— BS //—

Programmfäden

Betriebssysteme, ©Wolfgang Schröder-Preikschat

Überblick

- Aktivitätsträger: Koroutinen 2
  - asymmetrisches Aufrufmodell — „prozedurorientiert“ 6
  - symmetrisches Aufrufmodell — „prozessorientiert“ 9
- Laufzeitkontext, Kontrollflusswechsel, Implementierung 12
- Diskussion 24
- Zusammenfassung 28

Koroutinen als Aktivitätsträger

- **autonome Kontrollflüsse** innerhalb desselben Programms (Betriebssystem)
  - Programm(kontroll)faden, *thread of control*, TOC; kurz: *thread*
- mit zwei wesentlichen Unterschieden zu herkömmlichen Routinen/Prozeduren:
  1. die Ausführung beginnt immer an der letzten „Unterbrechungsstelle“ [1]
     - d.h., an der zuletzt die Kontrolle über den Prozessor abgegeben wurde
     - die Kontrollabgabe geschieht dabei grundsätzlich *kooperativ*
  2. der Zustand ist *invariant* zwischen zwei aufeinanderfolgenden Ausführungen
- sie treten praktisch wie „zustandsbehaftete Prozeduren“ in Erscheinung

[1] Wurde die Koroutine noch niemals ausgeführt, gibt es keine Unterbrechungsstelle. Der Koroutinenanfang ist dann „letzte Unterbrechungsstelle“. Dies erfordert spezielle Maßnahmen bei der Instanzenbildung von Koroutinen.

Koroutine (1)

An autonomous program which communicates with adjacent modules as if they were input or output subroutines.

[. . .]

Coroutines are subroutines all at the same level, each acting as if it were the master program. [2] [2]

[2] Koroutinen tauchten erstmalig auf in der von Conway entwickelten klassischen Architektur eines Fließbandübersetzer (*pipeline compiler*). Darin wurden Parser konzeptionell als Datenflussfließbänder zwischen Koroutinen aufgefasst. Die Koroutinen repräsentierten *first-class* Prozessoren wie z.B. Lexer, Parser und Codegenerator.

Koroutine (2)

- Koroutinen sind Prozeduren ähnlich, es fehlt jedoch die Aufrufhierarchie:

  Beim Verlassen einer Koroutine geht anders als beim Verlassen einer Prozedur die Kontrolle nicht automatisch an die aufrufende Routine zurück. Stattdessen wird mit einer *resume*-Anweisung beim Verlassen einer Koroutine explizit bestimmt, welche andere Koroutine als nächste ausgeführt wird. [6], S. 49

- ein **programmiersprachliches Mittel** zur Prozessorweitergabe an Prozesse

Routinen vs. Koroutinen

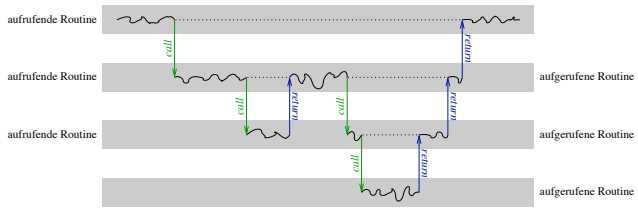
- der Unterschied wird u.a. auch durch die verschiedenen *Aufrufmodelle* deutlich:

  asymmetrisches Aufrufmodell von Routinen
  - die Beziehung zwischen den Routinen ist nicht gleichberechtigt
    - ein Spezial-/Problemfall stellt ({,in}direkte) Rekursion dar
  - es besteht eine Hierarchie zwischen aufrufende und aufgerufene Routine

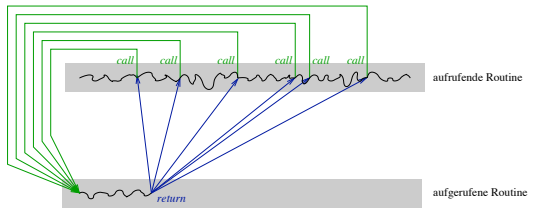
  symmetrisches Aufrufmodell von Koroutinen
  - zwischen Koroutinen ist keine Aufrufhierarchie definiert
    - ggf. jedoch eine Aktivierungsreihenfolge (*scheduling*)
  - die Beziehung zwischen den Koroutinen ist gleichberechtigt

- zwischen {,Ko}Routinen kann jedoch die gleiche Benutztbeziehung [7] bestehen

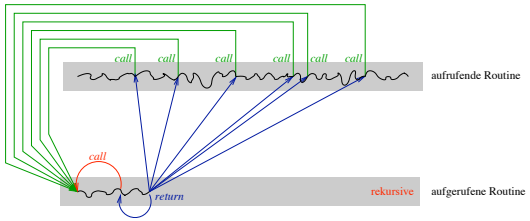
Asymmetrisches Aufrufmodell (1) Aufrufhierarchie



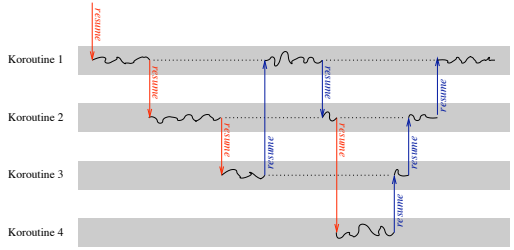
Asymmetrisches Aufrufmodell (2) Wiederverwendung



Asymmetrisches Aufrufmodell (3) — Rekursion



Symmetrisches Aufrufmodell — Koroutinen



Gemeinsamkeiten

- {,Ko}Routinen sind zu reaktivieren, um weiter ausgeführt werden zu können:

  Routine beim *Rücksprung* aus der aufgerufenen Instanz
  Koroutine beim *Suspendieren* der Kontrolle abgebenden Instanz

- jeder Aufruf hinterlässt seinen „Fußabdruck“ im Aktivierungsblock
  - die Rückkehradresse zur aufrufenden {,Ko}Routine wird gespeichert
  - die von der {,Ko}Routine belegten Register werden gesichert[3]

- der Aufbau des Aktivierungsblocks ist prozessor- und übersetzerabhängig

[3] Im Falle der Routine werden die Register erst in der aufgerufenen Instanz gesichert. Dagegen werden im Fall der Koroutine die Register schon in der aufrufenden Instanz gesichert.

Unterschiede

- eine Koroutine besitzt *eigene Betriebsmittel* zur Aktivierungsblockverwaltung
  - Art/Menge der Betriebsmittel ist {prozessor,übersetzer,problem}abhängig
    - CISC (Stapel) vs. RISC (Register und/oder Stapel)
    - Laufzeitmodell der jeweiligen Programmiersprache
    - von der Koroutine jeweils zu bewältigenden Aufgabe
  - die Verfügbarkeit eigener Betriebsmittel begründet die Unabhängigkeit

- eine Routine muss sich diese Betriebsmittel mit anderen Routinen teilen

Aktivierungsblock — *Activation Record*

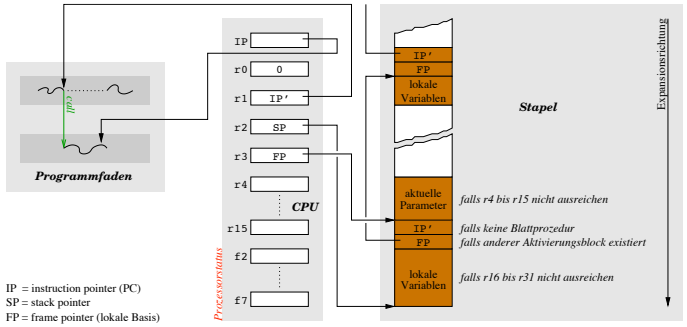
- definiert den Kontext einer aufgerufenen Routine bzw. suspendierten Koroutine
  - Rücksprungadresse, Stapelzeiger, lokale Basis und ggf.
  - aktuelle Parameter, lokale Variablen, „Zusammengewürfeltes“ (*scratch*)

Fallstudie g++ und der i860 von Intel[4]:

- die CPU implementiert 64 Arbeitsregister, jeweils 32 Bit breit
- der g++ unterteilt die Arbeitsregister in zwei Bereiche:
  - nicht-flüchtige Register `r0` – `r15` und `f2` – `f7`
  - flüchtige Register `r16` – `r31` und `f8` – `f31`
- Rücksprungadresse in `r1`, Stapelzeiger ist `r2`, lokale Basis ist `r3`

[4] Ein 32/64-Bit superskalarer Prozessor in RISC-Technologie [1]

Laufzeitkontext g++/i860



„Zustandsbehaftete Prozeduren“

- programmiersprachlich gesehen entspricht eine Koroutine einer Prozedur
  - die symbolische Bezeichnung einer Anweisungsfolge, eines Unterprogramms
- die Prozedur wird jedoch nie aufgerufen, sondern sie wird instanziiert
  - die Instanzenbildung schafft einen Laufzeitkontext für die Koroutine
  - die Aktivierung des Laufzeitkontextes (re-) aktiviert die Koroutine
  - der Laufzeitkontext ist invariant nur in Phasen der Koroutineninaktivität
  - die „Prozedur“ besitzt einen Zustand während sie nicht ausgeführt wird
- da die Prozedur nie aufgerufen wird, kann sie auch nirgendwohin zurückkehren

resume — „wieder aufnehmen“

- eine ELOP mit zwei fundamentalen Eigenschaften zum Kontrollflusswechsel:
  1. die Sicherung des Laufzeitkontextes der abgebenden Koroutine
     - den Prozessorstatus der laufenden Koroutine „einfrieren“
     - die den Laufzeitkontext repräsentierende Variable beschreiben
  2. die Herstellung des Laufzeitkontextes der aufzunehmenden Koroutine
     - den Prozessorstatus der suspendierten Koroutine „auftauen“
     - die den Laufzeitkontext repräsentierende Variable lesen
- im Regelfall wird durch diese Maßnahme der Prozessorstatus ausgetauscht

resume — ELOP der Ebene$_4$

- typischerweise CPU-abhängig in Assemblersprache implementiert als Prozedur
  - aus der nicht die Koroutine (jetzt) zurückkehrt, die den Aufruf getätigt hat
  - d.h., bei der Ausführung der Prozedur wechselt (im Regelfall) die Koroutine[5]
- eine ELOP mit zwei Operanden: `resume (op1&, const op2&)`

  op1 die Adresse der Kontextvariablen der laufenden Koroutine
  op2 die Adresse der Kontextvariablen der suspendierten Koroutine
- die prozedurale Herangehensweise eröffnet einen naheliegenden „Lösungstrick“

[5] Je nach Implementierung der `resume`-ELOP und der Werte ihrer Operanden könnte es auch möglich sein, dass ein Koroutinenwechsel nicht wirklich erfolgen muss.

resume — „Lösungstrick“

- der Aktivierungsblock des *resume*-Aufrufs enthält bereits Kontextinformation
  - nur der noch nicht gesicherte Kontextanteil ist „von Hand“ einzufrieren
  - im Regelfall[6] sollte dies nur nicht-flüchtige Register betreffen müssen
- die Kontextvariable einer Koroutine kann verschiedener Art sein:

  lokale Variable der *resume*-Prozedur → Stapel
  globale Variable des *resume* aufrufenden Programms → Halde
- Austausch des Stapelzeigers und Prozedurrückkehr aktiviert die Koroutine **!**

[6] Der Aufruf erfolgt aus einem Programm der Ebene$_5$ heraus, d.h., aus einem Hochsprachenprogramm. In dem Fall ist das Laufzeitmodell des Übersetzers maßgeblich, das zudem noch Optimierungspotential eröffnet.

resume — i860

```
_resume:
    st.l r1,R1(r16);   st.l r2,R2(r16);   st.l r3,R3(r16);   st.l r4,R4(r16)
    st.l r5,R5(r16);   st.l r6,R6(r16);   st.l r7,R7(r16);   st.l r8,R8(r16)
    st.l r9,R9(r16);   st.l r10,R10(r16); st.l r11,R11(r16); st.l r12,R12(r16)
    st.l r13,R13(r16); st.l r14,R14(r16); st.l r15,R15(r16)

    fst.d f2,F2(r16);  fst.d f4,F4(r16);  fst.d f6,F6(r16)

    fld.d F6(r17),f6;  fld.d F4(r17),f4;  fld.d F2(r17),f2

    ld.l R15(r17),r15; ld.l R14(r17),r14; ld.l R13(r17),r13
    ld.l R12(r17),r12; ld.l R11(r17),r11; ld.l R10(r17),r10; ld.l R9(r17),r9
    ld.l R8(r17),r8;   ld.l R7(r17),r7;   ld.l R6(r17),r6;   ld.l R5(r17),r5
    ld.l R4(r17),r4;   ld.l R3(r17),r3;                      ld.l R1(r17),r1

    bri  r1            /* return */
    ld.l R2(r17),r2    /* load stack pointer */
```


resume — m68k

```
_resume:
    movel  %sp@(4),%a0              /* grab context of this coroutine */
    moveml %d2-%d7/%a2-%a7,%a0@     /* save non-volatile registers */
    movel  %sp@(8),%a0              /* grab context of next coroutine */
    moveml %a0@,%d2-%d7/%a2-%a7     /* restore non-volatile registers */
    rts                             /* return */
```

lokale/globale Kontextvariable Die vorliegende Implementierung (wie auch die für den i860) abstrahiert durch die gewählte Adressierungsart davon, wo die Kontextvariable der beiden beteiligten Koroutinen lokalisiert ist. Ob es sich um eine lokale oder globale Variable handelt, d.h., ob die Variable auf dem Stapel oder der Halde angelegt ist, hängt nur von der Wahl der aktuellen Parameter ab. Mischformen sind möglich.

resume — ELOP anderer Ebenen

Ebene$_2$ konventionelle Maschinenebene

- kommerzielle Hardware mit integriertem Koroutinenkonzept ist unüblich
- dort, wo es solche Hardware gibt, werden die Konzepte nicht immer genutzt

Ebene$_3$ Betriebssystemebene

- Systemaufrufe zur Koroutinenverwaltung sind wenig sinnvoll
- beachte: *Threads* sind (viel) zu mächtig und mehr als Koroutinen

Ebene$_5$ problemorientierte Programmiersprachenebene

- Sprachen mit integriertem Koroutinenkonzept sind nicht weit verbreitet
- auch hier ist gleiches zu beachten wie für Ebene$_3$: *Thread* $\neq$ Koroutine

Instanzenbildung von Koroutinen

- damit eine (Ebene$_5$) Prozedur Koroutine werden kann, ist Vorsorge zu treffen:
  - der Stapel (*stack*) ist anzulegen und passend zu initialisieren
  - ein initialer Kontext ist zu erzeugen, der mittels *resume* aktivierbar ist
  - die versehentliche Rückkehr aus der „Koroutinenprozedur“ ist zu unterbinden
- eine ELOP mit drei Operanden: `create (cp*, sp*, pc*)`

  cp die Adresse der Kontextvariablen der zu instanziierenden Koroutine
  sp der initiale Stapelzeiger (*stack pointer*)
  pc die (Prozedur-) Startadresse der Koroutine
- Zerstören einer Koroutine geht einher mit Freigabe ihrer Kontextvariablen

create — i860

```
void create (long* cp, long* sp, void (*pc)()) {
    cp[R1] = (long)lifter;
    cp[R2] = (long)sp;
    cp[R4] = (long)pc;
}
```

```
_lifter:
    calli r4
    nop
    nop
1:  br    1b
    lock
```

„Notbremse“ *create* sorgt dafür, dass die Koroutine initial als Prozedur aufgerufen wird, nachdem diese mittels *resume* aktiviert worden ist. Damit kann aus der „Koroutinenprozedur“ zurückgekehrt werden. Im Falle der Rückkehr wird die Koroutine „ge*trap*t“. Der *lifter* implementiert die entsprechenden Aktionen: er führt den Aufruf aus und zwingt die Koroutine in die „Falle“.

create — „Untiefen“

- die Instanzenbildung von Koroutinen ist zutiefst CPU- und übersetzerabhängig
  - auch wenn keine Assemblerprogrammierung anfallen würde[7]
- ebenso ist die Berechnung des initialen Stapelzeigers maschinenabhängig:
  - die Notwendigkeit eines Stapels ist überhaupt CPU-abhängig !
  - die Expansionsrichtung des Stapels ist CPU- bzw. übersetzerabhängig !
  - die Ausrichtung (*alignment*) des Stapelzeiges ist CPU-abhängig !
- desweiteren ist es nicht einfach, die richtige Größe des Stapels zu bestimmen

[7] Es ist nicht zwingend notwendig *lifter* in Assembler zu programmieren. Je nach CPU kann diese Funktion auch in Hochsprache realisiert werden. Sie bliebe dann aber immer noch unportabel.

Ein Beispiel — das zu denken gibt . . .

```
long* life;
long* next;

main () {
    long foo[CONTEXT_SIZE];
    long bar[CONTEXT_SIZE + 16 * 1024];

    create(&bar[0],
           &bar[CONTEXT_SIZE + 16 * 1024],
           coroutine);

    life = &foo[0];
    next = &bar[0];

    coroutine();
}
```

```
void coroutine () {
    for (int i = 0; i < 10; i++) {
        printf("%d", i);

        long* self = life;
        life = next;
        next = self;

        resume(self, life);
    }
}
```


Alternative (1) — Nebenläufige Blöcke

- strukturierte programmiersprachliche Beschreibung nebenläufiger Prozesse
  - in der Tradition blockorientierter Sprachen wie z.B. CSP [5]
  - mehrere Blöcke (derselben Prozedur) können nebenläufig ausgeführt werden
  - „Klammerkonstrukte“ identifizieren die Blöcke: $[\ B_1 \parallel B_2 \parallel \ldots \parallel B_n\ ]$
- der Kontrollfluss innerhalb eines Programms (einer Prozedur) wird aufgespalten
  - jeder einzelne nebenläufige Block definiert einen *Kindprozess*
  - es gibt einen *Elternprozess*, der den nebenläufigen Block aktiviert
  - ein nebenläufiger Block terminiert, wenn jeder Kindprozess terminiert ist
  - terminiert der Block, geht die Kontrolle zum Elternprozess zurück
- der Ablauf wird auch als **Aufspaltung** und **Sammlung** bezeichnet [4]

Alternative (2) — Prozessabzweigung

- die Abzweigung geschieht durch `fork` und impliziert einen Prozeduraufruf [3]
  - verschiedentlich auch als „**asynchroner Prozeduraufruf**“ bezeichnet
  - die „asynchrone Prozedur“ wird durch einen eigenen Kindprozess ausgeführt
  - der Rückgabewert von `fork` dient der Prozessidentifizierung
- mittels join kann auf die Terminierung von Kindprozessen gewartet werden
  - entweder auf alle oder, über die Prozessidentifizierung, auf einen speziellen
  - ist der Kindprozess bereits terminiert, wartet der Elternprozess nicht
- fork/join kann an *jeder* Stelle im Programm verwendet werden +/−
  - sehr flexible Abläufe bzw. unregelmäßige Schachtelungen sind möglich

Sprachunterstützung? Nein danke . . .

. . . **Ja bitte!** Aber auch nur dann, wenn dadurch keine Entwurfs- und Implementierungsentscheidungen vorweg genommen werden [8].

- Java-*Threads*, z.B., sind deshalb eher ungeeignet zum Betriebssystembau
  - nicht nur, dass sie vergleichsweise zu „schwergewichtig“ sind
  - viel kritischer: mit ihnen sind *Scheduling-Strategien* vorgegeben
  - ähnlich verhält es sich mit den sogenannten *threads packages*
  - die Konzepte können nicht bzw. nur sehr schwer verschlankt werden
- die richtige Systemprogrammiersprache ist noch nicht erfunden worden

Zusammenfassung

- Programmfäden werden auf Basis von Koroutinen implementiert
  - ein Faden ist auch eine Koroutine, aber eine Koroutine ist kein Faden
  - Grundgerüst der Implementierung bilden zwei Elementaroperationen
- eine Koroutine ist mehr als eine Routine, sie hat einen eigenen Laufzeitkontext
  - der Aktivierungsblock ist automatisch Bestandteil des Koroutinenkontextes
  - weiteres Bestandteil ist der noch nicht berücksichtigte Prozessorstatusanteil
- Koroutinen müssen kooperativ sein, damit Nebenläufigkeit funktionieren kann

Referenzen

[1] *i860 Processor Manual*. Intel Corporation, 1989.

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

[3] J. B. Dennis and E. C. van Horn. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 11(5):143–155, 1966.

[4] Deutsches Institut für Normung. *Informationsverarbeitung — Begriffe*. DIN 43000. Beuth-Verlag, Berlin, Köln, 1985.

[5] E. W. Dijkstra. Cooperating Sequential Processes. Technical report, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 1965. (Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996).

[6] R. G. Herrtwich and G. Hommel. *Kooperation und Konkurrenz — Nebenläufige, verteilte und echtzeitabhängige Programmsysteme*. Springer-Verlag, 1989. ISBN 3-540-51701-4.

[7] D. L. Parnas. Some Hypotheses About the "Uses" Hierarchy for Operating Systems. Technical Report BS I 75/2, TH Darmstadt, 1975.

[8] W. Schröder-Preikschat. Operating-System Engineering. http://www4.informatik.uni-erlangen.de, 2002.