

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil B – VI.2 Rechnerorganisation: Maschinenprogramme

3. Juli 2025

Rüdiger Kapitza

(© Wolfgang Schröder-Preikschat, Rüdiger Kapitza)



Lehrstuhl für Informatik 4
Systemsoftware



Friedrich-Alexander-Universität
Technische Fakultät

Einführung

Hybrid

Programmhierarchie

Hochsprachenkonstrukte

Assemblersprachenanweisungen

Betriebssystembefehle

Organisationsprinzipien

Funktionen

Komponenten

Zusammenfassung

Einführung

Hybrid

Programmhierarchie

Hochsprachenkonstrukte

Assemblersprachenanweisungen

Betriebssystembefehle

Organisationsprinzipien

Funktionen

Komponenten

Zusammenfassung

- Maschinenprogramm als Entität einer **hybriden Schicht** verstehen
 - Instruktionen an die Befehlssatzebene, die direkt ausgeführt werden
 - Instruktionen an das Betriebssystem, die partiell interpretiert werden
- Ebene_[2,3] als **Programmhierarchie** virtueller Maschine vertiefen
 - indem exemplarisch für x86 und Linux das Zusammenspiel dieser Maschinen zur Diskussion gestellt wird
 - dabei die prinzipielle Funktionsweise von Systemaufrufen erkennen
- **Grobstruktur** von Maschinenprogrammen im Ansatz kennenlernen
 - mit dem Laufzeitsystem und den Systemaufrufstümpfen als zwei zentrale Bestandteile der Systemsoftware
 - inklusive Anwendungsroutinen zusammengebunden zum **Lademodul**

Auch wenn wir die Programmbeispiele symbolisch dargestellt sehen, ist zu beachten, dass Maschinenprogramme letztlich numerischer Natur sind. (vgl. [3, S. 18])

Einführung

Hybrid

Hybride Schicht in einem Rechensystem

- Maschinenprogramme enthalten zwei Sorten von Befehlen:
 - i **Maschinenbefehle** der Befehlssatzebene (ISA)
 - normalerweise direkt interpretiert durch die Zentraleinheit¹
 - ausnahmsweise partiell interpretiert durch das Betriebssystem
 - ii **Systemaufrufe** an das Betriebssystem
 - normalerweise partiell interpretiert durch das Betriebssystem

Hybrid (lat. *hybrida* Bastard, Mischling, Frevelkind)^a

^agr. *hýbris* Übermut, Anmaßung

„etwas Gebündeltes, Gekreuztes oder Gemischtes“ [6]

- ein System das zwei Techniken miteinander kombiniert:
 - i Interpretation von Programmen der Befehlssatzebene
 - ii partielle Interpretation von Maschinenprogrammen
- ein Maschinenprogramm ist **Hybridsoftware**, die auf Ebene_[2,3] läuft

¹central processing unit, CPU

Betriebssystem $\equiv$ Programm der Befehlssatzebene

- ein Betriebssystem implementiert die Maschinenprogrammebene
 - es zählt damit selbst nicht zur Klasse der Maschinenprogramme
 - es setzt normalerweise keine Systemaufrufe (an sich selbst) ab
 - es unterbricht sich normalerweise niemals von selbst
 - gleichwohl sollten Betriebssysteme es zulassen, in der Ausführung eigener Programme unterbrochen werden zu können
 - nicht durch Systemaufrufe
 - aber durch *Traps* oder *Interrupts* — **Ausnahmen**
- ↪ sie interpretieren die eigenen Programme nur eingeschränkt partielle

Teilinterpretation von Betriebssystemprogrammen

Bewirkt **indirekt rekursive Programmausführungen** im Betriebssystem^a und erfordert daher die Fähigkeit zum **Wiedereintritt** (re-entrance). Je nach Operationsprinzip^b des Betriebssystems ist dies zulässig oder (temporär) unzulässig.

^aausgelöst durch synchrone/asynchrone Unterbrechungen

^bnichtblockierende/blockierende Synchronisation

Einführung

Hybrid

Programmhierarchie

Hochsprachenkonstrukte

Assemblersprachenanweisungen

Betriebssystembefehle

Organisationsprinzipien

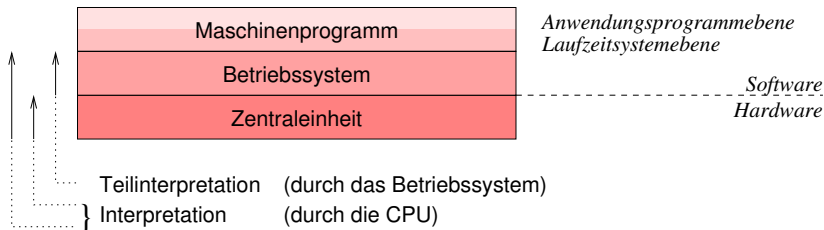
Funktionen

Komponenten

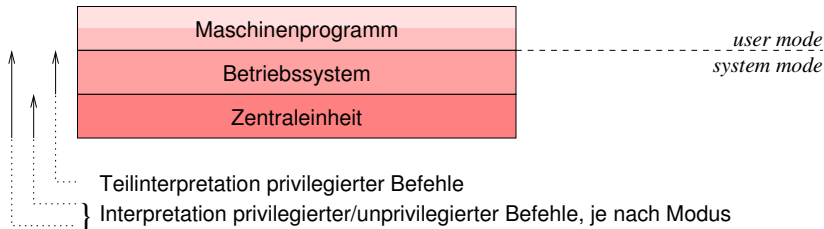
Zusammenfassung

- Maschinenprogramme setzen sich aus Anweisungen zusammen, die **ohne Übersetzung** von einem Prozessor ausführbar sind
 - gleichwohl werden sie (normalerweise) durch Übersetzung generiert
 - nahezu ausschließlich automatisch: Kompilierer, Assemblierer, Binder
 - in seltenen Fällen manuell: **nativer Kode** (*native code*)²
 - sie repräsentieren sich technisch als **Lademodul** (*load module*)
 - erzeugt durch Dienstprogramme (*utilities*): `gcc(1)`, `as(1)`, `ld(1)`
 - geladen, verarbeitet und entsorgt durch Betriebssysteme
 - d.h., als **ausführbares Programm** und in numerischer Form
- Grundlage für die Entwicklung von Maschinenprogrammen bilden Hoch- und Assemblersprachen, und zwar für jede Art Software:
 - Anwendungsprogramme, Laufzeitsysteme und Betriebssysteme
 - symbolisch repräsentiert auf Ebene_[4,5], numerisch auf Ebene₃

²Binärkode des realen Prozessors, auch: Maschinenkode.



- **Maschinenprogramm** = Anwendungsprogramm + Laufzeitsystem
 - beide Teilebenen liegen im selben **Adressraum**, der zudem (logisch) per **Speicherschutz** von anderen Adressräumen isoliert ist
 - einfache Unterprogrammaufrufe aktivieren das Laufzeitsystem
- **Ausführungsplattform** = Betriebssystem + Zentraleinheit (CPU)
 - Verarbeitung eines Maschinenprogramms durch einen Prozessor, der in Hard- und Software implementiert vorliegt
 - komplexe **Systemaufrufe** (*system calls*) aktivieren das Betriebssystem



- **Maschinenprogramm** = Benutzerebene (*user level, user space*)
 - eingeschränkter Umgang mit Merkmalen der Befehlssatzebene in Bezug auf Maschinenbefehle, Hardwarekomponenten und Peripheriegeräte
 - nur **unprivilegierte Operationen** werden direkt ausgeführt, privilegierte Operationen erfordern den **Moduswechsel** $\leadsto$ Systemaufruf
- **Ausführungsplattform** = Systemebene (*system level, kernel space*)
 - uneingeschränkter Umgang mit den Merkmalen der Befehlssatzebene
 - alle Maschinenbefehle werden direkt ausgeführt, alle Operationen gültig

Programmhierarchie

Hochsprachenkonstrukte

- ein auf Ebene₅ symbolisch repräsentiertes Programm der Ebene₃:

```
1 void echo() {  
2     char c;  
3     while (read(0, &c, 1) == 1) write(1, &c, 1);  
4 }
```

echo.c

Funktion read(2) überträgt ein Zeichen von Standardeingabe (0) an die Arbeitsspeicheradresse der lokalen Variablen c, deren Inhalt anschließend mit der Funktion write(2) zur Standardausgabe (1) gesendet wird. Die Schleife terminiert durch Unterbrechung, unter UNIX z.B. nach Eingabe von ^C.

Programmhierarchie

Assemblersprachenanweisungen

- dasselbe Programm (S. 13) symbolisch repräsentiert auf Ebene 4:³

```
1  .file "echo.c"
2  .text
3  .p2align 4,15
4  .globl echo
5  .type echo, @function
6  echo:
7      pushl %ebx
8      subl $40, %esp
9      leal 28(%esp), %ebx
10     jmp .L2
11     .p2align 4,7
12     .p2align 3
13     .L3:
14         movl $1, 8(%esp)
15         movl %ebx, 4(%esp)
16         movl $1, (%esp)
17         call write
18     .L2:
19         movl $1, 8(%esp)
20         movl %ebx, 4(%esp)
21         movl $0, (%esp)
22         call read
23         cmpl $1, %eax
24         je .L3
25         addl $40, %esp
26         popl %ebx
27     ret
```

- **unaufgelöste Referenzen** der Systemfunktionen `read(2)` und `write(2)` werden vom Binder `ld(1)` aufgelöst $\mapsto$ `libc.a`

³Übersetzung von `echo.c` mit `-S` liefert `echo.s`

Programmhierarchie

Betriebssystembefehle

■ Stümpfe der Systemfunktionen auf Ebene ₃, symbol. aufbereitet:

```
1  read:
2      push %ebx
3      movl 16(%esp),%edx
4      movl 12(%esp),%ecx
5      movl 8(%esp),%ebx
6      mov  $3,%eax
7      int  $0x80
8      pop  %ebx
9      cmp  $-4095,%eax
10     jae  __syscall_error
11     ret
```

```
12  write:
13     push %ebx
14     movl 16(%esp),%edx
15     movl 12(%esp),%ecx
16     movl 8(%esp),%ebx
17     mov  $4,%eax
18     int  $0x80
19     pop  %ebx
20     cmp  $-4095,%eax
21     jae  __syscall_error
22     ret
```

- nach Kompilation⁴ Verwendung der disassemble-Operation von gdb(1)

■ Systemaufruf wird durch `int $0x80` (*software interrupt*) ausgelöst

- Operationskode in %eax
- Parameter in %ebx, %ecx und %edx
- Resultat in %eax zurück

```
23  __syscall_error:
24     neg  %eax
25     mov  %eax,errno
26     mov  $-1,%eax
27     ret
28
29     .comm  errno,16
```

⁴Übersetzung von `echo.c` inklusive abschließender Bindung.

■ Systemaufrufzuteiler (system call dispatcher):

- ein auf Ebene₄ symbolisch repräsentiertes Programm der Ebene₂
- kernel-source-2.4.20/arch/i386/kernel/entry.S (Auszug)

Prolog

```

1  system_call:
2      pushl %eax
3      cld
4      pushl %es
5      pushl %ds
6      pushl %eax
7      pushl %ebp
8      pushl %edi
9      pushl %esi
10     pushl %edx
11     pushl %ecx
12     pushl %ebx
13     ...

```

Abruf und Ausführung

```

14     ...
15     cmpl $(NR_syscalls),%eax
16     jae  badsys
17     call *sys_call_table(,%eax,4)
18     movl %eax,24(%esp)
19     ret_from_sys_call:
20     ...
21     jmp  restore_all
22     badsys:
23     movl $-ENOSYS,24(%esp)
24     jmp  ret_from_sys_call

```

Epilog

```

25     restore_all:
26     popl  %ebx
27     popl  %ecx
28     popl  %edx
29     popl  %esi
30     popl  %edi
31     popl  %ebp
32     popl  %eax
33     popl  %ds
34     popl  %es
35     addl  $4,%esp
36     iret

```

4-12

- Sicherung des Prozessorzustands des Maschinenprogramms

7-12

- Übernahme der aktuellen Parameter von Systemaufrufen

15-18

- Überprüfung des Operationskodes und Aufruf der Systemfunktion

26-34

- Wiederherstellung des gesicherten Prozessorzustands

36

- Wiederaufnahme der Ausführung des Maschinenprogramms

- **Befehlsabruf- und -ausführungszyklus** (*fetch-execute cycle*) zur Ausführung von Systemaufrufen
 1. Prozessorstatus des unterbrochenen Programms sichern.....Prolog
 - Aufforderung der CPU zur Teilinterpretation nachkommen
 2. Systemaufruf interpretieren Abruf und Ausführung
 - i Systemaufrufnummer (Operationskode) abrufen
 - ii auf Gültigkeit überprüfen und ggf. Fehlerbehandlung auslösen
 - iii bei gültigem Operationskode, zugeordnete Systemfunktion ausführen
 3. Prozessorstatus wiederherstellen und zurückspringen Epilog
 - Beendigung der Teilinterpretation der CPU „mitteilen“
 - Ausführung des unterbrochenen Programms wieder aufnehmen
- mangels „echter“ **Systemimplementierungssprache**⁵ ist hier in dem Kontext der Einsatz von Assemblersprache erforderlich
 - Teilinterpretation erfordert kompletten Zugriff auf den Prozessorstatus
 - dieser ist nicht mehr Teil des Programmiermodells einer Hochsprache

⁵Höhere Programmiersprache mit hardwarenahen Sprachelementen.

- ein auf Ebene₅ symbolisch repräsentiertes Programm der Ebene₂:
 - kernel-source-2.4.20/fs/read_write.c (Auszug)

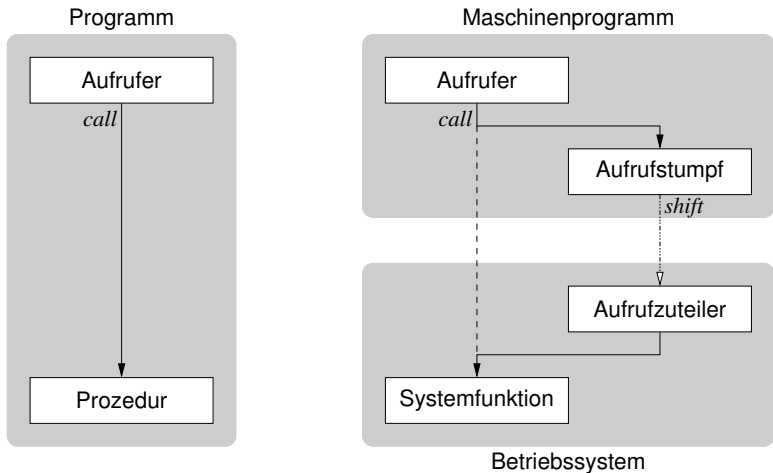
```
1  asmlinkage
2  ssize_t sys_read(unsigned int fd, char *buf, size_t count) {
3      ssize_t ret;
4      struct file *file;
5
6      ret = -EBADF;
7      file = fget(fd);
8      if (file) {
9          ...
10     }
11     return ret;
12 }
13
14 asmlinkage ssize_t sys_write ...
```

- **Systemfunktion** (Implementierung) innerhalb des Betriebssystems
 - aktiviert durch `call *sys_call_table(,%eax,4)` (S. 18, Zeile 17)

1 ■ weist den Kompilierer an, Parameter auf dem Stapel zu übergeben⁶

⁶Standardmäßig werden die ersten Parameter der Systemfunktionen von Linux in Registern übergeben, für x86-32: `eax`, `ecx` und `edx`.

Prozedur- vs. Systemaufruf



- Systemaufruf als adressraumübergreifender Prozeduraufruf
 - verlagert (*shift*) die weitere Prozedurausführung ins Betriebssystem

Einführung

Hybrid

Programmhierarchie

Hochsprachenkonstrukte

Assemblersprachenanweisungen

Betriebssystembefehle

Organisationsprinzipien

Funktionen

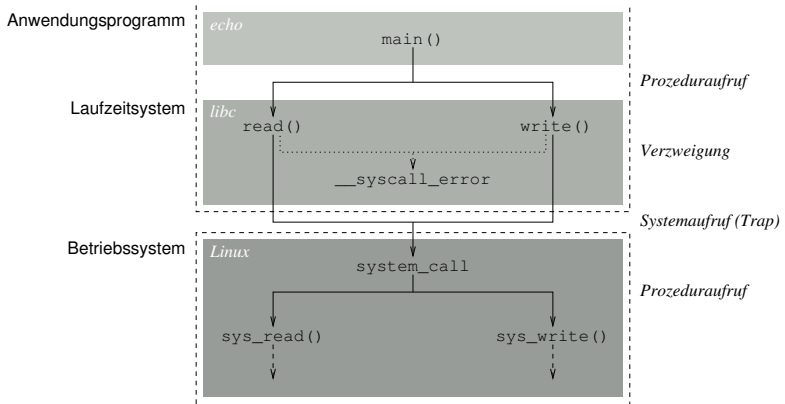
Komponenten

Zusammenfassung

Organisationsprinzipien

Funktionen

Domänenübergreifende Aufrufhierarchie



■ „obere“ Domäne (Ebene₃, □)

- Anwendungsmodus
- unprivilegiert (graduell)
- räumlich isoliert (total)
- transient (logisch)

■ „untere“ Domäne (Ebene₂, □)

- Systemmodus
- privilegiert (graduell)
- räumlich isoliert (partiell)
- resident (logisch)

Systemaufrufchnittstelle (system call interface)⁷

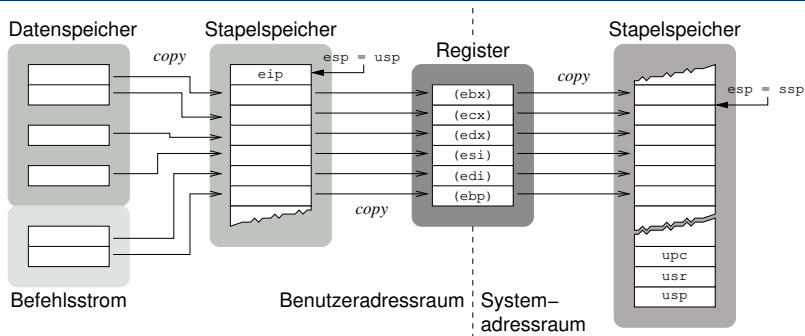
```
1 read:
2     push %ebx
3     movl 16(%esp),%edx
4     movl 12(%esp),%ecx
5     movl 8(%esp),%ebx
6     mov $3,%eax
7     int $0x80
8     pop %ebx
9     cmp $-4095,%eax
10    jae __syscall_error
11    ret
```

- „Grenzübergangsstelle“ **Aufrufstumpf**
 - einerseits erscheint ein Systemaufruf als normaler **Prozeduraufruf**
 - andererseits bewirkt der Systemaufruf einen **Moduswechsel**
- sorgt für **Ortstransparenz** (funktional)
 - die Lokalität der aufgerufenen Systemfunktion muss nicht bekannt sein

- Systemaufrufe sind **Prozedurfernaufrufe**, um **Prozessdomänen** in kontrollierter Weise zu überwinden

- 3–5** ▪ tatsächliche Parameter (Argumente) in Registern übergeben
- 6** ▪ Systemaufrufnummer (Operationskode) in Register übergeben
- 7** ▪ Domänenwechsel (Ebene₃ ↦ Ebene₂) auslösen
 - Aufruf abfangen (*trap*) und dem Betriebssystem zustellen
- 9–10** ▪ Status überprüfen und ggf. Fehlerbehandlung durchführen

⁷UNIX Programmers Manual (UPM), Lektion 2 — `man(2)`



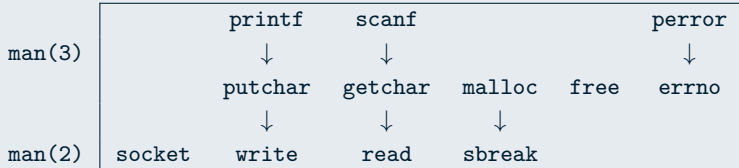
- **Werteübergabe (*call by value*)** für alle Parameter
 - Variable: Befehlsoperand ist Adresse im Datenspeicher inkl. Register
 - Direktwert: Bestandteil des Befehls im Befehlsstrom
- stark abhängig vom **Programmiermodell** der Befehlssatzebene⁸
 - die Registeranzahl bestimmt die Anzahl direkter Parameter
 - ggf. sind weitere Parameter indirekt über den Stapelzeiger zu laden

⁸...und der problemorientierten Programmiersprachenebene, des Kompilers.

Laufzeitumgebung (*runtime environment*)⁹

- **Programmbausteine** in Form eines zur Laufzeit zur Verfügung gestellten universellen Satzes von Funktionen und Variablen
 - Lesen/Schreiben von Dateien, Ein-/Ausgabegeräte steuern
 - Daten über Netzwerke transportieren oder verwalten
 - formatierte Ein-/Ausgabe, ...

Laufzeitbibliothek von C unter UNIX (Auszug)



⁹UNIX Programmers Manual (UPM), Lektion 3 — man(3)

■ **Anwendungsroutinen** (des Rechners)

- bei C/C++ die Funktion `main()` und anderes Selbstgebautes
- setzen u.a. Betriebssystem- oder Laufzeitsystemaufrufe ab

■ **Laufzeitsystemfunktionen** (des Kompilierers/Betriebssystems)

- bei C z.B. die Bibliotheksfunktionen `printf(3)` und `malloc(3)`
- setzt Betriebssystem- oder (andere) Laufzeitsystemaufrufe ab

■ **Systemaufrufstümpfe** (des Betriebssystems)

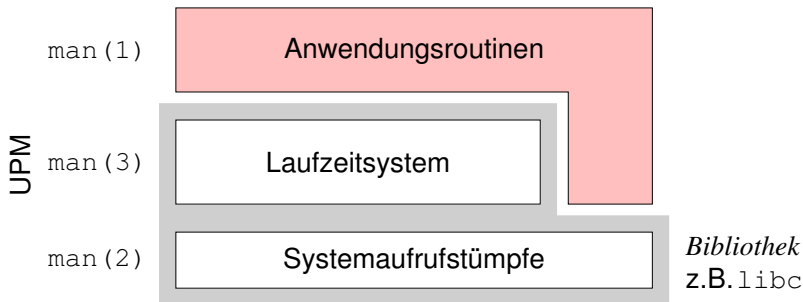
- bei UNIX z.B. die Bibliotheksfunktionen `read(2)` und `write(2)`
- setzen Aufrufe an das Betriebssystem ab
 - Systemaufruf $\mapsto$ Abfangstelle im Betriebssystem $\sim$ *Trap*

■ bilden zusammengebunden das **Maschinenprogramm** (Lademodul)

Organisationsprinzipien

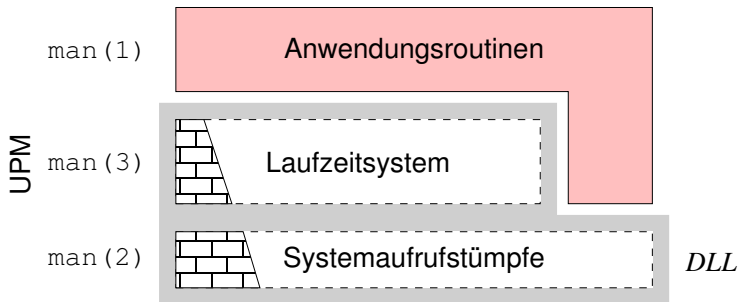
Komponenten

Grobstruktur von Maschinenprogrammen I



- statisch gebundenes Programm
 - zum Ladezeitpunkt des Programms sind alle Referenzen aufgelöst
 - Compiler und Assembler lösen lokale (interne) Referenzen auf
 - der Binder löst globale (`extern`, `.globl`) Referenzen auf
 - Schalter `-static` bei `gcc(1)` oder `ld(1)`
- Laufzeitüberprüfung von Bibliotheksreferenzen entfällt

Grobstruktur von Maschinenprogrammen II



- dynamisch gebundenes Programm
 - Bibliotheksfunktionen erst bei Bedarf (vom Betriebssystem) einbinden
 - Ebene_[2,3] erkennt einen **Bindungsfehler** (*link trap*, Multics [4])
 - den ein **bindender Lader** (*linking loader*) im Betriebssystem behandelt
 - dynamische Bibliothek (*shared library*, *dynamic link library* (DLL))
- Laufzeitüberprüfung von Bibliotheksref. $\leadsto$ **Teilinterpretation**

Einführung

Hybrid

Programmhierarchie

Hochsprachenkonstrukte

Assemblersprachenanweisungen

Betriebssystembefehle

Organisationsprinzipien

Funktionen

Komponenten

Zusammenfassung

- Bedeutung der Maschinenprogrammebene als **Hybrid** skizziert
 - **Maschinenbefehle** der Befehlssatzebene und **Betriebssystembefehle**
 - letztere als **Systemaufrufe** abgesetzt und partiell interpretiert
 - Betriebssysteme als Programme der Befehlssatzebene eingeordnet
- Ebene_[2,3] als **Programmhierarchie** virtueller Maschinen erklärt
 - Repräsentation einer **Systemfunktion** in Hochsprache, Assemblersprache und symbolischen Maschinenkode behandelt
 - in dem Zusammenhang die Implementierung von Systemaufrufen erörtert: **Systemaufrufstumpf** und **Systemaufrufzuteiler**
 - Befehlsabruf- und ausführungszyklus eines Betriebssystems und damit die Funktion als **Interpreter** (von Betriebssystembefehlen) verdeutlicht
- **Organisationsprinzipien** von Maschinenprogrammen präsentiert
 - domänenübergreifende **Aufrufhierarchie** von Funktionen verschiedener Abstraktionsebenen im Zuge der Ausführung eines Systemaufrufs
 - Ebene₃-Programme sind ein Ensemble von (a) Anwendungsroutinen und (b) Laufzeitsystem und Systemaufrufstümpfen
 - Komplex (b) ist Teil einer statischen/dynamischen **Programmbibliothek**

Zusammenfassung

Bibliographie

- [1] FOG, A. :
Optimization Manuals.
4. Instruction Tables.
: Technical University of Denmark, Dez. 2014
- [2] INTEL CORPORATION (Hrsg.):
Addendum—Intel Architecture Software Developer's Manual.
2: Instruction Set Reference.
: Intel Corporation, 1997.
(243689-001)
- [3] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :
Virtuelle Maschinen.
In: LEHRSTUHL INFORMATIK 4 (Hrsg.): *Systemprogrammierung.*
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien), Kapitel 5.1

- [4] ORGANICK, E. I.:
The Multics System: An Examination of its Structure.
MIT Press, 1972. –
ISBN 0-262-15012-3
- [5] VASUDEVAN, A. ; YERRABALLI, R. ; CHAWLA, A. :
A High Performance Kernel-Less Operating System Architecture.
In: ESTIVILL-CASTRO, V. (Hrsg.) ; Australian Computer Society
(Veranst.): *Proceedings of the Twenty-Eighth Australasian
Computer Science Conference (ACSC2005)* Bd. 38 Australian
Computer Society, CRPIT, 2005. –
ISBN 1-920682-20-1, S. 287-296
- [6] WIKIPEDIA:
<http://de.wikipedia.org/wiki/Hybrid>.
2015

Anhang

Betriebssystembefehle

■ Kontext eines Programmablaufs

- der für einen bestimmten Programmablauf relevante Prozessorstatus
- vorgegeben durch die im Programm festgelegte Berechnungsvorschrift
- je nach Art und Mächtigkeit der Maschinenbefehle unterschiedlich groß

Prozessorstatus

Der im Programmiermodell der CPU für einen (abstrakten/realen) Prozessor definierte Zustand, manifestiert in den im **Registersatz** dieser CPU gespeicherten Daten.

■ Kontextwechsel müssen **Konsistenz** des Prozessorstatus wahren

- hier: Unterprogrammaufrufe, Systemaufrufe, ..., Koroutinenaufrufe
- vorgegeben durch die **Aufrufkonventionen** des jeweiligen Prozessors
 - des Kompilers einerseits und des Betriebssystems andererseits

flüchtige Register – Inhalt gilt als unbeständig, darf verändert werden
– bei Aufrufender gespeichert (*caller saved*)¹⁰

nichtflüchtige Register – Inhalt gilt als beständig, muss unverändert bleiben
– bei Aufgerufener gespeichert (*callee saved*)

¹⁰x86: eax, ecx, edx

Programmbeispiel: Speicherzelleninhalte austauschen

■ Ebene₅

```
1 void swap(long *one, long *other) {
2     long aux = *one;
3     *one = *other;
4     *other = aux;
5 }
6 extern long foo, bar;
7
8 swap(&foo, &bar);
```

■ Ebene₄ beziehungsweise Ebene_[3,2] im symbolischen Maschinencode

```
10 swap:
11     pushl    %ebp
12     movl     %esp, %ebp
13     pushl    %esi
14     movl     12(%ebp), %eax
15     movl     8(%ebp), %ecx
16     movl     (%ecx), %edx
17     movl     (%eax), %esi
18     movl     %esi, (%ecx)
19     movl     %edx, (%eax)
20     popl     %esi
21     popl     %ebp
22     retl
```

```
23     pushl    $_bar
24     pushl    $_foo
25     calll    swap
```

23–24 Parameterübergabe

25 Unterprogrammaufruf

11–12 lokale Basis einrichten

13 Register sichern

14–15 Parameterübernahme

16 lokale Variable definieren

17–19 Tausch bewerkstelligen

20–22 Epilog und Rücksprung

Varianten von Aktivierungsblöcken

- funktional gleich auf allen Ebenen, aber nichtfunktional ist Ebene₅ ungleich gegenüber Ebene_[4,3,2] in räum- und zeitlicher Hinsicht

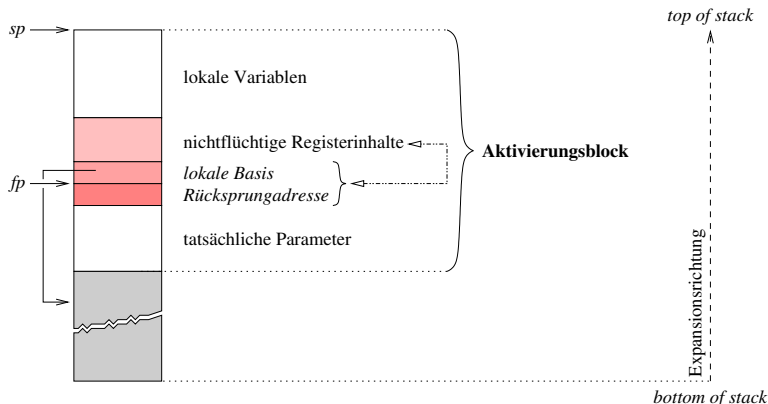
mit lokaler Basis

```
1  swap:
2      pushl    %ebp
3      movl     %esp, %ebp
4      pushl    %esi
5      movl     12(%ebp), %eax
6      movl     8(%ebp), %ecx
7      movl     (%ecx), %edx
8      movl     (%eax), %esi
9      movl     %esi, (%ecx)
10     movl     %edx, (%eax)
11     popl     %esi
12     popl     %ebp
13     retl
```

ohne lokaler Basis (-fomit-frame-pointer)

```
21  swap:
22      #
23      #
24      pushl    %esi
25      movl     12(%esp), %eax
26      movl     8(%esp), %ecx
27      movl     (%ecx), %edx
28      movl     (%eax), %esi
29      movl     %esi, (%ecx)
30      movl     %edx, (%eax)
31      popl     %esi
32      #
33      retl
```

- Art der Lokalisierung der Argumente, aber auch lokaler Variablen
 - relativ zum Basiszeiger (*base pointer*), ein **fester Bezugspunkt** oder
 - relativ zum Stapelzeiger (*stack pointer*), logisch **variabler Bezugspunkt**



■ Prozessorregister der Befehlssatzebene zur Unterprogrammverwaltung

- sp** ■ *stack pointer*, markiert die Oberseite des Stapels
- fp** ■ *frame pointer* (optional)¹¹, die lokale Basis eines Unterprogramms
- Zeiger auf die lokale Basis des umgebenden Unterprogramms

¹¹gcc -fomit-frame-pointer speichert/verwaltet keine lokale Basis (S. 40).

Relevante Merkmale der Befehlssatzebene

- die Expansionsrichtung des Stapels verläuft...
 - von hohen zu niedrigen Adressen (*top-down stack*, x86) oder
 - von niedrigen zu hohen Adressen (*bottom-up stack*)
- der Stapelzeiger adressiert...
 - das zuletzt auf dem Stapel abgelegte Datum (x86) oder
 - den nächsten freien Platz an der Oberseite des Stapels
- eine Adresse auf eine Speicherzelle im Stapel ist...
 - repräsentiert durch eine beliebige Bytenummer (x86) oder
 - ausgerichtet passend zur Operandengröße der nächsten Stapeloperation

Jenseits von Assemblersprache oder nativem Kode

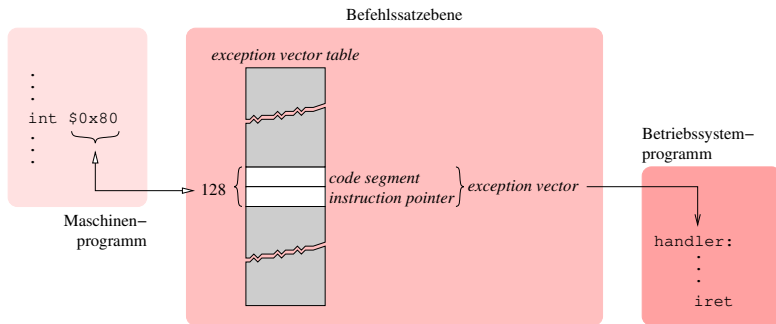
Jedes einzelne dieser Merkmale ist eine prozessorabhängige Größe, die die Software, um den Kontext von Programmläufen zu speichern, zu verwalten oder zu wechseln nicht übertragbar macht.

→ Aspekte, die insbesondere für Systemsoftware bedeutsam sind

Anhang

Systemaufrufe

Systemaufruf mittels Unterbrechungsbefehl¹²



■ die CPU durchläuft ihren gewöhnlichen **Unterbrechungszyklus**

- `int`
 - (minimalen) Prozessorstatus sichern
 - Befehlszählerregister vom Ausnahmevektor laden
 - privilegierten Betriebsmodus aktivieren
- `iret`
 - (minimalen) Prozessorstatus wieder herstellen
 - nichtprivilegierten Betriebsmodus reaktivieren, zurückspringen

¹² `int` (x86), `trap` (m68k, PDP11), `sc` (PowerPC), ..., `svc` (System/370)

Ausnahme ohne wirkliche Ausnahmesituation

- den Systemaufruf konventionell über eine **Abfangstelle** (*trap*) laufen zu lassen, ist vergleichsweise „schwergewichtig“
 - Systemaufruf (`int n/iret`) in Relation zu Prozeduraufruf (`call/ret`)
 - je nach x86-Modell, Faktor 3–30 mehr an Latenz (Prozessortakte, [1])
- im Zusammenhang mit der Funktionsweise gängiger Betriebssysteme (z.B. Linux) ist dies zudem unzweckmäßig
 - der im Rahmen der Unterbrechungsbehandlung gesicherte Prozessorstatus entspricht nicht der Wirklichkeit des unterbrochenen Prozesses
 - vielmehr geschieht diese Statussicherung, bevor die Prozessorregister zur Argumentenübergabe verwendet werden (vgl. S. 25, Zeile 2)
 - die Statussicherung durch das Betriebssystem bleibt **inkonsistent** (S. 18)
- der eigentlich bedeutsame Aspekt eines Systemaufrufs ist jedoch der **Domänenwechsel**, der „leichtgewichtig“ bewirkt werden kann
 - für x86-Prozessoren wurden hierfür dedizierte Ebene₂-Befehle eingeführt
 - `sysenter/sysexit` (Intel, [2]) und `syscall/sysret` (AMD)
 - diese ändern lediglich den **Betriebsmodus** des Ebene₂-Prozessors (CPU)

`sysenter/syscall` unprivilegiert $\mapsto$ privilegiert (d.h., Ebene_{3 $\mapsto$ 2})
`sysexit/sysret` privilegiert $\mapsto$ unprivilegiert (d.h., Ebene_{2 $\mapsto$ 3})

■ Verwendung im Maschinenprogramm (Ebene₃) für Linux:

Umschaltung hin zur Ebene₂

```
1 __kernel_vsyscall:
2     pushl %ecx
3     pushl %edx
4     pushl %ebp
5     movl %esp,%ebp
6     sysenter
```

- Aufruf ersetzt `int $0x80` im Systemaufrufstumpf
- `sysenter` bewirkt Sprung zu `sysenter_entry` im Kern

Fortsetzung auf Ebene₃

```
7 SYSENTER_RETURN:
8     popl %ebp
9     popl %edx
10    popl %ecx
11    ret
```

Sysexit erwartet den PC in `%edx` und den SP in `%ecx`, Werte die der Kern definiert:
► `%ecx` $\leftarrow$ `%ebp` und
► `%edx` $\leftarrow$ &Zeile 7.
Die Registerinhalte müssen daher auf Ebene₃ gesichert und wiederhergestellt werden.

- Ausführung von `sysexit` auf Ebene₂ bewirkt Rücksprung an Zeile 7
 - der Wert von `SYSENTER_RETURN` ist eine „Betriebssystemkonstante“
- der Mechanismus kann die Systemaufruflatenz des Ebene₂-Prozessors signifikant verringern (z.B. von 181 auf 92 Taktzyklen [5])