

Betriebssysteme (BS)

VL 7 – Koroutinen und Fäden

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 09 – 02. Dezember 2009

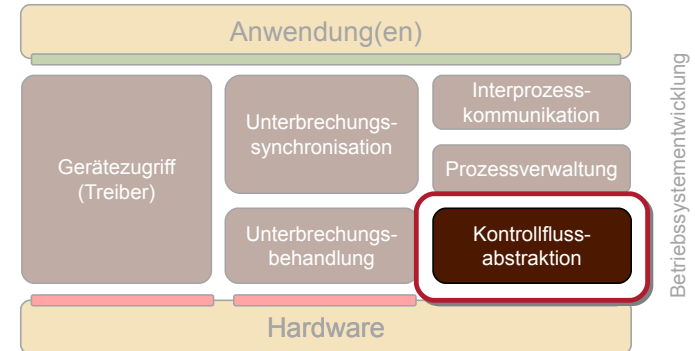


Agenda

- Motivation: Quasi-Parallelität
 - Einige Versuche
- Grundbegriffe
 - Routine und Kontrollfluss
 - Koroutine, Kontrollfluss und Programmfaden
 - asymmetrisches und symmetrisches Fortsetzungsmodell
- Implementierung von Koroutinen
 - Fortsetzungen
 - Elementaroperationen
- Ausblick
 - Koroutinen als Hilfsmittel für das BS
 - Mehrfädigkeit
- Zusammenfassung



Überblick: Einordnung dieser VL



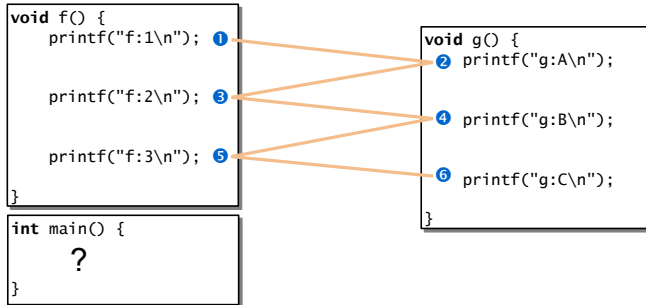
Agenda

- **Motivation: Quasi-Parallelität**
 - **Einige Versuche**
- Grundbegriffe
 - Routine und Kontrollfluss
 - Koroutine, Kontrollfluss und Programmfaden
 - asymmetrisches und symmetrisches Fortsetzungsmodell
- Implementierung von Koroutinen
 - Fortsetzungen
 - Elementaroperationen
- Ausblick
 - Koroutinen als Hilfsmittel für das BS
 - Mehrfädigkeit
- Zusammenfassung



Motivation: Quasi-Parallelität

- **Gegeben:** Funktionen **f** und **g**
- **Ziel:** **f** und **g** sollen „versetzt“ ablaufen



Motivation: Quasi-Parallelität – Versuch 1

```

void f() {
    printf("f:1\n");

    printf("f:2\n");

    printf("f:3\n");
}

int main() {
    f();
    g();
}
    
```

```

void g() {
    printf("g:A\n");

    printf("g:B\n");

    printf("g:C\n");
}
    
```

```

Tohmann@fai48a>gcc routine.c -o routine
Tohmann@fai48a>./routine
f:1
f:2
f:3
g:A
g:B
g:C
    
```

Das funktioniert so natürlich nicht...



Motivation: Quasi-Parallelität – Versuch 2

```

void f() {
    printf("f:1\n");
    g();

    printf("f:2\n");
    g();

    printf("f:3\n");
    g();
}

int main() {
    f();
}
    
```

```

void g() {
    printf("g:A\n");

    printf("g:B\n");

    printf("g:C\n");
}
    
```

```

Tohmann@fai48a>gcc routine.c -o routine
Tohmann@fai48a>./routine
f:1
g:A
g:B
g:C
f:2
...
    
```

So geht es wohl auch nicht...



Motivation: Quasi-Parallelität – Versuch 3

```

void f() {
    printf("f:1\n");
    g();

    printf("f:2\n");
    g();

    printf("f:3\n");
    g();
}

int main() {
    f();
}
    
```

```

void g() {
    printf("g:A\n");
    f();

    printf("g:B\n");
    f();

    printf("g:C\n");
    f();
}
    
```

```

Tohmann@fai48a>gcc routine.c -o routine
Tohmann@fai48a>./routine
f:1
g:A
f:1
g:A
...
Segmentation fault
    
```

So schon gar nicht!



Motivation: Quasi-Parallelität – Versuch 4

```
void f_start() {
    printf("f:1\n");
    f = &&l1; goto *g;

l1: printf("f:2\n");
    f = &&l2; goto *g;

l2: printf("f:3\n");
    goto *g;
}

void (*volatile f)();
void (*volatile g)();

int main() {
    f = f_start;
    g = g_start;
    f();
}
```

```
void g_start() {
    printf("g:A\n");
    g = &&l1; goto *f;

l1: printf("g:B\n");
    g = &&l2; goto *f;

l2: printf("g:C\n");
    exit(0);
}
```

Und so?

Motivation: Quasi-Parallelität – Versuch 4

```
void f_start() {
    printf("f:1\n");
    f = &&l1; goto *g;

l1: printf("f:2\n");
    f = &&l2; goto *g;

l2: printf("f:3\n");
    goto *g;
}

void (*volatile f)();
void (*volatile g)();

int main() {
    f = f_start;
    g = g_start;
    f();
}
```

```
void g_start() {
    printf("g:A\n");
    g = &&l1; goto *f;

l1: printf("g:B\n");
    g = &&l2; goto *f;

l2: printf("g:C\n");
    exit(0);
}
```

```
lohmann@fau148a>gcc-2.95 -fomit-frame-
pointer -o coroutine coroutine.c
lohmann@fau148a>./coroutine
f:1
g:A
f:2
g:B
f:3
g:C
```

Klappt!

Motivation: Quasi-Parallelität – Versuch 4

```
void f_start() {
    printf("f:1\n");
    f = &&l1; goto *g;

l1: printf("f:2\n");
    f = &&l2; goto *g;

l2: printf("f:3\n");
    goto *g;
}

void (*volatile f)();
void (*volatile g)();

int main() {
    f = f_start;
    g = g_start;
    f();
}
```

```
void g_start() {
    printf("g:A\n");
    g = &&l1; goto *f;

l1: printf("g:\n");
    g = &&l2; goto *f;

l2: printf("g:C\n");
    exit(0);
}
```

```
lohmann@fau148a>gcc-2.95 -fomit-frame-
pointer -o coroutine coroutine.c
lohmann@fau148a>./coroutine
f:1
g:A
f:2
g:B
f:3
g:C
```

Warum?

Quasi-Parallelität: Erstes Fazit

- Quasi-Parallelität zwischen zwei Funktions-Ausführungen kann nicht erreicht werden durch Funktionsaufrufe
 - einfache Funktionsaufrufe (Versuche 1 und 2)
 - → laufen immer komplett durch
 - rekursive Funktionsaufrufe (Versuch 3)
 - → dito, deshalb Endlosrekursion und Stapelüberlauf
- Wir brauchen Funktionen, die „während der Ausführung“ verlassen und wieder betreten werden können
 - also ungefähr so wie in Versuch 4
 - PC der Ausführungen wird gespeichert, mit goto wieder aufgenommen
 - aber bitte ohne die damit einhergehenden Probleme
 - direkte Sprünge aus und in Funktionen sind in C undefiniert! (goto über Zeiger ist ein gcc-„Feature“)
 - Zustand besteht aus mehr als dem PC – Was ist mit Registern, Stapel?

Agenda

- Motivation: Quasi-Parallelität
 - Einige Versuche
- **Grundbegriffe**
 - **Routine und Kontrollfluss**
 - **Koroutine, Kontrollfluss und Programmfaden**
 - **asymmetrisches und symmetrisches Fortsetzungsmodell**
- Implementierung von Koroutinen
 - Fortsetzungen
 - Elementaroperationen
- Ausblick
 - Koroutinen als Hilfsmittel für das BS
 - Mehrfädigkeit
- Zusammenfassung



Grundbegriffe: Routine, Kontrollfluss

- Routinen-Kontrollflüsse werden erzeugt, gesteuert und zerstört mit speziellen **Elementaroperationen**
 - $\langle f \rangle$ **call** g (Ausführung $\langle f \rangle$ erreicht Anweisung **call** g)
 - **erzeugt** neue Ausführung $\langle g \rangle$ von g
 - **suspendiert** die Ausführung $\langle f \rangle$
 - **aktiviert** die Ausführung $\langle g \rangle$ (erste Anweisung wird ausgeführt)
 - $\langle g \rangle$ **ret** (Ausführung $\langle g \rangle$ erreicht Anweisung **ret**)
 - **zerstört** die Ausführung $\langle g \rangle$
 - **reaktiviert** die Ausführung des erzeugenden Kontrollflusses



Grundbegriffe: Routine, Kontrollfluss

- **Routine**: eine endliche Sequenz von Anweisungen
 - z.B. die Funktion f
 - Sprachmittel fast aller Programmiersprachen
 - wird ausgeführt durch (Routinen-)Kontrollfluss
- **(Routinen-)Kontrollfluss**: eine (Routinen-)Ausführung
 - **Ausführung** und **Kontrollfluss** sind synonyme Begriffe
 - z.B. die Ausführung $\langle f \rangle$ der Funktion f
 - beginnt bei Aktivierung mit der ersten Anweisung von f

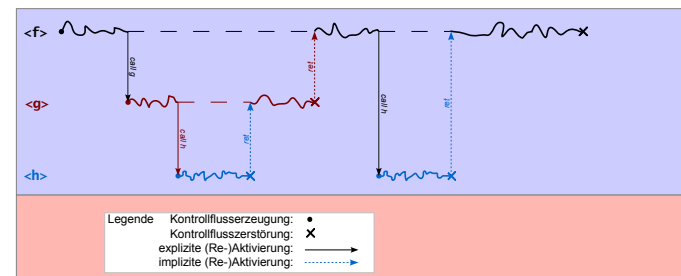
Zwischen Routinen und Ausführungen besteht eine Art Schema-Instanz-Relation. Zur klaren Unterscheidung werden Ausführungen deshalb hier in spitzen Klammern gesetzt:

$\langle f \rangle$, $\langle f' \rangle$, $\langle f'' \rangle$, bezeichnen Ausführungen der Funktion f .



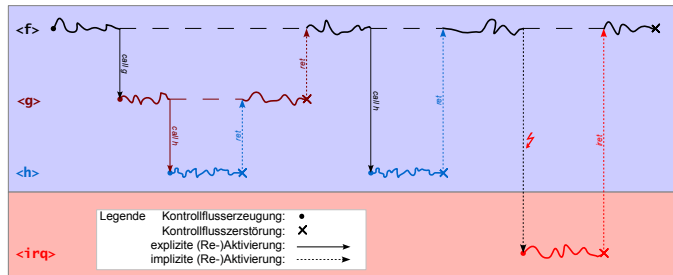
Routinen → asymmetrisches Fortsetzungsmodell

- Routinen-Kontrollflüsse bilden eine **Fortsetzungshierarchie**
 - Vater-Kind-Relation zwischen Erzeuger und Erzeugtem
- Aktivierte Kontrollflüsse werden nach **LIFO** fortgesetzt.
 - Der zuletzt aktivierte Kontrollfluss terminiert immer zuerst.
 - Vater wird erst fortgesetzt, wenn Kind terminiert



Routinen → asymmetrisches Fortsetzungsmodell

- Das gilt auch bei **Unterbrechungen**
 - `<f>` \neq `<irq>` wie `call`, nur implizit
 - `<irq>` `iret` wie `ret`
- Unterbrechungen können als **implizit** erzeugte und aktivierte Routinen-Ausführungen verstanden werden.



Grundbegriffe: Koroutine

- **Koroutine** (engl. *Coroutine*): verallgemeinerte Routine
 - erlaubt zusätzlich: expliziten Austritt und Wiedereintritt
 - Sprachmittel einiger Programmiersprachen
 - z.B. Modula-2, Simula-67, Stackless Python
 - wird ausgeführt durch Koroutinen-Kontrollfluss
- **Koroutinen-Kontrollfluss**: eine Koroutinen-Ausführung
 - Kontrollfluss mit eigenem, unabhängigem Zustand
 - Stapel, Register
 - Im Prinzip ein eigenständiger Faden (engl. *Thread*) – **dazu später mehr**

Koroutinen und Koroutinen-Kontrollflüsse stehen ebenfalls in einer Schema-Instanz-Relation.

In der Literatur ist diese Unterscheidung unüblich. Koroutinen-Kontrollflüsse werden häufig ebenfalls als Koroutinen bezeichnet.

Grundbegriffe: Koroutine

- Koroutinen-Kontrollflüsse werden erzeugt, gesteuert und zerstört über zusätzliche **Elementaroperationen**
 - **create** `g`
 - erzeugt neue Koroutinen-Ausführung `<g>` von `g`
 - `<f>` **resume** `<g>`
 - suspendiert die Koroutinen-Ausführung `<f>`
 - (re-)aktiviert die Koroutinen-Ausführung `<g>`
 - **destroy** `<g>`
 - zerstört die Koroutinen-Ausführung `<g>`

Grundbegriffe: Koroutine

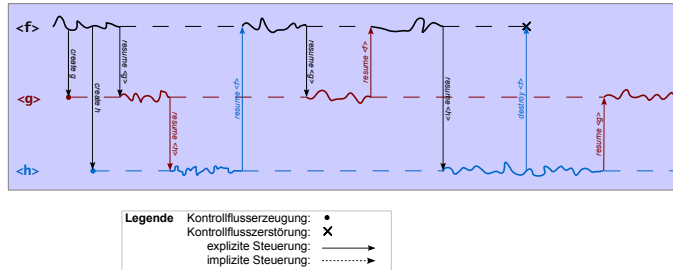
- Koroutinen-Kontrollflüsse werden erzeugt, gesteuert und zerstört über zusätzliche **Elementaroperationen**
 - **create** `g`
 - erzeugt neue Koroutinen-Ausführung `<g>` von `g`
 - `<f>` **resume** `<g>`
 - suspendiert die Koroutinen-Ausführung `<f>`
 - (re-)aktiviert die Koroutinen-Ausführung `<g>`
 - **destroy** `<g>`
 - zerstört die Koroutinen-Ausführung `<g>`

Unterschied zu Routinen-Kontrollflüssen:

Aktivierung und Reaktivierung sind zeitlich entkoppelt von Erzeugung und Zerstörung

Koroutinen → symmetrisches Fortsetzungsmodell

- Koroutinen-Kontrollflüsse bilden eine **Fortsetzungsfolge**
 - Koroutinenzustand bleibt über Ein-/Austritte hinweg erhalten
- Alle Koroutinen-Kontrollflüsse sind **gleichberechtigt**
 - kooperatives Multitasking
 - Fortsetzungsreihenfolge ist beliebig



Koroutinen und Programmfäden

- Koroutinen-Kontrollflüsse werden oft auch bezeichnet als
 - kooperative **Fäden** (engl. *cooperative Threads*)
 - **Fasern** (engl. *Fibers*)
- Das ist *im Prinzip* richtig, die Begriffe entstammen jedoch aus verschiedenen Welten
 - Koroutinen-Unterstützung ist historisch (eher) ein **Sprachmerkmal**
 - Mehrfädigkeit ist historisch (eher) ein **Betriebssystemmerkmal**
 - Die Grenzen sind fließend
 - Sprachfunktion – (Laufzeit-)bibliothekfunktion – Betriebssystemfunktion
- Wir verstehen Koroutinen als *technisches* Mittel
 - um Mehrfädigkeit im BS zu **implementieren**
 - insbesondere später auch nicht-kooperative Fäden

Agenda

- Motivation: Quasi-Parallelität
 - Einige Versuche
- Grundbegriffe
 - Routine und Kontrollfluss
 - Koroutine, Kontrollfluss und Programmfaden
 - asymmetrisches und symmetrisches Fortsetzungsmodell
- **Implementierung von Koroutinen**
 - **Fortsetzungen**
 - **Elementaroperationen**
- Ausblick
 - Koroutinen als Hilfsmittel für das BS
 - Mehrfädigkeit
- Zusammenfassung

Implementierung: Fortsetzungen

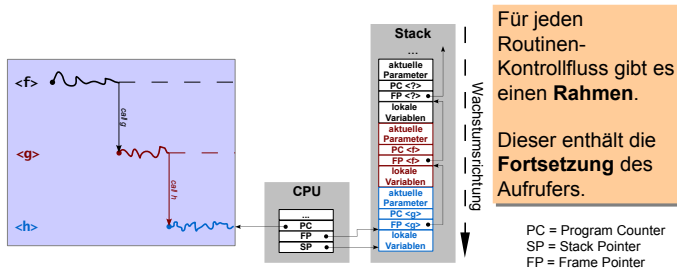
- **Fortsetzung** (engl. *Continuation*): Rest einer Ausführung
 - Eine Fortsetzung ist ein **Objekt**, das einen **suspendierten Kontrollfluss** repräsentiert.
 - Programmzähler, Register, lokale Variablen, ...
 - kurz: gesamter Kontrollflusszustand
 - wird benötigt, um den Kontrollfluss zu reaktivieren

Continuations sind ursprünglich entstanden als ein Beschreibungsmittel in der **denotationalen Semantik**.

Sprachen wie Haskell oder Scheme bieten *Continuations* als eigenes Sprachmittel an.

Routinen → asymmetrisches Fortsetzungsmodell

- Routinen-Fortsetzungen werden auf **dem Stapel** instantiiert
 - in Form von **Stapel-Rahmen**, erzeugt und zerstört durch
 - **Übersetzer** (und CPU) bei **call, ret**
 - **Kopplungsroutine** (und CPU) bei **Unterbrechung, iret**
 - Stapel wird durch die Hardware (CPU-Stapel) bereitgestellt
 - Anweisungen wie call, ret, push, pop verwenden diesen implizit



Koroutinen → symmetrisches Fortsetzungsmodell

- Ein Koroutinen-Kontrollfluss benötigt einen eigenen Stapel
 - für lokale Variablen - sind Teil des Zustands
 - für Subroutinen-Aufrufe - darauf wollen wir nicht ganz verzichten
 - während der Ausführung ist dieser Stapel der CPU-Stapel

Koroutinen-Kontrollflüsse können demnach Routinen-Kontrollflüsse auf ihrem Stapel erzeugen und aktivieren!

Koroutinen → symmetrisches Fortsetzungsmodell

- Ein Koroutinen-Kontrollfluss benötigt einen eigenen Stapel
 - für lokale Variablen - sind Teil des Zustands
 - für Subroutinen-Aufrufe - darauf wollen wir nicht ganz verzichten
 - während der Ausführung ist dieser Stapel der CPU-Stapel
- **Ansatz:** Koroutinen-Fortsetzungen werden als **Stapel-Rahmen** auf **ihrem Stapel** instantiiert
 - Ein Kontrollfluss-Kontext wird **repräsentiert** durch den Stapel
 - Das oberste Stapelelement enthält immer die Fortsetzung
 - Kontrollfluss-Wechsel entspricht Stapelwechsel und „Rücksprung“

Im Prinzip werden bei diesem Ansatz Koroutinen-Fortsetzungen mit Hilfe von Routinen-Fortsetzungen implementiert.

Implementierung: *resume*

- **Aufgabe:** Koroutinen-Kontrollfluss wechseln

```
// Typ für Stapelzeiger (Stapel ist Feld von void*)
typedef void** SP;

extern „C“ void resume( SP& from_sp, SP& to_sp ) {
    /* aktueller Stapel-Rahmen ist Fortsetzung des zu
       suspendierenden Kontrollflusses (Aufrufer von resume) */

    < sichere CPU-Stapelzeiger in from_sp >
    < lade CPU-Stapelzeiger aus to_sp >

    /* aktueller Stapel-Rahmen ist Fortsetzung des zu
       reaktivierenden Kontrollflusses */
} // Rücksprung
```

Implementierung: *resume*

- **Aufgabe:** Koroutinen-Kontrollfluss wechseln

```
// Typ für Stapelzeiger (Stapel ist Feld von void*)
typedef void** SP;

extern „C“ void resume( SP& from_sp, SP& to_sp ) {
    /* aktueller Stapel-Rahmen ist Fortsetzung des zu
       suspendierenden Kontrollflusses (Aufrufer von resume) */

    < sichere CPU-Stapelzeiger in from_sp >
    < lade CPU-Stapelzeiger aus to_sp >

    /* aktueller
       reaktivierender
    } // Rücksprung
```

Problem: nicht-flüchtige Register

Der Stapel-Rahmen enthält **keine nicht-flüchtigen Register**, da der Aufrufer davon ausgeht, dass diese nicht verändert werden.

Wir springen jedoch **in einen anderen Aufrufer** zurück!

Implementierung: *resume*

- **Problem:** nicht-flüchtige Register
 - Stapel-Rahmen enthält keine nicht-flüchtigen Register
 - also müssen diese extra gesichert und restauriert werden
- Implementierungsvarianten
 - nicht-flüchtige Register werden in eigener Struktur gesichert
 - oder einfach als „lokale Variablen“ auf dem Stapel:

```
extern „C“ void resume( SP& from_sp, SP& to_sp ) {
    /* aktueller Stapel-Rahmen ist Fortsetzung des zu
       suspendierenden Kontrollflusses (Aufrufer von resume) */

    < lege nicht-flüchtige Register auf den Stapel >
    < sichere CPU-Stapelzeiger in from_sp >
    < lade CPU-Stapelzeiger aus to_sp >
    < hole nicht-flüchtige Register vom Stapel >
    /* aktueller Stapel-Rahmen ist Fortsetzung des zu
       reaktivierenden Kontrollflusses */

} // Rücksprung
```

Implementierung: *resume*

- Implementierung von *resume* ist architekturabhängig
 - Aufbau der Stapel-Rahmen
 - nicht-flüchtige Register
 - Wachstumsrichtung des Stapels
- Außerdem muss man Register bearbeiten → **Assembler**

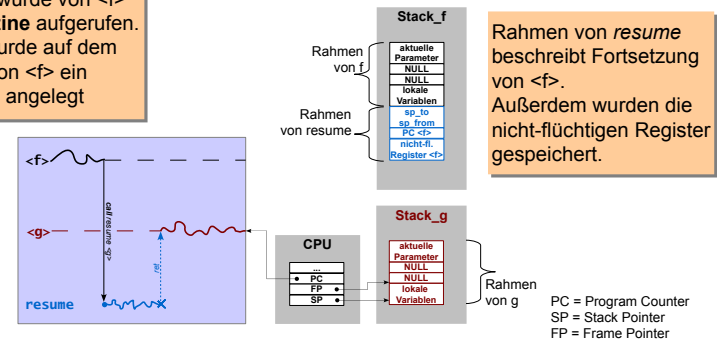
Beispiel Motorola 68000

```
// extern „C“ void resume( SP& sp_from, SP& sp_to )
resume:
    move.l 4(sp), a0          // a0 = &sp_from
    move.l 8(sp), a1          // a1 = &sp_to
    movem.l d2-d7/a2-a6, -(sp) // nf-Register auf den Stapel
    move.l sp, (a0)           // sp_from = sp
    move.l (a1), sp           // sp = sp_to
    movem.l (sp)+, d2-d7/a2-a6 // hole nf-Register vom Stapel
    rts                       // „Rücksprung“
```

Beispiel: Verwendung von *resume*

- Koroutinen-Kontrollfluss <f> übergab an <g>
 - <f> ist suspendiert, <g> ist aktiv

resume wurde von <f> als **Routine** aufgerufen. Dabei wurde auf dem Stapel von <f> ein Rahmen angelegt



Rahmen von *resume* beschreibt Fortsetzung von <f>. Außerdem wurden die nicht-flüchtigen Register gespeichert.

Implementierung: *create*

- **Aufgabe:** Koroutinen-Kontrollfluss <start> erzeugen
 - Wir brauchen
 - **Stapelspeicher** (irgendwo, global) `static void* stack_start[256];`
 - einen **Stapelzeiger** `SP sp_start = &stack_start[256];`
 - eine **Startfunktion** `void start(void* param){...}`
 - **Parameter** für die Startfunktion
 - Koroutinen-Kontrollfluss wird suspendiert erzeugt
 - Stapel repräsentiert den Kontext
 - Erst durch ein *resume* soll die Ausführung beginnen
- **Ansatz:** *create* erzeugt zwei Stapel-Rahmen
 - so als hätte die Startfunktion schon einmal *resume* aufgerufen:
 - den Rahmen der Startfunktion (erzeugt vom „virtuellen Aufrufer“)
 - den Rahmen von *resume* (enthält Fortsetzung in der Startfunktion)
 - Erstes *resume* macht „Rücksprung“ an den Beginn der Startfunktion

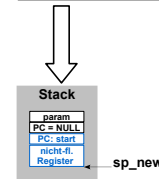


Implementierung: *create*

Beispiel Motorola 68000

```
void create( SP& sp_new, void (*start)(void*), void* param) {
    *(&sp_new) = param; // Parameter von Startfunktion
    *(&sp_new) = 0;     // Aufrufer (gibt es nicht!)

    *(&sp_new) = start; // Startadresse
    sp_new -= 11;       // nicht-flüchtige Register (Werte egal)
}
```

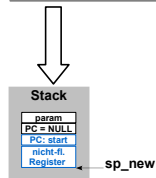


Implementierung: *create*

Beispiel Motorola 68000

```
void create( SP& sp_new, void (*start)(void*), void* param) {
    *(&sp_new) = param; // Parameter von Startfunktion
    *(&sp_new) = 0;     // Aufrufer (gibt es nicht!)

    *(&sp_new) = start; // Startadresse
    sp_new -= 11;       // nicht-flüchtige Register (Werte egal)
}
```



Da der Rücksprung an den **Anfang** einer Funktion erfolgt, sind die Rahmen sehr einfach aufgebaut.

Zu diesem Fortsetzungspunkt hat eine Funktion

- noch keinen lokalen Variablen (und somit auch keinen FP) auf den Stapel gelegt
- keine Parameter (für resume) auf den Stapel gelegt
- keine Annahmen über die Inhalte von nf-Registern



Implementierung: *destroy*

- **Aufgabe:** Koroutinen-Kontrollfluss zerstören
- **Ansatz:** Kontrollfluss-Kontext freigeben
 - entspricht Freigabe der Kontextvariablen (Stapelzeiger)
 - Stapelspeicher kann anschließend anderweitig verwendet werden

Das ist wenigstens mal einfach :-)



Agenda

- Motivation: Quasi-Parallelität
 - Einige Versuche
- Grundbegriffe
 - Routine und Kontrollfluss
 - Koroutine, Kontrollfluss und Programmfaden
 - asymmetrisches und symmetrisches Fortsetzungsmodell
- Implementierung von Koroutinen
 - Fortsetzungen
 - Elementaroperationen
- **Ausblick**
 - **Koroutinen als Hilfsmittel für das BS**
 - **Mehrfädigkeit**
- Zusammenfassung



Ausblick: Betriebssystemfäden

- Koroutinen sind (eigentlich) ein Sprachkonzept
 - Multitasking auf der Sprachebene
 - wir haben es hier für C „nachgerüstet“
 - Kontextwechsel erfordert keine Systemprivilegien (muss also auch nicht zwingend im BS-Kern erfolgen)
- Voraussetzung für Multitasking ist jedoch: **Kooperation**
 - Anwendungen müssen als Koroutinen implementiert werden
 - Anwendungen müssen sich gegenseitig kennen
 - Anwendungen müssen sich gegenseitig aktivieren

Für den uneingeschränkten Mehrprogramm-Betrieb ist das **unrealistisch!**



Ausblick: Betriebssystemfäden

- **Alternative:** „Kooperationsfähigkeit“ als Aufgabe des Betriebssystems auffassen
- **Ansatz:** Anwendungen „unbemerkt“ als eigenständige Fäden ausführen
 - **BS** sorgt für die **Erzeugung** der Koroutinen-Kontrollflüsse
 - jede Anwendung wird als Routine aus einer **BS-Koroutine** aufgerufen
 - so dass indirekt jede Anwendung als Koroutine implementiert ist
 - **BS** sorgt für die **Suspendierung** laufender Koroutinen-Kontrollflüsse
 - so dass Anwendungen nicht kooperieren müssen
 - erfordert einen **Verdrängungsmechanismus**
 - **BS** sorgt für die **Auswahl** des nächsten Koroutinen-Kontrollflusses
 - so dass Anwendungen sich nicht gegenseitig kennen müssen
 - erfordert einen **Scheduler**



Ausblick: Betriebssystemfäden

- **Alternative:** „Kooperationsfähigkeit“ als Aufgabe des Betriebssystems auffassen
- **Ansatz:** Anwendungen „unbemerkt“ als eigenständige Fäden ausführen
 - **BS** sorgt für die **Erzeugung** der Koroutinen-Kontrollflüsse
 - jede Anwendung wird als Routine aus einer **BS-Koroutine** aufgerufen
 - so dass indirekt jede Anwendung als Koroutine implementiert ist
 - **BS** sorgt für die **Suspendierung** laufender Koroutinen-Kontrollflüsse
 - so dass Anwendungen nicht kooperieren müssen
 - erfordert einen **Verdrängungsmechanismus**
 - **BS** sorgt für die **Auswahl** des nächsten Koroutinen-Kontrollflusses
 - so dass Anwendungen sich nicht gegenseitig kennen müssen
 - erfordert einen **Scheduler**

Dazu mehr in der Übung und in der nächsten Vorlesung



Agenda

- Motivation: Quasi-Parallelität
 - Einige Versuche
- Grundbegriffe
 - Routine und Kontrollfluss
 - Koroutine, Kontrollfluss und Programmfaden
 - asymmetrisches und symmetrisches Fortsetzungsmodell
- Implementierung von Koroutinen
 - Fortsetzungen
 - Elementaroperationen
- Ausblick
 - Koroutinen als Hilfsmittel für das BS
 - Mehrfädigkeit
- **Zusammenfassung**



Zusammenfassung

- Unser Ziel war die Ermöglichung von „Quasi-Parallelität“
 - Funktionen „abwechselnd“ jeweils „ein wenig“ ausführen
 - Suspendierung und Reaktivierung von Funktions-Ausführungen
 - Begriff der Fortsetzung
- Routinen → asymmetrisches Fortsetzungsmodell
 - Ausführung in LIFO-Reihenfolge (und damit nicht „quasi-parallel“)
 - CPU und Übersetzer stellen die Elementaroperationen bereit
- Koroutinen → symmetrisches Fortsetzungsmodell
 - Ausführung in beliebiger Reihenfolge
 - erfordert eigenen Kontext: Register, Stapel
 - Elementaroperationen i.a. nicht durch CPU/Übersetzer bereitgestellt
- Fäden sind vom BS verwaltete Koroutinen

