

# Systemprogrammierung

*Grundlage von Betriebssystemen*

Teil B – V.2 Rechnerorganisation: Maschinenprogramme

Jürgen Kleinöder

28. Mai 2019



## Gliederung

Einführung  
Hybrid

Programmhierarchie  
Hochsprachenkonstrukte  
Assembliersprachenanweisungen  
Betriebssystembefehle

Organisationsprinzipien  
Funktionen  
Komponenten

Zusammenfassung



## Agenda

Einführung

Hybrid

Programmhierarchie

Hochsprachenkonstrukte  
Assembliersprachenanweisungen  
Betriebssystembefehle

Organisationsprinzipien

Funktionen  
Komponenten

Zusammenfassung



## Lehrstoff

- Maschinenprogramm als Entität einer **hybriden Schicht** verstehen
  - Instruktionen an die Befehlssatzebene, die direkt ausgeführt werden
  - Instruktionen an das Betriebssystem, die partiell interpretiert werden
- Ebene<sub>[2,3]</sub> 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 Anwendungs Routinen zusammengebunden zum **Lademodul**

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



## Hybride Schicht in einem Rechensystem

- Maschinenprogramme enthalten zwei Sorten von Befehlen:
  - i **Maschinenbefehle** der Befehlssatzebene (ISA)
    - normalerweise direkt interpretiert durch die Zentraleinheit<sup>1</sup>
    - ausnahmsweise partiell interpretiert durch das Betriebssystem
  - ii **Systemaufrufe** an das Betriebssystem
    - normalerweise partiell interpretiert durch das Betriebssystem

### Hybrid (lat. *hybrida* Bastard, Mischling, Frevelkind)<sup>a</sup>

<sup>a</sup>gr. *hybris* Übermut, Anmaßung

„etwas Gebündeltes, Gekreuztes oder Gemischtes“ [6]

- ein System, in dem zwei Techniken miteinander kombiniert werden:
  - i Interpretation von Programmen der Befehlssatzebene
  - ii partielle Interpretation von Maschinenprogrammen
- ein Maschinenprogramm ist **Hybridsoftware**, die auf Ebene<sub>[2,3]</sub> läuft

<sup>1</sup>central processing unit, CPU

© wosch

SP (SS 2019, B – V.2 )

1.1 Einführung – Hybrid

V.2/5

## Gliederung

Einführung  
Hybrid

Programmhierarchie  
Hochsprachenkonstrukte  
Assembliersprachenanweisungen  
Betriebssystembefehle

Organisationsprinzipien  
Funktionen  
Komponenten

Zusammenfassung



© wosch

SP (SS 2019, B – V.2 )

2. Programmhierarchie

V.2/7

## Betriebssystem ≡ 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 interpretiert die eigenen Programme nur eingeschränkt partiell

### Teilinterpretation von Betriebssystemprogrammen

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

<sup>a</sup>ausgelöst durch synchrone/asynchrone Unterbrechungen

<sup>b</sup>nichtblockierende/blockierende Synchronisation

- gleichwohl sollten Betriebssysteme es zulassen, in der Ausführung eigener Programme unterbrochen werden zu können
  - nicht durch Systemaufrufe aber durch Traps oder Interrupts...



© wosch

SP (SS 2019, B – V.2 )

1.1 Einführung – Hybrid

V.2/6

## Maschinensprache(n)

- 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*)<sup>2</sup>
  - 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 Assembliersprachen, und zwar für jede Art Software:
  - Anwendungsprogramme, Laufzeitsysteme und Betriebssysteme
  - symbolisch repräsentiert auf Ebene<sub>[4,5]</sub>, numerisch auf Ebene<sub>3</sub>



<sup>2</sup>Binärkode des realen Prozessors, auch: Maschinenkode.

© wosch

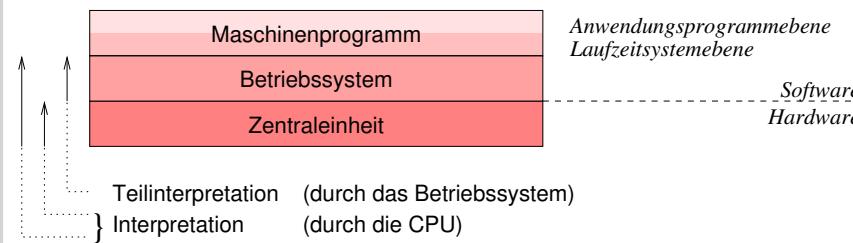
SP (SS 2019, B – V.2 )

2. Programmhierarchie

V.2/8

## „Triumvirat“

... zur Ausführung von Anwendungsprogrammen



- Maschinenprogramm = Anwendungsprogramm + Laufzeitsystem
  - beide Teilebenen liegen im selben **Adressraum**, der zudem (logisch) per **Specherschutz** 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



© wosch

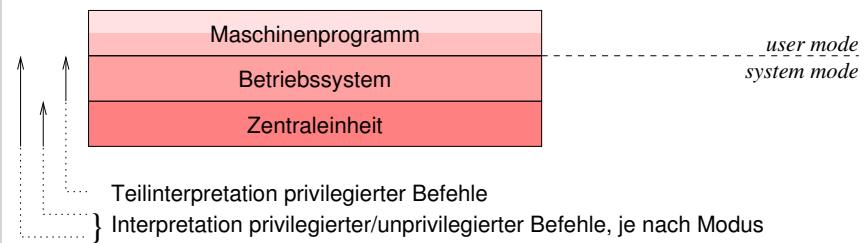
SP (SS 2019, B – V.2 )

2. Programmhierarchie

V.2/9

## Benutzer-/Systemgrenze

Betriebsmodus



- 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** ↗ Systemaufruf
- Ausführungsplattform = Systemebene (*system level, kernel space*)
  - uneingeschränkter Umgang mit den Merkmalen der Befehlssatzebene
  - alle Maschinenbefehle werden direkt ausgeführt, alle Operationen gültig



© wosch

SP (SS 2019, B – V.2 )

2. Programmhierarchie

V.2/10

## Anwendungsprogramm: Hochsprache

C

- ein auf Ebene 5 symbolisch repräsentiertes Programm der Ebene 3:

```
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`.



© wosch

SP (SS 2019, B – V.2 )

2.1 Programmhierarchie – Hochsprachenkonstrukte

V.2/11

## Anwendungsprogramm: Assembliersprache

ASM

- dasselbe Programm symbolisch repräsentiert auf Ebene 4

■ `gcc -O4 -fomit-frame-pointer -m32 -S echo.c ~ echo.s:`

```
.file "echo.c"
.text
.p2align 4,,15
.global echo
.type echo, @function
echo:
    pushl %ebx
    subl $40, %esp
    leal 28(%esp), %ebx
    jmp .L2
    .p2align 4,,7
    .p2align 3
.L3:
    movl $1, 8(%esp)
    movl %ebx, 4(%esp)
    movl $1, (%esp)
    call write
.L2:
    movl $1, 8(%esp)
    movl %ebx, 4(%esp)
    movl $0, (%esp)
    call read
    cmpb $1, %eax
    je .L3
    addl $40, %esp
    popl %ebx
    ret
```

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



© wosch

SP (SS 2019, B – V.2 )

2.2 Programmhierarchie – Assembliersprachenanweisungen

V.2/12

- **Stümpfe** der Systemfunktionen auf Ebene<sub>3</sub>, symbolisch aufbereitet:

```

1      read:           12    write:
2      push %ebx          13    push %ebx
3      movl 16(%esp),%edx 14    movl 16(%esp),%edx
4      movl 12(%esp),%ecx 15    movl 12(%esp),%ecx
5      movl 8(%esp),%ebx  16    movl 8(%esp),%ebx
6      mov $3,%eax        17    mov $4,%eax
7      int $0x80          18    int $0x80
8      pop %ebx          19    pop %ebx
9      cmp $-4095,%eax   20    cmp $-4095,%eax
10     jae __syscall_error 21    jae __syscall_error
11     ret                22    ret
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
  ■ gcc -O4 -fomit-frame-pointer -m32 -static echo.c
  ■ 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
  
```



# Betriebssystem: Interpreter

- **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 **Systemimplementierungssprache**<sup>3</sup> ist in dem Kontext der Einsatz von Assemblersprache erforderlich
  - Teilinterpretation erfordert kompletten Zugriff auf den Prozessorstatus
  - dieser ist nicht mehr Teil des Programmiermodells einer Hochsprache

<sup>3</sup>Höhere Programmiersprache mit hardwarenahen Sprachelementen.



- **Systemaufrufzuteiler** (*system call dispatcher*):

- ein auf Ebene<sub>4</sub> symbolisch repräsentiertes Programm der Ebene<sub>2</sub>
- 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
```

```
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
```

```
25     ...
```

```
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
```

## 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
```

```
25     ...
```

```
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
```

## 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
```



# Betriebssystem: Hochsprache

- ein auf Ebene<sub>5</sub> symbolisch repräsentiertes Programm der Ebene<sub>2</sub>:

- 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
```

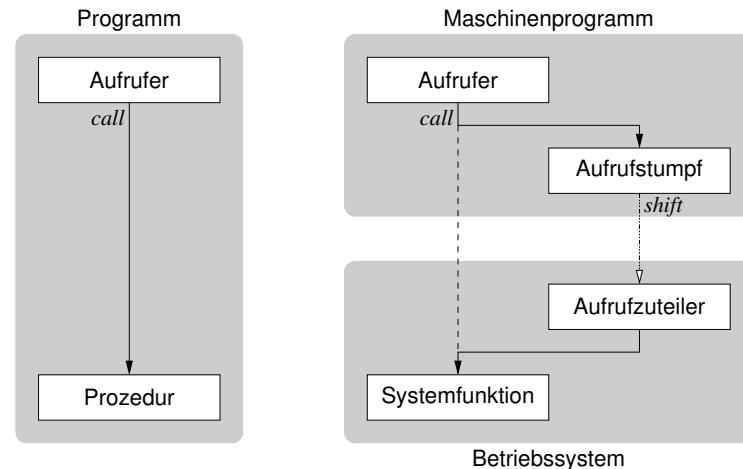
```
14 asmlinkage ssize_t sys_write ...
```

- **Systemfunktion** (Implementierung) innerhalb des Betriebssystems

- aktiviert durch **call \*sys\_call\_table(%eax,4)** (S. 14, Zeile 17)



## Prozedur- vs. Systemaufruf



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

© wosch

SP (SS 2019, B – V.2 )

2.3 Programmhierarchie – Betriebssystembefehle

V.2/17

## Gliederung

Einführung

Hybrid

Programmhierarchie

Hochsprachenkonstrukte

Assembliersprachenanweisungen

Betriebssystembefehle

Organisationsprinzipien

Funktionen

Komponenten

Zusammenfassung

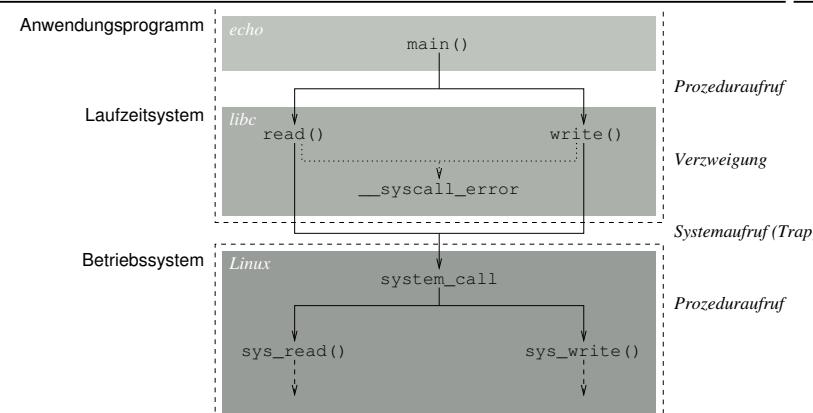
© wosch

SP (SS 2019, B – V.2 )

3. Organisationsprinzipien

V.2/18

## Domänenübergreifende Aufruhierarchie



- „obere“ Domäne (Ebene<sub>3</sub>, ⊔)
  - Anwendungsmodus
  - unprivilegiert (graduell)
  - räumlich isoliert (total)
  - transient (logisch)
- „untere“ Domäne (Ebene<sub>2</sub>, ⊓)
  - Systemmodus
  - privilegiert (graduell)
  - räumlich isoliert (partiell)
  - resident (logisch)

© wosch

SP (SS 2019, B – V.2 )

3.1 Organisationsprinzipien – Funktionen

V.2/19

## Systemaufrufschnittstelle (*system call interface*)<sup>4</sup>

```

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 (Ebene<sub>3</sub>  $\mapsto$  Ebene<sub>2</sub>) auslösen
    - Aufruf abfangen (*trap*) und dem Betriebssystem zustellen
  - 9–10 ■ Status überprüfen und ggf. Fehlerbehandlung durchführen

<sup>4</sup>UNIX Programmers Manual (UPM), Lektion 2 — *man(2)*

© wosch

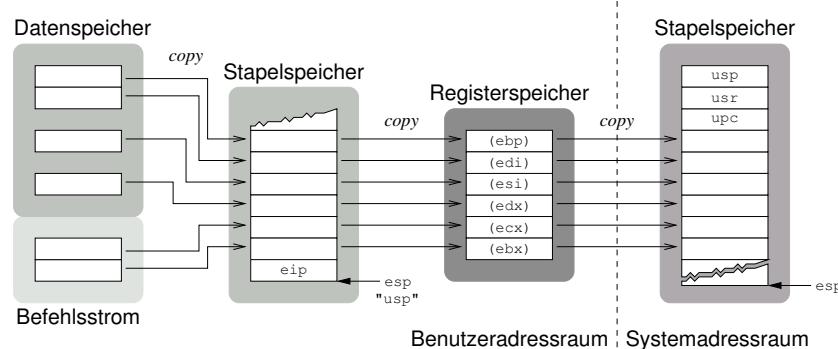
SP (SS 2019, B – V.2 )

3.1 Organisationsprinzipien – Funktionen

V.2/20

## Parametertransfer: Linux

IA-32 passend



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

© wosch

SP (SS 2019, B – V.2 )

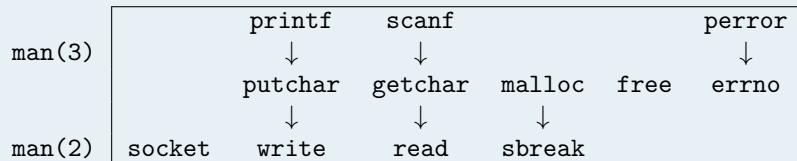
3.1 Organisationsprinzipien – Funktionen

V.2/21

## Laufzeitumgebung (*runtime environment*)<sup>5</sup>

- **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)



<sup>5</sup> UNIX Programmers Manual (UPM), Lektion 3 — man(3)

© wosch

SP (SS 2019, B – V.2 )

3.1 Organisationsprinzipien – Funktionen

V.2/22

## Ensemble problemspezifischer Prozeduren

- **Anwendungs Routinen** (des Rechners)
  - bei C/C++ die Funktion main() und andere 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)

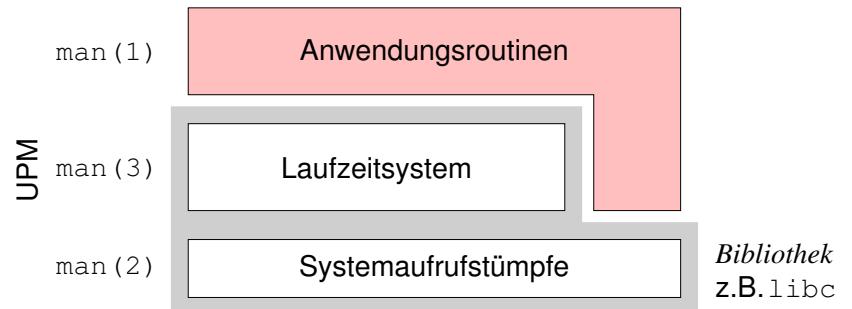
© wosch

SP (SS 2019, B – V.2 )

3.1 Organisationsprinzipien – Funktionen

V.2/23

## Grobstruktur von Maschinenprogrammen I



- statisch gebundenes Programm
  - zum Ladezeitpunkt des Programms sind alle Referenzen aufgelöst
    - Kompilierer und Assemblierer 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

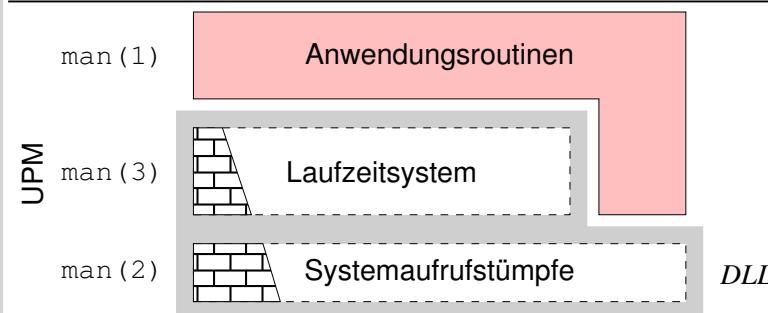
© wosch

SP (SS 2019, B – V.2 )

3.2 Organisationsprinzipien – Komponenten

V.2/24

## Grobstruktur von Maschinenprogrammen II



- dynamisch gebundenes Programm
  - Bibliotheksfunktionen erst bei Bedarf (vom Betriebssystem) einbinden
    - Ebene<sub>[2,3]</sub> 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 Bibliotheksreferenzen ↼ **Teilinterpretation**

© wosch

SP (SS 2019, B – V.2 )

3.2 Organisationsprinzipien – Komponenten

V.2/25

## Resümee

... Maschinenprogramme gibt es nicht ohne Betriebssystem

- 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<sub>[2,3]</sub> 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
  - Ebene<sub>3</sub>-Programme sind ein Ensemble von (a) Anwendroutinen und (b) Laufzeitsystem und Systemaufrufstümpfen
  - Komplex (b) ist Teil einer statischen/dynamischen **Programmbibliothek**

© wosch

SP (SS 2019, B – V.2 )

4. Zusammenfassung

V.2/27

## Gliederung

Einführung

Hybrid

Programmhierarchie

Hochsprachenkonstrukte

Assemblersprachenanweisungen

Betriebssystembefehle

Organisationsprinzipien

Funktionen

Komponenten

Zusammenfassung

© wosch

SP (SS 2019, B – V.2 )

4. Zusammenfassung

V.2/26

## Literaturverzeichnis I

[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

© wosch

SP (SS 2019, B – V.2 )

4.1 Zusammenfassung – Bibliographie

V.2/28

## Literaturverzeichnis II

- [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

© wosch

SP (SS 2019, B – V.2 )

4.1 Zusammenfassung – Bibliographie

V.2/29

## Systemaufrufbeschleunigung I

- einen Systemaufruf konventionell über eine **Abfangstelle** (*trap*) laufen zu lassen, ist vergleichsweise „schwergewichtig“
  - Systemaufruf (*int n/iret*) in Relation zu Prozederaufruf (*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. 20, Zeile 2)
  - die Statussicherung durch das Betriebssystem bleibt **inkonsistent** (S. 14)
- der eigentlich bedeutsame Aspekt eines Systemaufrufs ist jedoch der **Domänenwechsel**, der „leichtgewichtig“ bewirkt werden kann
  - für x86-Prozessoren wurden hierfür dedizierte Ebene<sub>2</sub>-Befehle eingeführt
    - *sysenter/sysexit* (Intel, [2]) und *syscall/sysret* (AMD)
  - diese ändern lediglich den **Arbeitsmodus** des Ebene<sub>2</sub>-Prozessors (CPU)

© wosch

SP (SS 2019, B – V.2 )

5.1 Anhang – Systemaufrufe

V.2/30

## Systemaufrufbeschleunigung II

*fast system call interface* [2]

*sysenter/syscall* unprivilegiert  $\leftrightarrow$  privilegiert (d.h., Ebene<sub>3</sub> $\rightarrow$ 2)  
*sysexit/sysret* privilegiert  $\leftrightarrow$  unprivilegiert (d.h., Ebene<sub>2</sub> $\rightarrow$ 3)

- Verwendung im Maschinenprogramm (Ebene<sub>3</sub>) für Linux:

**Umschaltung hin zur Ebene<sub>2</sub>**

```
1 __kernel_vsyscall:          7 SYSENTER_RETURN:  
2     pushl %ecx             8     popl %ebp  
3     pushl %edx             9     popl %edx  
4     pushl %ebp            10    popl %ecx  
5     movl %esp,%ebp         11    ret
```

**Fortsetzung auf Ebene<sub>3</sub>**

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 Ebene<sub>3</sub> gesichert und wiederhergestellt werden.

- Aufruf ersetzt *int \$0x80* im Systemaufrufstumpf
- *sysenter* bewirkt Sprung zu *sysenter\_entry* im Kern
- der Mechanismus kann die Systemaufruflatenz des Ebene<sub>2</sub>-Prozessors signifikant verringern (z.B. von 181 auf 92 Taktzyklen [5])
- Ausführung von *sysexit* auf Ebene<sub>2</sub> bewirkt Rücksprung an Zeile 7
- der Wert von SYSENTER\_RETURN ist eine „Betriebssystemkonstante“

© wosch

SP (SS 2019, B – V.2 )

5.1 Anhang – Systemaufrufe

V.2/31