— BS //—

Prozesse

Betriebssysteme, ©Wolfgang Schröder-Preikschat

Überblick

- Begriffsdeutung 2
- Prozesskontrollblock (*process control block*, PCB) 11
- Prozessabfertigung (*process dispatching*) 17
- „*cross-cutting concern*“: präemptiver Prozesswechsel 21
- Zusammenfassung 32

Ein Prozess . . .

. . . is a program in execution. A process is controlled and scheduled by the operating system. Same as *task*.

Task Same as *process*.

Stallings, *Operating Systems*

. . . is an abstraction of a program in execution.

Galli, *Distributed Operating Systems*

. . . is basically a program in execution. Associated with each process is its **address space**, a list of memory locations from some minimum (usually 0) to some maximum, which the process can read and write.

Tanenbaum, *Modern Operating Systems*

. . . genauer eigentlich ein Rechenprozeß, ist ein Ablauf in einem Rechensystem, wobei dieser Ablauf eine Verwaltungseinheit in dem jeweiligen Betriebssystem ist.

Siegert/Baumgarten, *Betriebssysteme*

. . . ist definiert durch einen *Adreßraum*, eine darin gespeicherte *Handlungsvorschrift* in Form eines sequentiellen Programms und einen *Aktivitätsträger*, der mit der Handlungsvorschrift verknüpft ist und sie ausführt. *Prozesse* [1] sind dynamische Objekte, die sequentielle Aktivitäten in einem System repräsentieren.

Nehmer/Sturm, *Systemsoftware*

Prozess ≠ Programm

A process may involve the execution of more than one program; conversely, a single program or routine may be involved in more than one process. [. . .]

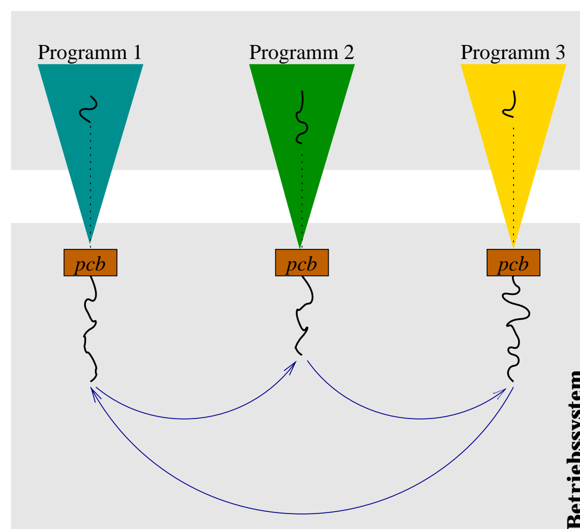
Hence the knowledge that a particular program is currently being executed does not tell us much about what activity is being pursued or what function is being implemented. It is largely for this reason that the concept of a process is more useful than that of a program when talking about operating systems. [2]

Koroutinen implementieren Prozesse

- die Koroutine „mechanisiert“ den autonomen Kontrollfluss eines Prozesses
  - durch sie wird ein Prozess zur Instanz
  - sie legt den Grundstein für eine Prozessinkarnation
  - mit ihr wird die Prozessorzuteilung an (ausführbereite) Prozesse möglich
- Kontextvariablen von Koroutinen sind Prozessdeskriptoren (→ p. 11) sehr ähnlich
  - ein Prozessdeskriptor „enthält“/„ist“ mindestens eine solche Variable
  - die Notwendigkeit weiterer Attribute ist sehr problemspezifisch
- Betriebssysteme sind (logisch) Programme mit vielen solchen Kontextvariablen
  - erst dadurch werden z.B. Mehrfädigkeit bzw. Mehrprogrammbetrieb erreicht

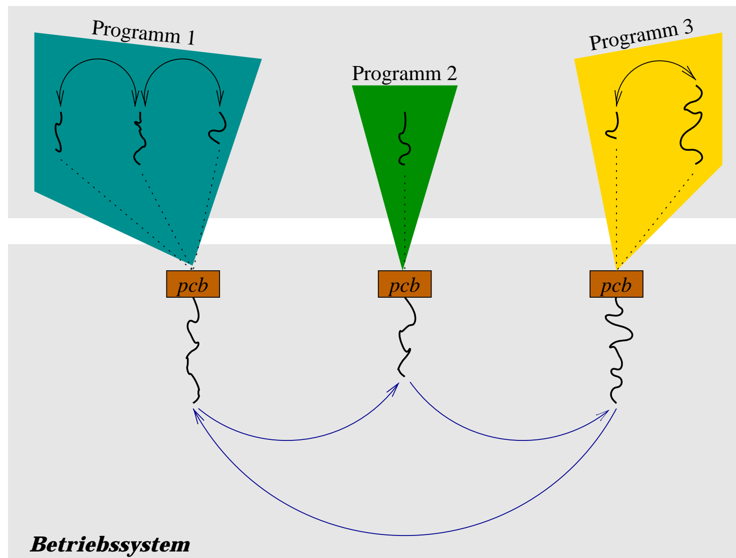
Mehrprogrammbetrieb vs. Koroutinen

(1) Der kooperative Wechsel zwischen Benutzerprogrammen ist auf der Ebene mit Koroutinen nicht erreichbar. Dazu müsste ein Programm Zugriff auf Kontextvariablen haben, die Koroutinen anderer Programme repräsentieren.



(2) Der kooperative Wechsel zwischen Koroutinen des Betriebssystems ist unproblematisch, da alle Kontextvariablen dem Programm „Betriebssystem“ bekannt sind. Jede Koroutine ist dabei ein abstrakter Prozessor für ein Benutzerprogramm.

Mehrfädige Programme im Mehrprogrammbetrieb



(1) Die Mehrfädigkeit von Benutzerprogrammen wird erreicht, indem der abstrakte Prozessor „Betriebssystemprozess“ von mehreren Koroutinen des Benutzerprogramms geteilt wird. Dem Betriebssystem sind diese Programmfäden nicht bekannt, was auch bedeutet, dass der Wechsel zwischen ihnen ohne Betriebssystemintervention erfolgt.

(2) Dadurch, dass sich die Programmfäden des Benutzerprogramms denselben Betriebssystemprozess teilen, kann das Betriebssystem auch keine für diese Fäden spezifischen Verwaltungsmaßnahmen ergreifen. So betrifft dann etwa der Aufruf eines blockierenden Systemaufrufs alle Fäden.

Prozessmodelle

schwergewichtiger Prozess *heavyweight process*

- Prozess und Benutzeradressraum bilden eine Einheit (→ p. 5)
- Prozesswechsel bedeutet zwei Adressraumwechsel $AR_x \Rightarrow BS \Rightarrow AR_y$

leichtgewichtiger Prozess *lightweight process*

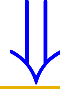
- Prozess und Adressraum sind voneinander entkoppelt (→ p. 6)
- Prozesswechsel bedeutet einen Adressraumwechsel $AR_x \Rightarrow BS \Rightarrow AR_x$

federgewichtiger Prozess *featherweight process*

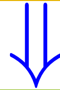
- Prozesse und Adressraum bilden eine Einheit
- Prozesswechsel bedeutet keinen Adressraumwechsel: *Betriebssystemprozesse*

Prozess**benutzt**hierarchie

schwergewichtiger Prozess



leichtgewichtiger Prozess



federgewichtiger Prozess

Prozess $\neq$ Prozessinkarnation

Prozess ist ein abstraktes Gebilde

- ein „Programm in Ausführung", ein asynchroner Programmablauf
- ein „Ablauf", der eine Verwaltungseinheit ist **?**

Prozessinkarnation ist ein konkretes Gebilde

- die „physische Instanz" des abstrakten Gebildes „Prozess"
  - gebunden an (Software-) Betriebsmittel
  - verbunden mit einer *Identität*
- die *Verwaltungseinheit*, die einen Prozess beschreibt und repräsentiert

Betriebsmittel einer Prozessinkarnation

Hardware konkrete, „greifbare“ Betriebsmittel

- Prozessor: zur Ausführung der Elementaroperationen *CPU*
- Peripherie: zur Durchführung der Ein-/Ausgabeoperationen *I/O*
- Speicher: zur Aufbewahrung von Programm und Daten *RAM/ROM*

Software abstrakte, „un(be)greifbare“ Betriebsmittel

- Prozess{deskriptoren,identifikation}
- Puffer, „Geheimlager“ (*cache*), Blöcke, {{E/A,Datei}-}Deskriptoren
- Seiten{,rahmen,tabellen} und/oder Segment{e,tabellen}

Prozessdeskriptor (1)

- Dreh- und Angelpunkt, der alle prozessbezogenen Betriebsmittel bündelt
  - Speicher- und, ggf., Adressraumbelegung †
    - Text-, Daten-, Stapelsegmente (*code, data, stack*)
  - Dateideskriptoren und -knoten („*inode*“) †
    - {„Versteck“ (*cache*),Puffer}deskriptoren, Datenblöcke
  - Datei, die das vom Prozess ausgeführte Programm repräsentiert †

- zentrales Objekt, das Prozess- und Prozessorzustände beschreibt
  - Laufzeitkontext des zugeordneten Programmfadens
  - gegenwärtiger Abfertigungszustand (*Scheduling*-Informationen) †
  - anstehende Ereignisse bzw. erwartete Ereignisse †
  - Benutzerzuordnung und -rechte †

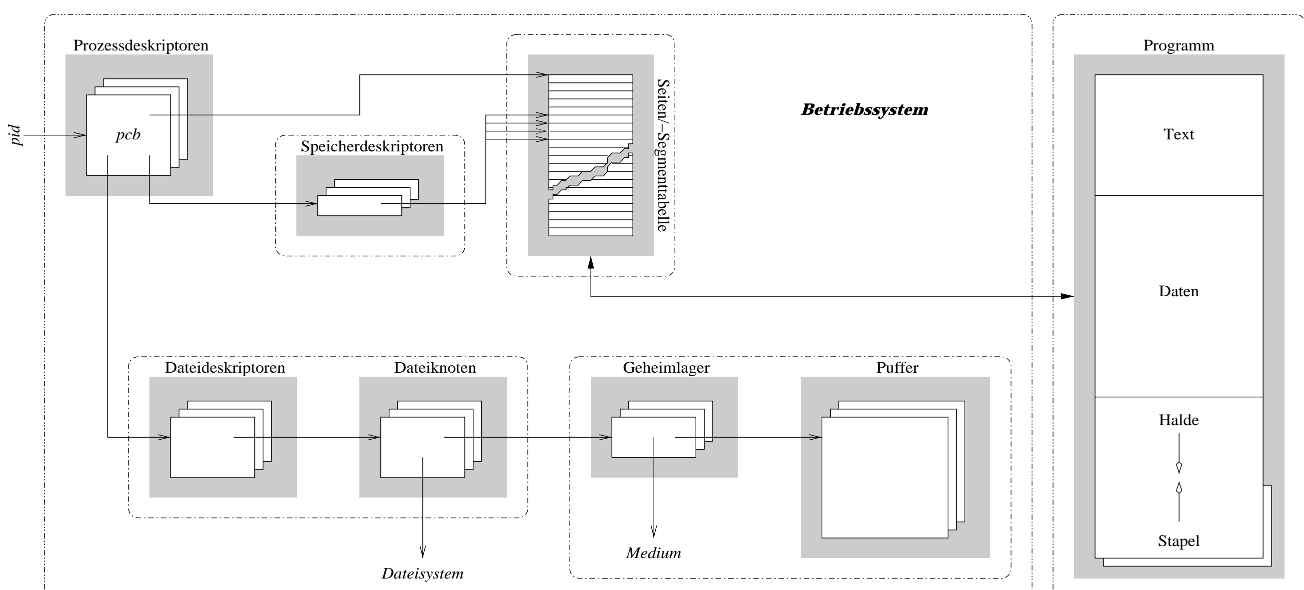
Prozessdeskriptor (2)

- Aufbau und Struktur ist höchst abhängig von Betriebsart und Zweck

  † (1) Adressraumdeskriptoren sind nur notwendig im Falle von Systemen, die eine Adressraumisolation erfordern. (2) Für ein (Spezialzweck-) Sensor-/Aktorsystem haben Dateideskriptoren/-knoten wenig Bedeutung. (3) In ROM-basierten (Spezialzweck-) Systemen durchlaufen Prozesse oft immer ein und dasselbe Programm. (4) In Einbenutzersystemen ist es wenig sinnvoll, prozessbezogene Benutzerrechte verwalten zu wollen. (5) Bei statischem Scheduling ist die Buchführung von Abfertigungszuständen durchaus verzichtbar. (6) Ebenso fällt Ereignisverwaltung nur an bei ereignisgesteuerten und/oder präemptiven Systemen. (7) . . .

- es gibt *nicht* die eine Ausprägung — sie „erzwingen“ zu wollen, ist falsch

Dynamische Datenstruktur „Prozessinkarnation“



Prozessdeskriptor — problemorientierte Kontextvariable

- *process control block*: Spezialisierung der Kontextvariablen der Koroutine

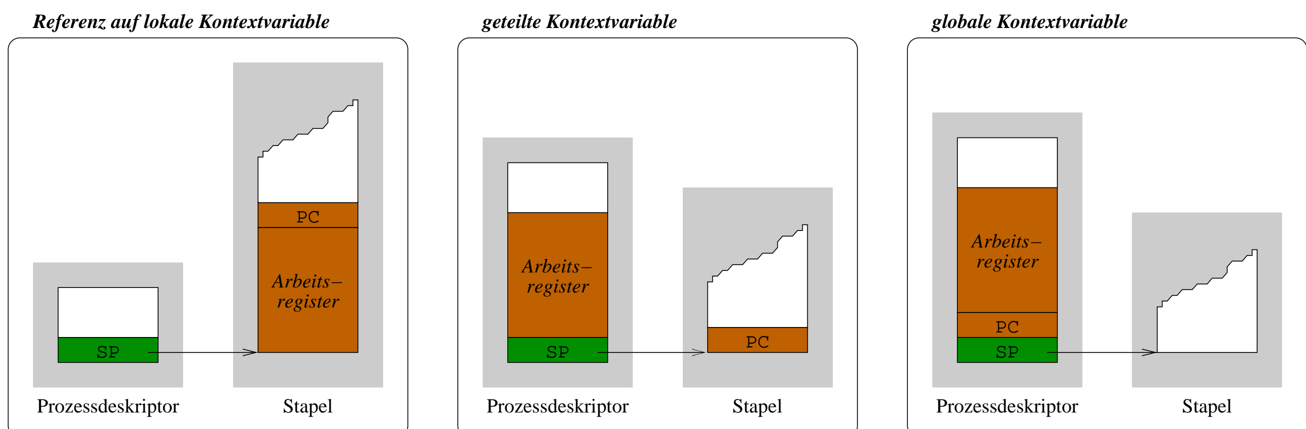
  Kontextvariablenreferenz Der PCB speichert die Referenz „SP“ auf den als lokale *resume*-Variable vorliegenden Koroutinenzustand — Minimierung des Bedarfs an statischem Speicher.

  geteilte Kontextvariable Der PCB speichert die (nicht-flüchtigen) Arbeitsregister und die Referenz „SP“ auf die lokale *resume*-Variable „PC“ — nur der automatisch von der CPU gesicherte Aktivierungsblock verbleibt im Stapel.

  globale Kontextvariable Der PCB speichert den als globale *resume*-Variable vorliegenden Koroutinenzustand — der gesamte Prozessorstatus des Prozesses bzw. der Koroutine ist direkt adressierbar.

- mindestens „enthalten“ ist der Laufzeitkontext des zugeordneten Fadens

Prozessdeskriptor vs. Fadenkontext



Objekt der Buchhaltung

- ein laufender Prozess kann das Verhalten „stehender" Prozesse beeinflussen
  - blockierende Synchronisation (Semaphor): `V()`$_x$ deblockiert `P()`$_y$
  - Ausnahmesituation(en) propagieren: `raise()`$_x$ an `catch()`$_y$
  - Ereignis signalisieren: `signal()`$_{Epilog_x}$ an `pickup()`$_y$
  - Jobkontrolle: `pause()`$_x$ appliziert auf `P`$_y$
- die Aktionen eines laufenden Prozesses sind vom Betriebssystem zu überwachen
  - beanspruchte Betriebsmittel aufzeichnen
  - bei der Betriebsmittelnutzung die Zugriffsrechte überprüfen
  - Kostenrechnung (*accounting*) beim Betriebsmittelgebrauch durchführen
  - beim Zugriff auf ein bereits belegtes unteilbares Betriebsmittel blockieren

Laufender Prozess — Aktive Prozessinkarnation

- zur Buchhaltung muss jederzeit die akuelle Prozessinkarnation definiert sein
  - „einfache" technische Lösung, einen Zeiger einrichten: `tocObject* soul;`
    `tocObject` implementiert den PCB eines Prozesses
    `soul` identifziert die jeweils aktive Prozessinkarnation („Seele")
  - ein *cross-cutting concern* kann diese Lösung aber zum Problem machen. . .
- die Prozessumschaltung besteht damit aus zwei elementaren Schritten:
  1. die Aktualisierung des Zeigers auf die neue Prozessinkarnation
  2. der Kontextwechsel zwischen den beiden beteiligten Koroutinen
- der „Prozessabfertiger" (*process dispatcher*) führt diese Schritte durch

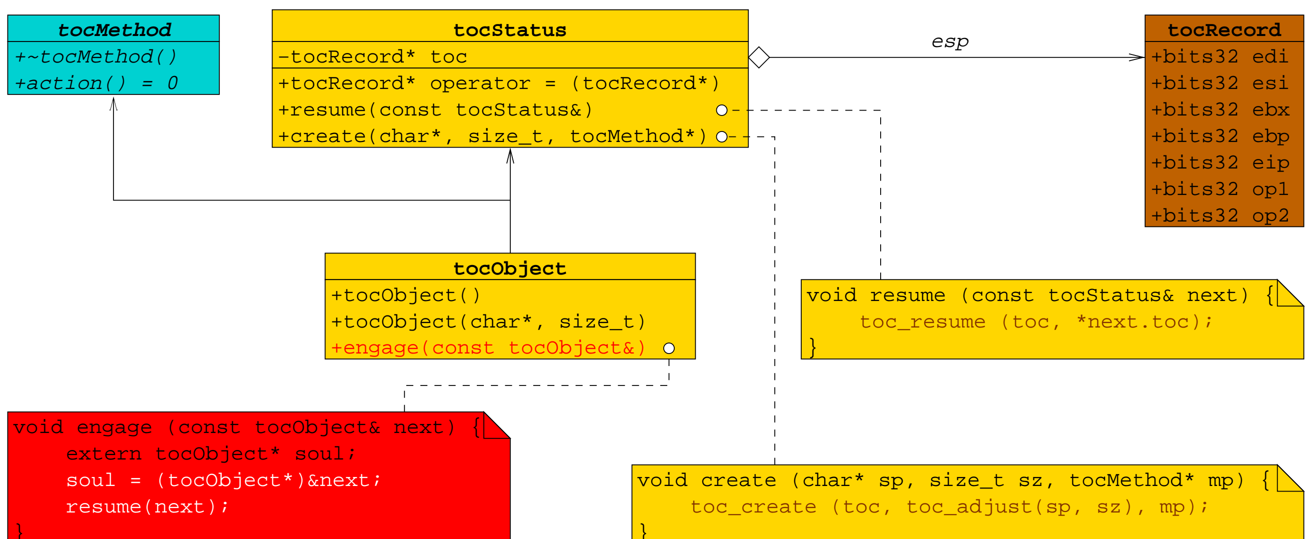
Aufgabenteilung zwischen *Scheduler* und *Dispatcher*

process scheduler trifft strategische Entscheidungen zur Prozessorvergabe

- betrachtet wird immer eine Menge lauffähiger Prozesse
  - die Prozesse sind allgemein in einer CPU-Warteschlange aufgereiht
  - welche Position darin ein PCB einnimmt, obliegt der *Scheduling*-Strategie
- der aktuell laufende Prozess ist immer von der Entscheidung mit betroffen
  - dazu muss der PCB des laufenden Prozesses „jederzeit greifbar“ sein
  - vor der Umschaltung ist dieser PCB (beim *Dispatching*) zu vermerken
- ein ausgewählter neuer Prozess wird dem *Dispatcher* übergeben

process dispatcher setzt die Entscheidungen durch und schaltet Prozesse um

thread of control — toc (1)



thread of control — toc (2)

```
#include "tocStatus.h"
#include "tocMethod.h"

void toc_lifter (tocMethod* ap) { for (;;) ap->action(); }

void toc_create (tocRecord*& toc, char* sp, tocMethod* ap) {
    *--(bits32*)sp = 0;
    *--(bits32*)sp = (bits32)ap;
    *--(bits32*)sp = 0;
    *--(bits32*)sp = (bits32)toc_lifter;
    *--(bits32*)sp = 0;
    *--(bits32*)sp = 0;
    *--(bits32*)sp = 0;
    *--(bits32*)sp = 0;
    toc = (tocRecord*)sp;
}

inline char* toc_adjust (char* p, size_t l) { return p + l; }
```

```
.text
    .align 2
    .globl toc_resume
    .globl _toc_resume
toc_resume:
_toc_resume:
    push %ebp
    push %ebx
    push %esi
    push %edi
    movl 20(%esp),%eax
    movl %esp,0(%eax)
    movl 24(%esp),%esp
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
```

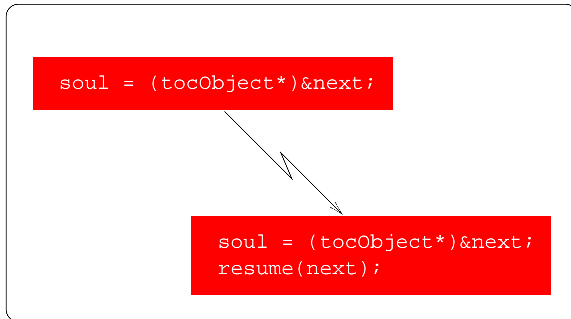

„*Cross-Cutting Concern*“

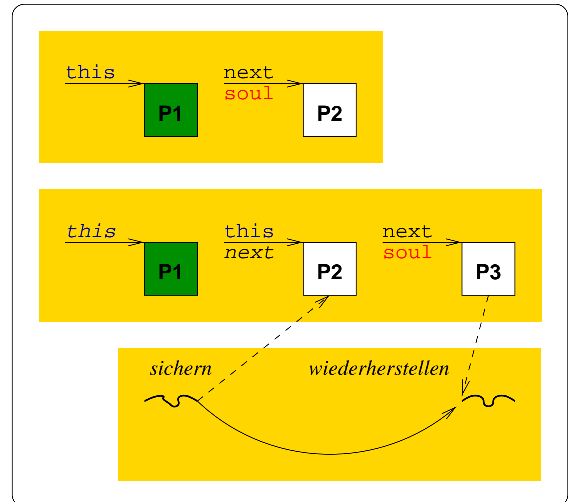
- die Prozessumschaltung/-abfertigung ist unter bestimmten Umständen kritisch

  `engage()` Angenommen, ein Prozess gibt freiwillig die Kontrolle über die CPU ab. Der nächste auszuführende Prozess wurde bereits durch umsetzen von `soul` vermerkt und ein Interrupt tritt auf, bevor die Umschaltung (durch `resume()`) wirksam werden kann. Der Interrupt führt zur Verdrängung des laufenden Prozesses durch den *Scheduler*, der zur Durchsetzung der Prozessorzuteilung den *Dispatcher* aktiviert und damit die überlappte Ausführung von `engage()` bewirkt. (→ p. 19)

- durch Interrupts ausgelöstes *präemptives Scheduling* ist höchst problematisch

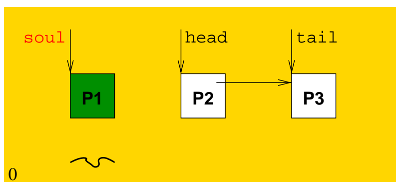
Fehlgeleitete Zustandssicherung



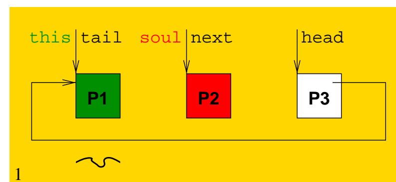


Im Fall einer Verdrängung wird der Zustand des laufenden Prozesses P_1 fälschlicherweise in den PCB von P_2 gesichert, da im *Dispatcher* soul vor dem Interrupt bereits auf P_2 umgesetzt wurde. Der vom Interrupt aktivierte *Scheduler* versteht den über soul identifizierten Prozess immer als den laufenden Prozess, den es zu verdrängen gilt. Wirklich aktiv ist jedoch immer noch die Prozessinkarnation von P_1, so dass die Zustandssicherung für P_1 fehlgeleitet wird in den PCB von P_2! Und die Reaktivierung von P_1?

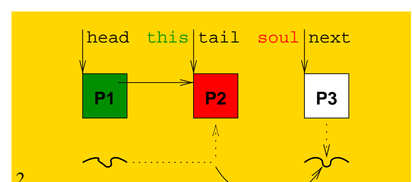
Logbuch einer Irrfahrt



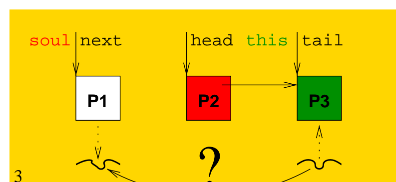
P1 läuft, P2 und P3 sind laufbereit.



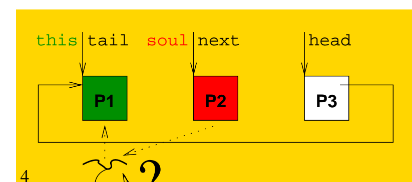
P1 gibt freiwillig die CPU ab, läuft jedoch noch.
P2 soll neuer laufender Prozess werden.



"P2" wird mit dem Zustand von P1 verdrängt.
P3 wird neuer laufender Prozess.



P3 gibt freiwillig die CPU ab.
P1 wird neuer laufender Prozess, läuft jedoch an seiner vorletzten Unterbrechungsstelle weiter.



P1 gibt freiwillig die CPU ab. P2 wird reaktiviert und setzt seine Ausführung im Kontext von P1 fort.

Koordination der Prozessumschaltung

- die Abfertigung der beiden beteiligten Prozesse bildet einen kritischen Abschnitt
  - kritisch sind in dem Zusammenhang zwei Variablen: `soul` und `SP`
  - die Aktualisierung beider Variablen muss (potentiell) atomar erfolgen[1]
- es zeichnen sich zwei Lösungswege für das Nebenläufigkeitsproblem ab:
  1. die überlappte Aktualisierung der beiden kritischen Variablen unterbinden
  2. aus zwei mach eine: die eine Variable (`soul`) auf die andere (`SP`) abbilden
- die Lösungen sollten die Wiederverwendbarkeit von `engage` nicht einschränken

[1] Wenn aus irgendwelchen Gründen die überlappte Ausführung der Prozessumschaltung möglich werden könnte.

„*Cross-Cutting Concern*" — Prozessumschaltung

```
#include "lock.h"

void engage (const tocObject& next) {
    extern tocObject* soul;

    lock.enter();
    soul = (tocObject*)&next;
    resume(next);
    lock.leave();
}
```

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

void engage (const tocObject& next) {
    extern tocObject* soul;

    cli();
    soul = (tocObject*)&next;
    resume(next);
    sti();
}
```

Gute Lösung [warum?] Schlechte Lösung

Hochfahren eines Prozesses

- die Prozessumschaltung in einem aktiven kritischen Abschnitt ist „knifflig“:
  - unproblematisch ist der Wechsel zurück zu einem unterbrochenen Prozess
    - er kehrt zum *Dispatcher* in den kritischen Abschnitt zurück
    - damit wird er auch selbst (aktiv) den kritischen Bereich verlassen
  - problematisch ist dagegen der Wechsel hin zu einem neu erzeugten Prozess
- erstmalig aktivierte Prozesse sind „ordentlich“ hochzufahren
  - sie müssen kritische Abschnitte verlassen, die sie selbst nie betreten haben
  - sie müssen Vorgänge abschließen, die ihr Vorgänger noch begonnen hatte
- jedoch woher sind die ggf. jeweils noch abzuschließenden Vorgänge bekannt?

„*Cross-Cutting Concern*“ — Prozesshochfahren

```
#include "tocMethod.h"

void toc_lifter (tocMethod* ap) {
    lock.leave();
    ...
    for (;;) ap->action();
}
```

- wieviel ist überhaupt noch zu leisten?
  - kritischen Abschnitt freigeben. . .
  - *Dispatcher* „sauber“ verlassen. . .
  - das System aufräumen. . .
  - die Reihenfolge beachten. . .
  - Systemzustände „bereinigen“. . .
- wie letztlich zu verfahren ist, hängt vom Einsatz-/Anwendungsszenario ab
  - je nach Systemausprägung gibt es mehr oder weniger viele Hochfahrvarianten
- ggf. besser: die Aktionen in einer Spezialisierung der Prozessabstraktion kapseln

Zweckentfremdung des Stapelzeigers

- angenommen, ein Prozessdeskriptor liegt im Stapelspeicher seines Prozesses
  - der Stapelzeiger ist dann auch ein Zeiger auf den PCB des Prozesses
  - `soul` ist damit obsolet, denn die PCB-Variable wurde auf `SP` abgebildet
  - aktualisieren des PCB-Zeigers und umsetzen von `SP` sind dann atomar[2]
- der Prozessdeskriptor muss dazu an einer berechenbaren Stapeladresse liegen
  - die Adresse liegt dabei im oberen oder unteren Bereich des Stapels
  - eine Funktion liefert die PCB-Adresse anhand eines „beliebigen“ `SP`-Wertes
- die Berechnung der Prozessdeskriptoradresse ist unkritisch und wenig aufwendig

[2]Sofern die CPU die erforderliche Zuweisungsoperation an `SP` in einem Schritt (d.h., atomar) ausführen kann.

Prozessdeskriptor als „lokale Variable“

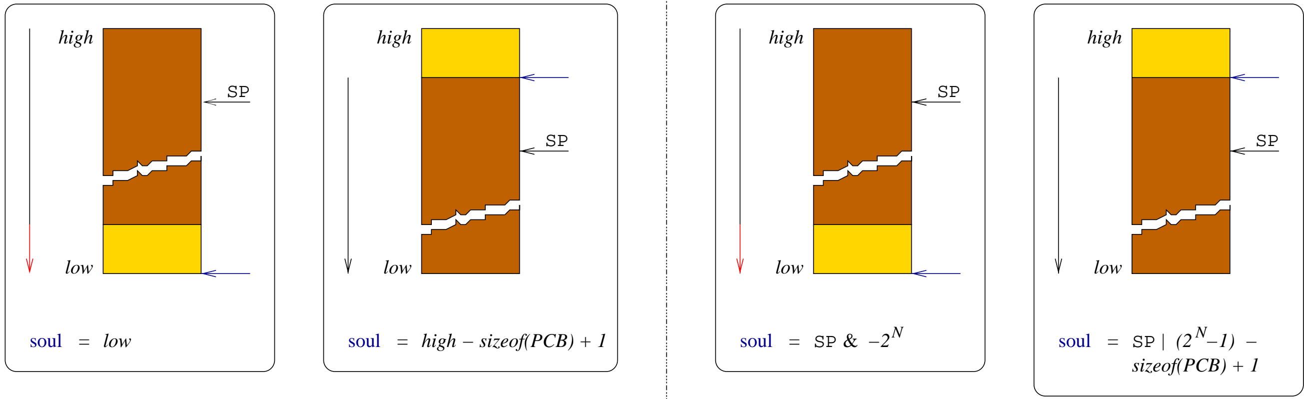
- die Berechnung der PCB-Adresse hängt ab vom etablierten Adressraummodel

  physikalischer Adressraum die PCB-Adressen sind alle verschieden
  - die effektive Adresse wird durch Software berechnet
  - Annahmen über Lage und Größe des Stapels sind zu treffen

  logischer Adressraum die PCB-Adressen sind alle gleich
  - die effektive Adresse wird durch Hardware (MMU/CPU)[3] berechnet
  - Annahmen über bestimmte Prozessoreigenschaften sind zu treffen
- nicht jeder Rechner ist mit einer MMU ausgestattet — die wenigsten! [3]

[3]Eine *memory management unit* (MMU) bildet logische auf physikalische Adressen ab. Eine CPU kann z.B. auf Grundlage der Adressierungsart „indirekt mit Versatz“ (*indirect with displacement*) gleiches erreichen, wenn dazu ein Basisregister durchgängig zur Verfügung steht bzw. vom Übersetzer frei gehalten wird.

Berechnung der Prozessdeskriptoradresse



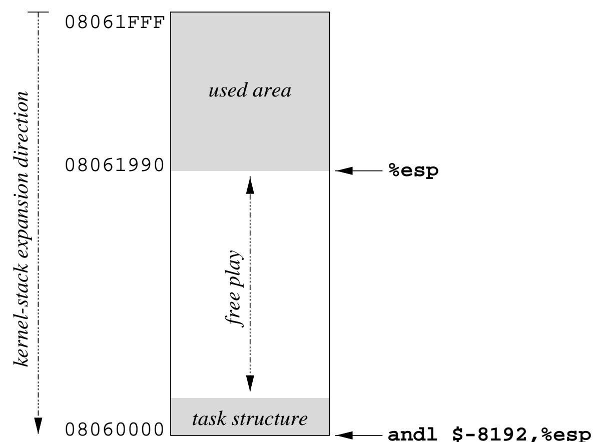
logische Adressen, MMU

physikalische Adressen, „*Alignment*“

Fallstudie Linux — „task structure“ Berechnung

```
#define GET_CURRENT(reg) \
    movl %esp, reg; \
    andl $-8192, reg;
```

Die *task structure* liegt an der niedrigen Adresse des „*supervisor mode*“-Stapels. Alle diese Stapel haben die gleiche Größe 2^N, $N = 13$ (8 KB) und beginnen an einer entsprechend dieser Größe ausgerichteten (logischen) Adresse. Damit kann die Adresse der *task structure* durch eine einfache Maskenoperation aus dem aktuellen Wert des (*supervisor*) SP berechnet werden.



Zusammenfassung

- Prozesse werden programmiertechnisch auf Basis von Koroutinen implementiert
  - Betriebssysteme sind Programme, die ggf. sehr viele Koroutinen enthalten
  - Betriebssysteme sind typischerweise mehrfädige, nebenläufige Programme
- Prozess(inkarnation) und Adressraum bilden nicht zwingend eine Einheit
  - beide Konzepte ergänzen einander, sie bedingen sich nicht
  - {schwer,leicht,feder}gewichtige Prozess{e,deskriptoren}
- die Prozessumschaltung setzt sich aus kritischen Anweisungsfolgen zusammen
  - kritischer Abschnitt vs. „Stapelzeigerzweckentfremdung"
  - die Koordinierung von Nebenläufigkeit ist ein *cross-cutting concern*

Referenzen

[1] E. W. Dijkstra. *Cooperating Sequential Processes*. Programming Languages. Academic Press, New York, 1968.

[2] A. M. Lister and R. D. Eager. *Fundamentals of Operating Systems*. The Macmillan Press Ltd., fifth edition, 1993. ISBN 0-333-59848-2.

[3] D. Tennenhouse. Proactive Computing. *Communications of the ACM*, 43(5):43–50, May 2000.