

# Systemprogrammierung

## Organisation von Rechensystemen

10./13. November 2008

# Überblick

## Organisation von Rechenystemen

Semantische Lücke

Mehrebenenmaschinen

Softwaremaschinen

Partielle Interpretation

Programmunterbrechung

Nebenläufigkeit

Virtualisierung

Zusammenfassung

Bibliographie

# 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*, [1])

*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  $\rightsquigarrow$  1 Summenformel

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

Ebene der Programmiersprache C  $\rightsquigarrow$  5 Komplexschritte

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

Ebene der Assemblersprache (x86)  $\rightsquigarrow$   $43+n$  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)  $\rightsquigarrow$  121 Bytes Programmtext

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



# Matrix-Matrix Multiplikation (Forts.)

## Resümee

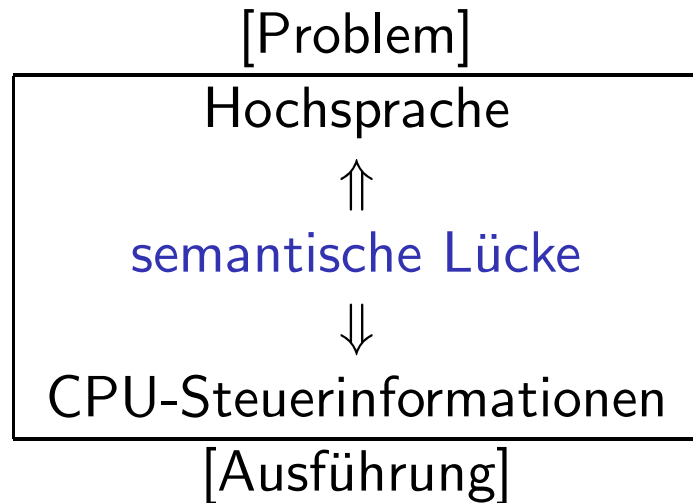
$$C_{i,j} = \sum_k A_{ik} \cdot B_{kj} \stackrel{?}{\iff} 5589E5 \dots 5F5DC3$$

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



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** (auch: abstrakte Prozessoren) auf die reale Maschine abbilden [2].

# Hierarchie virtueller Maschinen

## Interpretation und Übersetzung [2]

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)

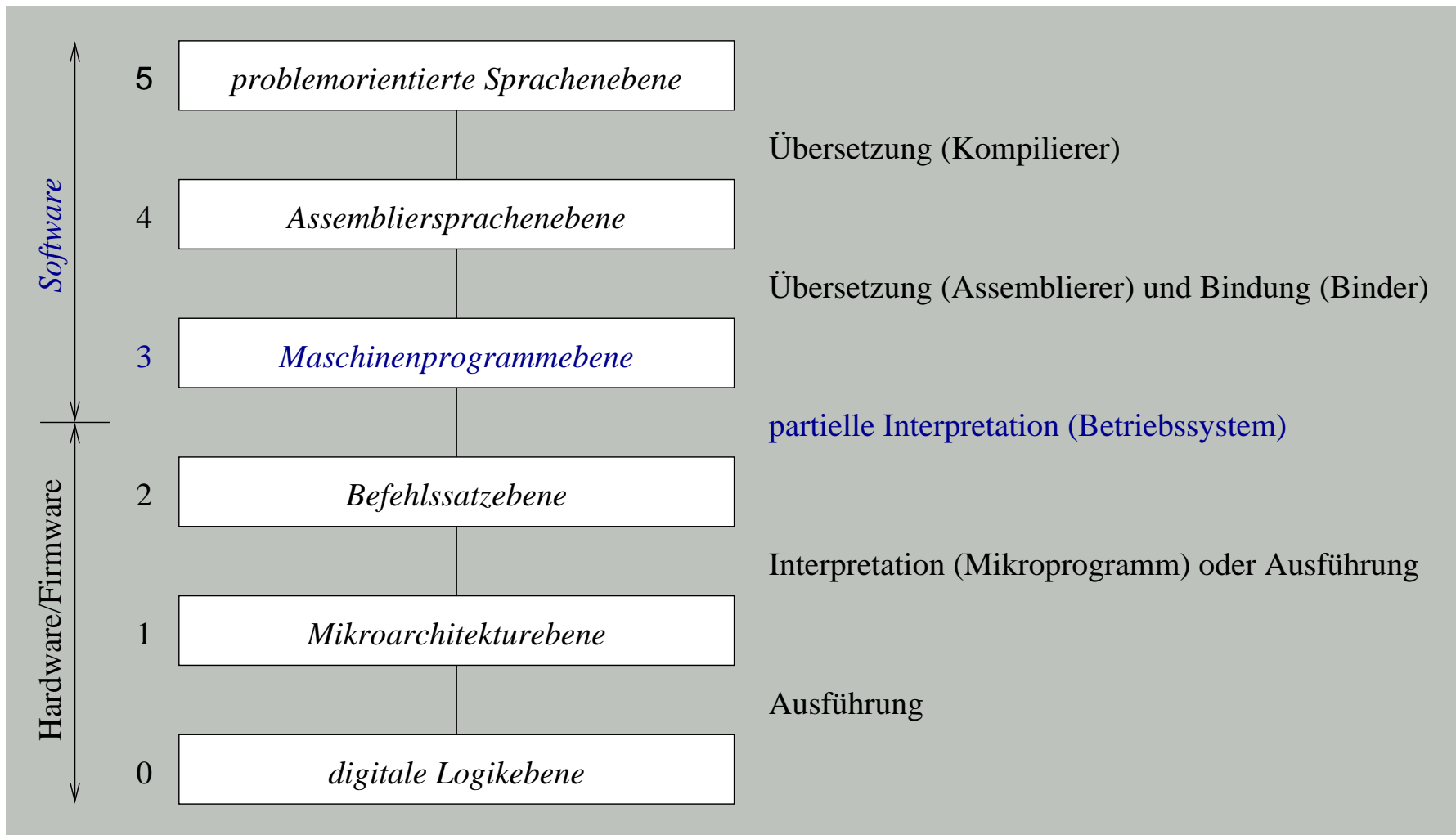
- ▶ **Softwareprozessor**, transformiert Programme einer *Quellsprache* in semantisch äquivalente Programme einer *Zielsprache*
  - ▶ {Ada, C, C++, Eiffel, Modula, Fortran, Pascal, ...}  $\mapsto$  Assembler
  - ▶ aber ebenso: C++  $\mapsto$  C  $\mapsto$  Assembler

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

- ▶ ein in Hard-, Firm- oder Software realisierter **Prozessor**, führt Programme einer bestimmten Quellsprache „direkt“ aus
  - ▶ z.B. Basic, Perl, C, sh(1)
- ▶ ggf. **Vorübersetzung** durch einen Kompilierer, um Programme in eine für die Interpretation günstigere Repräsentation zu bringen
  - ▶ z.B. Pascal P-Code, Java Bytecode, x86-Befehle

# Hardware/Software Hierarchie

Betriebssystem als Interpret [2]



# Elementaroperationen der einzelnen Ebenen

Softwaremaschinen — Übersetzer

## problemorientierte Programmiersprachenebene

[AuD/PFP]

- ▶ „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

## Assemblersprachenebene (*symbolischer Maschinenkode*)

[GROA]

- ▶ **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

## Maschinenprogrammzebene (*binärer Maschinenkode*) [SP]

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

## Betriebssysteme „ko-implementieren“ Maschinenprogrammzebene

- ▶ zum großen Teil kodiert in problemorientierte Programmiersprachen
  - ▶ vorwiegend C, zunehmend auch C++, kaum Java
- ▶ zum kleinen Teil kodiert in Assemblersprachen

# Elementaroperationen der einzelnen Ebenen (Forts.)

Firm-/Hardwaremaschinen — Interpretierer

**Befehlssatzebene** (engl. *instruction set architecture*, ISA) [GTI/GROA]

- ▶ 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**

[GTI]

- ▶ 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

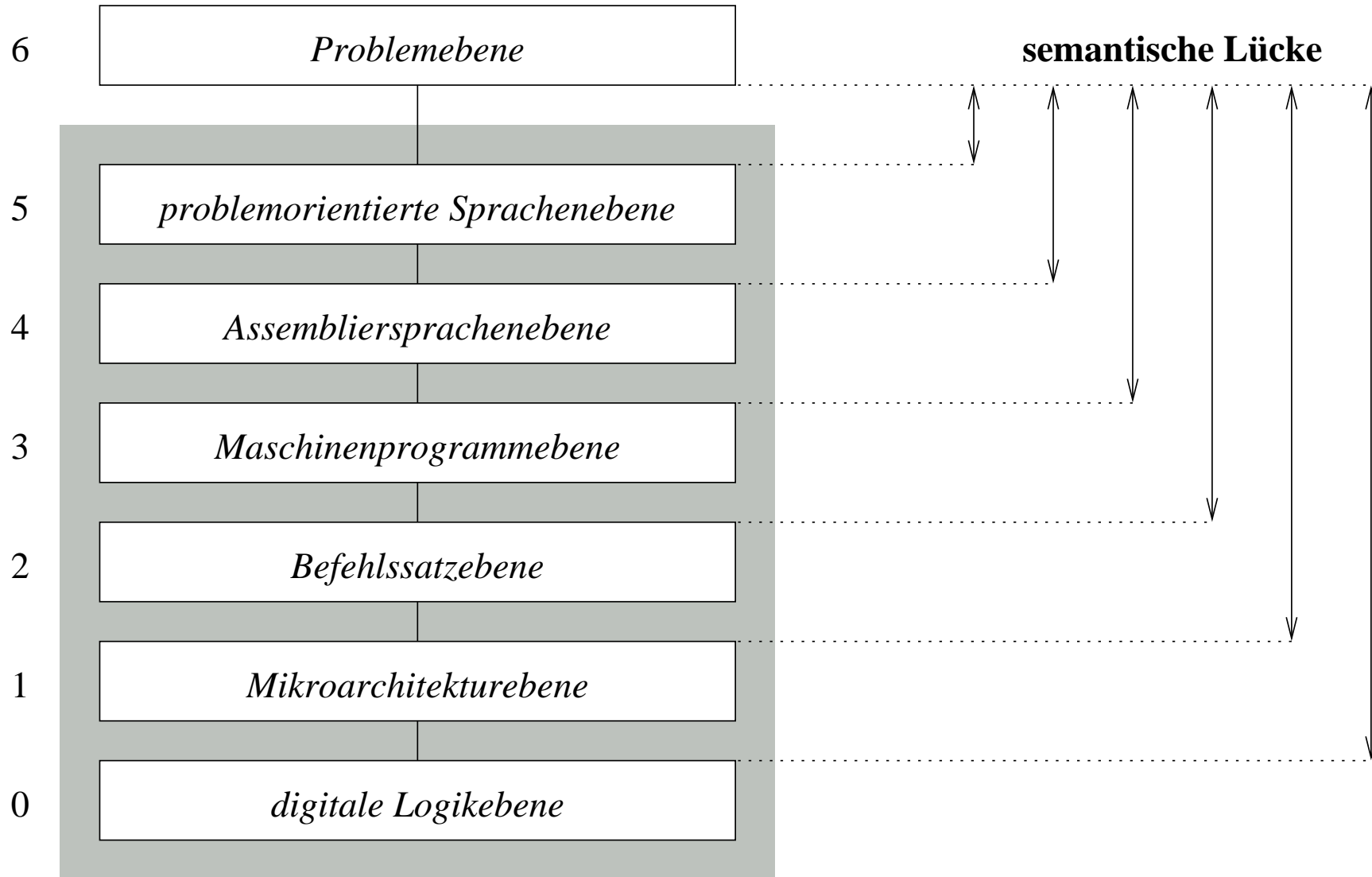
### digitale Logikebene (*Boolsche Algebra*)

[GTI]

- ▶ 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<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 als Sequenz von Assemblersprachenbefehlen
  - ▶ eine **semantisch äquivalente Befehlsfolge** generieren
- ▶ im Zuge der Transformation ggf. Optimierungsstufen durchlaufen

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

- ▶ Ebene<sub>4</sub>-Befehle „1:1“ in Ebene<sub>3</sub>-Befehle übersetzen
  - ▶ ein **Quellmodul** in ein **Objektmodul** umwandeln
  - ▶ mit **Bibliotheken** zum Maschinenprogramm zusammenbinden
- ▶ symbolischen Maschinencode („Mnemoniks“) auflösen
  - ▶ in binären Maschinencode umwandeln

# 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 den Ebene<sub>3</sub>-Befehlstrom „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>  $\rightsquigarrow$  **Kompilierer**

- ▶ Interpretation von Konstrukten/Anweisungen einer „Hochsprache“

Ebene<sub>4</sub>  $\rightsquigarrow$  **Assemblerer** und **Binder**

- ▶ Interpretation von Anweisungen einer Assemblersprache

Ebene<sub>3</sub>  $\rightsquigarrow$  **Betriebssystem**

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

Ebene<sub>2</sub>  $\rightsquigarrow$  **Zentraleinheit** (CPU)

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

# Ebene<sub>5</sub>-Programm realisiert mit Ebene<sub>5</sub>-Konzepten

Ebene<sub>6</sub>-Problem: Zeilenweises Echo einzeln eingegebener Zeichen...

```
myecho.c
```

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

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

## Ebene 5-Programm realisiert mit Ebene 4-Konzepten

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

```

**unaufgelöste Referenzen** werden vom Binder (ld(1)) behandelt

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

# Ebene 4-Programm realisiert mit Ebene 3-Konzepten

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
```

Binder (`ld(1)`) fügt diese Bibliotheksfunktionen später hinzu

Teilinterpretation (`int $0x80`) schaltet um von Ebene 2 zu Ebene 3

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



# Ebene 2-Programm realisiert mit Ebene 4-Konzepten

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

# Ebene<sub>2</sub>-Programm realisiert mit Ebene<sub>5</sub>-Konzepten

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;
}

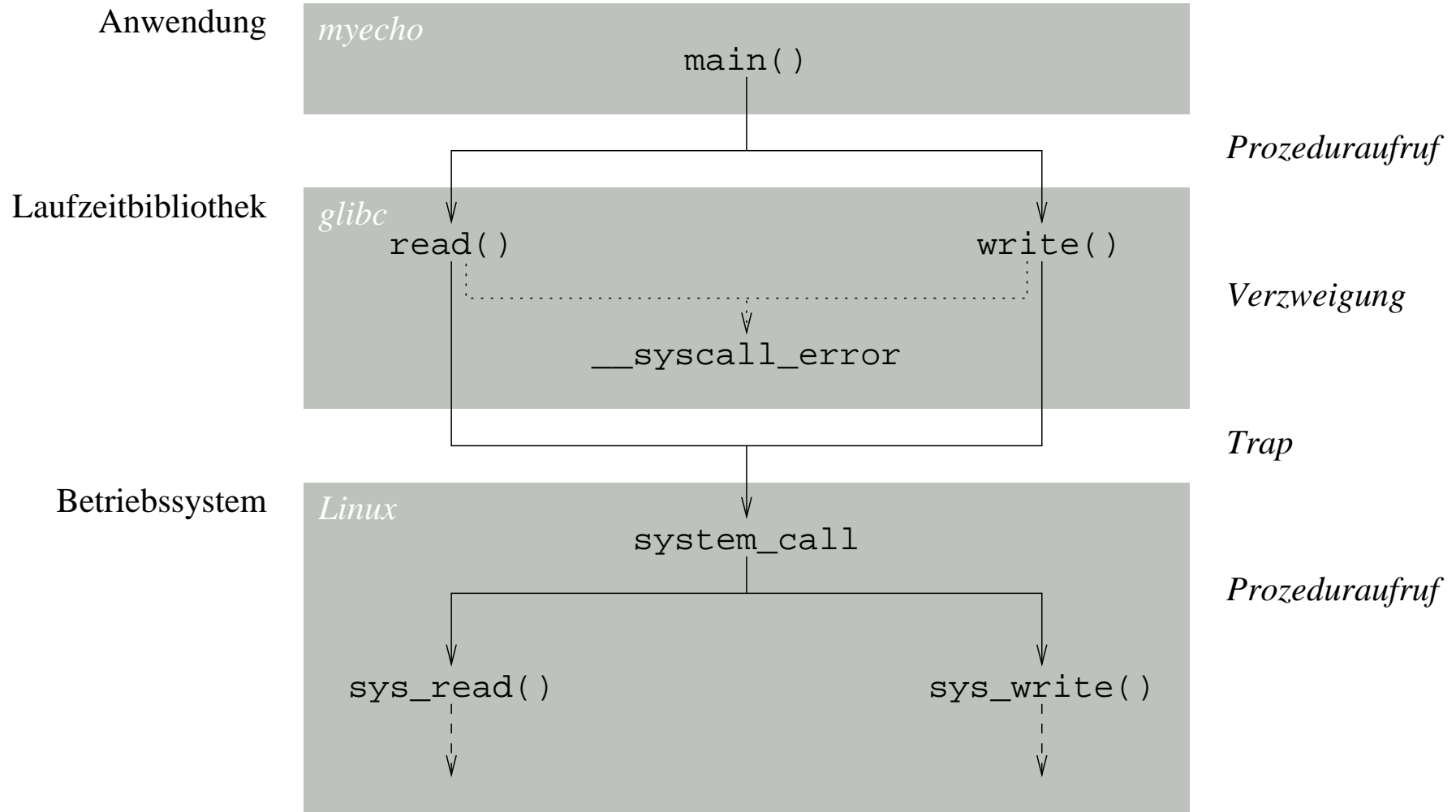
asmlinkage ssize_t sys_write ...
```

Ebene<sub>2</sub>-Programme zur Teilinterpretation der Maschinenprogramme

- ▶ Aufruf: `call *sys_call_table(,%eax,4)` (S. 5-24)

# 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 BS

- ▶ „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 BS
- ▶ Ein-/Ausgabeparameter in Registern übergeben, „Trap“ auslösen

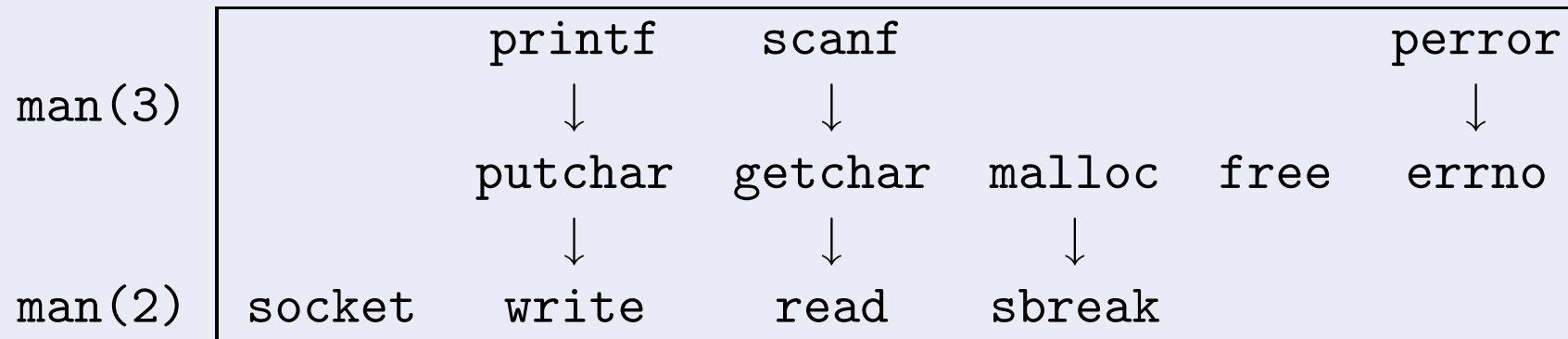
# Laufzeitumgebung (engl. *runtime environment*)

*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

Ensemble problemspezifischer Prozeduren: „Triumvirat“

## Anwendungsroutinen (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)

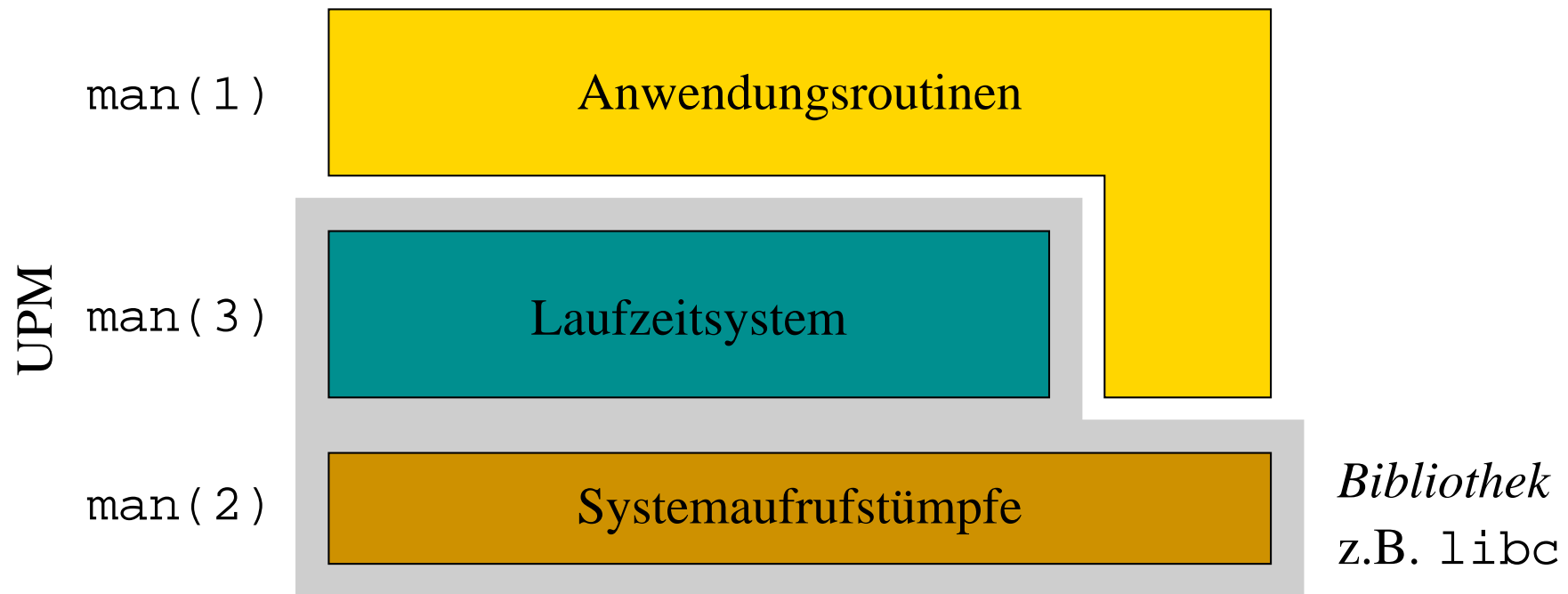
- ▶ bei UNIX z.B. die Bibliotheksfunktionen `write(2)` und `sbreak(2)`
- ▶ setzen synchrone Programmunterbrechungen (d.h. Traps) ab



bilden zusammengebunden ein **Anwendungsprogramm**

# Organisation von Maschinenprogrammen (Forts.)

Software(grob)struktur innerhalb eines Benutzeradressraums

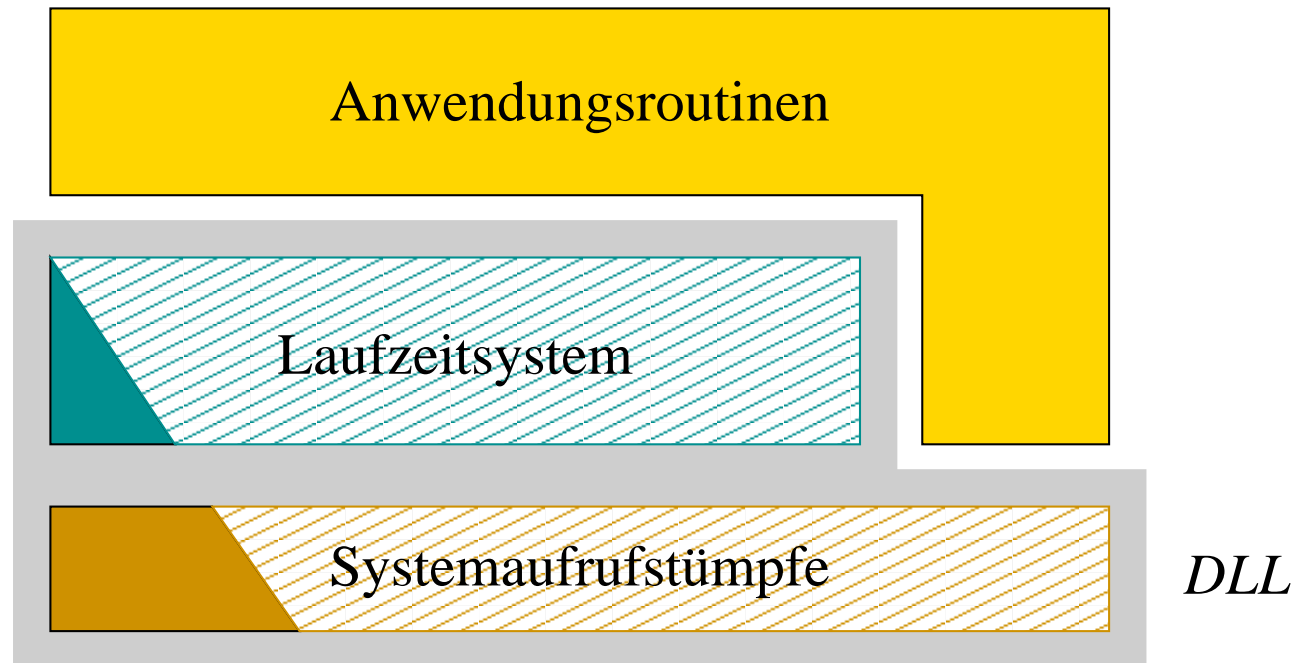


Modell für eine **statische Bibliothek** (`gcc` bzw. `ld -static ...`)

- ▶ 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 [3])
- ▶ 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>)

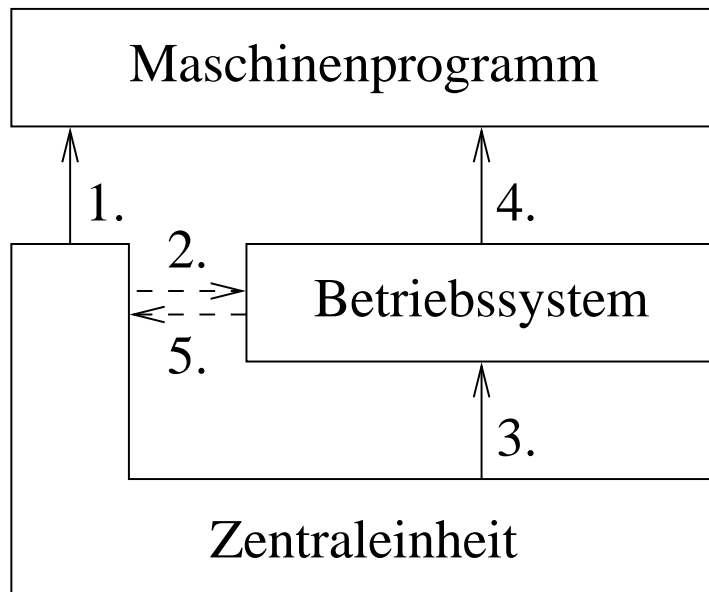
Ausführende Instanz ist immer die CPU, die nur Ebene<sub>2</sub>-Befehle kennt

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

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

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

Partielle Interpretation eines Maschinenprogramms durch das Betriebssystem



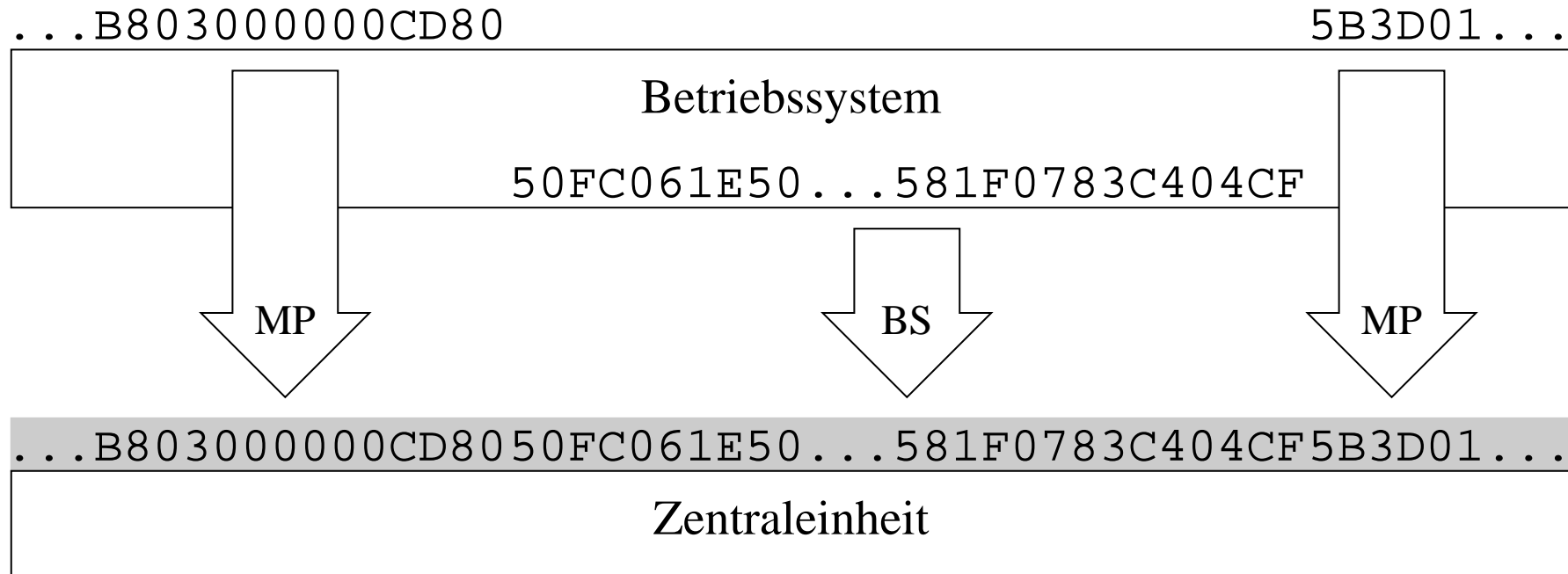
1. Die Zentraleinheit interpretiert das Maschinenprogramm befehlsweise,
2. setzt dessen Ausführung aus,
  - ▶ Ausnahmesituation
  - ▶ **Programmunterbrechung**
 startet das Betriebssystem und
3. interpretiert die Programme des Betriebssystems befehlsweise.

## Folge von 3., der Ausführung von Betriebssystemprogrammen...

4. Das Betriebssystem interpretiert das soeben oder zu einem früheren Zeitpunkt unterbrochene Maschinenprogramm befehlsweise und
5. instruiert die Zentraleinheit, die Ausführung des/eines zuvor unterbrochenen Maschinenprogramms wieder aufzunehmen.

# Zusammenspiel von Ebene<sub>2</sub> und Ebene<sub>3</sub> (Forts.)

Logischer Aufbau des Befehlsstroms für die Zentraleinheit



Eine bei Ausführung eines Maschinenprogramms vom **realen Prozessor** (Ebene<sub>2</sub>, Zentraleinheit) gestartete Behandlung einer Ausnahmesituation bewirkt — bildlich gesprochen — das „Einschieben“ von Befehlsfolgen jener Programme in den Befehlsstrom, die den **abstrakten Prozessor** „Betriebssystem“ (Ebene<sub>3</sub>) implementieren.

# 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**: Linux/x86), programmiert
- ▶ im Falle von Ausnahmesituationen, nicht programmiert

. . . und deaktivieren sich immer selbst, programmiert (**CF**: x86)

# 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

- ▶ vom jeweiligen realen *und* abstrakten Prozessor
- ▶ genauer: von **Zentraleinheit** und **Betriebssystem**

# Synchrone Programmunterbrechung

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

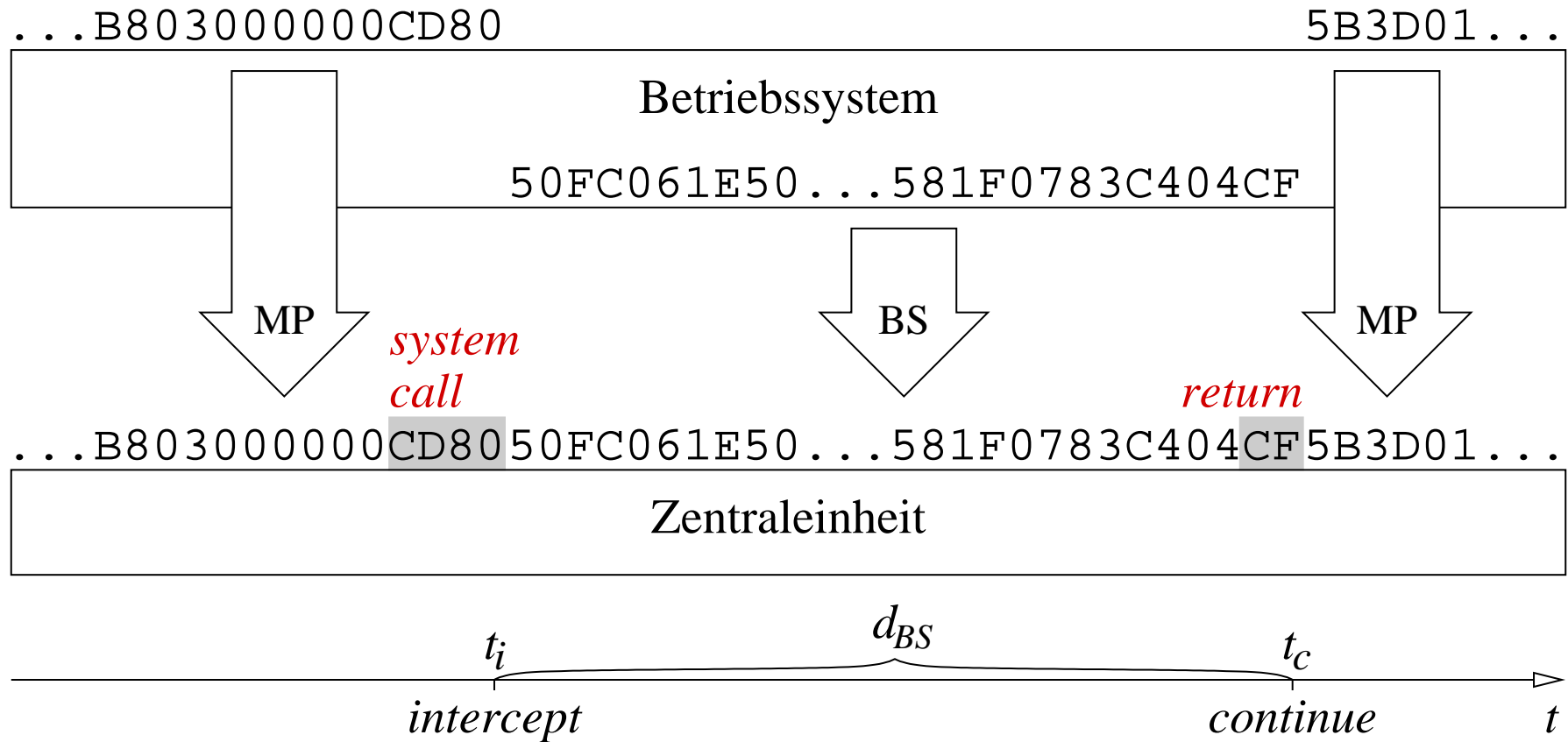
## Trap — synchron, vorhersagbar, reproduzierbar

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



Trapvermeidung ohne Behebung der Ausnahmebedingung unmöglich

# Synchrone Programmunterbrechung — *Trap*



Unterbrechungsverzögerung (engl. *trap latency*)

$d_{BS}$  Ebene 3; begrenzt bei Echtzeitfähigkeit, unbegrenzt sonst

# Asynchrone Programmunterbrechung

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

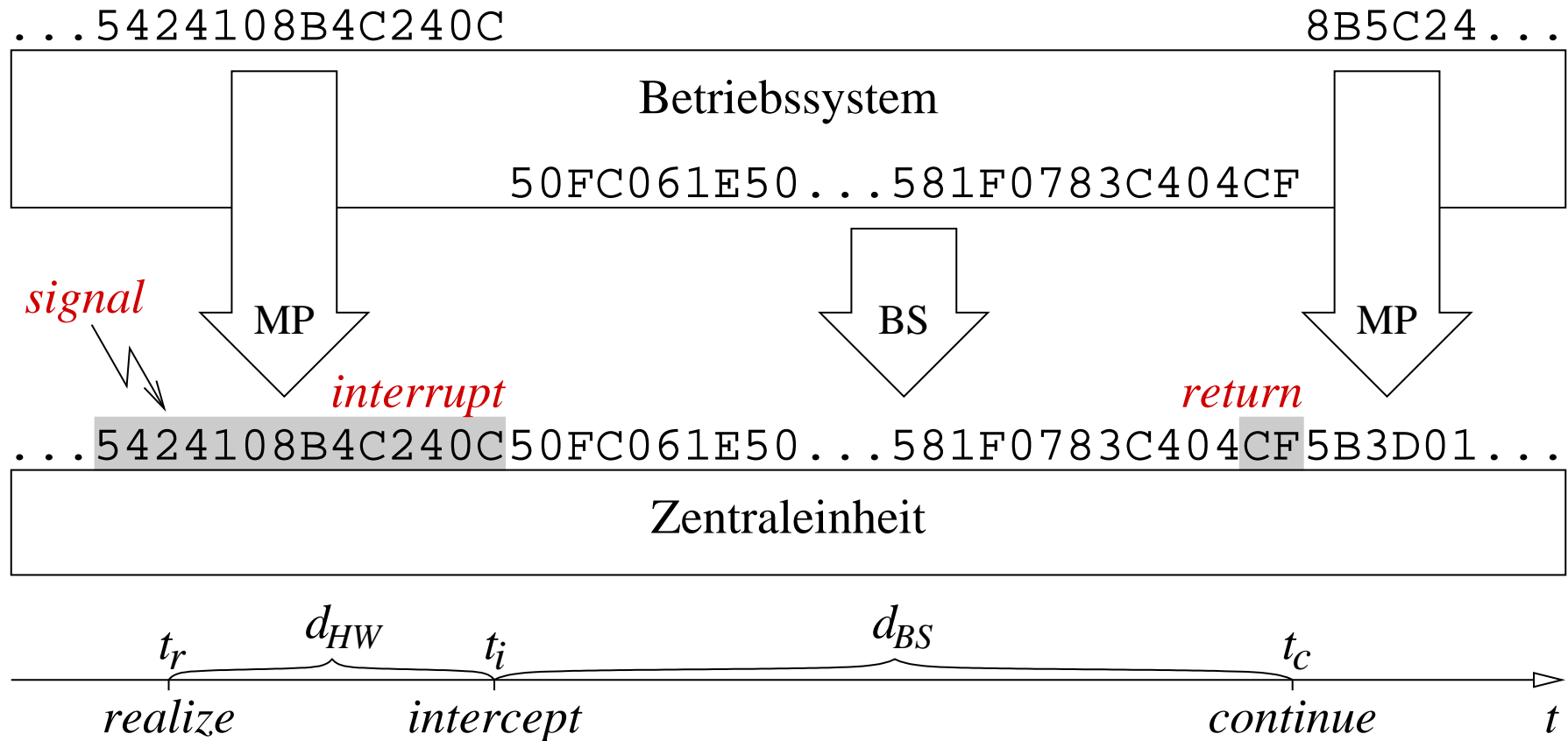
**Interrupt — asynchron, unvorhersagbar, nicht reproduzierbar**

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



# Asynchrone Programmunterbrechung — *Interrupt*



Unterbrechungsverzögerung (engl. *interrupt latency*)

$d_{HW}$  Ebene<sub>2</sub>; konstant bei RISC, variabel (begrenzt) bei CISC

$d_{BS}$  Ebene<sub>3</sub> ~ *Trap* (S. 5-38) ..... ⚡ **BS kann  $d_{HW}$  beeinträchtigen**

# Trap oder Interrupt?

```
#include <stdlib.h>

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

## Division (durch Null)

- ▶ 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 **↪ reproduzierbar** ..... Interrupt
  - ▶ je nach Anfangswert (*random seed*) und Güte des Zufallsgenerators

# 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: die **Ersetzungsstrategie** (engl. *replacement policy*)<sup>1</sup>

**lokal**  $\rightsquigarrow$  Stelle **vorhersagbar**, Unterbrechung **reproduzierbar**.....Trap

**global**  $\rightsquigarrow$  Stelle  $\neg$ **vorhersagbar**, Unterbrechung  $\neg$ **reproduzierbar**.....Int.

---

<sup>1</sup>Optionales Merkmal der Speicherverwaltung, vgl. Teil C, Kapitel 12.

# 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*, [4])

### 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/Kompilierer) 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 „Kompilierer“

```
gcc
```

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

`__attribute__((interrupt))`

- ▶ Generierung der speziellen Maschinenbefehle durch den **Kompilierer**
  - ▶ 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 weitergefahren 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
- ▶ nicht vergessen: Berücksichtigung 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

# Echtzeitbedingungen (Forts.)

Fest  $\longleftrightarrow$  Hart

fest/hart  $\mapsto$  Terminverletzung ist nicht ausgeschlossen<sup>2</sup>

- ▶ die Terminverletzung wird vom Betriebssystem erkannt

fest  $\rightsquigarrow$  plangemäß weiterarbeiten

- ▶ das Betriebssystem bricht den Arbeitsauftrag ab
- ▶ der nächste Arbeitsauftrag wird gestartet
- ▶ ist transparent für die Anwendung

hart  $\rightsquigarrow$  sicheren Zustand finden

- ▶ das Betriebssystem löst eine **Ausnahmesituation** aus
- ▶ die Ausnahmebehandlung führt zum sicheren Zustand
- ▶ ist **intransparent für die Anwendung**

---

<sup>2</sup>Auch wenn Ablaufplan und Betriebssystem auf dem Blatt Papier Determinismus zeigen, kann das im Feld eingesetzte technische System von Störeinflüssen betroffen sein, die ggf. die Verletzung auch eines harten Termins nach sich ziehen.

# Parallele und Funktionale Programmierung, 2. Semester

Ergänzung, Verfeinerung bzw. Vertiefung von PFP



- ▶ Teil I, 3. **Datensynchronisation** [5], speziell...
  - ✓ Wettlaufsituationen
  - ✓ gemeinsamer Zustand
  - ✓ kritische Abschnitte

... bei asynchronen Programmunterbrechungen, genauer:

- ▶ durch Abstraktion verborgene „Untiefen“ in Maschinenprogrammen
- ▶ überlappungsfreie Ausführung kritischer Abschnitte auf Ebene<sub>2</sub>
  - ▶ verdeutlicht durch Programme der Ebene<sub>4</sub> (Assemblersprache)

# Nichtdeterministisches Programm

## Asynchronität von Programmunterbrechungen

Welche wheel-Werte gibt main() aus?

```
unsigned int wheel = 0;
```

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

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

**normalerweise** fortlaufende Werte im Bereich<sup>3</sup>  $[0, 2^{32} - 1]$ , Schrittweite 1  
**ausnahmsweise** dito, allerdings mit Schrittweite  $n$ ,  $0 \leq n \leq 2^{32} - 1$

- ▶ jenachdem, wie oft die Ausführung von train() die von main() in einem bestimmten **kritischen Abschnitt** überlappt
- ▶  $n = 1$  impliziert nicht, dass keine Überlappung stattgefunden hat

---

<sup>3</sup>Annahme: `sizeof(unsigned int) = 4 Bytes je acht Bits, d.h. 32 Bits.`

# Nichtdeterministisches Programm (Forts.)

## Teilbarkeit von Operationen

### wheel++

- ▶ eine **Elementaroperation** (kurz: Elop) der Ebene<sub>5</sub>
  - ▶ der abstrakte C-Prozessor führt wheel++ **atomar**, d.h. **unteilbar** aus
- ▶ nicht zwingend auch eine Elop der Ebene<sub>4</sub> (und tiefer)
  - ▶ der reale x86-Prozessor führt wheel++ **teilbar** aus

	main()	train()	# Elop	
Ebene <sub>5</sub>	wheel++		1	unteilbar
Ebene <sub>4</sub>	movl wheel,%edx incl %edx movl %edx,wheel	movl wheel,%eax incl %eax movl %eax,wheel	3	<b>teilbar</b>

### Unterbrechungsfall

- ▶ zeitliche Überlappung einer Veränderung des wheel-Wertes
  - ▶ train()-Ausführung überlappt ggf. main()-Ausführung

# Unterbrechungsbedingte Überlappungseffekte

## Kritischer Programmtext

### Nebenläufiges Zählen

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

### Unterbrechungsfall

- ▶ zweimal wheel++ durchlaufen (main() und train())
- ▶ zweimal gezählt, den Wert von wheel aber nur um eins erhöht



# 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

`wheel++`

- ▶ **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  $\rightsquigarrow$  Virtualisierung

## Modularisieren

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

## Achtung...

Schrittweiten ungleich 1 bei der Ausgabe sind weiterhin möglich:

- ▶ Verdrängungslatenz
- ▶ Unterbrechungsfrequenz

## 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

- ▶ optional in Hardware oder Software implementierte Befehlssätze
  - ▶ Koprozessor (z.B. *floating-point unit*, FPU)
  - ▶ rekonfigurierbare Hardware (z.B. *field-programmable array* FPGA)
- ▶ Flexibilisierung der wirklichen Implementierungsebene
  - ▶ 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>*i*</sub>-Programm emuliert werden
- ▶ Ebene<sub>*i*</sub>-Programm kann durch Ebene<sub>*i*</sub>-Befehl implementiert werden

## Emulation $\mapsto$ Nachbildung

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

- ▶ Nachahmung der Eigenschaften eines ggf. anderen Rechnersystems
- ▶ bei **Selbstvirtualisierung** emuliert sich ein Rechnersystem selbst

### Nachbilder (engl. *emulator*)

Interpretiert die von einer 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 „alte“ Apple-Rechner. . .

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

# Selbstvirtualisierung

**Voraussetzung:** bei Ausführung des Maschinenprogramms „*trappt*“ die CPU *sensitive Befehle*<sup>4</sup> im „nicht-privilegierten Arbeitsmodus“

- ▶ beim x86 (Pentium) z.B. die Befehle [6]: `call`, `int`, `jmp`, `lar`, `lsl`, `mov`, `pop`, `popf`, `push`, `pushf`, `ret`, `sgdt/sidt`, `sldt`, `smsw`, `str`, `verr` und `verw`
- ▶ Problem: diese werden jedoch nicht wie gefordert abgefangen!
  - ▶ partielle Interpretation nicht möglich  $\leadsto$  **Paravirtualisierung**

## 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

---

<sup>4</sup>Befehle, deren direkte Ausführung durch die virtuelle Maschine nicht tolerierbar ist.

# Selbstvirtualisierung (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 BS)
  - ▶ 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

# Paravirtualisierung

Verzahnung von Betriebssystem und VMM  $\mapsto$  **alternative Ebene<sub>3</sub>**

- ▶ es wird nicht gefordert, auch Betriebssysteme unverändert auf einer virtuellen Maschine ablaufen zu lassen
- ▶ stattdessen: im BS spezielle **Virtualisierungsunterstützung** vorsehen
  - (a) um das Problem nicht-privilegierter sensibler Befehle zu lösen
  - (b) aber auch, um Virtualisierung zu „beschleunigen“

Präparation des Betriebssystems zum Zwecke der Ausführung auf einer durch einen *Hypervisor*<sup>a</sup> implementierten virtuellen Maschine

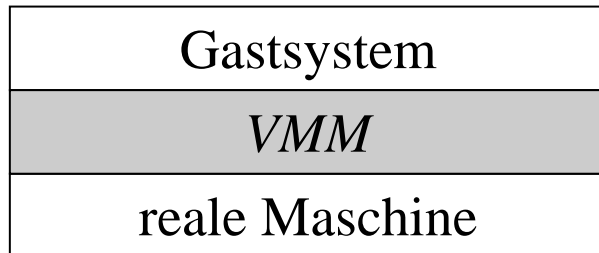
- ▶ das Betriebssystem läuft dem VMM untergeordnet auf der CPU ab
  - ▶ privilegierte Befehle werden weiterhin vom VMM abgefangen/emuliert
- ▶ der VMM liegt **intransparent** zwischen Betriebssystem und CPU
  - ▶ VMM kapselt sensitive Befehle in „komplexen“ Elementaroperationen
  - ▶ BS nutzt diese Operationen, nicht die „nackten“ CPU-Befehle

---

<sup>a</sup>Der Überlieferung nach, ein bei der Entwicklung von VM/370 geprägter Begriff zur Abgrenzung vom auch als *Supervisor* bezeichneten Betriebssystem.

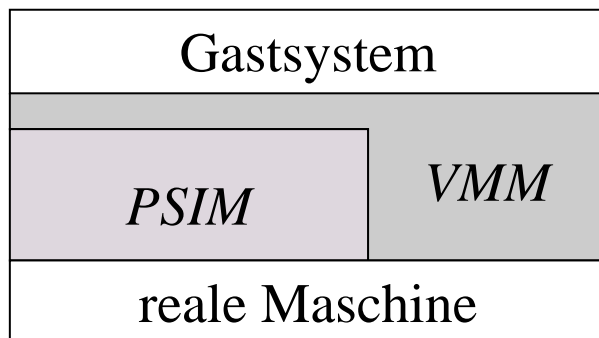
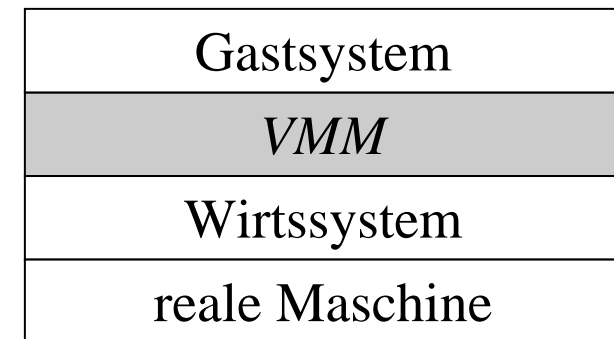


# Ausprägungen von Virtualisierungssystemen [7, 8]



**Typ 1 VMM** privilegierte Befehle werden vom VMM abgefangen

**Typ 2 VMM** privilegierte Befehle werden vom Wirtssystem abgefangen und an den VMM hochgereicht

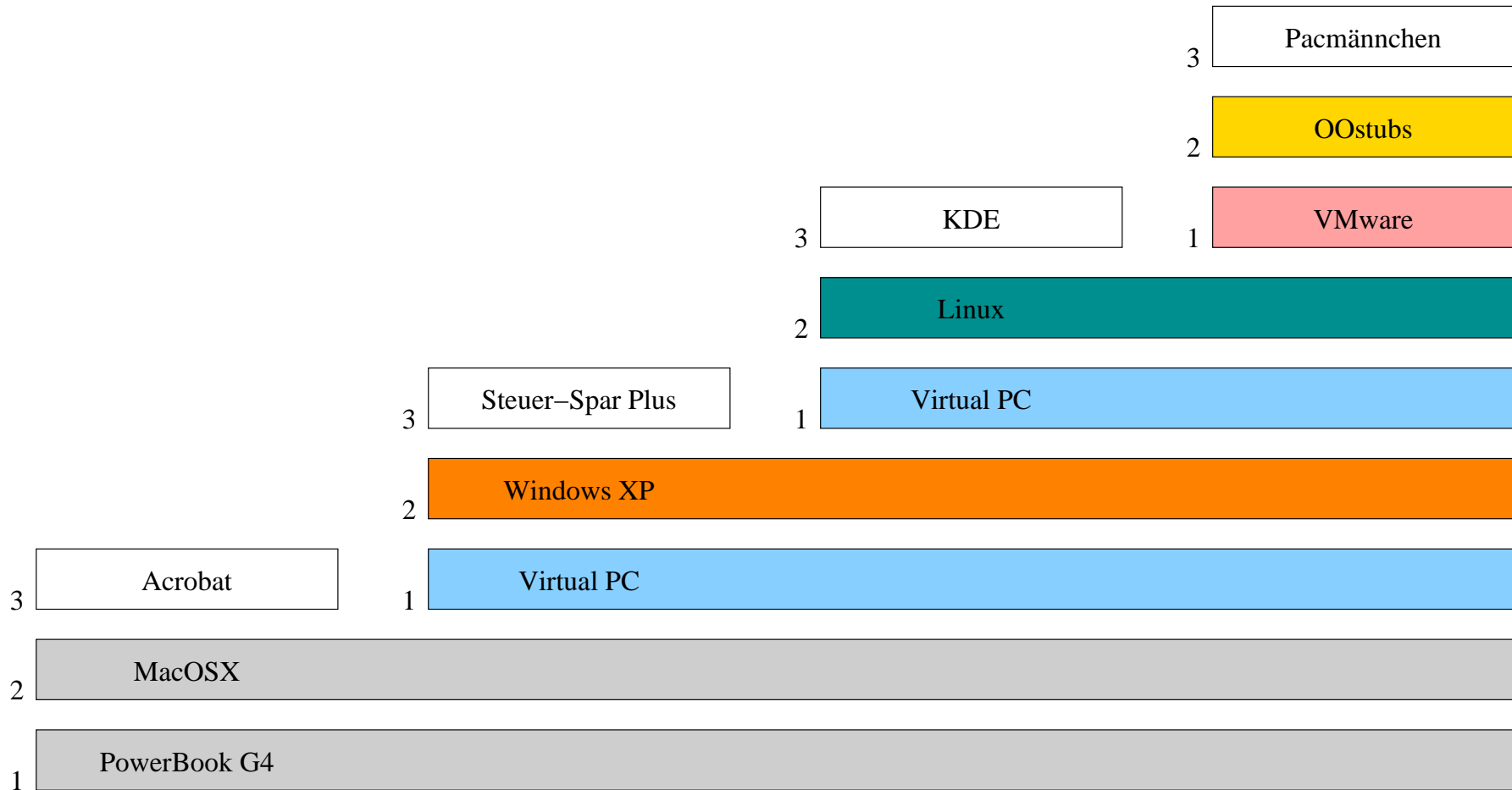


**hybrider VMM (HVM)** privilegierte Befehle werden vom VMM abgefangen, laufen jedoch auf einer partiell in Software implementierten Maschine (PSIM) ab

## Hypervisor

- ▶ Typ 1 oder 2 VMM, je nach Geschmack: **ungenau** Verwendung...

# 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 (einzeln 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  $\mapsto$  Windows XP

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

# Rechnerorganisation

## Strukturierte Organisation von Rechenystemen

### 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 der Eigenschaften von (abstrakten) Prozessoren

# Literaturverzeichnis

- [1] <http://www.hyperdictionary.com/computing/semantic+gap>.
- [2] Andrew Stuart Tanenbaum.  
*Structured Computer Organization*.  
Prentice-Hall, Inc., fourth edition, 1999.
- [3] Elliot I. Organick.  
*The Multics System: An Examination of its Structure*.  
MIT Press, 1972.
- [4] John Bannister Goodenough.  
Exception handling: Issues and a proposed notation.  
*Communications of the ACM*, 18(12):683–696, 1975.

## Literaturverzeichnis (Forts.)

[5] Michael Philippsen.

Parallele und funktionale programmierung.

<http://www2.cs.fau.de/Lehre/SS2008/PFP/material/>, 2008.

Lecture Notes.

[6] John Scott Robin and Cynthia E. Irvine.

Analysis of the Intel Pentium's ability to support a secure virtual machine monitor.

In *Proceedings of the 9th USENIX Security Symposium*, pages 10–10, Denver, CO, USA, August 14–17, 2000. USENIX Association.

[7] Robert P. Goldberg.

*Architectural Principles for Virtual Computer Systems.*

PhD thesis, Division of Engineering and Applied Physics, Harvard University, Cambridge, MA, USA, February 1973.

# Literaturverzeichnis (Forts.)

- [8] Gerald J. Popek and Robert P. Goldberg.  
Formal requirements for virtualizable third generation architectures.  
*Communications of the ACM*, 17(7):412–421, 1974.