

## Teil III

# Organisation von Rechensystemen

# Überblick

## Virtuelle Maschinen

Semantische Lücke

Mehrebenenmaschinen

Softwaremaschinen

Partielle Interpretation

Programmunterbrechung

Nebenläufigkeit

Virtualisierung

Zusammenfassung

# Verschiedenheit zwischen Quell- und Zielsprache

Faustregel:  $\left\{ \begin{array}{l} \text{Quellsprache} \rightarrow \text{höheres} \\ \text{Zielsprache} \rightarrow \text{niedrigeres} \end{array} \right\}$  Abstraktionsniveau

engl. *semantic gap* [13]

*The difference between the complex operations performed by high-level constructs and the simple ones provided by computer instruction sets. It was in an attempt to try to close this gap that computer architects designed increasingly complex instruction set computers.*

 Kluft zwischen gedanklich Gemeintem und sprachlich Geäußertem

# Matrix-Matrix Multiplikation

## Problemskizze

Multiplikation von zwei  $2 \times 2$  Matrizen:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

Daraus lässt sich für  $C = A \times B$  allgemein ableiten:

$$C_{i,j} = \sum_k A_{ik} \cdot B_{kj}$$

# Matrix-Matrix Multiplikation (Forts.)

Umsetzung in ein Programm (vi multiply.c)

## Implementierung in C, $N = 2$

```
typedef int Matrix [N][N];
void multiply (const Matrix a, const Matrix b, Matrix c) {
    unsigned int i, j, k;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++) {
            c[i][j] = 0;
            for (k = 0; k < N; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
}
```

# Matrix-Matrix Multiplikation (Forts.)

Umwandlung in ein semantisch äquivalentes Programm (`gcc -S multiply.c`)

`_multiply:`

```
    pushl %ebp
    movl  %esp,%ebp
    pushl %edi
    pushl %esi
    pushl %ebx
    subl  $12,%esp
    movl  $0, -16(%ebp)
    movl  16(%ebp),%edi
```

`L16:`

```
    movl  $0,-20(%ebp)
    movl  -16(%ebp),%eax
    movl  -16(%ebp),%ebx
    sall  $3,%eax
    addl  %ebx,%ebx
    movl  %eax,-24(%ebp)
    .align 16
```

`L15:`

```
    movl  $0, (%edi,%ebx,4)
    movl  -20(%ebp),%edx
    xorl  %esi,%esi
    movl  12(%ebp),%eax
    leal  (%eax,%edx,4),%ecx
    movl  8(%ebp),%eax
    movl  -24(%ebp),%edx
    addl  %eax,%edx
```

`L14:`

```
    movl  (%ecx),%eax
    incl  %esi
    addl  $8,%ecx
    imull (%edx),%eax
    addl  $4,%edx
    addl  %eax, (%edi,%ebx,4)
```

```
    cmpl  $1,%esi
    jbe  L14
    incl  -20(%ebp)
    incl  %ebx
    cmpl  $1,-20(%ebp)
    jbe  L15
    incl  -16(%ebp)
    cmpl  $1,-16(%ebp)
    jbe  L16
    addl  $12,%esp
    popl  %ebx
    popl  %esi
    popl  %edi
    popl  %ebp
    ret
```

# Matrix-Matrix Multiplikation (Forts.)

Verschiedenheit zwischen Quell- und Zielsprache

Ebene der Problemskizze  1 Summenformel

- ▶ welches Problem behandelt wird, ist (nahezu) offensichtlich
- ▶ eine semantische Lücke ist eigentlich nicht vorhanden

Ebene der Programmiersprache C  5 Komplexschritte

- ▶ welches Problem behandelt wird, ist (für Experten) noch erkennbar
- ▶ die semantische Lücke ist vergleichsweise klein

Ebene der Assemblersprache (x86)   $43+n$  Maschinenanweisungen

- ▶ welches Problem behandelt wird, ist (eigentlich) nicht erkennbar
- ▶ die semantische Lücke ist vergleichsweise sehr groß

# Matrix-Matrix Multiplikation (Forts.)

Objektmodul — das fast ausführbare Programm

## Maschinencode (x86) in Hex

```
5589E557565383EC0CC745F0000000008B7D10C745EC0000  
00008B45F08B5DF0C1E00301DB8945E8908DB42600000000  
C7049F000000008B55EC31F68B450C8D0C908B45088B55E8  
01C28B014683C1080FAF0283C20401049F83FE0176ECFF45  
EC43837DEC0176C8FF45F0837DF00176A283C40C5B5E5F5D  
C3
```

Ebene der Maschinsprache (x86)  121 Bytes

- ▶ welches Problem behandelt wird, ist überhaupt nicht mehr erkennbar
- ▶ die semantische Lücke ist (nahezu) unendlich groß

# Matrix-Matrix Multiplikation (Forts.)

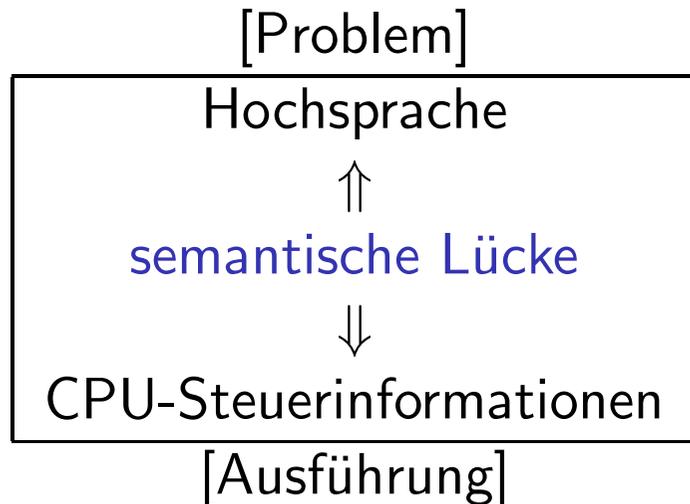
$$C_{i,j} = \sum_k A_{ik} \cdot B_{kj} \stackrel{?}{\iff} 5589E5 \dots 5F5DC3$$

Die Diskrepanz zwischen der vom Menschen skizzierten Problemlösung und dem dazu korrespondierenden, von der Maschine „x86“ ausführbaren Programm, ist beträchtlich.

Abstraktion half, sich auf das Wesentliche konzentrieren zu können

- ▶ eine **virtuelle Maschine** zur Matrixmultiplikation entstand
- ▶ die schrittweise abgebildet wurde auf eine **reale Maschine**

# Semantische Lücke schrittweise schließen



Die Ausdehnung der Lücke variiert:

- ▶ bei gleich bleibendem Problem mit der Plattform (dem System)
- ▶ bei gleich bleibender Plattform mit dem Problem (der Anwendung)

Lückenschluss ist ganzheitlich zu sehen

Problemlösungen über [virtuelle Maschinen](#) auf die reale Maschine abbilden

# Hierarchie virtueller Maschinen

## Interpretation und Übersetzung

Ebene		
$n$	virtuelle Maschine $M_n$ mit Maschinensprache $S_n$	Programme in $S_n$ werden von einem auf einer tieferen Maschine laufenden Interpreter gedeutet oder in Programme tieferer Maschinen übersetzt
$\vdots$	$\vdots$	$\vdots$
2	virtuelle Maschine $M_2$ mit Maschinensprache $S_2$	Programme in $S_2$ werden von einem auf $M_1$ bzw. $M_0$ laufenden Interpreter gedeutet oder nach $S_1$ bzw. $S_0$ übersetzt
1	virtuelle Maschine $M_1$ mit Maschinensprache $S_1$	Programme in $S_1$ werden von einem auf $M_0$ laufenden Interpreter gedeutet oder nach $S_0$ übersetzt
0	reale Maschine $M_0$ mit Maschinensprache $S_0$	Programme in $S_0$ werden direkt von der Hardware ausgeführt

# Programme leisten die Abbildung

Kompilierer (engl. *compiler*) und Interpretierer (engl. *interpreter*)

## **Kom|pi|la|tor** *lat.* (Zusammenträger)

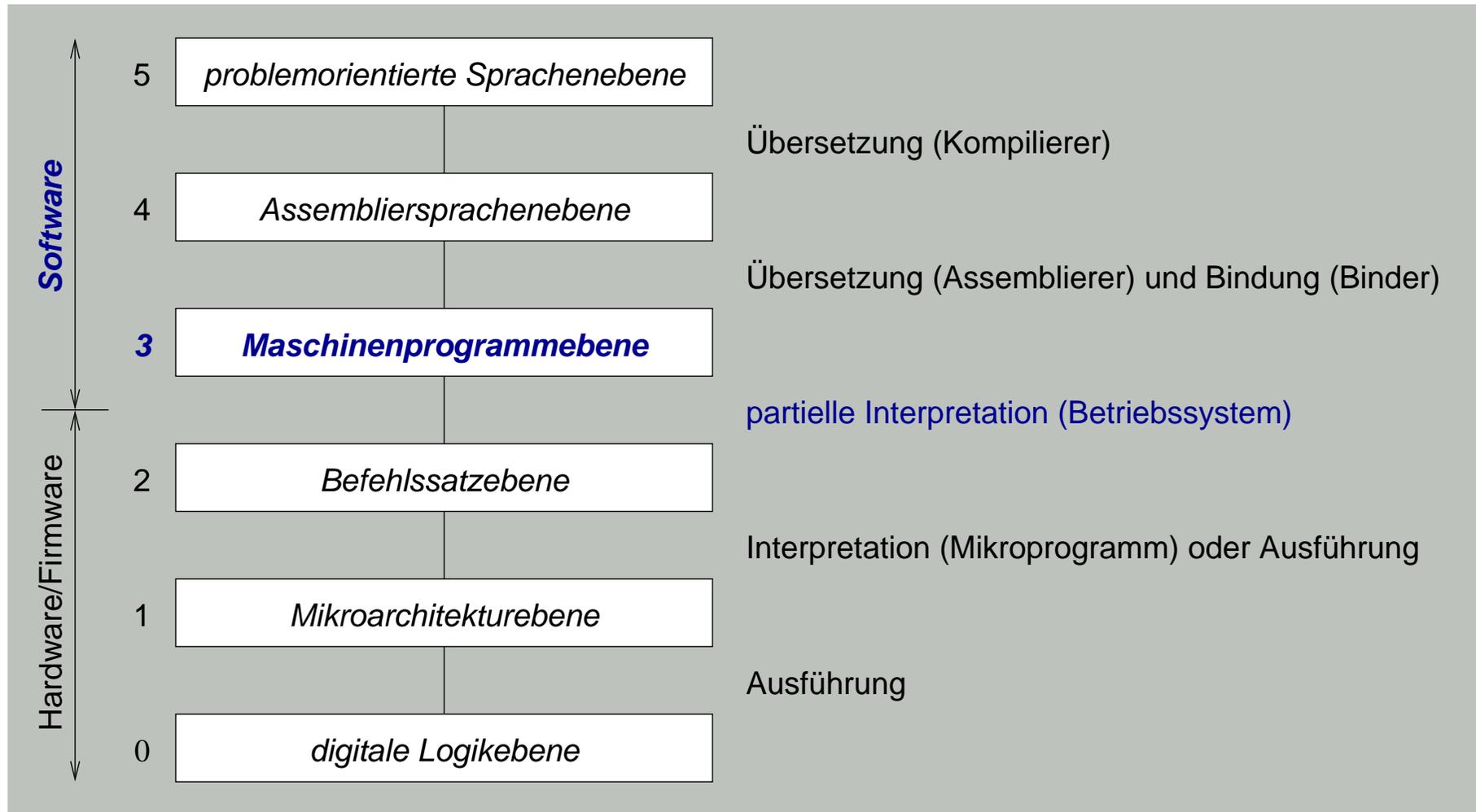
Ein typischerweise in Software realisierter *Prozessor*, der Programme einer bestimmten *Quellsprache* (z.B. C++) in semantisch äquivalente Programme einer bestimmten *Zielsprache* (z.B. C oder Assembler) transformiert.

## **In|ter|pret** *lat.* (Ausleger, Erklärer, Deuter)

Ein in Hard-, Firm- oder Software realisierter *Prozessor*, der Programme einer bestimmten Quellsprache (z.B. Basic, Perl, C, sh(1)) „direkt“ ausführt. Bei *Vorübersetzung* durch einen Kompilierer werden die Programme zunächst in eine für die Interpretation günstigere Repräsentation (z.B. Pascal P-Code, Java Bytecode, x86-Befehle) transformiert.

# Hardware/Software Hierarchie

Betriebssystem als Interpret



# Elementaroperationen der einzelnen Ebenen

## Softwaresmaschinen

### Problemorientierte Programmiersprachenebene

„Höhere Programmiersprachen“ erlauben die abstrakte und plattformunabhängige Formulierung von Problemlösungen. Programme setzen sich zusammen aus Konstrukten zur Selektion und Iteration, zur Formulierung von Sequenzen, Blockstrukturen, Prozeduren, zur Beschreibung von elementaren und abstrakte Datentypen und (getypten) Operatoren.

### Assemblersprachenebene

Pseudobefehle (Assembler/Binder), mnemonisch ausgelegte Maschinenbefehle (ISA) und symbolisch bezeichnete Operanden (Speicheradressen, Register) und Adressierungsarten bilden die Programme (*symbolischer Maschinenkode*).

# Elementaroperationen der einzelnen Ebenen (Forts.)

## Softwaremaschinen

### Maschinenprogrammzebene

Legt die Betriebsarten fest, verwaltet die Betriebsmittel des Rechners und steuert bzw. überwacht die Abwicklung von Programmen. *Systemaufrufe* und *Maschinenbefehle* (ISA) bilden die Elementaroperationen der Programme (*binärer Maschinenkode*).

Brennpunkt von  $SOS_I$ : **Betriebssysteme** implementieren diese Ebene

- ▶ auf Basis der problemorientierten und Assemblersprachenebenen

# Elementaroperationen der einzelnen Ebenen (Forts.)

Firm-/Hardwaremaschinen

## Befehlssatzebene (engl. *instruction set architecture, ISA*)

Implementiert das *Programmiermodell* der CPU (z.B. CISC, RISC, VLIW). Programme bestehen aus *Mikroanweisungen* oder Konstrukten einer *Hardwarebeschreibungssprache* (z.B. VHDL, SystemC).

## Mikroarchitekturebene

Beschreibt den Aufbau der Operations-/Steuerwerke, der Zwischenspeicher und die Befehlsverarbeitung. Programme bestehen aus Konstrukten einer *Hardwarebeschreibungssprache* (z.B. VHDL, SystemC).

# Elementaroperationen der einzelnen Ebenen (Forts.)

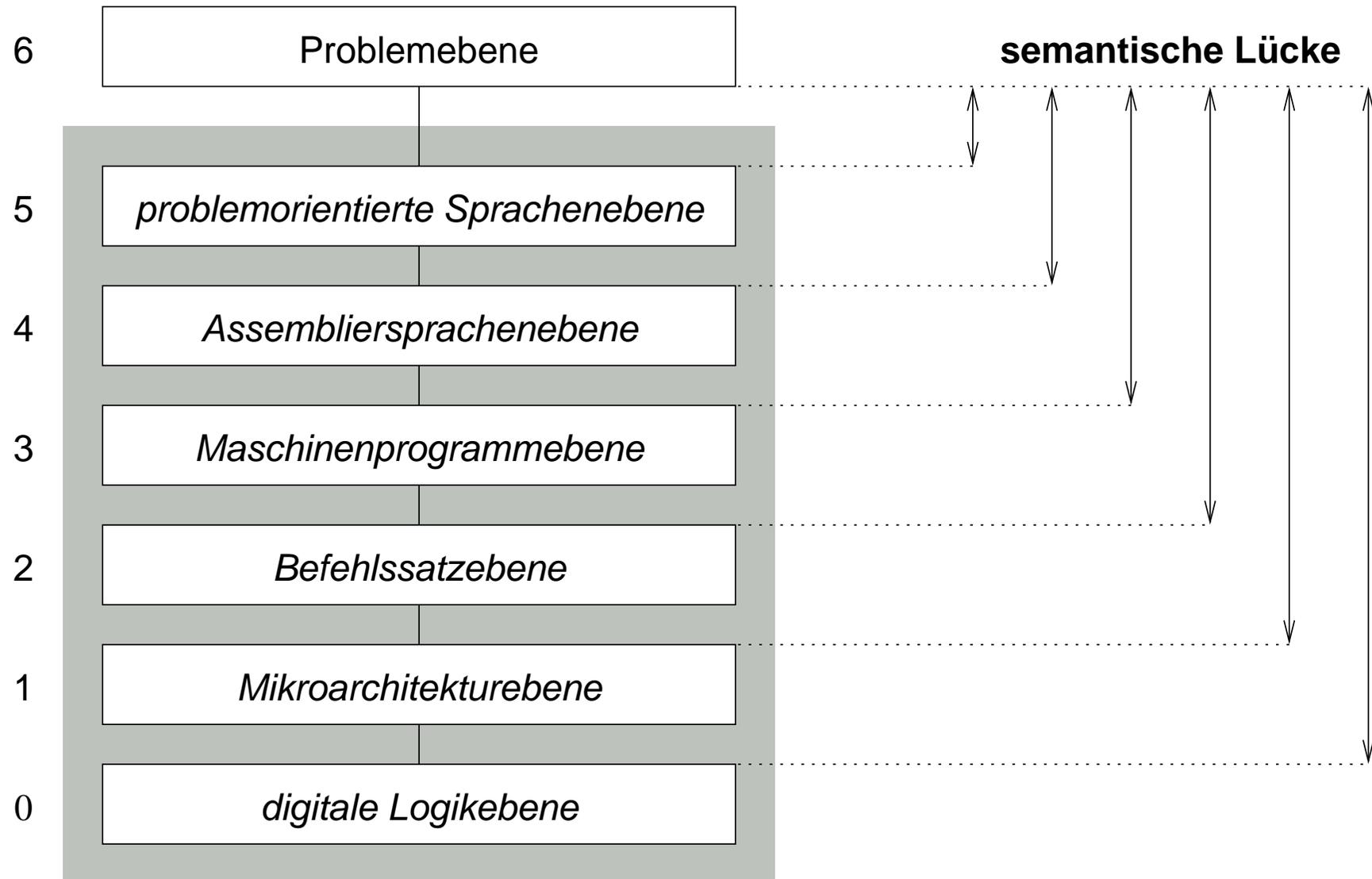
## Hardwaremaschinen

### Digitale Logikebene

Bildet auf Basis von Transistoren, Gattern, Schaltnetzen und Schaltwerken die wirkliche Hardware des Rechners. Programme bestehen aus Elementen der *Boolschen Algebra*.

- ▶ maximale Flexibilität
- ▶ minimale Benutzerfreundlichkeit
- ▶ maximale Distanz von sehr vielen Problemdomänen

# Abstraktionsniveau vs. Semantische Lücke



# Abbildung durch Übersetzung

## Ebene<sub>5</sub> $\mapsto$ Ebene<sub>4</sub>: Kompilierung

Ebene<sub>5</sub>-Befehle „1:N“ in Ebene<sub>4</sub>-Befehle übersetzen

- ▶ ein Hochsprachenbefehl ist eine Folge von Assemblersprachenbefehlen  
    ☞ *semantisch äquivalente* Befehlsfolge
- ▶ im Zuge der Transformation ggf. Optimierungsstufen durchlaufen

## Ebene<sub>4</sub> $\mapsto$ Ebene<sub>3</sub>: Assemblierung und Binden

Ebene<sub>4</sub>-Befehle („Mnemoniks“) „1:1“ in Ebene<sub>3</sub>-Befehle übersetzen

- ▶ ein **Quellmodul** in ein **Objektmodul** umwandeln
- ▶ mit **Bibliotheken** zum Maschinenprogramm zusammenbinden

# Abbildung durch Interpretation

Ebene<sub>3</sub>  $\mapsto$  Ebene<sub>2</sub>: Teilinterpretation (auch *partielle Interpretation*)

Ebene<sub>3</sub>-Befehle als Folgen von Ebene<sub>2</sub>-Befehlen ausführen

- ▶ Systemaufrufe aus dem Ebene<sub>3</sub>-Befehlsstrom „herausfiltern“
- ▶ ein Ebene<sub>3</sub>-Befehl aktiviert ein Ebene<sub>2</sub>-Programm

Ebene<sub>2</sub>  $\mapsto$  Ebene<sub>1</sub>: Interpretation

Ebene<sub>2</sub>-Befehle als Folgen von Ebene<sub>1</sub>-Befehlen ausführen

- ▶ Abruf- und Ausführungszyklus (engl. *fetch-execute-cycle*) der CPU
- ▶ ein Ebene<sub>2</sub>-Befehl löst Ebene<sub>1</sub>-Steueranweisungen aus

# Prozessoren implementieren die Abbildungen

## Ebene<sub>5</sub> **Kompilierer**

- ▶ Interpretation von Konstrukten/Anweisungen einer „Hochsprache“

## Ebene<sub>4</sub> **Assemblerer** und **Binder**

- ▶ Interpretation von Anweisungen einer Assemblersprache

## Ebene<sub>3</sub> **Betriebssystem**

- ▶ Interpretation von Systemaufrufen
- ▶ Ausführung von Ebene<sub>3</sub>-Programmen (durch Teilinterpretation)

## Ebene<sub>2</sub> **Zentraleinheit** (CPU)

- ▶ Interpretation von Instruktionen (an die ALU, FPU, MMU, ...)
- ▶ Ausführung von Ebene<sub>2</sub>-Programmen

# Programm der problemorientierten Sprachenebene

Echo

```
myecho.c
```

```
main () {  
    char c;  
    while (write(1, &c, read(0, &c, 1)) != -1) {}  
}
```

Die Funktion `read(2)` überträgt ein Zeichen von Standardeingabe (0) an die Speicheradresse `&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`.

# Programm der Assemblersprachenebene

myecho.s (generiert mit „gcc -O6 -S myecho.c“)

```
main () {...
```

```
main:
```

```
    pushl %ebp
    movl  %esp,%ebp
    pushl %esi
    pushl %ebx
    subl  $16,%esp
    leal  -9(%ebp),%ebx
    andl  $-16,%esp
    movl  %ebx,%esi
    .align 16
```

```
while (...) {}
```

```
.L2:
```

```
    movl  %esi,4(%esp)
    movl  $1,%edx
    movl  %ebx,%esi
    movl  %edx,8(%esp)
    movl  $0,(%esp)
    call  read
    movl  %eax,8(%esp)
    movl  %ebx,4(%esp)
    movl  $1,(%esp)
    call  write
    incl  %eax
    jne  .L2
```

```
... }
```

```
    leal  -8(%ebp),%esp
    popl  %ebx
    popl  %esi
    popl  %ebp
    ret
```

# Programm der Assemblersprachenebene (Forts.)

Auszüge aus der C-Bibliothek (libc.a) des Kompilers (gcc(1))

```
read:
    push %ebx
    movl 16(%esp),%edx
    movl 12(%esp),%ecx
    movl 8(%esp),%ebx
    mov $3,%eax
    int $0x80
    pop %ebx
    cmp $-4095,%eax
    jae __syscall_error
    ret
```

```
__syscall_error:
    neg %eax
    mov %eax,errno
    mov $-1,%eax
    ret

    .comm errno,16
```

```
write:
    push %ebx
    movl 16(%esp),%edx
    movl 12(%esp),%ecx
    movl 8(%esp),%ebx
    mov $4,%eax}
    int $0x80
    pop %ebx
    cmp $-4095,%eax
    jae __syscall_error
    ret
```

- ☞ Der **Binder** (ld(1)) wird diese Funktionen später hinzufügen
- ☞ **Teilinterpretation**: **int \$0x80** schaltet um von Ebene<sub>2</sub> zu Ebene<sub>3</sub>

# Programm der Assemblersprachenebene (Forts.)

Auszüge aus Linux (kernel-source-2.4.20/arch/i386/kernel/entry.S)

## Sichern

```
system_call:
    pushl %eax
    cld
    pushl %es
    pushl %ds
    pushl %eax
    pushl %ebp
    pushl %edi
    pushl %esi
    pushl %edx
    pushl %ecx
    pushl %ebx
    ...
```

## Behandeln

```
    ...
    cmpl $(NR_syscalls),%eax
    jae  badsys
    call *sys_call_table(,%eax,4)
    movl %eax,24(%esp)
ret_from_sys_call:
    ...
badsys:
    movl $-ENOSYS,24(%esp)
    jmp  ret_from_sys_call
```

## Wiederherstellen

```
    ...
    popl  %ebx
    popl  %ecx
    popl  %edx
    popl  %esi
    popl  %edi
    popl  %ebp
    popl  %eax
    popl  %ds
    popl  %es
    addl  $4,%esp
    iret
```

# Programm der problemorientierten Sprachenebene (Forts.)

Auszüge aus Linux (kernel-source-2.4.20/fs/read\_write.c)

## Systemaufrufe implementierende Programme

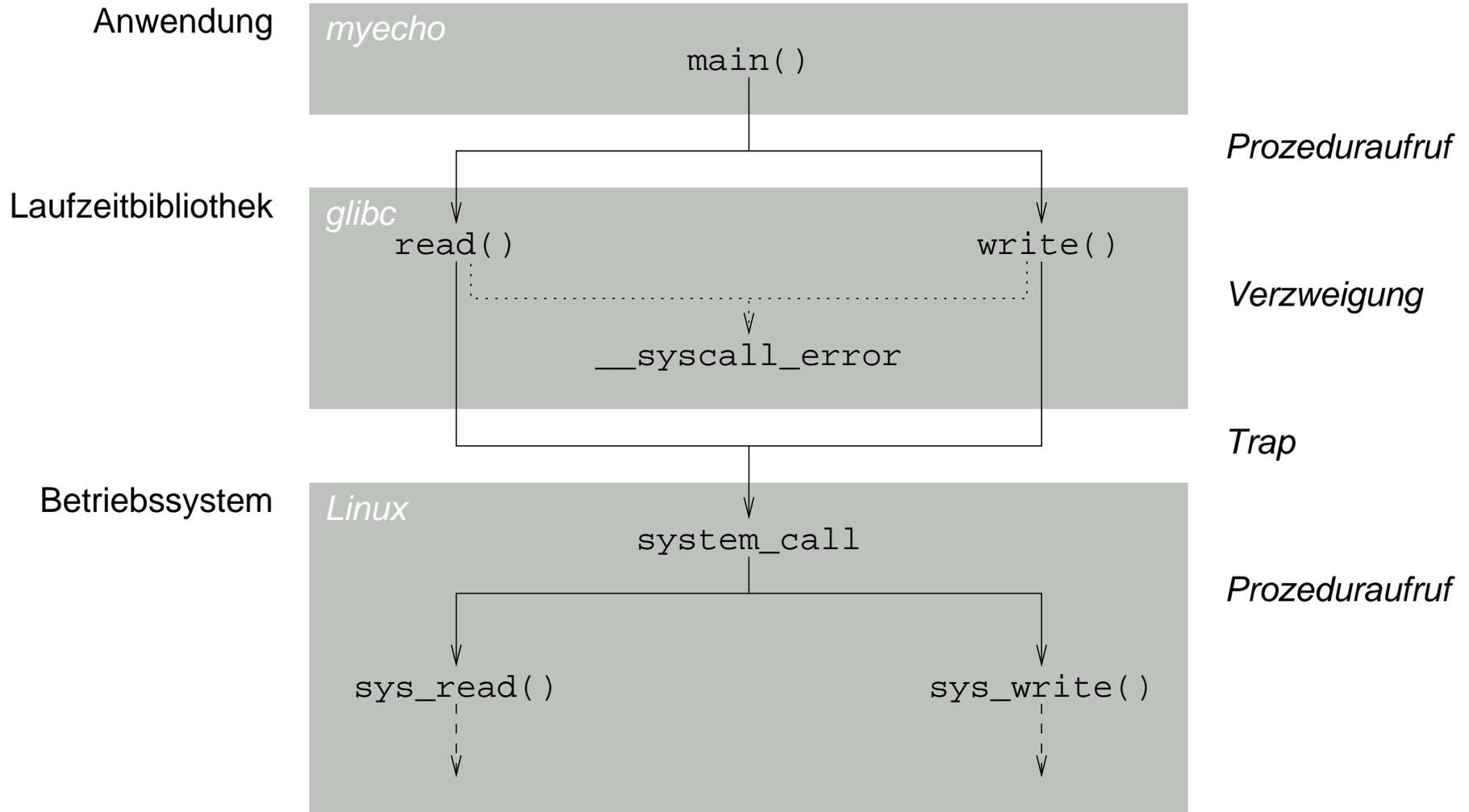
```
asmlinkage ssize_t sys_read(unsigned int fd, char * buf, size_t count) {
    ssize_t ret;
    struct file * file;

    ret = -EBADF;
    file = fget(fd);
    if (file) {
        ...
    }
    return ret;
}

asmlinkage ssize_t sys_write ...
```

# Softwaresystem „myecho“

## Aufrufhierarchie



# Systemaufrufchnittstelle (engl. *system call interface*)

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

```
read:
  push %ebx
  movl 16(%esp),%edx
  movl 12(%esp),%ecx
  movl 8(%esp),%ebx
  mov $3,%eax
  int $0x80
  pop %ebx
  cmp $-4095,%eax
  jae __syscall_error
  ret
```

**Aufrufstümpfe** verbergen die technische Auslegung der Interaktion zwischen Anwendungsprogramm und Betriebssystem

- ▶ „nach außen“ erscheint ein Systemaufruf als normaler **Prozeduraufruf**
- ▶ „nach innen“ setzt ein Systemaufruf eine (synchrone) **Programmunterbrechung** ab

Systemaufrufe sind spezielle „Prozedurfernaufrufe“, die ggf. bestehende Schutzdomänen in kontrollierter Weise überwinden müssen

- ▶ getrennte Adressräume für Anwendungsprogramm und Betriebssystem
- ▶ Ein-/Ausgabeparameter in Registern übergeben, „Trap“ auslösen

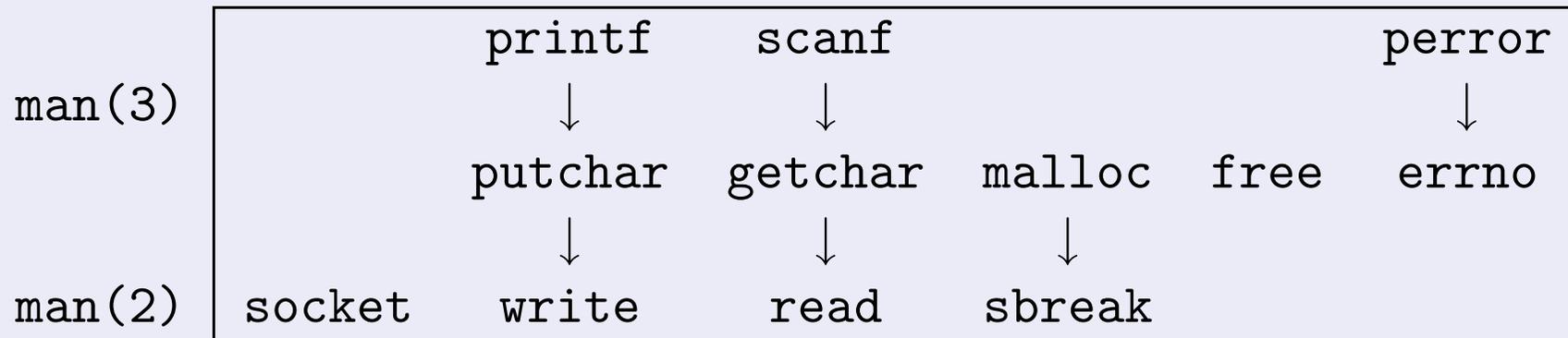
# Laufzeitumgebung (engl. *runtime environment*)

*UNIX Programmers Manual (UPM)*, Lektion 3 — `man(3)`

**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)



# Organisation von Maschinenprogrammen

„Die Drei von der Tankstelle“

## Anwendungsroutinen (des Rechners)

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

## Laufzeitsystem (des Kompilers/Betriebssystems)

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

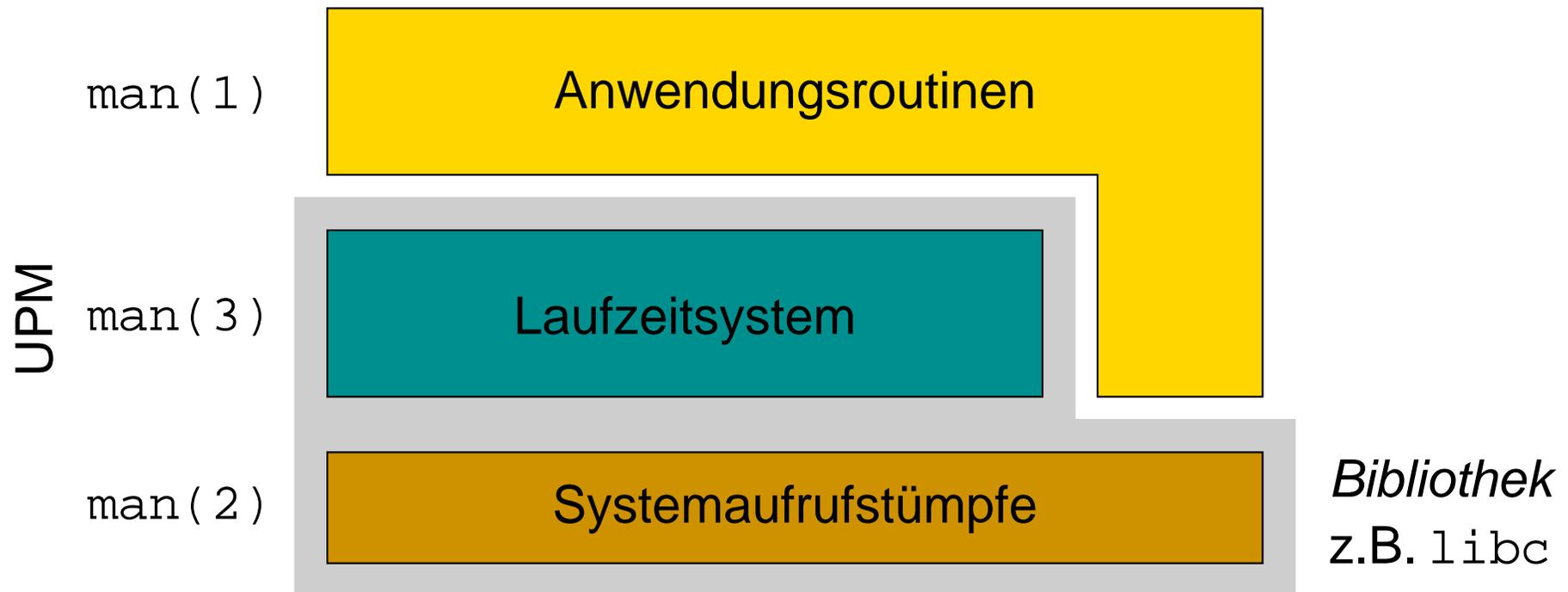
## Systemaufrufstümpfe (des Betriebssystems)

- ▶ bei UNIX z.B. die Bibliotheksfunktionen `write(2)` und `sbreak(2)`
- ▶ setzen synchrone Programmunterbrechungen (d.h. Traps) ab

 bilden zusammengebunden ein **Anwendungsprogramm**

# Organisation von Maschinenprogrammen (Forts.)

Software(grob)struktur innerhalb eines Benutzeradressraums

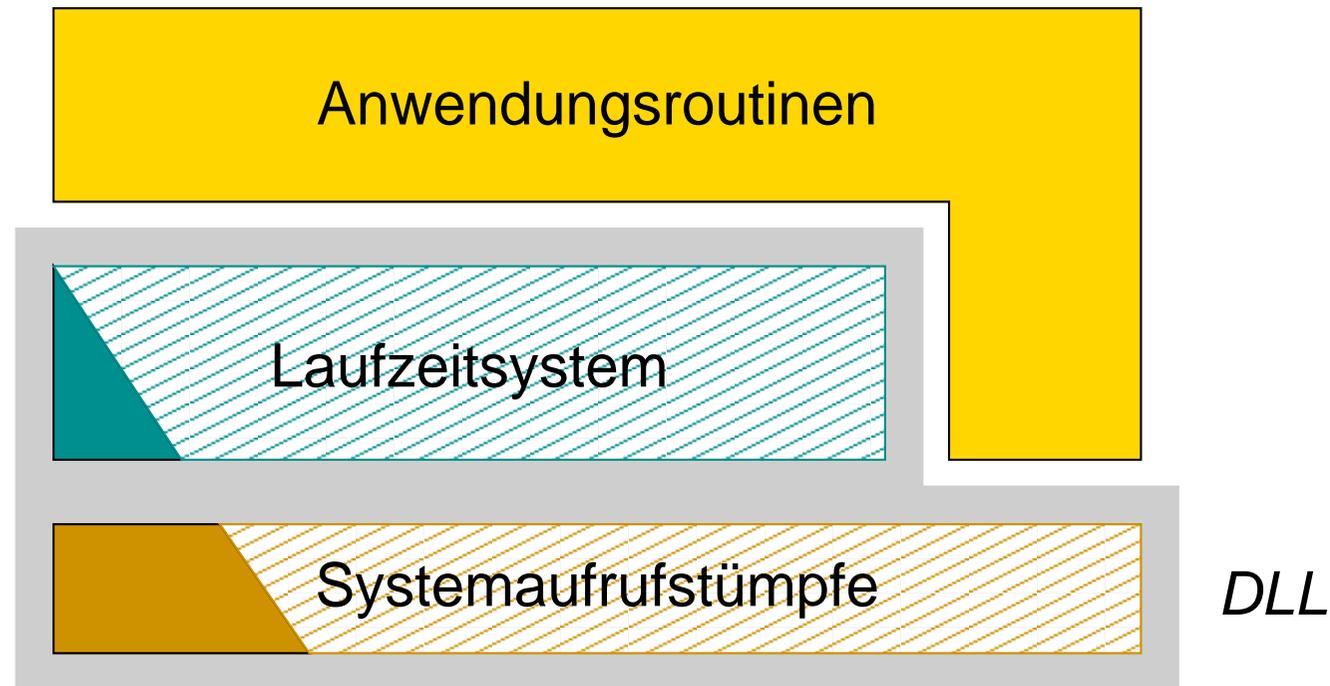


Modell für eine **statische Bibliothek** (`gcc` bzw. `ld -static ...`)

- ▶ auch Dienstprogramme (z.B. `ls(1)`) sind so repräsentiert
- ▶ der Aufbau spiegelt jedoch nur die **logische Struktur** wieder
- ▶ dynamisches Binden von Bibliotheken liefert eine andere Sicht...

# Organisation von Maschinenprogrammen (Forts.)

Dynamische Bibliothek (engl. *shared library*, UNIX; *dynamic link library* (DLL), Windows)



Bibliotheksfunktionen erst bei Bedarf (vom Betriebssystem) einbinden

- ▶ z.B. beim erstmaligen Aufruf („*trap on use*“, Multics [14])
- ▶ enorme Speicherplatzersparnis — im Hintergrundspeicher (Platte) !!!
- ▶ ein **bindender Lader** ist Bestandteil des Betriebssystems

# Zusammenspiel von Ebene<sub>2</sub> und Ebene<sub>3</sub>

## Elementaroperationen der Maschinenprogrammzebene

Maschinenprogramme umfassen zwei Sorten von Befehlen:

1. Aufrufe an das Betriebssystem (Ebene<sub>3</sub>)
  - ▶ explizit als **Systemaufruf** (engl. *system call*) kodiert
  - ▶ implizit als **Programmunterbrechung** (engl. *trap, interrupt*) ausgelöst
2. Anweisungen an die CPU (Ebene<sub>2</sub>)

Ausführende Instanz ist immer die CPU, die nur Ebene<sub>2</sub>-Befehle kennt

- ▶ Ebene<sub>3</sub>-Befehle  $\left\{ \begin{array}{l} \text{werden „wahrgenommen“, nicht ausgeführt} \\ \text{signalisieren eine **Ausnahme** (engl. *exception*)} \end{array} \right.$

Das Betriebssystem fängt Ebene<sub>3</sub>-Befehle ab, behandelt Ausnahmen

# Zusammenspiel von Ebene 2 und Ebene 3 (Forts.)

Programmunterbrechungen bewirken partielle Interpretation

## Maschinenprogramm

Ebene 3

5589E55653...538B5424108B4C240C8B5C2408B803000000CD805B3D01F0FFFF731EC3...8D65F85B5E5DC3

## Anwendungsprogramm

Ebene 3

5589E55653...538B5424108B4C240C8B5C2408B803000000CD805B3D01F0FFFF731EC3...8D65F85B5E5DC3

*trap/interrupt*

*system call*

50FC061E50...581F0783C404CF

50FC061E50...581F0783C404CF

Ebene 2

*return from exception*

*return from exception*

## Betriebssystemprogramme

# Programme der Maschinenprogrammzebene

## Hybride Ebene

Ebene<sub>3</sub>-Befehle...

- ▶ sind „normale“ Befehle der Ebene<sub>2</sub>, die die CPU ausführt
- ▶ sind „ausnahmebedingte“ Befehle, die das Betriebssystem ausführt

...implementieren z.B. Adressräume, Dateien, Prozesse

- ▶ Interpret dieser zusätzlichen Befehle ist das Betriebssystem

Betriebssysteme werden aktiviert...

- ▶ im Falle eines Systemaufrufs (CD80), programmiert
- ▶ im Falle von Ausnahmesituationen, nicht programmiert

...und deaktivieren sich immer selbst, programmiert (CF)

# Unterbrechungsarten und Ausnahmesituationen

Zwei Kategorien von Ausnahmesituationen werden unterschieden:

1. die „Falle“ (engl. *trap*)
2. die „Unterbrechung“ (engl. *interrupt*)

**Unterschiede** ergeben sich hinsichtlich. . .

- ▶ Quelle
- ▶ Synchronität
- ▶ Vorhersagbarkeit
- ▶ Reproduzierbarkeit

**Behandlung** ist zwingend und grundsätzlich prozessorabhängig