

# Überblick

## Organisation von Rechensystemen

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

## Semantische Lücke (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:

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

Formel für  $C = A \times B$ :

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

# 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 Assembliersprache (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

## Maschinenkode (x86) in Hex

```
5589E557565383EC0CC745F0000000008B7D10C745EC0000  
00008B45F08B5DF0C1E00301DB8945E8908DB4260000000  
C7049F000000008B55EC31F68B450C8D0C908B45088B55E8  
01C28B014683C1080FAF0283C20401049F83FE0176ECFF45  
EC43837DEC0176C8FF45F0837DF00176A283C40C5B5E5F5D  
C3
```

Ebene der Maschinensprache (x86)  121 Bytes

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

# Matrix-Matrix Multiplikation (Forts.)

Resümee

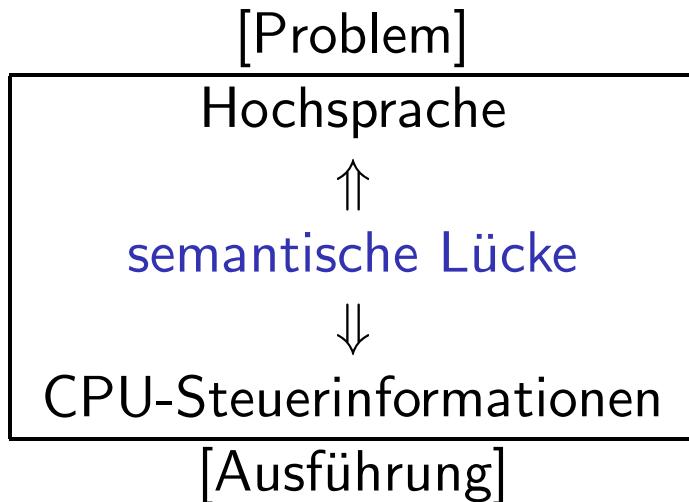
$$C_{i,j} = \sum_k A_{ik} \cdot B_{kj} \xrightarrow{?} 5589E5 \dots 5F5DC3$$

Die Diskrepanz zwischen der vom Menschen skizzierten Problemlösung und dem dazu korrespondierenden Maschinenprogramm 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
:	:	:
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

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

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

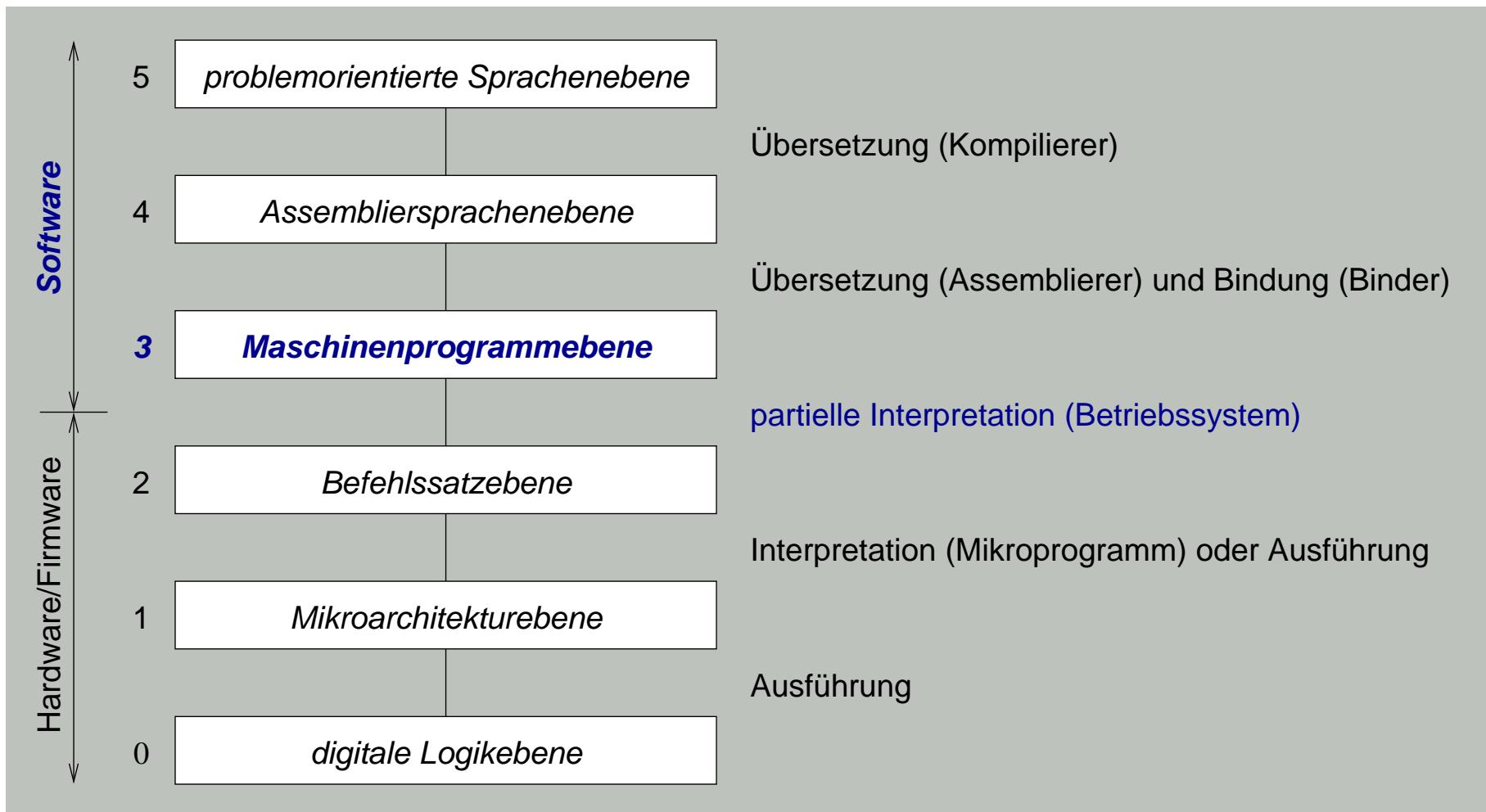
- ▶ **Softwareprozessor**, der Programme einer *Quellsprache* in semantisch äquivalente Programme einer *Zielsprache* transformiert
  - ▶ {Ada, C, C++, Eiffel, Modula, Fortran, Pascal, ...}  $\mapsto$  Assembler
  - ▶ aber ebenso: C++  $\mapsto$  C  $\mapsto$  Assembler

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

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

# Hardware/Software Hierarchie

Betriebssystem als Interpret



# Elementaroperationen der einzelnen Ebenen

Softwaremaschinen — Übersetzer

## 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 abstrakten Datentypen und (getypten) Operatoren

## Assembliersprachenebene (*symbolischer Maschinenkode*)

- ▶ **maschinenabhängige Programme**, erzeugt/zusammengestellt von Assemblierer (engl. *assembler*) und Binder (engl. *linker*)
- ▶ Programme bestehen aus Pseudobefehle, mnemonisch ausgelegte Maschinenbefehle (ISA), symbolisch bezeichnete Operanden (Speicheradressen, Register) und Adressierungsarten

# Elementaroperationen der einzelnen Ebenen (Forts.)

Softwaremaschinen — Interpretierer

## Maschinenprogrammebene (*binärer Maschinenkode*)

- ▶ legt die Betriebsarten fest, verwaltet die Betriebsmittel des Rechners und steuert bzw. überwacht die Abwicklung von Programmen
- ▶ Programme bestehen aus **Systemaufrufe** (Betriebssystem) und **Maschinenbefehle** (ISA)

Brennpunkt von SOS 1: **Betriebssysteme** implementieren diese Ebene

- ▶ auf Basis der problemorientierten und Assembliersprachenebenen

# Elementaroperationen der einzelnen Ebenen (Forts.)

Firm-/Hardwaremaschinen — Interpretierer

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

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

## Mikroarchitekturebene

- ▶ beschreibt den Aufbau der Operations- und Steuerwerke, der Zwischenspeicher und die Befehlsverarbeitung
- ▶ Programme setzen sich zusammen aus den Konstrukten einer Hardwarebeschreibungssprache (z.B. VHDL, SystemC)

# Elementaroperationen der einzelnen Ebenen (Forts.)

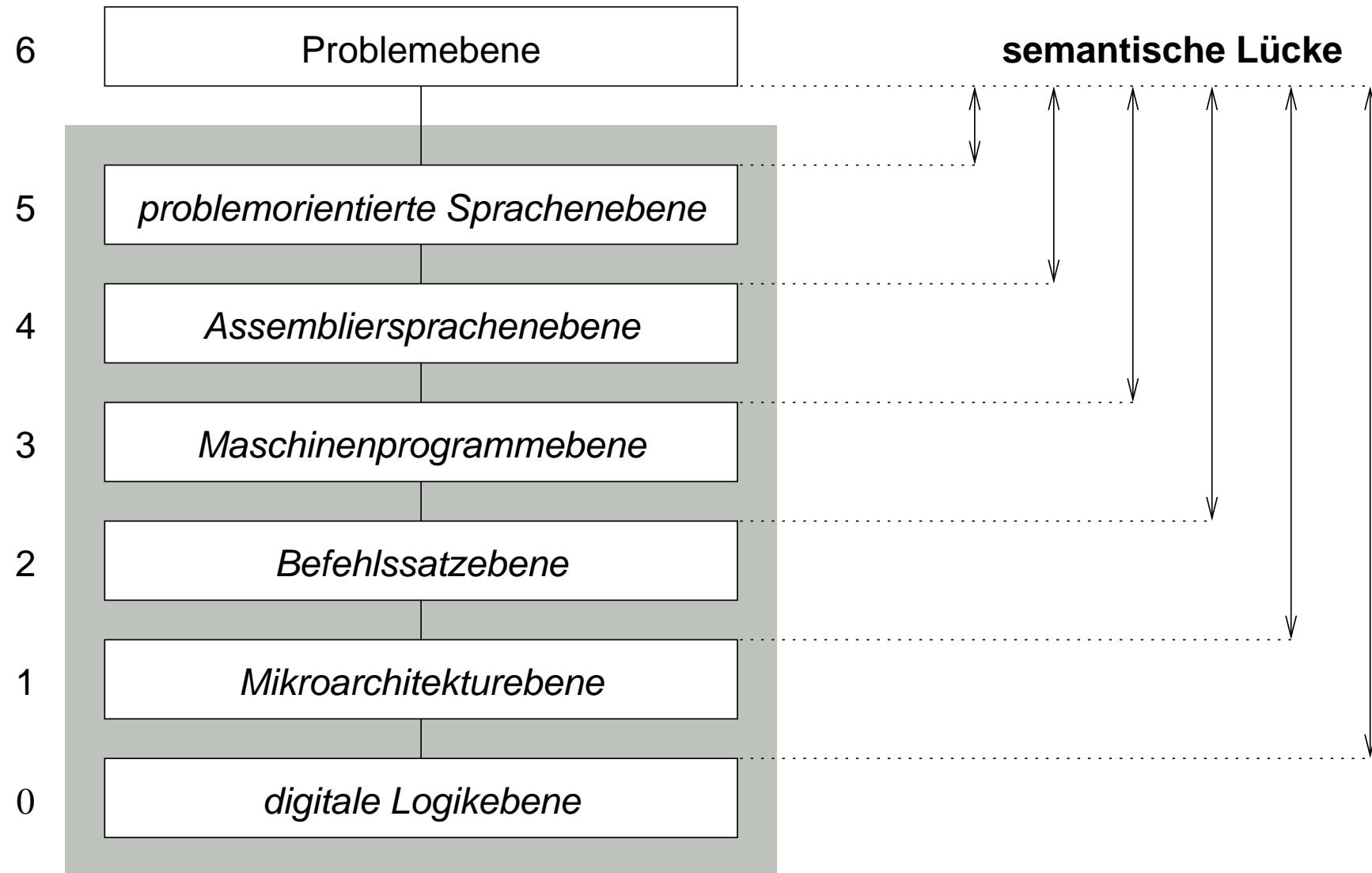
## Hardwaremaschinen

### digitale Logikebene (*Boolsche Algebra*)

- ▶ bildet auf Basis von Transistoren, Gattern, Schaltnetzen und Schaltwerken die wirkliche Hardware des Rechners
- ▶ Programme bestehen aus Elementen der **Schaltalgebra**
  - ▶ UND, ODER und NICHT bzw. NAND oder NOR

- ▶ maximale Flexibilität
- ▶ minimale Benutzer(un)freundlichkeit
- ▶ maximale Distanz von sehr vielen Problemdomänen

# Abstraktionsniveau vs. Semantische Lücke



# Abbildung durch Übersetzung

## Ebene 5 $\mapsto$ Ebene 4 (Kompilierung)

- ▶ Ebene 5-Befehle „1:N“ in Ebene 4-Befehle übersetzen
  - ▶ ein Hochsprachenbefehl als Sequenz von Assembliersprachenbefehlen
  - ▶ eine **semantisch äquivalente Befehlsfolge** generieren
- ▶ im Zuge der Transformation ggf. Optimierungsstufen durchlaufen

## Ebene 4 $\mapsto$ Ebene 3 (Assemblierung und Binden)

- ▶ Ebene 4-Befehle „1:1“ in Ebene 3-Befehle übersetzen
  - ▶ ein **Quellmodul** in ein **Objektmodul** umwandeln
  - ▶ mit **Bibliotheken** zum Maschinenprogramm zusammenbinden
- ▶ symbolischen Maschinenkode („Mnemoniks“) in binären umwandeln

# Abbildung durch Interpretation

Ebene  $3 \rightarrow$  Ebene  $2$  (Teilinterpretation, auch *partielle Interpretation*)

- ▶ Ebene  $3$ -Befehle als Folgen von Ebene  $2$ -Befehlen ausführen
  - ▶ Systemaufrufe aus den Ebene  $3$ -Befehlstrom „herausfiltern“
- ▶ ein Ebene  $3$ -Befehl aktiviert ein Ebene  $2$ -Programm

Ebene  $2 \rightarrow$  Ebene  $1$  (Interpretation)

- ▶ Ebene  $2$ -Befehle als Folgen von Ebene  $1$ -Befehlen ausführen
  - ▶ Abruf- und Ausführungszzyklus (engl. *fetch-execute-cycle*) der CPU
- ▶ ein Ebene  $2$ -Befehl löst Ebene  $1$ -Steueranweisungen aus

# Prozessoren implementieren die Abbildungen

Ebene 5  $\leadsto$  Kompilierer

- ▶ Interpretation von Konstrukten/Anweisungen einer „Hochsprache“

Ebene 4  $\leadsto$  Assemblierer und Binder

- ▶ Interpretation von Anweisungen einer Assemblersprache

Ebene 3  $\leadsto$  Betriebssystem

- ▶ Interpretation von Systemaufrufen
- ▶ Ausführung von Ebene 3-Programmen (durch Teilinterpretation)

Ebene 2  $\leadsto$  Zentraleinheit (CPU)

- ▶ Interpretation von Instruktionen (an die ALU, FPU, MMU, ...)
- ▶ Ausführung von Ebene 2-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 Assembliersprachenebene

`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
```

unauflöste Referenzen werden vom Binder (`ld(1)`) behandelt

- ▶ Folge ist u.a. das statische Einbinden von `read(2)` und `write(2)`

# Programm der Assembliersprachenebene (Forts.)

Auszüge aus der C-Bibliothek (libc.a) des Kompilierers (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
```

Binder (ld(1)) fügt diese Bibliotheksfunctionen später hinzu

Teilinterpretation (int \$0x80) schaltet um von Ebene<sub>2</sub> zu Ebene<sub>3</sub>

- ▶ das BS behandelt den Systemaufruf (S. 5-24)

# Programm der Assembliersprachenebene (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
    ...
```

## Interpretieren

```
...
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
```

1. Prozessorstatus des unterbrochenen Programms sichern
2. Systemaufrufnummer überprüfen, Systemaufruf interpretieren
3. Prozessorstatus wiederherstellen und zurückspringen

# Programm der problemorientierten Sprachenebene (Forts.)

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

## Systemaufrufe implementierende Programme

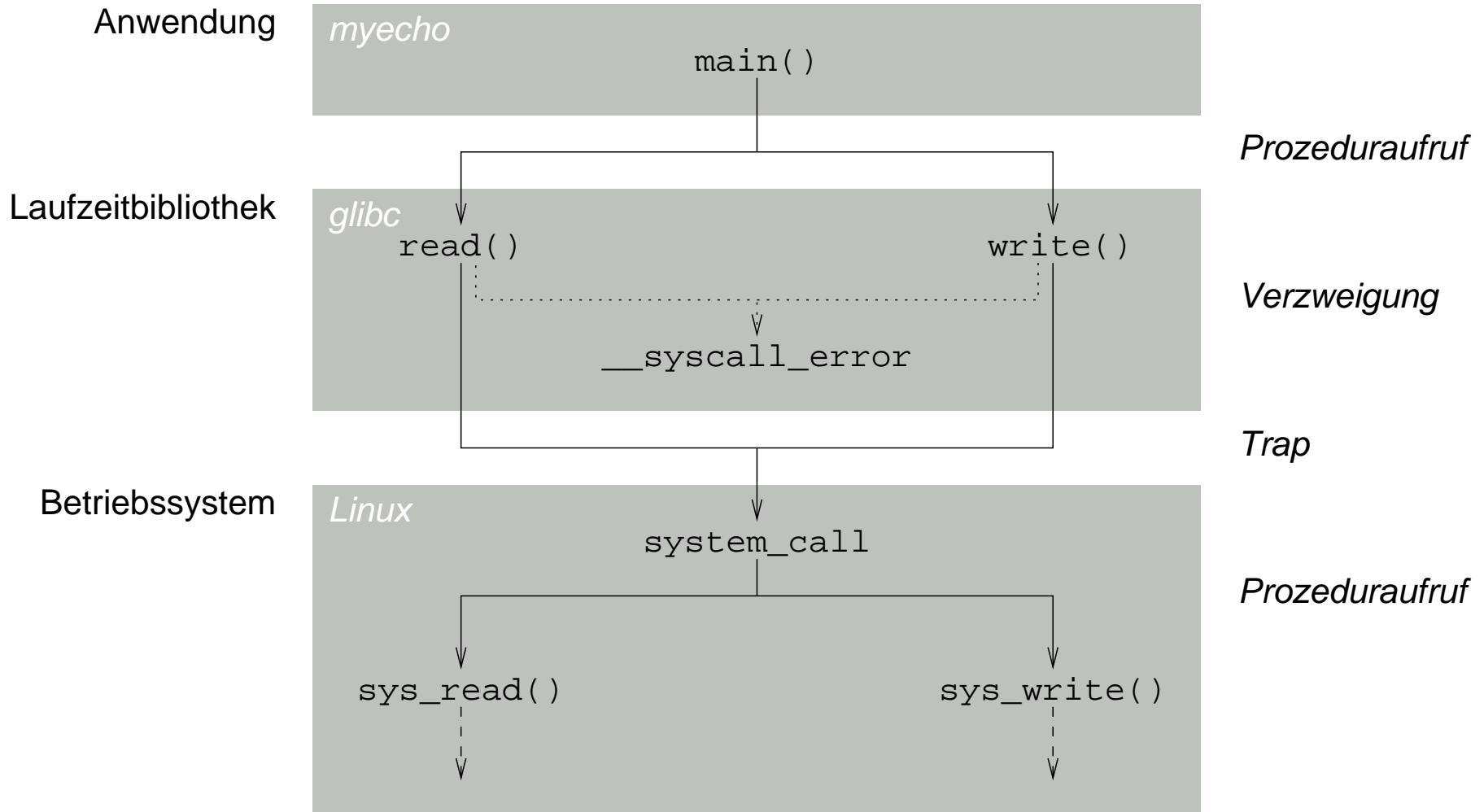
```
asmlinkage ssize_t sys_read(unsigned int fd, char * buf, size_t count) {  
    ssize_t ret;  
    struct file * file;  
  
    ret = -EBADF;  
    file = fget(fd);  
    if (file) {  
        ...  
    }  
    return ret;  
}  
  
asmlinkage ssize_t sys_write ...
```

## Ebene 2-Programme zur Teilinterpretation der Maschinenprogramme

- ▶ Aufruf: **call \*sys\_call\_table(,%eax,4)** (S. 5-24)

# Softwaresystem „myecho“

## Aufrufhierarchie



# Systemaufrufschnittstelle (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 BS

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

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

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

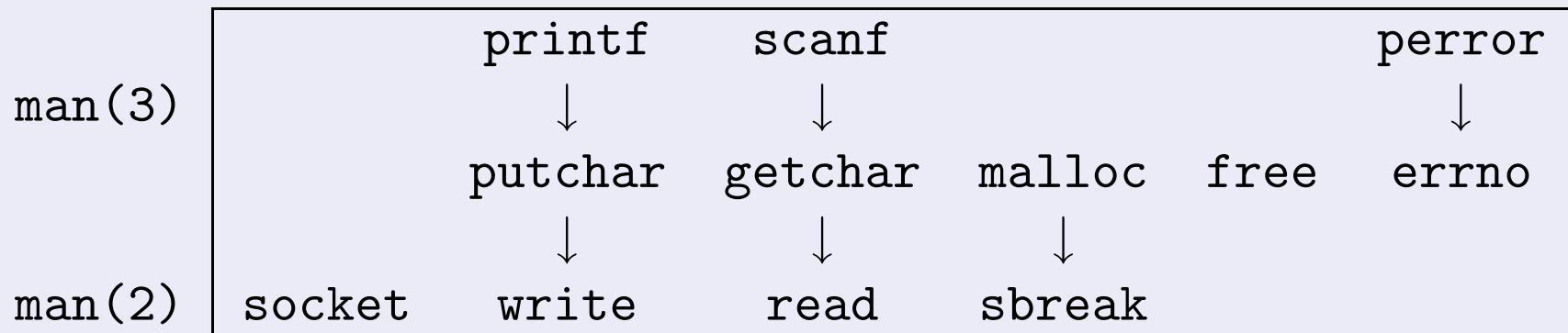
# Laufzeitumgebung (engl. *runtime environment*)

*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“

## Anwendungs routinen (des Rechners)

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

## Laufzeitsystem (des Kompilierers/Betriebssystems)

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

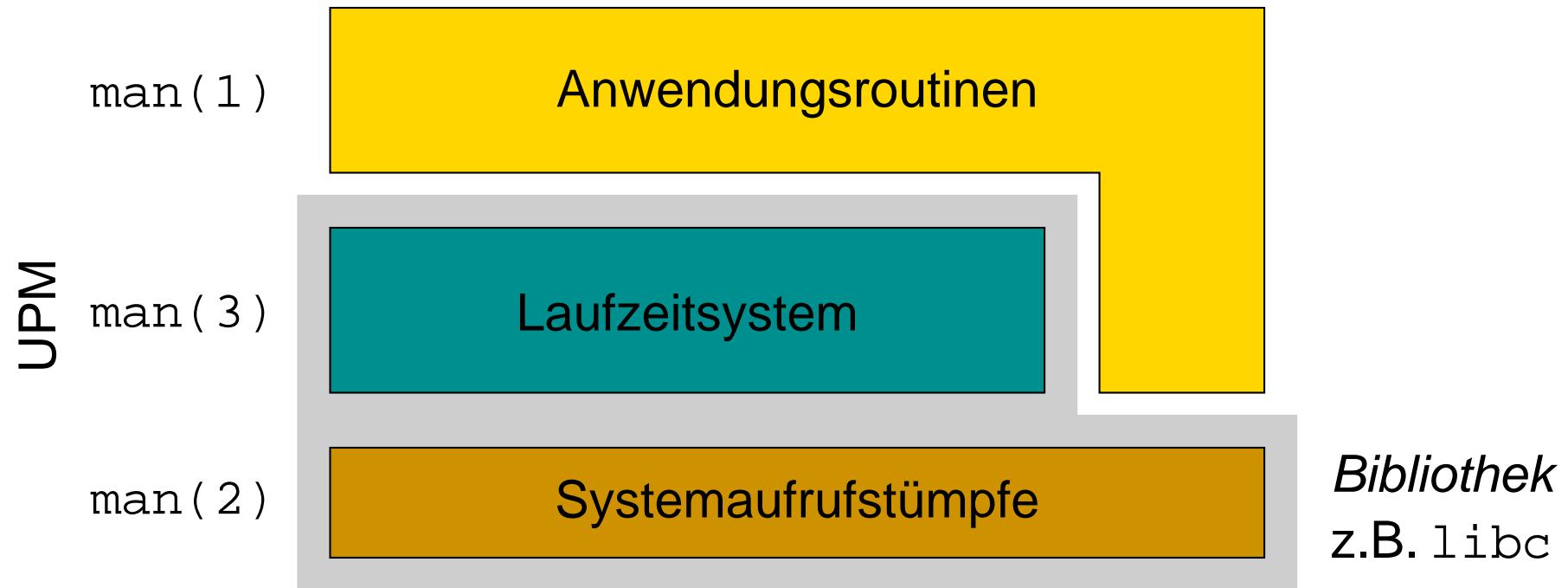
## Systemaufrufstümpfe (des Betriebssystems)

- ▶ bei UNIX z.B. die Bibliotheksfunktionen `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

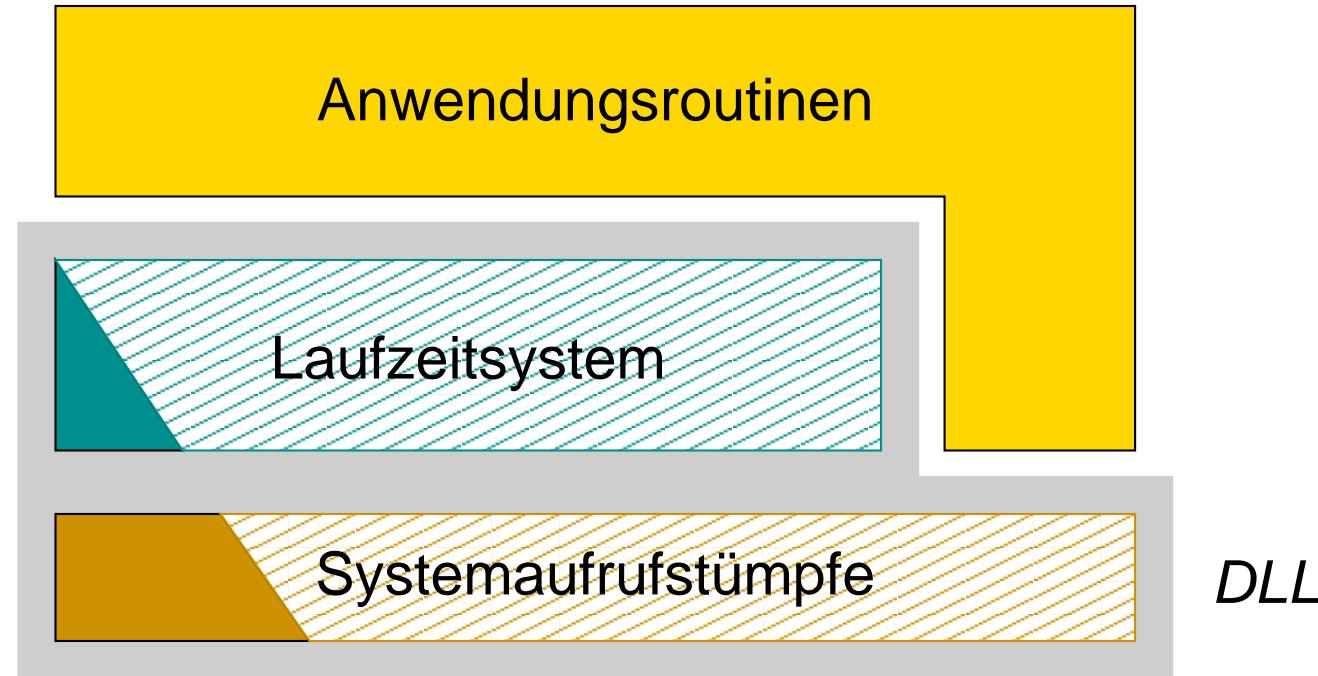


Modell für eine **statische Bibliothek** (gcc bzw. ld -static ...)

- ▶ 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*, *dynamic link library* (DLL))



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 Maschinenprogrammebene

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 { werden „wahrgenommen“, nicht ausgeführt  
signalisieren eine **Ausnahme** (engl. *exception*)

- ☞ Betriebssysteme fangen Ebene<sub>3</sub>-Befehle ab, behandeln Ausnahmen

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

Programmunterbrechungen bewirken partielle Interpretation

## Maschinenprogramm

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

Ebene 3

## Anwendungsprogramm

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

Ebene 3

trap/interrupt

system call

50FC061E50...581F0783C404CF  
return from exception

50FC061E50...581F0783C404CF  
return from exception

Ebene 2

## Betriebssystemprogramme

# Programme der Maschinenprogrammebene

## Hybride Ebene

Ebene 3-Befehle . . .

- ▶ sind „normale“ Befehle der Ebene 2, 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

Ausnahmesituationen (der Ebene<sub>2</sub>) fallen in zwei Kategorien:

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

Unterschiede ergeben sich hinsichtlich...

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

Behandlung ist zwingend und grundsätzlich prozessorabhängig

# Synchrone Programmunterbrechung

- ▶ unbekannter Befehl, falsche Adressierungsart oder Rechenoperation
- ▶ Systemaufruf, Adressraumverletzung, unbekanntes Gerät
- ▶ Seitenfehler im Falle lokaler Ersetzungsstrategien

## Trap — synchron, vorhersagbar, reproduzierbar

Ein in die Falle gelaufenes („getrapptes“) Programm, das unverändert wiederholt und jedesmal mit den selben Eingabedaten versorgt auf ein und demselben Prozessor zur Ausführung gebracht wird, wird auch immer wieder an der selben Stelle in die selbe Falle tappen.

- ☞ Trapvermeidung ist ohne Behebung der Ausnahmebedingung unmöglich

# Asynchrone Programmunterbrechung

- ▶ Signalisierung „externer“ Ereignisse
- ▶ Beendigung einer DMA- bzw. E/A-Operation
- ▶ Seitenfehler im Falle globaler Ersetzungsstrategien

**Interrupt — asynchron, unvorhersagbar, nicht reproduzierbar**

Ein „externer Prozess“ (z.B. ein Gerät) signalisiert einen Interrupt unabhängig vom Arbeitszustand des gegenwärtig sich in Ausführung befindlichen Programms. Ob und ggf. an welcher Stelle das betreffende Programm unterbrochen wird, ist nicht vorhersehbar.

☞ Ausnahmesituationsbehandlung muss **nebeneffektfrei** verlaufen

# Trap oder Interrupt?

```
#include <stdlib.h>

float frandom () {
    return random()/random();
}
```

Division durch 0 ...

- ▶ Programmunterbrechung  
(je nach CPU)
- ▶ wird zufällig geschehen

Programmierfehler, der sich jedoch nicht zwingend auswirken muss:

- ▶ die Unterbrechung verläuft **synchron** zum Programmablauf ..... Trap
- ▶ die Unterbrechungsstelle im Programm ist **vorhersagbar** ..... Trap
- ▶ der Zufall macht die Unterbrechung **nicht reproduzierbar** .... Interrupt

## Trap oder Interrupt? (Forts.)

```
extern edata, end;  
int main () {  
    char* p = (char*)&edata;  
    do *p++ = 0;  
    while (p != (char*)&end);  
}
```

Indirekte Adressierung (`*p++`) ...

- ▶ Programmunterbrechung  
(je nach Systemauslastung)
- ▶ trotz korrektem Text

Seitenfehler (engl. *page fault*), vorbehaltlich eines virtuellen Speichers

- ▶ die Unterbrechung verläuft **synchron** zum Programmablauf ..... Trap

Diskussionsstoff liefert die Ersetzungsstrategie (engl. *replacement policy*):

**lokal** ~> Stelle **vorhersagbar**, Unterbrechung **reproduzierbar** ..... Trap

**global** ~> Stelle **unvorhersagbar**, Unterbr. **nicht reproduzierbar** ... Interrupt

# Ausnahmesituationen sind Betriebssystemnormalität

Ereignisse, oftmals unerwünscht aber nicht immer eintretend:

- ▶ Signale von der Peripherie (z.B. E/A, Zeitgeber oder „Wachhund“)
- ▶ Wechsel der Schutzdomäne (z.B. Systemaufruf)
- ▶ Programmierfehler (z.B. ungültige Adresse)
- ▶ unerfüllbare Speicheranforderung (z.B. bei Rekursion)
- ▶ Einlagerung auf Anforderung (z.B. beim Seitenfehler)
- ▶ Warnsignale von der Hardware (z.B. Energiemangel)

Ereignisbehandlung, die problemspezifisch zu gewährleisten ist:

- ▶ als Ausnahme während der „normalen“ Programmausführung

# Bezug zur Softwaretechnik

Ausnahmebehandlung (engl. *exception handling*, [15])

## Wiederaufnahmemodell (engl. *resumption model*)

- ▶ die Behandlung der Ausnahmesituation führt zur **Fortsetzung** der Ausführung des unterbrochenen Programms
- ▶ ein Trap kann, ein Interrupt muss so behandelt werden

## Beendigungsmodell (engl. *termination model*)

- ▶ die Behandlung der Ausnahmesituation führt zum **Abbruch** der Ausführung des unterbrochenen Programms

- ▶ ggf. als Folge eines schwerwiegenden Laufzeitfehlers

- ▶ ein Trap kann, ein Interrupt darf niemals so behandelt werden

☞ Auslösung (engl. *raising*) einer Ausnahme impliziert **Kontextwechsel**

# Ausnahmebehandlung impliziert Kontextwechsel

Abrupter Zustandswechsel

Programmunterbrechungen implizieren **nicht-lokale Sprünge**:

vom  $\left\{ \begin{array}{l} \text{unterbrochenen} \\ \text{behandelnden} \end{array} \right\}$  Programm zum  $\left\{ \begin{array}{l} \text{behandelnden} \\ \text{unterbrochenen} \end{array} \right\}$  Programm

Sprünge (und Rückkehr davon), die Kontextwechsel nach sich ziehen:

- ▶ erfordert Maßnahmen zur Zustandssicherung/-wiederherstellung
- ▶ Mechanismen liefert das behandelnde Programm/die tiefere Ebene

☞ der **Prozessorstatus** unterbrochener Programme muss invariant sein

## Prozessorstatus invariant halten

Ebene<sub>2</sub> (CPU) sichert bei Ausnahmen einen Zustand minimaler Größe

- ▶ Statusregister (SR) und Befehlszeiger (engl. *program counter*, PC)
- ▶ möglicherweise aber auch den kompletten Registersatz
- ▶ je nach CPU werden dabei wenige bis sehr viele Daten(bytes) bewegt

Ebene<sub>3/5</sub> (Betriebssystem/Komplizierter) sichert den restlichen Zustand

- ▶ d.h., alle  $\left\{ \begin{array}{l} \text{dann noch ungesicherten} \\ \text{im weiteren Verlauf verwendeten} \end{array} \right\}$  CPU-Register

☞ die zu ergreifenden Maßnahmen sind höchst **prozessorabhängig**

# Prozessorstatus sichern und wiederherstellen

Prozessor „Betriebssystem“

Zeile

1:  
2:  
3:  
4:  
5:

x86

```
train:  
    pushal  
    call handler  
    popal  
    iret
```

m68k

```
train:  
    moveml d0-d7/a0-a6,a7@-  
    jsr handler  
    moveml a7@+,d0-d7/a0-a6  
    rte
```

**train** (trap/interrupt):

- ▶ Arbeitsregisterinhalte im RAM sichern (2) und wiederherstellen (4)
- ▶ Unterbrechungsbehandlung durchführen (3)
- ▶ Ausführung des unterbrochenen Programms wieder aufnehmen (5)

# Prozessorstatus sichern und wiederherstellen (Forts.)

Prozessor „Komplizierer“

gcc

```
void __attribute__ ((interrupt)) train () {  
    handler();  
}
```

## \_\_attribute\_\_ ((interrupt))

- ▶ Generierung der speziellen Maschinenbefehle durch den Komplizierer
    - ▶ zur Sicherung/Wiederherstellung der Arbeitsregisterinhalte
    - ▶ zur Wiederaufnahme der Programmausführung
  - ▶ nicht jeder „Prozessor“ (für C/C++) implementiert dieses Attribut
- ☞ „prozessorabhängig“ bedeutet nicht immer gleich „CPU-abhängig“ !!!

# Unvorhersagbare Laufzeitvarianzen

Unterbrechungen verzögern Programmabläufe

Problem für **determinierte Programme** . . .

- ▶ lassen bei ein und derselben Eingabe verschiedene Abläufe zu
- ▶ alle Abläufe liefern jedoch stets das gleiche Resultat

. . . da asynchrone Unterbrechungen sie **nicht-deterministisch** machen

- ▶ nicht zu jedem Zeitpunkt ist bestimmt, wie weitergefahrene wird

Nichtdeterminismus ist kritisch für **echtzeitabhängige Programme**

- ▶ die Laufzeitumgebung dieser Programme muss echtzeitfähig sein
  - ▶ ermöglicht einem sich in Ausführung befindlichem Programm alle Zeit- und Terminvorgaben seiner Umgebung einzuhalten
  - ▶ die Echtzeitbedingungen (des Programms) sind weich, fest oder hart
- ▶ eine Anforderung unter Einbezug aller Last- und Fehlerbedingungen !!!

# Echtzeitbedingungen

Unterbrechungen erschweren Echtzeitprogrammierung

weich (engl. *soft*) auch „schwach“

- ▶ das Ergebnis einer zu einem vorgegebenen Termin nicht geleisteten Arbeit ist weiterhin von Nutzen
- ▶ Terminverletzung ist tolerierbar

fest (engl. *firm*) auch „stark“

- ▶ das Ergebnis einer zu einem vorgegebenen Termin nicht geleisteten Arbeit ist wertlos und wird verworfen
- ▶ Terminverletzung ist tolerierbar, führt zum Arbeitsabbruch

hart (engl. *hard*) auch „strikt“

- ▶ das Versäumnis eines fest vorgegebenen Termins kann eine „Katastrophe“ hervorrufen
- ▶ Terminverletzung ist keinesfalls tolerierbar

# Nichtdeterministisches Programm

Asynchronität von Programmunterbrechungen

Welche wheel-Werte gibt main() aus?

```
unsigned int wheel = 0;

void __attribute__((interrupt)) train () {
    wheel++;
}

int main () {
    for (;;)
        printf("%10u", wheel++);
}
```

0 1 2 4 ... 13 13 14 15 16 ... 4711 4711 2002 ... ???

# Nichtdeterministisches Programm (Forts.)

## Teilbarkeit von Operationen

wheel++ ist Elementaroperation (kurz: Elop) der Ebene<sub>5</sub>...

- ▶ ein Prozessor führt seine Elop atomar, d.h. unteilbar aus

... jedoch nicht notwendigerweise auch eine der Ebene<sub>4</sub> (und tiefer)

### wheel++ in main()

```
movl wheel,%edx  
incl %edx  
movl %edx,wheel
```

### wheel++ in train()

```
movl wheel,%eax  
incl %eax  
movl %eax,wheel
```

- ☞ train() überlappt main() im Unterbrechungsfall !!!

# Unterbrechungsbedingte Überlappungseffekte

Kritischer Programmtext

## Nebenläufiges Zählen

main()		train()		wheel
x86-Befehl	%edx	x86-Befehl	%eax	
movl wheel,%edx	42	movl wheel,%eax incl %eax movl %eax,wheel	42 43 43	42 42 42 43
incl %edx	43			43
movl %edx,wheel	43			43

- ☞ zweimal durchlaufen (main() und train()), aber nur einmal gezählt

# Nichtsequentielles Programm

Laufgefahr (engl. *race hazard*)

## Wettlaufsituation (engl. *race condition*)

- ▶ zwei (oder mehr) Aktionen wetteifern um die Absicht, als erste ein Berechnungsergebnis herbeizuführen
    - ▶ Signal auslösen, Datum verändern, . . . , Betriebsmittel beanspruchen
  - ▶ eine Berechnung zeigt eine unerwartet kritische Abhängigkeit vom relativen Zeitverlauf von Ereignissen
    - ▶ fehlerhafte Stelle in einem (Hardware/Software) System
  - ▶ ein potentielles Problem, sobald die Ausführung von Programmen nebenläufig (d.h. überlappend oder parallel) möglich ist
- ☞ **kritischer Abschnitt** eines nebenläufig ausgeführten Programms

# Überlappungsschutz

Kritischen Abschnitt als Elementaroperation auslegen

Lösungsansatz (für den gegebenen Fall, Monoprozessoren):

- ▶ Schutz vor einer möglichen überlappenden Programmausführung
  - ▶ temporäres Abschalten asynchroner Programmunterbrechungen
  - ▶ „Synchronisationsklammern“ um den kritischen Abschnitt setzen
- ▶ auf Elop eines tieferen Prozessors (genauer: der CPU) abbilden
  - ▶ die **bessere Lösung**, sofern die CPU eine passende Elop dafür anbietet
  - ▶ ggf. praktikabel bei CISC (z.B. x86), nicht aber bei RISC (z.B. ppc)

Abstraktion als grundsätzliche Vorgehensweise: eine Elop schaffen...

- ▶ einen kritischen Abschnitt als **Modul** abkapseln
- ▶ den modularisierten Programmtext passend synchronisieren

# Überlappungsschutz (Forts.)

Abstraktion und Synchronisation  $\leadsto$  Virtualisierung

## Modularisieren

```
int main () {  
    for (;;) {  
        printf("%10u", incr(&wheel));  
    }  
}
```

Bei langer **Verdrängung** des **main()**-Prozesses und hoher **Unterbrechungsfrequenz** sind Schrittweiten ungleich 1 bei der Ausgabe weiterhin möglich:  
**Zahlenüberlauf !!!**

## Komplexbefehl verwenden

```
inline int incr (int* ip) {  
    asm ("incl %0" : : "g" (*ip));  
    return *ip;  
}
```

## Interrupts abschalten

```
inline int incr (int* ip) {  
    asm ("cli"); *ip += 1; asm ("sti");  
    return *ip;  
}
```

# Äquivalenz von Hardware und Software

Ebene<sub>2</sub>-Befehle sind Teil der ISA, ihre Implementierungen nur bedingt

- ▶ Befehlssätze, sind optional in Hardware oder Software implementiert
  - ▶ Koprozessor (z.B. *floating-point unit*, FPU)
  - ▶ rekonfigurierbare Hardware (z.B. *field-programmable array* FPGA)
- ▶ die Festlegung der wirklichen Implementierungsebene erfolgt später

Ebene<sub>2</sub>-Befehle können durch Ebene<sub>2</sub>-Programme *emuliert* werden

Allgemein gilt:

- ▶ Ebene<sub>i</sub>-Befehl kann durch Ebene<sub>j</sub>-Programm emuliert werden
- ▶ Ebene<sub>i</sub>-Programm kann durch Ebene<sub>j</sub>-Befehl implementiert werden

# Emulation $\mapsto$ Nachbildung

Spezialfall der Simulation, bei dem das Verhalten einer Maschine durch eine andere Maschine vollständig nachgebildet wird:

- ▶ die Nachahmung der Eigenschaften eines ggf. anderen Rechtersystems
- ▶ bei **Selbstvirtualisierung** emuliert sich ein Rechtersystem selbst

**Nachbilder** (engl. *emulator*) interpretiert die von einer wirklichen, realen Maschine nicht ausführbaren Befehle:

- ▶ die betreffenden Befehle sind der (realen) Maschine bekannt
- ▶ sie müssen jedoch nicht zwingend auch in ihr implementiert sein

Extremfall der Emulation: z.B., Virtual PC für Apple-Rechner...

- ▶ x86 (Ebene<sub>2</sub>) wird auf PowerPC-Basis (Ebene<sub>2</sub>) nachgebildet
- ▶ Ebene<sub>3</sub>-Programm (MacOS/PowerPC) simuliert einen x86

# Selbstvirtualisierung durch Teilinterpretation

Voraussetzung ist, die reale Maschine „trappt“ **privilegierte Befehle** im nicht-privilegierten Arbeitsmodus

- ▶ beim x86 z.B. Befehle  $\left\{ \begin{array}{ll} \text{zur Ein-/Ausgabe} & (\text{in}, \text{out}) \\ \text{zur Synchronisation} & (\text{cli}, \text{sti}) \\ : & \\ \text{zum Moduswechsel} & (\text{int}, \text{iret}) \\ \text{zum ???} & (\text{popf}) \end{array} \right\}$

## Arbeitsmodi (einer CPU — jedoch nicht jeder)

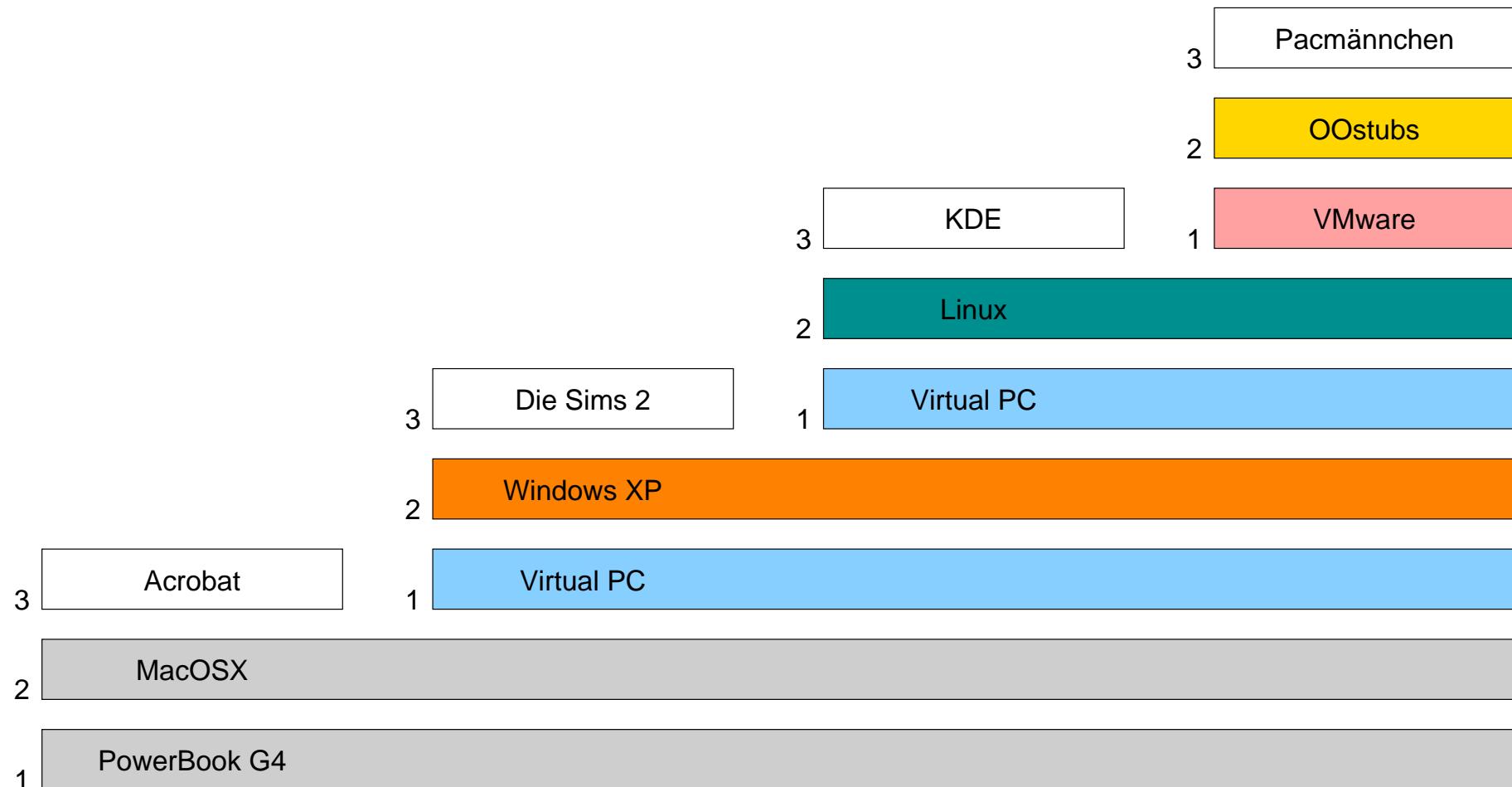
- ▶ im privilegierten Modus läuft nur das Betriebssystem
  - ▶ ggf. auch nur ein **Minimalkern** (*virtual machine monitor*, VMM) davon
- ▶ im nicht-privilegierten Modus laufen alle anderen Programme

## Selbstvirtualisierung durch Teilinterpretation (Forts.)

Ausgangspunkt ist die Anzeige des Eintritts einer **Ausnahmesituation** und die Umschaltung in den privilegierten Arbeitsmodus des Prozessors:

1. das Betriebssystem/der VMM analysiert die Unterbrechung
    - ▶ d.h., den Systemaufruf, Trap oder Interrupt
  2. für das unterbrochene Programm wird ein Emulator gestartet
    - ▶ Ausführung des die Unterbrechung hat verursachenden Befehls
    - ▶ Abbildung von Geräteaktionen auf E/A-Funktionen des Betriebssystems
    - ▶ Umsetzung von Adressraumzugriffen und privilegierten Befehlen
  3. das unterbrochene Programm wird weiter fortgeführt
    - ▶ Beendigung der Emulation des Befehls bzw. Zugriffs
    - ▶ Reaktivierung des nicht-privilegierten Arbeitsmodus
- ☞ **Abruf- und Ausführungszyklus** eines „fiktiven“ Prozessors durchlaufen

# Hierarchie virtueller Maschinen



# Grenzen der Emulation

## Funktionale vs. nicht-funktionale Eigenschaften

Nachahmung **funktionaler Eigenschaften** ist „leicht“ möglich

- ▶ d.h., in Funktionseinheiten gekapselter Fähigkeiten eines Prozessors:
  - ▶ Fließkommaeinheit, Vektoreinheit, Graphikbeschleuniger
  - ▶ Adressumsetzungs- und Kommunikationshardware
  - ▶ ISA eines beliebigen Prozessors
- ▶ solitäre („einzelne stehende“) Funktionen von Hardware oder Software

Schwierigkeiten bereiten **nicht-funktionale Eigenschaften**:

- ▶ ein emulierter Befehl wird durch ein „Unterprogramm“ ausgeführt
- ▶ er läuft dadurch langsamer ab und verbraucht auch mehr Energie

## Virtual PC: MacOS → Windows XP

Ein 1.25 GHz PowerPC G4 wird zum 293 MHz Pentium 686.

# Rechnerorganisation

## Strukturierte Organisation von Rechensystemen

### Hierarchie virtueller Maschinen (bzw. abstrakter Prozessoren)

- ▶ schrittweises Schließen der semantischen Lücke
- ▶ Mehrebenenmaschinen — Betriebssysteme implementieren Ebene  $_3$
- ▶ Ebene  $_i \mapsto$  Ebene  $_{i-1}$  durch Programme, für  $i > 1$
- ▶ Teilinterpretation (von Systemaufrufen) durch das Betriebssystem
- ▶ synchrone und asynchrone Programmunterbrechungen
- ▶ nebenläufige (bzw. überlappende) Ausführung von Programmen
- ▶ Nachahmung von Eigenschaften ggf. anderer (abstrakter) Prozessoren