Betriebssystemtechnik

Fadenumschaltung: Prozesseinlastung (*Dispatching*)

Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

12. Juni 2012

Gliederung

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

Datenstruktur zur Verwaltung von Prozessinkarnationen

Kopf eines Datenstrukturgeflechts zur Beschreibung und Verwaltung einer Prozessinkarnation 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 ein Exemplar vom Typ „PD“ repräsentiert

Kernfaden	• Variable des Betriebssystems
Benutzerfaden	• Variable des Anwendungsprogramms

- die Exemplaranzahl ist statisch (Systemkonstante) oder dynamisch

Objekt, das mit einer **Prozessidentifikation** (PID) assoziiert ist

- eine für die gesamte Lebensdauer des Prozesses gültige Bindung

Aspekte der Prozessauslegung

Verwaltungseinheit einer Prozessinkarnation

Dreh- und Angelpunkt, der **prozessbezogene Betriebsmittel** bündelt

- Speicher- und, ggf., Adressraumbelegung *1
  - Text-, Daten-, Stapelsegmente (*code, data, stack*)
- Dateideskriptoren und -köpfe (*inode*) *2
  - {Zwischenspeicher,Puffer}deskriptoren, Datenblöcke
- Datei, die das vom Prozess ausgeführte Programm repräsentiert *3

Datenstruktur, die Prozess- und Prozessorzustände beschreibt

- Laufzeitkontext des zugeordneten Programmfadens/Aktivitätsträgers
- gegenwärtiger Abfertigungszustand (*Scheduling*-Informationen) *4
- anstehende Ereignisse bzw. erwartete Ereignisse *5
- Benutzerzuordnung und -rechte *6

Aspekte der Prozessauslegung: Optionale Merkmale

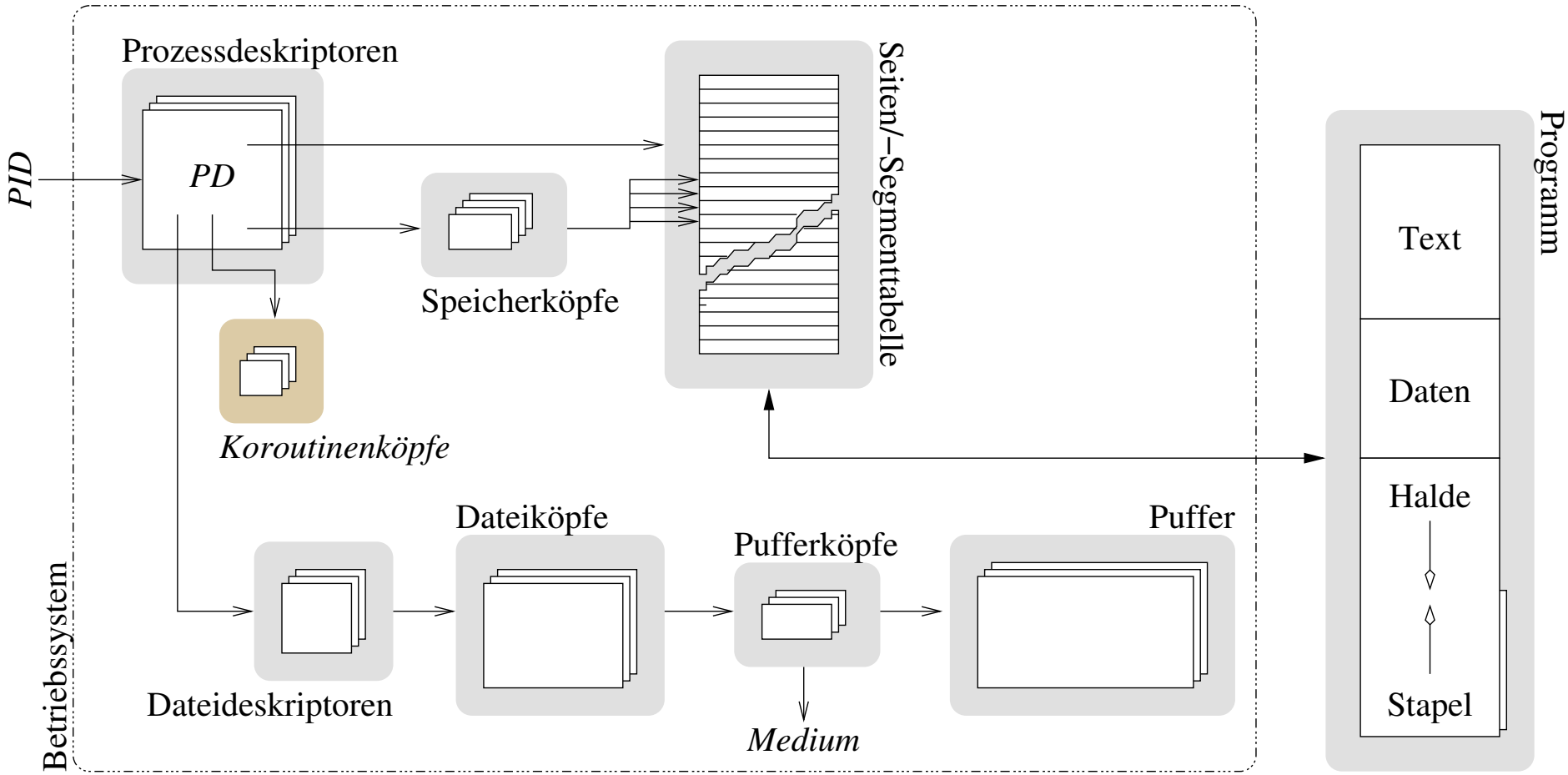
* Auslegung des PD ist höchst abhängig von Betriebsart und -zweck:
  1. Adressraumdeskriptoren sind nur notwendig in Anwendungsfällen, die eine Adressraumisolation erfordern
  2. für ein Sensor-/Aktorsystem haben Dateideskriptoren/-köpfe wenig Bedeutung
  3. in ROM-basierten Systemen durchlaufen die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

Problemspezifische Datenstruktur

- 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

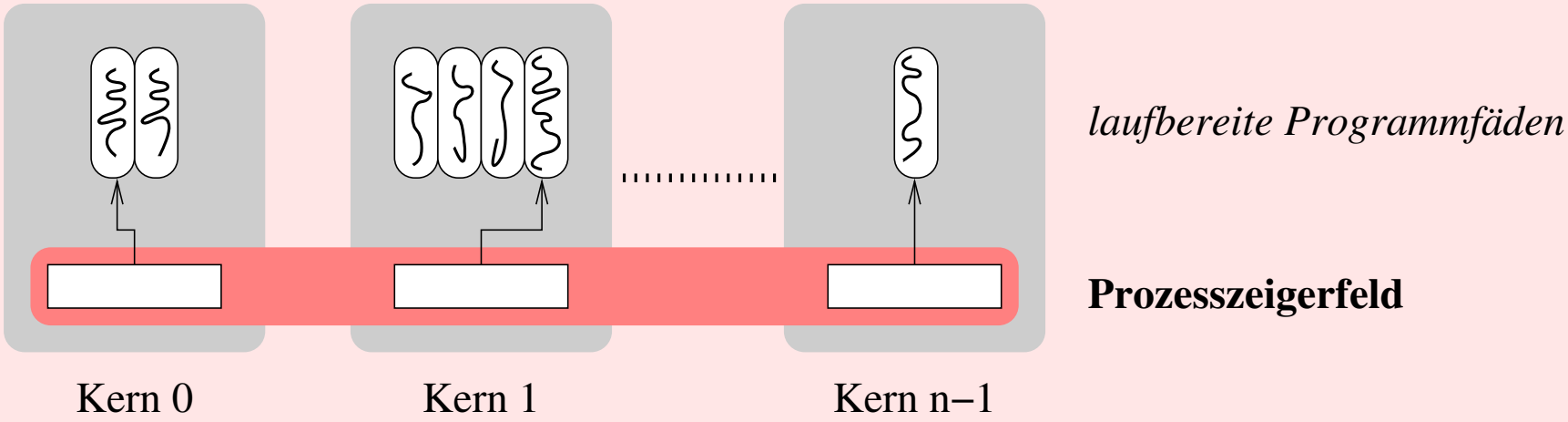
Buchführung über den aktuell laufenden Prozess

Zeiger auf den Kontrollblock des laufenden Prozesses: **Prozesszeiger**

- innerhalb des Betriebssystems ist darüber jederzeit bekannt, welcher Prozess, Faden, welche Koroutine die CPU „besitzt“
- wichtige Funktion der Einlastung von Prozessen ist es, den Zeiger im Zuge der Fadenumschaltung zu aktualisieren

Beachte: Mehr-/Vielkernige Prozessoren, Multiprozessoren

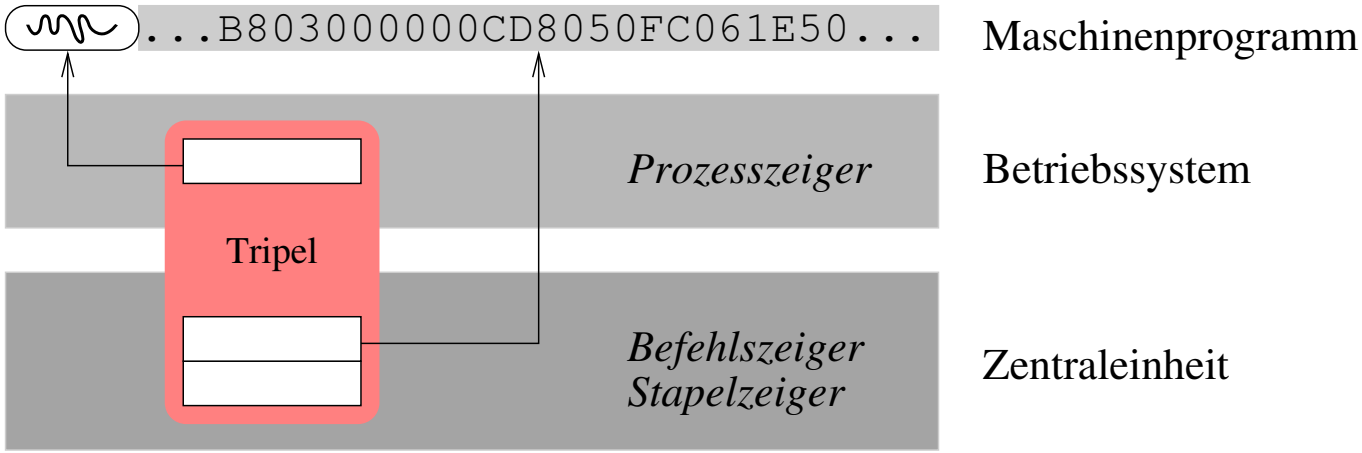
- für jeden Prozessor(kern) ist solch ein Prozesszeiger bereitzustellen



Lokalisierung eines Prozesses

Tripel aus Prozess-, Stapel- und Befehlszeiger

Logische Sicht:

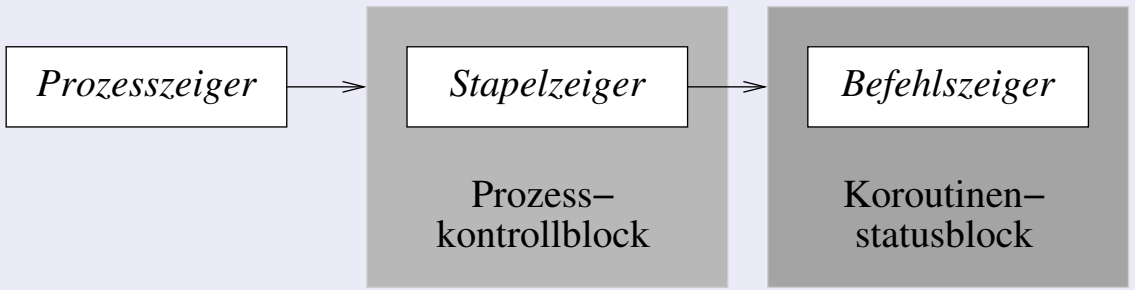


Physische Sicht: Prozesswechsel $\rightsquigarrow$ ein **Zeigerpaar** umsetzen

(a) Prozesszeiger

(b) Stapelzeiger

$\Rightarrow$ **kritischer Abschnitt!!!**



Gliederung

Funktionale/Prozedurale Abstraktion

Modulschnittstelle: `action.h` — minimale Koroutinenerweiterung

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

typedef struct action {
    :
    coaction_t act;                          /* associated coroutine action */
} action_t;

extern action_t *act_being();                /* return action pointer */
extern void      act_apply(action_t *); /* define action pointer */
extern void      act_board(action_t *); /* switch to specified action */
```

being
- den Prozesszeiger zum laufenden Progammfaden liefern

apply
- den Prozesszeiger des gegenwärtigen Prozessor(kern)s setzen

board
- einen Programmfaden „einschiffen“
- den Prozessor(kern) zwischen Programmfäden umschalten

Operationen auf dem Prozesszeiger

Prozesszeiger liefern

```
INLINE action_t *act_being() {
    return act_index[cpu_index()];
}
```

Prozesszeiger definieren

```
INLINE void act_apply(action_t *next) {
    act_index[cpu_index()] = next;
}
```

wobei das **Prozesszeigerfeld** wie folgt ausgelegt und vorgegeben ist:

Deklaration

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

extern action_t *act_index[];
```

Definition: act_index.c

```
#include "luxe/machine/cpu.h"
#include "luxe/action.h"

action_t *act_index[NCORE];
```

Beachte: Prozessor(kern)abhängigkeit

`cpu_index()`
- liefert die Nummer des ausführenden Prozessor(kern)s
- im Bereich $[0, NCORE - 1]$

`NCORE`
- spezifiziert die Anzahl der Prozessor(kern)e

Identifikation des ausführenden Prozessor(kern)s

Schnittstelle

```
#define NCORE 1    /* uni-processor */

extern int cpu_index();
```

Implementierung

```
INLINE int cpu_index() {
    return 0;
}
```

Optionen zur Identifikation eines Prozessor(kern)s für $NCORE > 1$:

i. Geräte- oder Prozessorregister mit festem, hardwaredefiniertem Inhalt

ii. Spezialregister mit variablem, betriebssystemdefiniertem Inhalt:

   a) ggf. noch zu normierende Nummer des Prozessor(kern)s
   b) Zeiger auf einen Prozessor(kern)deskriptor im Hauptspeicher
      - der u.a. die noch zu normierende Nummer des Prozessor(kern)s enthält

iii. vom Kompilierer freigehaltenes Arbeitsregister als „Spezialregister“

Beachte: Spezialregister

- Zugriff darauf ist eine *sensitive Operation*: **Virtualisierung**
- schreibender Zugriff darauf ist eine *privilegierte Operation*: **Schutz**

Reihenanordnung: *board*

Wechsel durch *regain*

```
INLINE void act_board(action_t *next) {
    action_t *self = act_being();
    act_apply(next);

    coa_regain(next->act, &self->act);
}
```

Wechsel durch *resume*

```
INLINE void act_board(action_t *next) {
    coaction_t last = coa_resume(next->act);

    act_being()->act = last;
    act_apply(next);
}
```

der abgebende Faden:

- sichert seinen Stapelzeiger selbst
- setzt den Prozesszeiger auf seinen Nachfolger

der annehmende Faden:

- sichert den Stapelzeiger seines Vorgängers
- setzt den Prozesszeiger auf sich selbst

Beachte: Arten der Auslegung von *regain*/*resume* (Kap. IV-2, S. 22)

Reihenanordnung	• unabhängige Fäden, variabler Kontextyp
Unterprogrammanordnung	• abhängige Fäden, fester Kontexttyp

Reihenanordnung: *board*, eingebettet (gcc -O6 -S)

board ↦ *regain* [a]

```
  movl  next, %eax
  movl  act_index, %edx
  movl  %eax, act_index
  movl  (%eax), %eax
  pushl %ebx
  pushl %ebp
  pushl %esi
  pushl %edi
  pushl $1f
  movl  %esp, (%edx)
  movl  %eax, %esp
  ret   # thread switch
1:
  popl  %edi
  popl  %esi
  popl  %ebp
  popl  %ebx
```

board ↦ *resume* [a]

```
  movl  next, %edx
  movl  (%edx), %ecx
  pushl %ebx
  pushl %ebp
  pushl %esi
  pushl %edi
  pushl $1f
  movl  %esp, %eax
  movl  %ecx, %esp
  ret   # thread switch
1:
  popl  %edi
  popl  %esi
  popl  %ebp
  popl  %ebx
  movl  act_index, %ecx
  movl  %edx, act_index
  movl  %eax, (%ecx)
```

board ↦ *resume* [b]

```
  movl  _next, %ecx
  movl  (%ecx), %edx
  pushl %ebx
  pushl %ebp
  pushl %esi
  pushl %edi
  pushl $1f
  movl  %esp, %eax
  movl  %edx, %esp
  ret   # thread switch
1:
  movl  %eax, %edx
  popl  %edi
  popl  %esi
  popl  %ebp
  popl  %ebx
  movl  _act_index, %eax
  movl  %edx, (%eax)
  movl  %ecx, _act_index
```

[a] gcc 4.4.5 (Debian)
[b] gcc 4.0.1 (Apple)

Unterprorgammanordnung: *board*

Wechsel durch *switch*

```
void act_board(action_t *next) {
    action_t *self = act_being();
    act_apply(next);

    self->act = coa_switch(next->act);
}
```

Beachte: Rücksprungadresse

- ist die Fortsetzungsadresse der Koroutine/des Fadens

$\Rightarrow$ innerhalb von *board* den Faden umschalten, ist überflüssig

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

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

    return (coaction_t)codo;
}
```

Minimale Erweiterung:

1. Kontext stapeln
2. Stapel umschalten
3. Kontext entstapeln

$\Rightarrow$ Abstraktion auflösen

Beachte: Art der Auslegung von *board*

- Implementierungsentscheidung: abhängige Fäden, fester Kontexttyp

Unterprorgammanordnung: *board* $\mapsto$ *switch*

`gcc -O6 -S act_board.c`

```
act_board:
  movl  4(%esp), %eax     # grab pointer to action descriptor of next thread
  movl  act_index, %edx   # grab pointer to action descriptor of this thread
  movl  (%eax), %ecx      # grab stack pointer of next thread
  movl  %eax, act_index   # book action of next thread to be switched to
#APP                      = begin of inlined switch
  pushl %ebx              # save context of this thread
  pushl %ebp              # ...
  pushl %esi              # ...
  pushl %edi              # ...
  movl  %esp, %eax        # keep context pointer of this thread
  movl  %ecx, %esp        # stack switch: change over to next stack
  popl  %edi              # restore context of next thread
  popl  %esi              # ...
  popl  %ebp              # ...
  popl  %ebx              # ...
#NO_APP                   = end of inlined switch
  movl  %eax, (%edx)      # save context pointer of this thread
  ret                     # thread switch: actually run next thread
```

Zwischenzusammenfassung

Argumente für die **Entwurfs- und Implementierungsentscheidung** zur Auslegungsart der Umschaltprozedur (*board*):

Reihenanordnung
- mehrere Umschaltstellen im Planer
  - $\Rightarrow$ Einsparung von Laufzeit
- Entscheidungsaufschub über die Fadenart
- Förderung koexstierender Planerdomänen

Unterprogrammanordnung
- eine zentrale Umschaltstelle im Planer
  - $\Rightarrow$ Einsparung von Speicherplatz
- Austausch des Umschalters im Betrieb

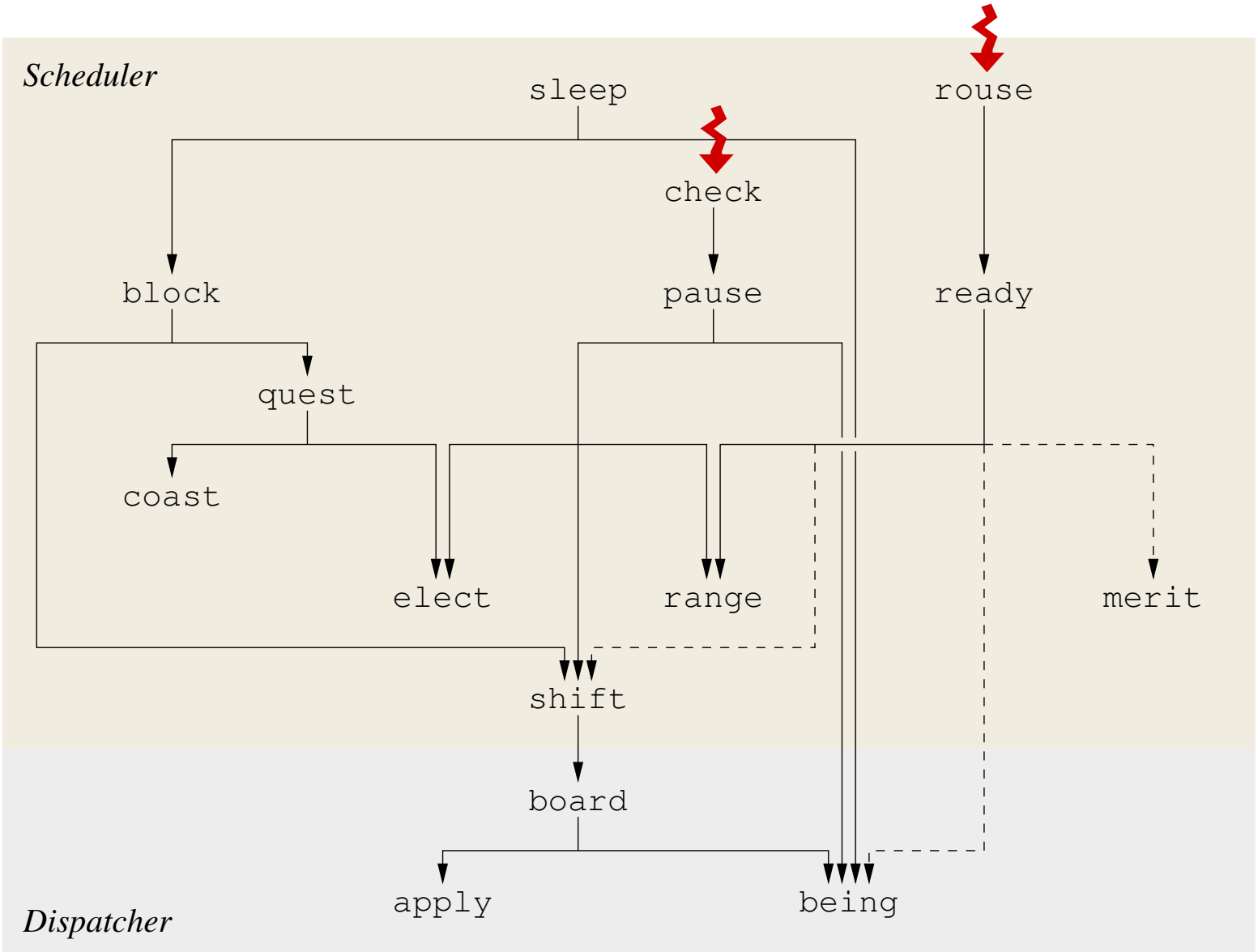
$\Rightarrow$ maßgeblich sind Aufbau, Struktur und Funktionsweise **des Planers**

Beachte: Abhängigkeit zum Operationsprinzip des Planers

- verdrängend arbeitender Planer führt zur Wettlaufsituation in *board*
- Problem: umsetzen von Prozess- und Stapelzeiger ist nicht atomar!

Gliederung

Planer und Umschalter im Verbund: Aufrufhierarchie



Funktionen des Planers

Ereignisbehandlung

sleep	• schlafen legen
rouse	• Schlafende aufwecken

Fadensteuerung

block	• Laufenden blockieren
pause	• Laufenden innehalten
ready	• Laufbereiten einstellen • ggf. Prozessor abtreten
check	• Laufenden prüfen

Leerlaufsteuerung

quest	• Laufbereiten suchen
coast	• Prozessor leerlaufen

Fadeneinplanung

elect	• Laufbereiten auswählen
range	• Laufbereiten einordnen
merit	• auf Vorzug prüfen

Prozessorzuteilung

shift	• Faden umschalten

Beachte: Operationsprinzip „Verdrängung“

synchron	• durch *ready*, im Falle vorranggesteuerter Einplanung
asynchron	• durch *check*, im Falle zeitscheibengesteuerter Umplanung • durch *rouse*, im Falle gerätegesteuerter Ereigniszustellung

Benutztbeziehung zwischen Planer und Umschalter

[4] Benutzthierarchie $\overset{[1,\ S.\ 151]}{\equiv}$ funktionale Hierarchie [2]

- für zwei Programme A und B bedeutet A „benutzt“ B, . . .
  - wenn es Situationen gibt, in denen das korrekte Funktionieren von A abhängt vom Vorhandensein einer korrekten Implementierung von B
- „benutzt“ bedeutet damit auch die **zyklenfreie Schichtanordnung**

ohne Verdrängung
- der Planer benutzt den Umschalter
- korrektes Funktionieren des Planers bedingt die Durchsetzung seiner Entscheidung im Umschalter

mit Verdrängung
- *dito* — und: der Umschalter benutzt den Planer
- korrektes Funktionieren des Umschalters hängt ab vom Schutz kritischer Abschnitte im Planer

$\Rightarrow$ der **Zyklus** ist entwurfstechnisch aufzulösen

Laufgefahr: Verdrängend arbeitende Prozesseinplanung

Einlastung meint zweierlei: Prozesszeiger- *und* Fadenumschaltung

- Verdrängung des dazwischen laufenden Fadens ist kritisch
  - genauer: zwischen *apply* und {*regain*, *resume*, *switch*} (vgl. S. 13–16)
- der Programmabschnitt dazu ist ein typischer **kritischer Abschnitt**
  - ungünstige Überlappung führt zur Fehlleitung der Zustandssicherung

Schutz dieses kritischen Abschnitts muss nur **pseudo-parallele Prozesse**[1] geeignet koordinieren, wozu folgende Optionen in Frage kommen:

(a) Verdrängung (d.h., Planeraktivierungen) zeitweise unterbinden
  - Verdrängungssperre

(b) Prozesszeiger aus dem Stapelzeiger berechnen
  - Zeigerverschmelzung
  - Stapelzeiger umschalten schaltet implizit den Prozesszeiger mit um

[1] Gleichzeitige Prozesse bedingt durch Verfahren zum Prozessormultiplexen.

Laufgefahr: Szenario mit drei Fäden F_1, F_2 und F_3

1. als Folge von *pause*, *block* oder *ready* tritt F_1 den Prozessor ab
   - F_2 wurde vom Planer ausgewählt und dem Umschalter übergeben
2. F_1 wird im Umschalter unterbrochen, der Planer wird erneut aktiviert:
   a) nach *apply* aber noch vor *regain* oder *switch* $\Rightarrow F_1$ läuft als F_2
   b) nach *resume* aber noch vor *apply* $\Rightarrow F_2$ läuft als F_1
3. als Folge von *check* oder *rouse* tritt F_1 bzw. F_2 den Prozessor ab
   - F_3 wurde vom Planer ausgewählt und dem Umschalter übergeben
   - der Prozessorstatus des laufenden Fadens wurde im Stapel abgelegt
4. der Stapelzeiger SP wird in den Prozessdeskriptor PD gesichert:
   zu a) $being = F_2 \Rightarrow SP_{F_1}$ wird *act* in PD_{F_2} zugewiesen
   - die Adresse des gesicherten Zustands von F_2 wird überschrieben
   - F_2 wird später mit dem Zustand von F_1 weiterlaufen

   zu b) $being = F_1 \Rightarrow SP_{F_2}$ wird *act* in PD_{F_1} zugewiesen
   - die Adresse des gesicherten Zustands von F_2 wurde nicht aktualisiert
   - F_2 wird später mit seinem „vorletzten“ Zustand weiterlaufen

Verdrängungsfreie kritische Abschnitte

NPCS, Abk. für (engl.) *non-preemptive critical section*

Ereignisse, die zur Verdrängung eines sich in einem kritischen Abschnitt befindlichen Prozesses führen könnten, werden unterbunden

enter
- Flagge zeigen, dass Entzug des Prozessors nicht stattfinden darf
- die mögliche Verdrängung des laufenden Prozesses zurückstellen

leave
- Flagge zeigen, dass Entzug des Prozessors stattfinden darf
- die ggf. zurückgestellte Verdrängungsanforderung weiterleiten

Aussetzen der Verdrängung des laufenden Prozesses ist durch (einfache) Maßnahmen an zwei Stellen der Prozessverwaltung möglich:

1. Einplanung eines freigestellten Prozesses zurückstellen
2. Einlastung eines zuvor eingeplanten Prozesses zurückstellen

Analogie zum Epilogkonzept [5, 3]

- wobei jedoch nur der „Verdrängungsepilog“ kontrolliert werden muss

Verdrängungsfreie Fadenumschaltung: *board* synchronisiert

```
INLINE void act_board(action_t *next) {
    CS_ENTER(&npcs);
    coaction_t last = coa_resume(next->act);

    act_being()->act = last;
    act_apply(next);
    CS_LEAVE(&npcs);
}
```

```
INLINE void act_board(action_t *next) {
    action_t *self = act_being();

    CS_ENTER(&npcs);
    act_apply(next);

    coa_regain(next->act, &self->act);
    CS_LEAVE(&npcs);
}
```

```
void act_board(action_t *next) {
    action_t *self = act_being();

    CS_ENTER(&npcs);
    act_apply(next);

    self->act = coa_switch(next->act);
    CS_LEAVE(&npcs);
}
```

Fadenwechsel geschehen damit immer innerhalb eines kritischen Abschnitts

- der den Prozessor abgebende Faden setzt die Verdrängungssperre
- der den Prozessor annehmende hebt die Verdrängungssperre auf

Beachte: Anlaufende Fäden

- ihnen wurde der Prozessor noch nie zuvor zugeteilt
- sie konnten *board* nicht ausführen, werden aber darüber aktiviert

$\Rightarrow$ Verdrängungssperre aufheben bevor der Faden wirklich startet

Implementierungsansatz für NPCS

Schutzvorrichtung (engl. *guard*) von recht einfacher Funktion:

i ein Bitschalter (engl. *flag*) zum Sperren von Verdrängungen
- durch *enter* gesetzt und *leave* zurückgesetzt

ii eine Warteschlange zurückgestellter Verdrängungsanforderungen
- durch *leave* abgearbeitet

Beachte: „Schleuse“ für Planeraufrufe

- neben *enter* und *leave* gibt es noch eine weitere Operation: *guide*
  - diese leitet die Verdrängungsanforderungen durch das System
  - sie schleust ein von Treibern aufgerufenes *check* oder *rouse* durch:
    a) Aktivierung, wenn der Bitschalter nicht gesetzt ist
    b) Zurückstellung und Einreihung in die Warteschlange, sonst
  - ihr Aufruf erfolgt im Nachspann einer Unterbrechungsbehandlung[a]
- alle Operationen lassen sich gut nichtblockierend implementieren

[a] AST, vgl. Kap. II-2, S. 16*ff.*

Konkrete Ausprägung der Laufgefahr

Gegenstand des Konflikts ist letztlich die Tatsache, zwei Zeiger zugleich (d.h., atomar) umsetzen zu müssen, nämlich:

i der **Prozesszeiger** auf den als nächster laufende Faden und

ii der **Stapelzeiger** auf den Prozessorzustand dieses Fadens

Problem dabei ist, dass beide Zeiger den jeweiligen Programmiermodellen eines abstrakten und realen Prozessors entstammen:

abstrakt
- Prozesszeiger $\mapsto$ Betriebssystem

real
- Stapelzeiger $\mapsto$ CPU (genauer: Kern einer CPU)

Ansatz: Prozesszeiger auf Stapelzeiger abbilden

- bei Fadenumschaltung ist dann nur der Stapelzeiger umzusetzen
  - schreiben ins Stapelzeigerregister der CPU verläuft atomar !!!
- der Prozesszeiger muss dann aber vom Stapelzeiger ableitbar sein
  - $\Rightarrow$ *being* muss einen Wert berechnen, nicht nur lesen
  - $\Rightarrow$ *apply* wird zur Nulloperation (engl. *no-operation*, NOP)

„Zweierpotenzen bündige“ Stapelspeicher

Platzierung des Prozessdeskriptors eines Fadens auf den **Boden** des ihm zugehörigen 2^k tiefen und ebenso bündigen Laufzeitstapels

- wobei $k < n$, mit n für die Adressbreite des Stapelzeigers
- z.B. L4/x86: $k = 12$ (d.h., 4 KiB gr. Stapelspeicher), $n = 32$

Folge aus den 2^k-Byte bündigen Stapelspeichern ist, dass die Adresse des Stapelbodens SB leicht aus dem Stapelzeiger SP ableitbar ist:

abwärts wachsender Stapel (x86)

- $SB = SP$ `or` $(2^k - 1)$
- $PD = SB - (sizeof(PD) - 1)$, Adresse des Prozessdeskriptors

aufwärts wachsender Stapel (8051)

- $SB = SP$ `and` $neg(2^k - 1)$
- $PD = SB$, Adresse des Prozessdeskriptors

Berechnung der Prozessdeskriptoradresse

Abwärts wachsender Stapel

```
INLINE thread_t *act_parts(void *tos) {
   return (thread_t *)(((unsigned)tos | ACT_STACKMASK) - (sizeof(thread_t) - 1));
}
```

Aufwärts wachsender Stapel

```
INLINE thread_t *act_parts(void *tos) {
   return (thread_t *)((unsigned)tos & ~ACT_STACKMASK);
}
```

Stapelparameter: 4 KiB

```
#define ACT_STACKSIZE (1 << 12)
#define ACT_STACKMASK (ACT_STACKSIZE - 1)
```

- nicht mehr als 4 KiB für μ-Kerne
- 8 KiB für Systeme à la Linux

Prozessdeskriptor: Muster

```
typedef struct thread {
    ...
    action_t act;
    ...
} thread_t;
```

⇒ Stapelspeichergröße eines Betriebssystemkerns: *system mode*

Verwendung zur Herleitung des Prozesszeigers

Prozesszeiger liefern

```
INLINE action_t *act_being() {
    return &act_parts(cpu_stack())->act;
}
```

Prozesszeiger definieren

```
INLINE void act_apply(action_t *nop) {
    /* no-operation */
}
```

Stackpointer liefern

```
INLINE void *cpu_stack() {
    register uint32_t tos __TOS;
    return (void *)tos;
}
```

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

Faden umschalten: wie gehabt

```
void act_board(action_t *next) {
    action_t *self = act_being();
    act_apply(next);

    self->act = coa_switch(next->act);
}
```

gcc -O6 -S act_being.c

```
act_being:
  movl %esp, %eax    # read stack pointer
  orl  $4095, %eax   # make pointer to bottom of stack (rightmost 12 bits are on)
  subl $3, %eax      # make pointer to thread descriptor (sample of 4 bytes size)
  ret
```

Unterprorgammanordnung: *board* $\mapsto$ *switch* (vgl. S. 16)

`gcc -O6 -S act_board.c`

```
act_board:
  movl  4(%esp), %edx     # grab pointer to action descriptor of next thread
  movl  %esp, %eax        # read stack pointer of this (i.e., running) thread
  movl  (%edx), %ecx      # grab stack pointer of next thread
#APP                      = begin of inlined switch
  pushl %ebx              # save context of this thread
  pushl %ebp              # ...
  pushl %esi              # ...
  pushl %edi              # ...
  movl  %esp, %edx        # keep context pointer of this thread
  movl  %ecx, %esp        # stack switch: change over to next stack
  popl  %edi              # restore context of next thread
  popl  %esi              # ...
  popl  %ebp              # ...
  popl  %ebx              # ...
#NO_APP                   = end of inlined switch
  orl   $4095, %eax       # make pointer to bottom of stack of this thread
  movl  %edx, -3(%eax)    # save context pointer of this thread
  ret                     # thread switch: actually run next thread
```

Zwischenzusammenfassung

Stapelspeicher, die auf Zweierpotenzgrößen bündig ausgerichtet sind, lassen den Fadenwechsel **inhärent frei von Wettlaufgefahr**

- der Prozesszeiger berechnet sich dann einfach aus dem Stapelzeiger
- um den am Stapelboden liegenden Prozessdeskriptor zu adressieren

Schutz des kritischen Abschnitts geschah durch **geschickte Wahl und Auslegung von Datenstrukturen** eines Betriebssystemkerns

- die Berechnung des Prozesszeigers erzeugt keinen Mehraufwand:

board	# Befehle	# Bytes
unsynchronisiert (S. 15)	16	32
Verdrängungssperre (S. 25)	26	51
bündige Stapelspeicher (S. 30)	16	28

Beachte: Konzept für Stapelspeicher garantiert maximaler Größe

- nur bedingt geeignet für Benutzerfäden (engl. *user-level threads*)

Gliederung

Resümee

- 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
- jeder Prozessor(kern) verfügt über einen eigenen **Prozesszeiger**
  - logisch in einem Prozesszeigerfeld $[0, NCORE - 1]$ gespeichert
  - mit einer Prozessor(kern)kennung als Index in dieses Feld:
    - Geräte-/Prozessorregister, Spezialregister, freigehaltenes Arbeitsregister
    - Zugriff darauf ausgelegt als sensitive bzw. privilegierte Operation
- **Verdrängung** macht die Einlastung zum **kritischen Abschnitt**
  (a) Verdrängung zeitweise unterbinden: Verdrängungssperre
    - verdrängungsfreie kritische Abschnitte schaffen, Epiloganalogie

  (b) Prozesszeiger aus dem Stapelzeiger berechnen
    - Stapelspeicher auf Zweierpotenzgrößen bündig auslegen
    - Prozessdeskriptor auf Stapelboden platzieren
    - Stapelbodenadresse durch Maskierung des Stapelzeigerwerts bestimmen

Literaturverzeichnis

[1] Garlan, D. ; Habermann, J. F. ; Notkin, D. :
Nico Habermann's Research: A Brief Retrospective.
In: *Proceedings of the 16th International Conference on Software Engineering (ICSE '94)*.
New York, NY, USA : ACM Press, 1994, S. 149–153

[2] Habermann, A. N. ; Flon, L. ; Cooprider, L. W.:
Modularization and Hierarchy in a Family of Operating Systems.
In: *Communications of the ACM* 19 (1976), Mai, Nr. 5, S. 266–272

[3] Lohmann, D. ; Kleinöder, J. :
Betriebssysteme.
http://www4.informatik.uni-erlangen.de/Lehre/WS07/V_BS, 2007

[4] Parnas, D. L.:
Some Hypothesis About the "Uses" Hierarchy for Operating Systems / TH Darmstadt, Fachbereich Informatik.
1976 (BSI 76/1). –
Forschungsbericht

[5] Schröder-Preikschat, W. :
The Logical Design of Parallel Operating Systems.
Upper Saddle River, NJ, USA : Prentice Hall International, 1994. –
ISBN 0–13–183369–3