Systemprogrammierung

Prozesseinlastung

13. Januar 2010

Überblick

Programmfaden

Einlastungseinheit (engl. *unit of dispatching*), Aktivitätsträger

Einlastung der CPU folgt mehr oder weniger zeitnah zur Ablaufplanung von Programmfäden, ist ihr nachgeschaltet

- ein Abfertiger (engl. *dispatcher*) führt die eingeplanten Fäden der CPU zur Verarbeitung zu
  - *Mechanismus* zur Prozessverarbeitung: CPU umschalten
- dazu nimmt er Aufträge vom Planer (engl. *scheduler*) entgegen
  - *Strategie* zur Prozessverarbeitung: Aufträge an die CPU sortieren

Umschalten der CPU bedeutet, zwischen zwei Aktivitätsträgern desselben oder verschiedener Programme zu wechseln

CPU-Stoß endet $\mapsto$ der laufende Aktivitätsträger wird weggeschaltet

CPU-Stoß beginnt $\mapsto$ ein laufbereiter Aktivitätsträger wird zugeschaltet

☞ Aktivitätsträger lassen sich adäquat durch Koroutinen repräsentieren

Routinenartige Komponente eines Programms

Gleichberechtigtes Unterprogramm

Koroutinen wurden erstmalig um 1963 in der von Conway entwickelten Architektur eines Fließbandübersetzers (engl. *pipeline compiler*) eingesetzt [1]. Darin wurden zentrale Komponenten des Übersetzers konzeptionell als Datenflussfließbänder zwischen Koroutinen aufgefasst. Die Koroutinen repräsentierten *first-class* Prozessoren wie z.B. Lexer, Parser und Codegenerator.

Ko{existierende, operierende}-Routine

> *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. [1]*

Autonomer Kontrollfluss eines Programms

Kontrollfaden (engl. *thread of control*, TOC)

Koroutinen konkretisieren Prozesse (implementieren Prozessinstanzen), sie repräsentieren die Aktivitätsträger von Programmen

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

Koroutine $\mapsto$ „zustandsbehaftete Prozedur“

Aktivierungskontext bleibt während Phasen der Inaktivität erhalten

Koroutine deaktivieren $\mapsto$ Kontext „einfrieren“ (sichern)

Koroutine aktivieren $\mapsto$ Kontext „auftauen“ (wieder herstellen)

Programmiersprachliches Mittel zur Prozessorweitergabe

Multiplexen des Prozessors zwischen Prozessinstanzen

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. [2, S. 49]*

Routine kein Kontrollflusswechsel bei Aktivierung/Deaktivierung

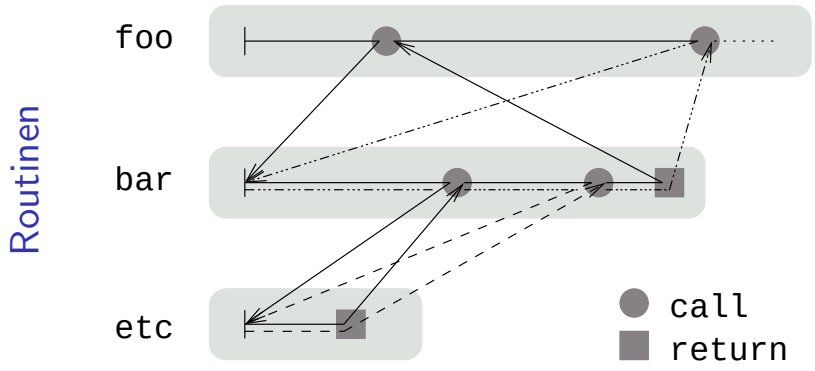
- asymmetrisch, ungleichberechtigte Rollen

Koroutine Kontrollflusswechsel bei Aktivierung/Deaktivierung

- symmetrisch, gleichberechtigte Rollen

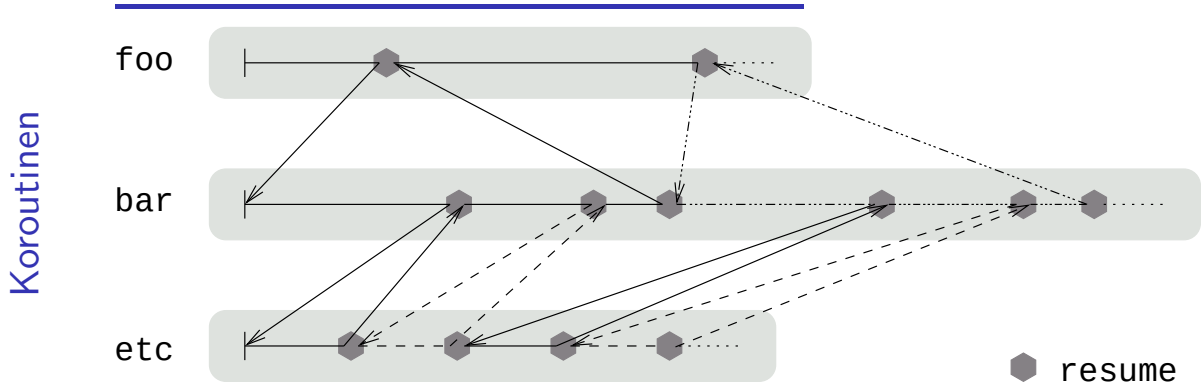
Routine vs. Koroutine

Gemeinsamkeiten und Unterschiede im Ablauf (vgl. S. 10-7)



Routinen werden aufgerufen, durchlaufen und beendet

Koroutinen werden erzeugt, gestartet und im Wechsel (freiwillig) unterbrochen und fortgesetzt



Routine vs. Koroutine (Forts.)

Gemeinsamkeiten und Unterschiede im Programmkode (korrespondierend zu S. 10-6)

Routinen

```
void foo () {
    ...
    call(bar);
    ...
    call(bar);
    ...
}
```

```
void bar () {
    ...
    call(etc);
    ...
    call(etc);
    ...
    return;
}
```

```
#define call(f) f()
```

```
void etc () {
    ...
    return;
}
```

```
void foo () {
    ...
    for (;;) {
        resume(bar);
        ...
    }
}
```

```
void bar () {
    ...
    for (;;) {
        resume(etc);
        ...
        resume(etc);
        ...
        resume(foo);
    }
}
```

```
void etc () {
    ...
    for (;;) {
        resume(bar);
        ...
    }
}
```

Koroutinen

Routine vs. Koroutine (Forts.)

Gemeinsamkeiten und Unterschiede im Konzept

Routine spezifischer als Koroutine

- ein einziger Einstiegspunkt
  - immer am Anfang
- ein einziger Ausstiegspunkt
  - kehrt nur einmal zurück
- Lebensdauer nach LIFO
  - *last in, first out*

Koroutine generischer als Routine

- ggf. mehrere Einstiegspunkte
  - dem letzten Ausstieg folgend
- ggf. mehrere Ausstiegspunkte
  - kehrt ggf. mehrmals zurück
- Lebensdauer nach LIAO
  - *last in, any out*

Routinen können durch Koroutinen implementiert werden [3]:

call $\mapsto$ *resume* der aufgerufenen Routine an ihrer Einsprungadresse
- Rücksprungkontext einfrieren, Aktivierungsblock aufsetzen

return $\mapsto$ *resume* der aufrufenden Routine an ihrer Rücksprungadresse
- Aktivierungsblock zurücksetzen, Rücksprungkontext auftauen

Routine vs. Koroutine (Forts.)

Gemeinsamkeiten und Unterschiede in der Implementierung

Mitbenutzung desselben Laufzeitstapels durch mehrere Koroutinen ist der von Routinen sehr ähnlich — und legt die Analogie auf S. 10-8 nahe:

- Unterbrechung und Fortsetzung von Koroutinen sind Spezialfälle des Ansprungs von Unterroutinen (engl. *jump to subroutine*, JSR)
- Prozessoren (Soft-/Hardware) stellen Elementaroperationen dafür zur Verfügung $\rightsquigarrow$ *Buchführung über Fortsetzungspunkte*

PDP-11/40 — Meilenstein in der Hardwareentwicklung

> *Another special case of the JSR instruction is JSR PC,@(SP)+ which exchanges the top element of the processor stack and the contents of the program counter. Use of this instruction allows two routines to swap program control and resume operation when recalled where they left off. Such routines are called "co-routines." [4, S. 4-58/59]*

Buchführung über Fortsetzungspunkte

Fortsetzung (engl. *continuation*) einer Programmausführung

Fortsetzungspunkt ist die Programmstelle, an der die Wiederaufnahme (engl. *resumption*) der Programmausführung möglich ist

- eine Adresse im Textsegment, an der ein Kontrollfluss (freiwillig, erzwungenermaßen) unterbrochen wurde
- die Stelle, an der der CPU-Stoß der einen Koroutine endet und der CPU-Stoß einer anderen Koroutine beginnt

Koroutinen zu implementieren bedeutet, Programmfortsetzungen zu verbuchen und Aktivierungskontexte zu wechseln:

- Fortsetzungsadressen sind dynamisch festzulegen und zu speichern
  - z.B. wie im Falle der Rücksprungadresse einer Prozedur
- ggf. sind weitere Laufzeitzustände zu sichern/wieder herzustellen
  - z.B. die Inhalte der von einer Koroutine benutzten Arbeitsregister

Kontrollflussverfolgung eines Routinenaufrufs

Programmfortsetzung und Aktivierungskontext

```
void* emuser (void *any) {
    return any;
}

void *foo, *bar;

int main () {
    bar = emuser(foo);
}
```

```
emuser:
    movl  4(%esp),%eax
    ret

main: ...
    pushl foo
    call  emuser
    movl  %eax,bar
    ...
```

Beispiel Stapelmaschine (x86):

- Rücksprungadresse und tatsächliche Parameter liegen auf dem Laufzeitstapel (engl. *runtime stack*) des Prozessors
- Aktivierungskontexte im Stapel sichern bzw. daraus wieder herstellen

Kontrollflussverfolgung eines Koroutinenwechsels

Programmfortsetzung verbuchen und Kontrollfluss wechseln

```
extern void* resume (void *);

void *foo, *bar;

int main () {
    bar = resume(foo);
}
```

```
resume:
    popl %eax
    movl (%esp),%ecx
    jmp  *%ecx
```

Beispiel Stapelmaschine (x86), Elementaroperation `resume()`:

- Rücksprungadresse ≡ Fortsetzungsadresse der abgebenden, genauer: `resume()` aufrufenden, Koroutine als Funktionswert zurückliefern
  - Rückgabewert (Arbeitsregister %eax) an die annehmende Koroutine
- tatsächlichen Parameter als Fortsetzungsadresse der annehmenden Koroutine lesen und dorthin verzweigen

Instanzenbildung und Aktivierung

Erzeugung einer Koroutine ↦ Gabelung des aktuellen Kontrollflusses

```
extern void* launch (void **);
extern void* resume (void *);

void *foo, *bar;

int main () {
    if ((bar = launch(&foo))
        /* Koroutine ohne Aufrufgeschichte */
        for (;;)
            bar = resume(bar);

    /* Koroutine mit Aufrufgeschichte */
    foo = resume(foo);
}
```

Funktion `launch()` sorgt dafür, dass ihre Rücksprungadresse initiale Fortsetzungsadresse einer neu geschaffenen Koroutine ist. Die „umgebende“ Routine wird in zwei Koroutinen aufgespaltet, eine (`main()`) mit und eine andere (*if*-Part) ohne Aufrufgeschichte: *letztere darf nicht terminieren.*

Instanzenbildung
Erzeugung des initialen Aktivierungskontextes

Koroutinen sind *first-class* Objekte, die im Regelfall dynamisch zur Laufzeit angelegt werden

- Objektzustand ist der Aktivierungskontext einer Koroutine
  - ihre Fortsetzungsadresse und ggf. ihr kompletter Prozessorzustand
- Startadresse ist eine „beliebige“ Programmadresse
  - hier: Rücksprungadresse der Funktion zur Instanzenbildung

```
void* launch (void **)                                    x86
launch:
    movl 4(%esp),%ecx # Referenzparameter lesen
    movl (%esp),%eax  # Rücksprungadresse lesen
    movl %eax,(%ecx)  # Rücksprungadresse sichern
    xorl %eax,%eax    # Rückgabewert aufsetzen: Ergebnis 0
    ret               # und zurück zum "Schöpfer"...
```


Intermezzo
Die Tücke liegt im Detail!

Frage Ist damit die Implementierung einer Prozessinstanz gegeben?

Antwort Im Prinzip ja, aber...

- die gemeinsame Benutzung desselben Laufzeitstapels durch mehrere Prozessinstanzen ist nur bedingt möglich
  1. die Prozesse dürfen nicht blockieren
  2. die Einplanung muss die Stapelmitbenutzung beachten [5]
- ein Prozess, der blockieren kann, benötigt einen eigenen Laufzeitstapel zur Sicherung seines Kontextes
  - die Fortsetzungsadresse einer Koroutine
  - je nach Prozess ggf. den kompletten Prozessorzustand
- die Aktivierung der diesbezüglichen Prozessinstanz bedingt den Wechsel des Laufzeitstapels — und ggf. mehr...

Eigenständige Koroutine ⇝ Federgewichtiger Prozess (S. 7-59)

- Koroutine mit eigenem Laufzeitstapel im gemeinsamen Adressraum

Koroutine ↦ Laufzeitstapel

Optionales Merkmal einer Prozessverwaltung

Signaturen/Schnittstellen der Koroutinenprimitiven wie gehabt:

```
resume:
    movl %esp,%eax
    movl 4(%esp),%esp
    ret
```

```
launch:
    movl 4(%esp),%ecx
    movl (%ecx),%eax
    leal -4(%eax),%eax
    movl %eax,(%ecx)
    movl %ecx,4(eax)
    movl (%esp),%ecx
    movl %ecx,(%eax)
    xorl %eax,%eax
    ret
```

`resume()` ⇝ Stapelumschaltung

- liefert alten Stapelzeiger
  - *ref* Fortsetzungsadresse
- zurück vom neuen Stapel
  - Koroutinenfortsetzung

`launch()` „vererbt“ ihren Aktivierungsblock an die neue Koroutine

- als wenn die neue Koroutine, die Funktion selbst aufgerufen hätte
- genauer: als wenn sie ein `resume()` bereits ausgeführt hätte

Instanzenbildung und Aktivierung (Forts.)

Einrichtung des Laufzeitstapels, Ausrichtung des Stapelzeigers

Delta zu `main()`, vgl. auch S. 10-13 x86

```
#define STACKSIZE 42*42

char area[STACKSIZE];

void *foo = area + (sizeof(area) - sizeof(int));
```

`area` ist Platzhalter für den Laufzeitstapel

- ein linear zusammenhängender Bereich von `STACKSIZE` Bytes

`foo` ist letztlich Platzhalter für den Stapelzeiger; Annahmen[1]:

- der Laufzeitstapel wächst „von oben nach unten“
- initialer Stapelzeiger ↦ hoher Adresswert, erster freier Platz
- ein Eintrag im Stapelspeicher belegt `sizeof(int)` Bytes
- der Stapelzeigerwert ist nicht Vielfaches von `sizeof(int)`

[1] Sämtliche Annahmen sind prozessor-, genauer: hardwareabhängig.

Prozessorstatus
Sicherung und Wiederherstellung

Unterbrechung der Koroutinenausführung $\mapsto$ Programmunterbrechung

- eine eigenständige, inaktive Koroutine ist davon abhängig, dass ihr Prozessorstatus invariant bleibt (S. 10-4)
- Koroutinenwechsel und -einrichtung sind fallspezifisch zu erweitern

Koroutinenwechsel $\rightsquigarrow$ Delta zum bisherigen `resume()`

- Prozessorstatus der abgebenden Koroutine auf den Stapel ablegen
- Prozessorstatus der annehmenden Koroutine vom Stapel nehmen

Koroutineneinrichtung $\rightsquigarrow$ Delta zum bisherigen `launch()`

- Prozessorstatus der erzeugenden Koroutine auf den Stapel der sich in Einrichtung befindlichen, neuen Koroutine ablegen

Prozessorstatus invariant halten
Analog zur Programmunterbrechung (vgl. S. 5-46*ff.*)

```
resume:
    __PUSH
    movl %esp,%eax
    movl __N+4(%esp),%esp
    __PULL
    ret
```

```
launch:
    movl 4(%esp),%ecx
    movl (%ecx),%eax
    leal -(__N+4)(%eax),%eax
    movl %eax,(%ecx)
    __DUMP(%eax)
    movl %ecx,__N+4(eax)
    movl (%esp),%ecx
    movl %ecx,__N(%eax)
    xorl %eax,%eax
    ret
```

`__N`	Umfang in Bytes
`__PUSH`	Sicherung
`__PULL`	Wiederherstellung
`__DUMP`	Übertragung

Entwurfsentscheidung: Alle Koroutinen sind vom selben „Typ“

- wobei der Typ Aufbau und Umfang des Prozessorstatus' definiert
- Alternative: Koroutinen verschiedener Typen $\rightsquigarrow$ BS/BST (S. 1-3)

Zwischenzusammenfassung

Koroutine *considered harmful*? Ja und nein!

Prozessinstanzen 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

Behauptung: Echte Systemprogrammiersprachen gibt es nicht mehr

- daher sind Koroutinen händisch in Assembliersprache bereitzustellen
- gleichwohl bleiben sie ein Programmiersprachenkonzept der Ebene 5

Koroutinen „mechanisieren“ Programmfäden

Technisches Detail zum Multiplexen der CPU zwischen Prozessen

Mehrprogrammbetrieb basiert auf Koroutinen des Betriebssystems

- pro auszuführendes Programm gibt es (wenigestens) eine Koroutine
  - ggf. für jeden Programmfaden $\rightsquigarrow$ leichtgewichtiger Prozess
- ist eine Koroutine aktiv, so ist das ihr zugeordnete Programm aktiv
  - der durch die Koroutine implementierte Programmfaden ist aktiv
- ein anderes Programm ausführen $\mapsto$ Koroutine wechseln

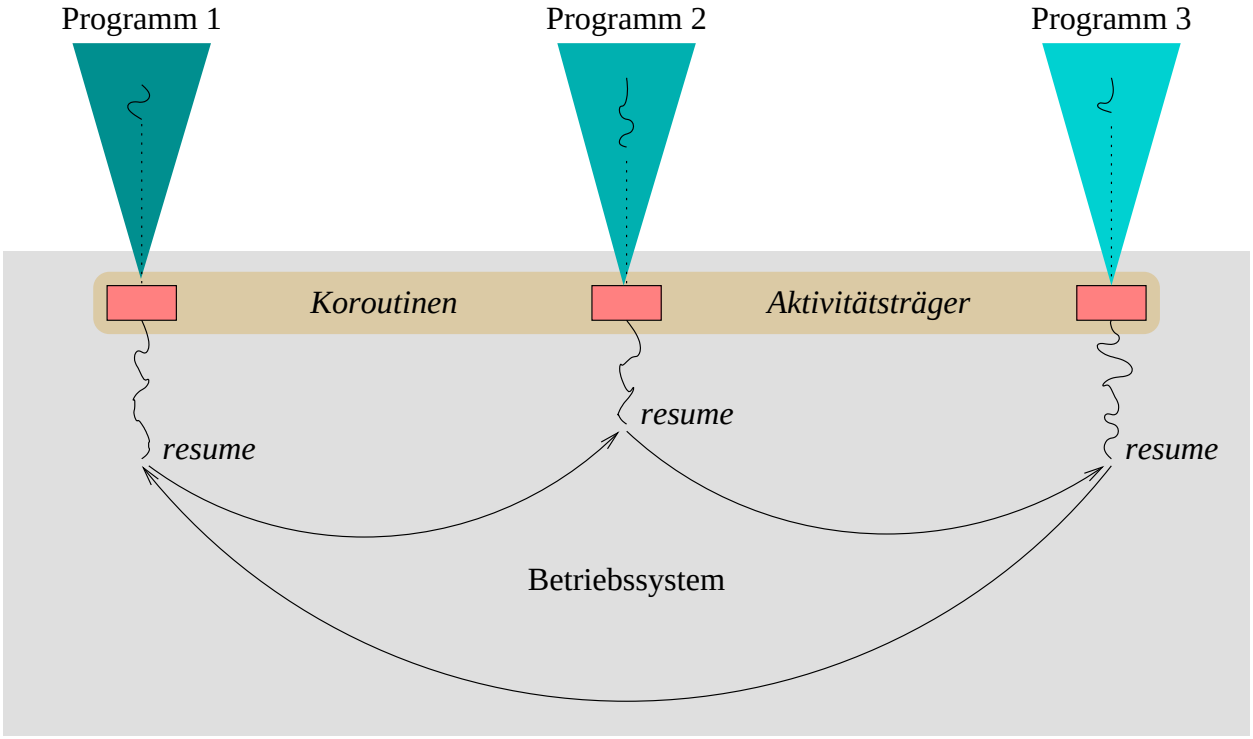
Koroutinen sind (autonome) Aktivitätsträger des Betriebssystems

- ihr Aktivierungskontext ist globale Variable des Betriebssystems
- für jede Prozessinstanz gibt es eine solche Betriebssystemvariable

☞ ein Betriebssystem ist Inbegriff für das nicht-sequentielle Programm

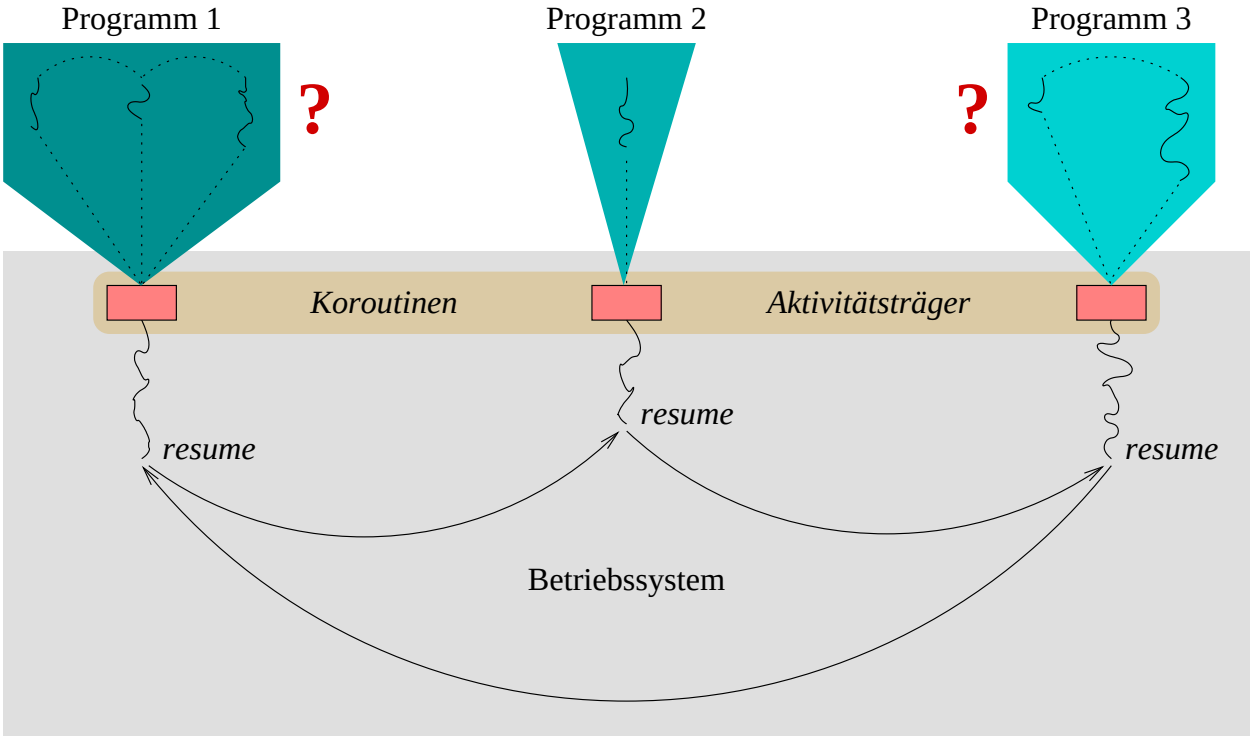
Verarbeitung sequentieller Programme

Koroutine als abstrakter Prozessor — Bestandteil des Betriebssystems



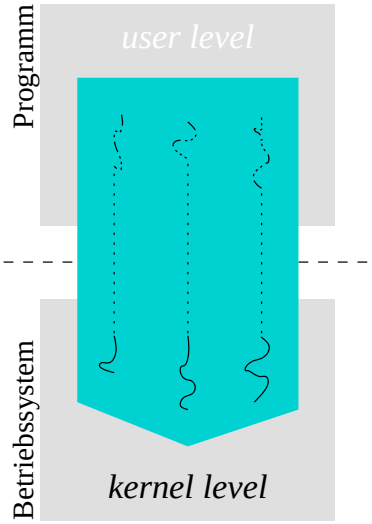
Verarbeitung nicht-sequentieller Programme

Multiplexen eines abstrakten Prozessors



Fäden der Kernebene

Klassische Variante von Mehrprozessbetrieb



Prozessinstanzen als Konzepte der Ebene $_3$ basieren auf Kernfäden (engl. *kernel-level threads*)

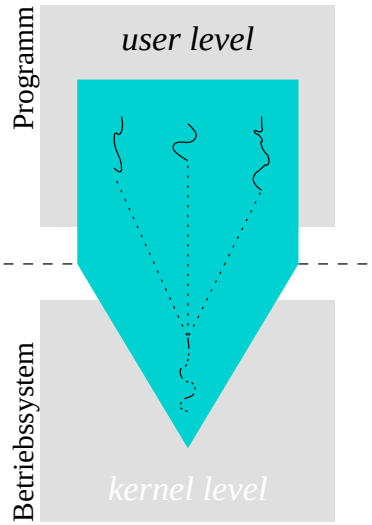
- egal, ob die Maschinenprogramme ein- oder mehrfädig ausgelegt sind
  - jeder Anwendungsfaden ist Kernfaden
  - nicht jeder Kernfaden ist Anwendungsfaden
- Kernfäden $\neq$ Prozessinstanzen der Ebene $_3$
  - Maschinenprogramme verwenden Fäden
  - im Programmiermodell des BS manifestiert
- Fäden realisiert durch Ebene $_2$-Programme

Einplanung und Einlastung der (leichtgewichtigen) Anwendungsprozesse sind Funktionen des Betriebssystem(kern)s

- Erzeugung, Koordination, Zerstörung von Fäden $\mapsto$ Systemaufrufe

Fäden der Benutzerebene

Ergänzung oder Alternative zu Kernfäden



Prozessinstanzen als Konzepte der Ebene $_{2/3}$ sind Benutzerfäden (engl. *user-level threads*)

- virtuelle Prozessoren bewirken die Ausführung der (mehrfädigen) Maschinenprogramme
  - Benutzerfäden = Koroutinen $\mapsto$ Ebene $_2$
  - 1 Kernfaden f. ≥ 2 Benutzerfäden $\mapsto$ Ebene $_3$
- der Kern stellt ggf. Planeransteuerungen (engl. *scheduler activations* [6]) bereit
  - zur Propagation von Einplanungsereignissen
- Fäden realisiert durch Ebene $_{2/3}$-Programme

Einplanung und Einlastung der (federgewichtigen) Anwendungsprozesse sind keine Funktionen des Betriebssystem(kern)s

- Erzeugung, Koordination, Zerstörung von Fäden $\mapsto$ Prozeduraufrufe

Prozesskontrollblock (engl. *process control block*, PCB)

Datenstruktur zur Verwaltung von Prozessinstanzen

Kopf eines Datenstrukturgeflechts zur Beschreibung einer Prozessinstanz und Steuerung eines Prozesses

- oft auch als Prozessdeskriptor (PD) bezeichnet
  - UNIX Jargon: *proc structure* (von „`struct proc`“)
- ein abstrakter Datentyp (ADT) des Betriebssystem(kern)s

Softwarebetriebsmittel zur Verwaltung von Programmausführungen

- jeder Faden wird durch eine Instanz vom Typ „PD“ repräsentiert

  Kernfaden Instanzenvariable des Betriebssystems

  Benutzerfaden Instanzenvariable des Anwendungsprogramms
- die Instanzenanzahl ist statisch (Systemkonstante) oder dynamisch

Objekt, das mit einer Prozessidentifikation (PID) assoziiert und für die gesamte Lebensdauer des betreffenden Prozesses gültig ist

- auch dann, wenn der Adressraum des Prozesses ausgelagert wurde

Aspekte der Prozessauslegung

Verwaltungseinheit einer Prozessinstanz

Dreh- und Angelpunkt, der alle prozessbezogenen Betriebsmittel bündelt

- Speicher- und, ggf., Adressraumbelegung †
  - Text-, Daten-, Stapelsegmente (*code, data, stack*)
- Dateideskriptoren und -köpfe (*inode*) †
  - {Zwischenspeicher,Puffer}deskriptoren, Datenblöcke
- Datei, die das vom Prozess ausgeführte Programm repräsentiert †

Datenstruktur, die Prozess- und Prozessorzustände beschreibt

- Laufzeitkontext des zugeordneten Programmfadens/Aktivitätsträgers
- gegenwärtiger Abfertigungszustand (*Scheduling*-Informationen) †
- anstehende Ereignisse bzw. erwartete Ereignisse †
- Benutzerzuordnung und -rechte †

Aspekte der Prozessauslegung (Forts.)

Prozessinstanz vs. Betriebsart

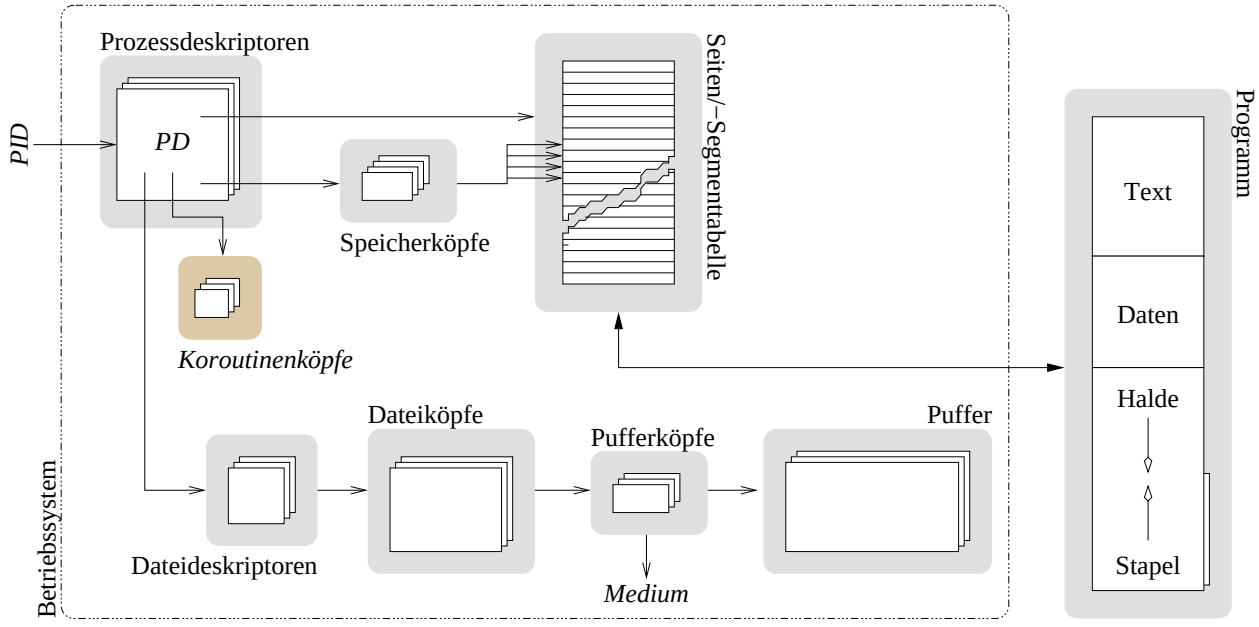
† Auslegung des PD ist höchst abhängig von Betriebsart und -zweck:

1. Adressraumdeskriptoren sind nur notwendig in Anwendungsfällen, die eine Adressraumisolation erfordern
2. für ein Sensor-/Aktorsystem haben Dateideskriptoren/-köpfe wenig Bedeutung
3. in ROM-basierten Systemen durchlaufen die Prozesse oft immer nur ein und dasselbe Programm
4. in Einbenutzersystemen ist es wenig sinnvoll, prozessbezogene Benutzerrechte verwalten zu wollen
5. bei statischer Prozesseinplanung ist die Buchführung von Abfertigungszuständen verzichtbar
6. Ereignisverwaltung fällt nur an bei ereignisgesteuerten und/oder verdrängend arbeitenden Systemen

☞ Festlegung auf eine Ausprägung grenzt Einsatzgebiete unnötig aus

Generische Datenstruktur

Logische Sicht eines Geflechts abstrakter Datentypen



einfädiger Prozess $\mapsto$ 1 Koroutinenkopf

mehrfädiger Prozess $\mapsto$ $N > 1$ Koroutinenköpfe

Instanzenvariable vom Typ „Prozessdeskriptor“

Buchführung über den aktuell laufenden Prozess

Zeiger auf den Kontrollblock des aktuell laufenden Prozesses

- für jeden Prozessor(kern)[2] ist solch ein Zeiger bereitzustellen
- innerhalb des Betriebssystems ist somit jederzeit bekannt, welcher Prozess, Faden, welche Koroutine die CPU „besitzt“
- wichtige Funktion der Einlastung ist es, den Zeiger zu aktualisieren

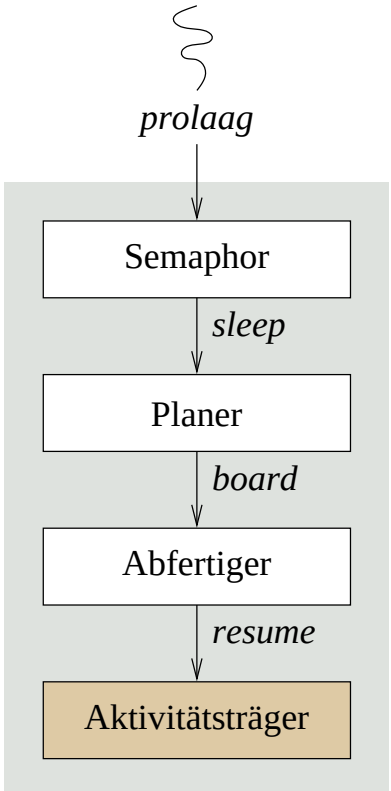
Aktualisierung des Zeigers ist, in Abhängigkeit von der Betriebsart, eine kritische Operation

- es droht Laufgefahr durch verdrängend arbeitende Einplanung
- Einlastung $\mapsto$ kritischer Abschnitt $\leadsto$ bedingt verdrängbar
- dennoch voll verdrängbar zu sein, ist knifflig $\leadsto$ BS/BST (S. 1-3)

[2] Im Falle mehr- oder vielkerniger (engl. *multi-core/many-core*) Prozessoren reicht ein Zeiger pro CPU nicht aus.

Gesamtzusammenhang

Funktionale Hierarchie typischer Komponenten einer Prozessverwaltung



prolaag bedingte Prozessblockade

- im P „hängen bleiben“...

sleep Prozessblockade und -auswahl

- laufenden Prozess Schlafen legen
- nächsten laufbereiten Prozess von der Warteliste nehmen

board Prozesseinlastung

- ausgewählten Prozess der CPU zuteilen
- Instanzenvariable des aktuell laufenden Prozesses aktualisieren

resume Koroutinenwechsel

- Prozessorstatus austauschen

Einlastung $\rightsquigarrow$ Umsetzung von Einplanungsentscheidungen

Koroutinenwechsel $\models$ Fadenwechsel $\models$ Prozesswechsel

- Koroutinen konkretisieren Prozesse, realisieren Prozessinstanzen
  - die Gewichtsklasse von Prozessen spielt eine untergeordnete Rolle
    - feder-, leicht-, schwergewichtige Prozesse basieren auf Koroutinen
  - ihr Aktivierungskontext überdauert Phasen der Inaktivität
    - gesichert („eingefroren“) im jeder Koroutine eigenen Stapelspeicher
- Programmfäden (engl. *threads*) sind durch Koroutinen repräsentiert
  - unterschieden in zwei Fadenarten, je nach Ebene der Abstraktion:

    Kernfaden implementiert durch Ebene$_2$-Programme

    Benutzerfaden implementiert durch Ebene$_{2/3}$-Programme
  - Einlastung eines Fadens führt einen Koroutinenwechsel nach sich
- der Prozessdeskriptor ist Objekt der Buchführung über Prozesse
  - Datenstruktur zur Verwaltung von Prozess- und Prozessorzuständen
    - insbesondere des Aktivierungskontextes der Koroutine eines Prozesses
  - Softwarebetriebsmittel zur Beschreibung einer Programmausführung

Literaturverzeichnis

[1] M. E. Conway.
Design of a separable transition-diagram compiler.
Communications of the ACM, 6(7):396–408, 1963.

[2] Ralf Guido Herrtwich and Günter Hommel.
Kooperation und Konkurrenz — Nebenläufige, verteilte und echtzeitabhängige Programmsysteme.
Springer-Verlag, 1989.

[3] Donald Ervin Knuth.
The Art of Computer Programming, volume 1, Fundamental Algorithms.
Addison-Wesley, Reading, MA, USA, 3rd edition, 1997.

[4] Digital Equipment Corporation, Maynard, MA, USA.
PDP-11/40 Processor Handbook, 1972.

Literaturverzeichnis (Forts.)

[5] Theodore P. Baker.
Stack-based scheduling of realtime processes.
Real-Time Systems, 3(1):67–99, 1991.

[6] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy.
Scheduler activations: Effective kernel support for the user-level management of parallelism.
In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, volume 25 of *Operating Systems Review*, pages 95–109, Pacific Grove, CA, 1991.