*

[4] WHARTON, J. :
An Introduction to the Intel MCS-51 Single-Chip Microcomputer Familiy / Intel Corporation, Application Notes.
1980 (AFN-01502A-01). –
Application Note

Gliederung

1. Rekapitulation
2. Koroutinendomäne
   - Schnittstelle
   - Stapelspeicher
   - Elementaroperationen
3. Koroutinenkontext
   - Schnittstelle
   - Prozessorstatus
   - Elementaroperationen
4. Zusammenfassung
5. Anhang

Vorgehensweise zur Implementierung in Assembliersprache

1. Formulierung des fraglichen Programms auf Basis einbettbarer Unterprogramme
   - `vi`
2. Verwendung des Kompilierers zur Erzeugung des dazu passenden Programms in Assembliersprache
   - `gcc -S`
3. Korrektur und Nachbesserung des Assembliersprachenprogramms „von Hand“, dabei Kommentare nicht vergessen
   - `vi`
4. Assemblierung des bearbeiteten Programms und Erzeugung des Bindemoduls
   - `as -o`
5. Archivierung des Bindemoduls zur späteren Verwendung
   - `ar -r`