

Betriebssysteme (BS)

VL 7 – Koroutinen und Fäden

Volkmar Sieh / Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 18 – 6. Dezember 2018

https://www4.cs.fau.de/Lehre/WS18/V_BS

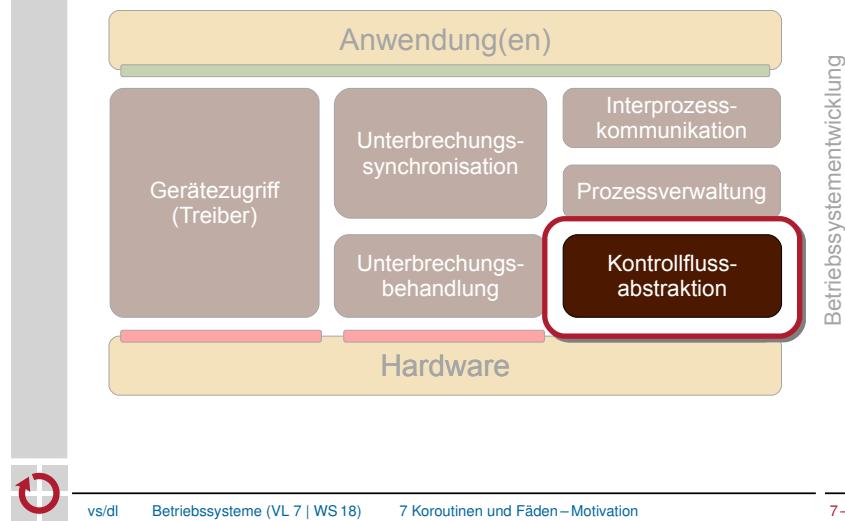


Agenda

- Motivation
- Grundbegriffe
- Implementierung
- Ausblick
- Zusammenfassung
- Referenzen



Überblick: Einordnung dieser VL



Betriebssystementwicklung

Agenda

- Motivation
- Einige Versuche
- Fazit
- Grundbegriffe
- Implementierung
- Ausblick
- Zusammenfassung
- Referenzen



Motivation: Quasi-Parallelität

```
void f() {
    printf("f:1\n");
    printf("f:2\n");
    printf("f:3\n");
}

void g() {
    printf("g:A\n");
    printf("g:B\n");
    printf("g:C\n");
}
```

```
int main() {
    ?
}
```

- **Gegeben:** Funktionen f() und g()
- **Ziel:** f() und g() sollen „verschränkt“ ablaufen

Im Folgenden einige Versuche...



Motivation: Quasi-Parallelität

```
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");
}
```

```
lohmann@faui48a>gcc routine.c -o routine
lohmann@faui48a>./routine
f:1
f:2
f:3
g:A
g:B
g:C
```

So funktioniert es natürlich nicht.



Motivation: Quasi-Parallelität

Versuch 2

```
void f() {
    printf("f:1\n");
    g();
    printf("f:2\n");
    g();
    printf("f:3\n");
    g();
}
```

```
void g() {
    printf("g:A\n");
    printf("g:B\n");
    printf("g:C\n");
}
```

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

```
lohmann@faui48a>gcc routine.c -o routine
lohmann@faui48a>./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();
}
```

```
void g() {
    printf("g:A\n");
    f();
    printf("g:B\n");
    f();
    printf("g:C\n");
    f();
}
```

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

```
lohmann@faui48a>gcc routine.c -o routine
lohmann@faui48a>./routine
f:1
g:A
f:1
g:A
...
Segmentation fault
```

So schon gar nicht!



Versuch 1

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 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);  
}
```

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

Und so?



vs/dl Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Motivation

7-5

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 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);  
}
```

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

```
lohmann@faui48a>gcc-2.95 -fomit-frame-pointer -o coroutine coroutine.c  
lohmann@faui48a>/coroutine  
f:1  
g:A  
f:2  
g:B  
f:3  
g:C
```

Klappt!



vs/dl Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Motivation

7-5

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 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);  
}
```

Bitte nicht zu Hause nachmachen!

```
lohmann@faui48a>gcc-2.95 -fomit-frame-pointer -o coroutine coroutine.c  
lohmann@faui48a>/coroutine  
f:1  
g:A  
f:2  
g:B  
f:3  
g:C
```



vs/dl Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Motivation

7-5

Quasi-Parallelität: Feststellungen

- C/C++ bietet keine Bordmittel für „verschränkte“ Ausführung
 - einfache Funktionsaufrufe (Versuche 1 und 2)
 - laufen immer komplett durch (*run-to-completion*)
 - rekursive Funktionsaufrufe (Versuch 3)
 - dito, \sim Endlosrekursion und Stapelüberlauf
- Wir brauchen **Systemunterstützung**, um Kontrollflüsse „während der Ausführung“ verlassen und wieder betreten zu können
 - ungefähr so wie in Versuch 4
 - „Fortsetzungs“-PC wird gespeichert, mit goto wieder aufgenommen
 - aber bitte ohne die damit einhergehenden Probleme!
 - *computed gotos* aus Funktionen sind **undefined**
 - Zustand besteht aus mehr als dem PC – was ist mit **Registern, Stapel, ...**

Anmerkung: Aus Systemsicht („von unten“) würde der PC reichen!

- (PC) \leftrightarrow minimaler Kontrollflusszustand
- alles weitere ist letztlich eine Entwurfsentscheidung des **Compilers** \sim [UE1]
- wird in der Praxis jedoch durch Hardwarehersteller nahegelegt (ISA, ABI)



vs/dl Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Motivation

7-6

Agenda

Motivation

Grundbegriffe

- Routine und asymmetrisches Fortsetzungsmodell
- Koroutine und symmetrisches Fortsetzungsmodell

Implementierung

Ausblick

Zusammenfassung

Referenzen



vs/dl

Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Grundbegriffe

7-7

Grundbegriffe: Routine, Kontrollfluss

- Routinen-Kontrollflüsse werden erzeugt, gesteuert, und zerstört mit speziellen **Elementaroperationen**
 - <f> call g** (Ausführung <f> erreicht Anweisung call g)
 - erzeugt** neue Ausführung <g> von g
 - suspendiert** die Ausführung <f>
 - aktiviert** die Ausführung <g>
(→ erste Anweisung wird ausgeführt)
 - <g> ret** (Ausführung <g> erreicht Anweisung ret)
 - zerstört** die Ausführung <g>
 - reaktiviert** die Ausführung des Vater-Kontrollflusses (z. B. <f>)



vs/dl

Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Grundbegriffe

7-9

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 Routine in Ausführung

- Ausführung** und **Kontrollfluss** sind synonome Begriffe
 - z. B. die Ausführung <f> der Funktion f
 - beginnt bei Aktivierung mit der ersten Anweisung von f

Zwischen **Routinen** und **Ausführungen** besteht eine **Schema–Instanz Relation**. Zur klaren Unterscheidung werden die Instanzen (→ Ausführungen) deshalb hier in spitzen Klammern gesetzt:

<f>, <f'>, <f''> sind Ausführungen von f .



vs/dl

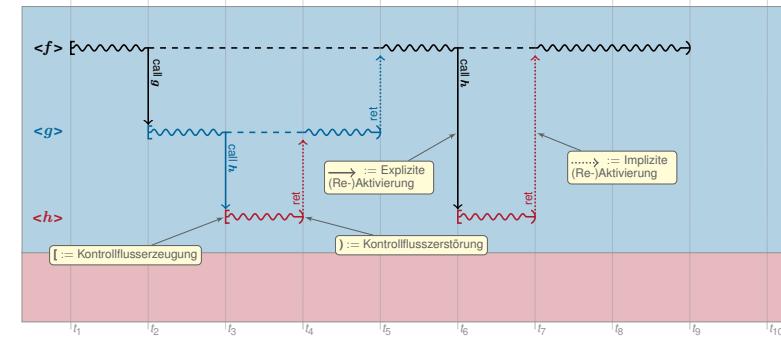
Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Grundbegriffe

7-8

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



vs/dl

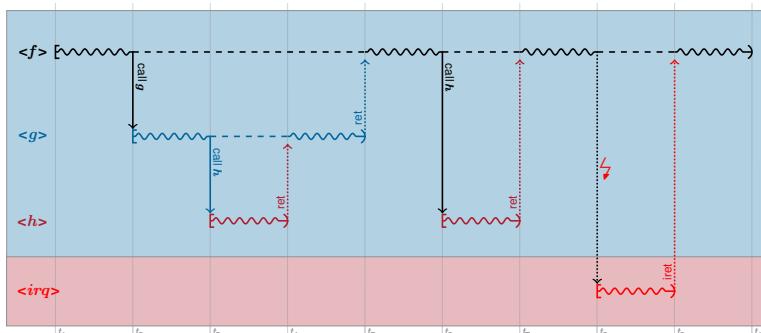
Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Grundbegriffe

7-10

Routinen \mapsto asymmetrisches Fortsetzungsmodell

- Das gilt auch bei Unterbrechungen
 - $<f>$ \downarrow irq ist wie `call`, nur implizit
- Unterbrechungen können als **implizit** erzeugte und aktivierte Routinen-Ausführungen verstanden werden



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: [SP, C 10-8]

Aktivierung und Reaktivierung sind **zeitlich entkoppelt** von Erzeugung und Zerstörung.
 \leadsto Koroutinen sind **echt mächtiger** als Routinen.



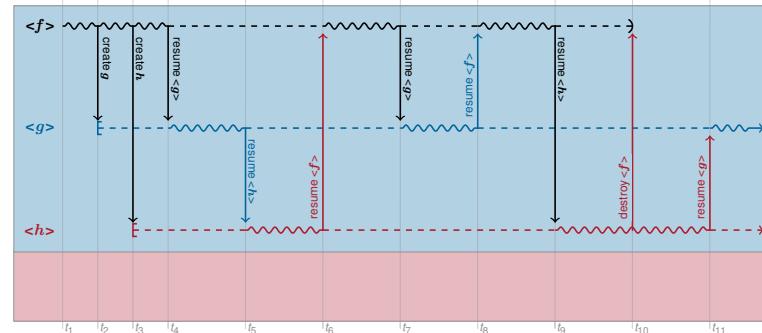
Grundbegriffe: Koroutine

- Koroutine (engl. Coroutine):** verallgemeinerte Routine [1]
 - 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 Koroutine in Ausführung
 - Kontrollfluss mit eigenem, unabhängigem Zustand
 - mindestens Programmzähler (PC)
 - zusätzlich je nach (zu unterstützendem) **Compiler / ABI / ISA:** weitere Register, Stapel, ...
 - 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 \leadsto Koroutinen-Kontrollflüsse werden (vereinfacht) ebenfalls als Koroutinen bezeichnet.



Koroutinen \mapsto symmetrisches Fortsetzungsmodell

- Koroutinen-Kontrollflüsse bilden eine **Fortsetzungsfolge**
 - Koroutenzustand bleibt über Ein-/Austritte hingewehrt 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 **Sprach**merkmal
 - Mehrfädigkeit ist historisch (eher) ein **Betriebssystem**merkmal
 - Die Grenzen sind fließend
 - Sprachfunktion — (Laufzeit-)Bibliotheksfunktion — Betriebssystemfunktion
- Wir verstehen Koroutinen als **technisches** Konzept
 - um Mehrfädigkeit im BS zu implementieren
 - insbesondere später auch nicht-kooperative Fäden



vs/dl

Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Grundbegriffe

7–15

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

Anmerkung: Fortsetzungen

- Continuations sind ursprünglich entstanden als ein Beschreibungsmittel der **denotationalen Semantik** [3].
- Sprachen wie Haskell oder Scheme bieten Continuations als eigenes Sprachmittel an.



vs/dl

Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Implementierung

7–17

Agenda

- Motivation
- Grundbegriffe
- Implementierung
 - Fortsetzungen
 - Elementaroperationen
- Ausblick
- Zusammenfassung
- Referenzen



vs/dl

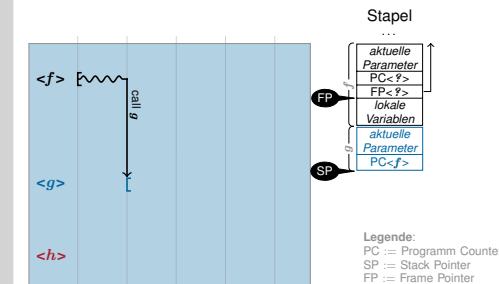
Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Implementierung

7–16

Routinen \mapsto asymmetrisches Fortsetzungsmodell

- Routinen-Fortsetzungen werden i. a. auf einem **Stapel** instantiiert
 - in Form von **Stapel-Rahmen**, erzeugt und zerstört durch
 - Compiler (explizit) und CPU (implizit) bei `call, ret`
 - Kopplungsfunktion (explizit) und CPU (implizit) bei `↳, iret`
 - Der Compiler verwendet dafür i. a. den CPU-Stapel
 - `call, ret, push, pop, ...` verwenden implizit den CPU-Stapel



Für jeden Routinen-Kontrollfluss legen CPU und Compiler einen Rahmen an. Dieser enthält die Fortsetzung des Aufrufers.

(vgl. auch [UE1, 07-8ff])



vs/dl

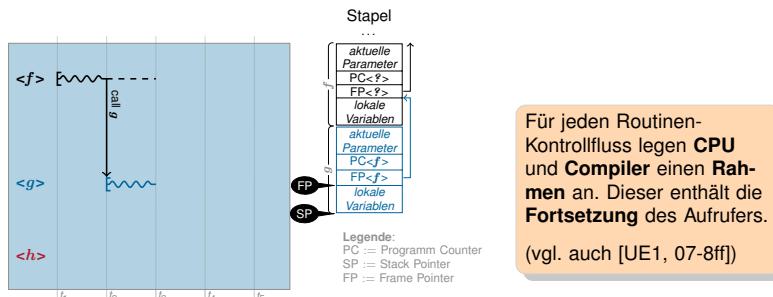
Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Implementierung

7–18

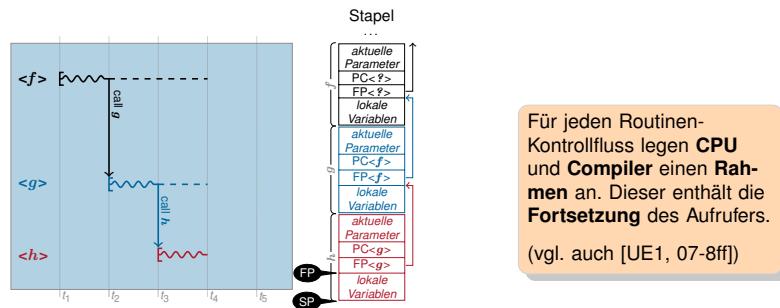
Routinen \mapsto asymmetrisches Fortsetzungsmodell

- Routinen-Fortsetzungen werden i. a. auf einem **Stapel** instantiiert
 - in Form von **Stapel-Rahmen**, erzeugt und zerstört durch
 - Compiler (explizit) und **CPU** (implizit) bei **call, ret**
 - Kopplungsfunktion (explizit) und **CPU** (implizit) bei **↳, iret**
 - Der Compiler verwendet dafür i. a. den CPU-Stapel
 - **call, ret, push, pop, ...** verwenden implizit den CPU-Stapel



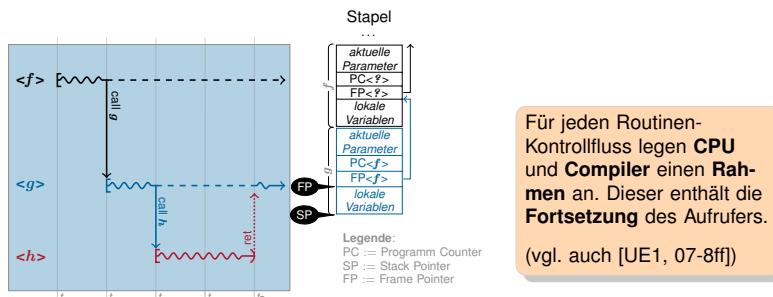
Routinen \mapsto asymmetrisches Fortsetzungsmodell

- Routinen-Fortsetzungen werden i. a. auf einem **Stapel** instantiiert
 - in Form von **Stapel-Rahmen**, erzeugt und zerstört durch
 - Compiler (explizit) und **CPU** (implizit) bei **call, ret**
 - Kopplungsfunktion (explizit) und **CPU** (implizit) bei **↳, iret**
 - Der Compiler verwendet dafür i. a. den CPU-Stapel
 - **call, ret, push, pop, ...** verwenden implizit den CPU-Stapel



Routinen \mapsto asymmetrisches Fortsetzungsmodell

- Routinen-Fortsetzungen werden i. a. auf einem **Stapel** instantiiert
 - in Form von **Stapel-Rahmen**, erzeugt und zerstört durch
 - Compiler (explizit) und **CPU** (implizit) bei **call, ret**
 - Kopplungsfunktion (explizit) und **CPU** (implizit) bei **↳, iret**
 - Der Compiler verwendet dafür i. a. den CPU-Stapel
 - **call, ret, push, pop, ...** verwenden implizit den CPU-Stapel



Koroutinen \mapsto symmetrisches Fortsetzungsmodell

- Koroutinen-Fortsetzungen werden i. a. nicht nativ unterstützt
- **Ansatz:** Koroutinen-Fortsetzungen durch **Routinen-Fortsetzungen** implementieren [2]
 - Ein **resume**-Aufruf sieht für den Compiler wie die Erzeugung und Aktivierung eines ganz normalen Routinen-Kontrollflusses aus.
 - Vor dem **ret** wird in **resume** jedoch intern der Koroutinen-Kontrollfluss gewechselt.
- **Folge:** Technisch gesehen, müssen wir das Routinen-Fortsetzungsmodell des **Compilers** bereitstellen
 - Registerverwendung \leadsto **nichtflüchtige Register** über Wechsel erhalten
 - Fortsetzungs-Stapel \leadsto **eigener Stapel** für jede Koroutinen-Instanz

Eine Koroutinen-Instanz wird durch ihren Fortsetzungs-Stapel repräsentiert

- während der Ausführung ist dieser Stapel der CPU-Stapel
- oberster Stapel-Rahmen enthält immer die Fortsetzung
- Koroutinen-Wechsel \rightarrow Stapel-Wechsel + **ret**



Implementierung: *resume*

Aufgabe: Koroutinen-Kontrollfluss wechseln

```
// Typ fuer 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 */

} // Ruecksprung
```

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!



vs/dl

Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Implementierung

7-20

Implementierung: *resume*

Problem: nicht-flüchtige Register

- Routinen-Fortsetzung enthält keine nicht-flüchtigen Register
- diese müssen explizit **gesichert** und **restauriert** werden

Viele Implementierungsvarianten sind denkbar

- nicht-flüchtige Register in eigener Struktur sichern (→ Übung)
- 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 */

} // Ruecksprung
```



vs/dl

Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Implementierung

7-21

Implementierung: *resume*

- Implementierung vom *resume* ist architekturabhängig
 - Aufbau der Stapel-Rahmen
 - nicht-flüchtige Register
 - Wachstumsrichtung des Stapsels
- 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                      // "Ruecksprung"
```



vs/dl

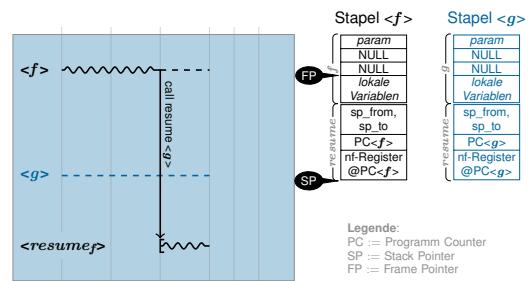
Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Implementierung

7-22

Beispiel: Verwendung von *resume*

Koroutinen-Kontrollfluss *<f>* übergibt an *<g>*:



- Koroutine *f* ist aktiv, Koroutine *g* ist suspendiert
- f* instantiiert den Routinen-Kontrollfluss *resume_g* und legt dazu Parameter (Stapelvariablen von *f* und *g*) sowie die Rücksprung-Adresse (→ Fortsetzung von *f*) auf den Stapel.
- resume_f* sichert nicht-flüchtige Register von *f* auf dem Stapel und eigenen SP in *sp_from*
- Wechselt des SP auf den Stapel von *g* (*sp_to*) → **Koroutinen-Wechsel**, nun läuft *resume_g*
- resume_g* holt nicht-flüchtige Register von *g* vom Stapel.
- Routinen-Kontrollfluss *resume_g* terminiert mit *ret*: *g* ist aktiv, *f* ist suspendiert



vs/dl

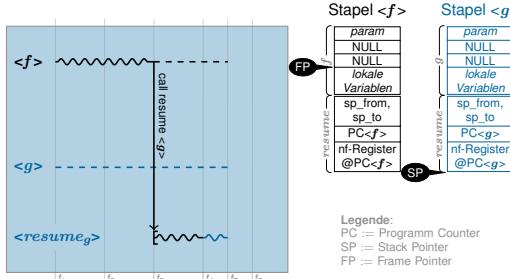
Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Implementierung

7-23

Beispiel: Verwendung von *resume*

Koroutinen-Kontrollfluss *<f>* übergibt an *<g>*:



1. Koroutine *<f>* ist aktiv, Koroutine *<g>* ist suspendiert
2. *<f>* instantiiert den Routinen-Kontrollfluss *<resume>* und legt dazu Parameter (Stapelvariablen von *<f>* und *<g>*) sowie die Rücksprung-Adresse (→ Fortsetzung von *<f>*) auf den Stapel.
3. *<resume>* sichert nicht-flüchtige Register von *<f>* auf dem Stapel und eigenen SP in *sp_from*.
4. Wechsel des SP auf den Stapel von *<g>* (*sp_to*) → **Koroutinen-Wechsel**, nun läuft *<resume>*
5. *<resume>* holt nicht-flüchtige Register von *<g>* vom Stapel.
6. Routinen-Kontrollfluss *<resume>* terminiert mit *ret*: *<g>* ist aktiv, *<f>* ist suspendiert



Implementierung: *create*

Aufgabe: Koroutinen-Kontrollfluss *<start>* erzeugen

■ Gebraucht wird dafür

1. **Stapelspeicher** (irgendwo, global)
2. **Stapelzeiger**
3. **Startfunktion**
4. **Parameter** für die Startfunktion

```
static void* stack_start[ 256 ];
SP sp_start = &stack_start[ 256 ];
void start( void* param ) ...
```

■ Koroutinen-Kontrollfluss wird suspendiert erzeugt

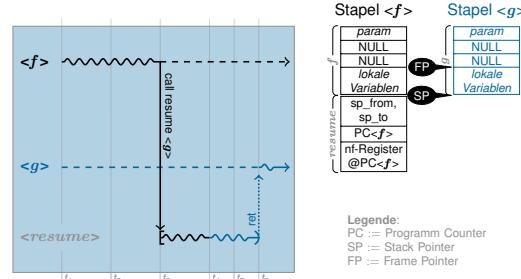
Ansatz: *create* erzeugt zwei Stapel-Rahmen

- so als hätte *<start>* bereits *resume* als **Routine** aufgerufen
 1. Rahmen der Startfunktion selber (erzeugt vom „virtuellen Aufrufer“)
 2. Rahmen von *resume* (enthält Fortsetzung in *<start>*)
- erstes *resume* macht „Rücksprung“ an den Beginn von *start*



Beispiel: Verwendung von *resume*

Koroutinen-Kontrollfluss *<f>* übergibt an *<g>*:



1. Koroutine *<f>* ist aktiv, Koroutine *<g>* ist suspendiert
2. *<f>* instantiiert den Routinen-Kontrollfluss *<resume>* und legt dazu Parameter (Stapelvariablen von *<f>* und *<g>*) sowie die Rücksprung-Adresse (→ Fortsetzung von *<f>*) auf den Stapel.
3. *<resume>* sichert nicht-flüchtige Register von *<f>* auf dem Stapel und eigenen SP in *sp_from*.
4. Wechsel des SP auf den Stapel von *<g>* (*sp_to*) → **Koroutinen-Wechsel**, nun läuft *<resume>*
5. *<resume>* holt nicht-flüchtige Register von *<g>* vom Stapel.
6. Routinen-Kontrollfluss *<resume>* terminiert mit *ret*: *<g>* ist aktiv, *<f>* ist suspendiert



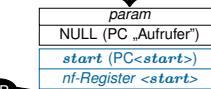
Implementierung: *create*

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)
}
```

ergibt



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

Zu diesem Fortsetzungspunkt hat ein Routinen-Kontrollfluss noch:

- keinen FP verwendet oder gesichert
- keine lokalen Variablen auf dem Stapel angelegt
- keine Annahmen über den Inhalt von nf-Registern



Implementierung: *destroy*

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

Das ist wenigstens mal einfach :-)



Ausblick: Betriebssystemfäden

- Koroutinen sind (eigentlich) ein **Sprachkonzept**
 - Multitasking auf Sprachebene
 - wir haben es hier für C/C++ (bzw. ein ABI) „nachgerüstet“
 - Kontextwechsel erfordert keine Systemprivilegien!
 - ~~ muss also **nicht zwingend** im BS-Kern erfolgen
- Voraussetzung für echtes Multitasking: **Kooperation**
 - Anwendungen müssen als Koroutinen implementiert sein
 - Anwendungen müssen sich gegenseitig kennen
 - Anwendungen müssen sich gegenseitig aktivieren
 - ...

Problem

Für uneingeschränkten Mehrprogramm-Betrieb ist das **unrealistisch**.



Agenda

- Motivation
- Grundbegriffe
- Implementierung
- Ausblick**
 - Koroutinen als Hilfsmittel für das BS
 - Mehrfähigkeit
- Zusammenfassung
- Referenzen



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
 - ~~ indirekt läuft jede Anwendung als Koroutine
 - **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**
- Mehr dazu in der nächsten Vorlesung!**



Agenda

- Motivation
- Grundbegriffe
- Implementierung
- Ausblick
- Zusammenfassung
- Referenzen



vs/dl

Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Zusammenfassung

7–30

Zusammenfassung: Quasi-Parallelität

- Ziel war die Ermöglichung von „Quasi-Parallelität“
 - Verschränkte Ausführung von Funktionen
 - Suspendierung und Reaktivierung von Funktions-Ausführungen
 - Begriff der Fortsetzung
- Routinen → asymmetrisches Fortsetzungsmodell
 - Ausführung nach LIFO (und damit nicht „quasi-parallel“)
 - CPU und Übersetzer stellen Elementaroperationen bereit
- Koroutinen → symmetrisches Fortsetzungsmodell
 - Ausführung in beliebiger Reihenfolge
 - erfordert eigenen Kontext: minimal PC, i. a. auch Register und Stapel
 - CPU und Übersetzer stellen i. a. keine Elementaroperationen bereit
- Fäden → vom BS verwaltete Koroutinen



vs/dl

Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Zusammenfassung

7–31

Referenzen

-  Melvin E. Conway. „Design of a separable transition-diagram compiler“. In: *Communications of the ACM* 6 (7 Juli 1963), S. 396–408. ISSN: 0001-0782. doi: 10.1145/366663.366704.
-  Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms, Third Edition*. Addison-Wesley, 1997. ISBN: 978-0201896831.
-  Michael Philippsen. *Grundlagen des Übersetzerbaus*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 2, 2015 (jährlich). URL: <https://www2.cs.fau.de/teaching/WS2015/UE1/index.html>.
-  John C. Reynolds. „The discoveries of continuations“. In: *Lisp Symb. Comput.* 6 (3-4 Nov. 1993), S. 233–248. ISSN: 0892-4635. doi: 10.1007/BF01019459.
-  Wolfgang Schröder-Preikschat. *Systemprogrammierung*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4, 2015 (jährlich). URL: <https://www4.cs.fau.de/Lehre/WS15/V-SP>.



vs/dl

Betriebssysteme (VL 7 | WS 18)

7 Koroutinen und Fäden – Referenzen

7–32