Systemprogrammierung

Rechnerorganisation: Schichtenstruktur

Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

14. Mai 2014

Gliederung

Verschiedenheit zwischen Quell- und Zielsprache

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

Semantische Lücke (engl. *semantic gap*, [6])

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

Beispiel: Matrizenmultiplikation

Problemraum

(Mathematik)

Lösungsraum

(Informatik)

- „gedanklich gemeint“ ist ein Verfahren aus der linearen Algebra
- „sprachlich geäußert“ auf verschiedenen Ebenen der **Abstraktion**

Ebene mathematischer Sprache: Lineare Algebra

Multiplikation von zwei 2×2 Matrizen:

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

Zwei Matrizen werden multipliziert, indem die Produktsummenformel auf Paare aus einem Zeilenvektor der ersten und einem Spaltenvektor der zweiten Matrix angewandt wird.

Produktsummenformel für $C = A \times B$:

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

Ebene informatischer Sprache: C

Skalarprodukt oder „inneres Produkt“

```
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];
        }
}
```

Konkretisierung der Multiplikation von zwei $N \times N$ Matrizen: $\mathtt{c} = \mathtt{a} \times \mathtt{b}$

- ausgelegt als Unterprogramm: Prozedur $\mapsto$ *C function*
- `vi multiply.c`: Quellmodul erstellen

Ebene informatischer Sprache: ASM [3, 2]

```
  .file "multiply.c"
  .text
  .p2align 4,,15
  .globl multiply
  .type multiply,@function
multiply:
  pushl %ebp
  movl  %esp,%ebp
  pushl %edi
  pushl %esi
  pushl %ebx
  subl  $4,%esp
  movl  16(%ebp),%esi
  movl  $0,-16(%ebp)
.L2:
  movl  8(%ebp),%edi
  xorl  %ebx,%ebx
```

```
  addl  -16(%ebp),%edi
  .p2align 4,,7
  .p2align 3
.L4:
  movl  12(%ebp),%eax
  xorl  %edx,%edx
  movl  $0,(%esi,%ebx,4)
  leal  (%eax,%ebx,4),%ecx
  .p2align 4,,7
  .p2align 3
.L3:
  movl  (%ecx),%eax
  addl  $400,%ecx
  imull (%edi,%edx,4),%eax
  addl  $1,%edx
  addl  %eax,(%esi,%ebx,4)
  cmpl  $100,%edx
```

```
jne   .L3
addl  $1,%ebx
cmpl  $100,%ebx
jne   .L4
addl  $400,-16(%ebp)
addl  $400,%esi
cmpl  $40000,-16(%ebp)
jne   .L2
addl  $4,%esp
popl  %ebx
popl  %esi
popl  %edi
popl  %ebp
ret
.size multiply, .-multiply
.ident "GCC: (Debian 4.3.2-1.1) 4.3.2"
.section .note.GNU-stack,"",@progbits
```

Kompilierung der Quelle in ein semantisch äquivalentes Programm

- `gcc -O6 -S -DN=100 multiply.c`: *C function* $\mapsto$ ASM/x86
- Schalter `-S`: Übersetzung der Quelle vor der Assemblierung beenden

Ebene informatischer Sprache: „Unix-Dialekt“ `a.out` [3, 1]

```
0000000 457f 464c 0101 0001 0000 0000 0000 0000
0000020 0001 0003 0001 0000 0000 0000 0000 0000
0000040 0114 0000 0000 0000 0034 0000 0000 0028
0000060 0009 0006 0000 0000 0000 0000 0000 0000
0000100 8955 57e5 5356 ec83 8b04 1075 45c7 00f0
0000120 0000 8b00 087d db31 7d03 90f0 748d 0026
0000140 458b 310c c7d2 9e04 0000 0000 0c8d 9098
0000160 018b c181 0190 0000 af0f 9704 c283 0101
0000200 9e04 fa83 7564 83e9 01c3 fb83 7564 81d1
0000220 f045 0190 0000 c681 0190 0000 7d81 40f0
0000240 009c 7500 83ae 04c4 5e5b 5d5f 00c3 0000
0000260 4700 4343 203a 4428 6265 6169 206e 2e34
0000300 2e33 2d32 2e31 2931 3420 332e 322e 0000
0000320 732e 6d79 6174 0062 732e 7274 6174 0062
0000340 732e 7368 7274 6174 0062 742e 7865 0074
0000360 642e 7461 0061 622e 7373 2e00 6f63 6d6d
0000400 6e65 0074 6e2e 746f 2e65 4e47 2d55 7473
0000420 6361 006b 0000 0000 0000 0000 0000 0000
0000440 0000 0000 0000 0000 0000 0000 0000 0000
0000460 0000 0000 0000 0000 0000 0000 001b 0000
0000500 0001 0000 0006 0000 0000 0000 0040 0000
0000520 006d 0000 0000 0000 0000 0000 0010 0000
0000540 0000 0000 0021 0000 0001 0000 0003 0000
0000560 0000 0000 00b0 0000 0000 0000 0000 0000
0000600 0000 0000 0004 0000 0000 0000 0027 0000
0000620 0008 0000 0003 0000 0000 0000 00b0 0000
0000640 0000 0000 0000 0000 0000 0000 0004 0000
0000660 0000 0000 002c 0000 0001 0000 0000 0000
0000700 0000 0000 00b0 0000 001f 0000 0000 0000
0000720 0000 0000 0001 0000 0000 0000 0035 0000
0000740 0001 0000 0000 0000 0000 0000 00cf 0000
0000760 0000 0000 0000 0000 0000 0000 0001 0000
0001000 0000 0000 0011 0000 0003 0000 0000 0000
0001020 0000 0000 00cf 0000 0045 0000 0000 0000
0001040 0000 0000 0001 0000 0000 0000 0001 0000
0001060 0002 0000 0000 0000 0000 0000 027c 0000
0001100 0080 0000 0008 0000 0007 0000 0004 0000
0001120 0010 0000 0009 0000 0003 0000 0000 0000
0001140 0000 0000 02fc 0000 0015 0000 0000 0000
0001160 0000 0000 0001 0000 0000 0000 0000 0000
0001200 0000 0000 0000 0000 0000 0000 0001 0000
0001220 0000 0000 0000 0000 0004 fff1 0000 0000
0001240 0000 0000 0000 0000 0003 0001 0000 0000
0001260 0000 0000 0000 0000 0003 0002 0000 0000
0001300 0000 0000 0000 0000 0003 0003 0000 0000
0001320 0000 0000 0000 0000 0003 0005 0000 0000
0001340 0000 0000 0000 0000 0003 0004 000c 0000
0001360 0000 0000 006d 0000 0012 0001 6d00 6c75
0001400 6974 6c70 2e79 0063 756d 746c 7069 796c
0001420 0000
```

Assemblierung der kompilierten Quelle und Ausgabeaufbereitung

1. `as multiply.s`: ASM/x86 $\mapsto$ `a.out`/x86 (Binde-/Lademodul)
2. `od -x a.out` $\rightsquigarrow$ auf x86-Prozessoren ausführbarer Binärkode

Verschiedenheit zwischen Quell- und Zielsprache (Forts.)

Ebene der **Modellsprache** „Lineare Algebra“ $\leadsto$ 1 Produktsummenformel

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

Ebene der **Programmiersprache** C $\leadsto$ 5 Komplexschritte

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

Ebene der **Assembliersprache** ASM $\leadsto$ $35+n$ Elementarschritte

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

Ebene der **Maschinensprache** x86 $\leadsto$ 97 Bytes Programmtext

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

Zwischenzusammenfassung

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

Die Diskrepanz zwischen der vom Menschen skizzierten Lösung des Problems und dem dazu korrespondierenden, von einem Prozessor ausführbaren Maschinenprogramm ist beträchtlich.

Abstraktion half, sich auf das Wesentliche konzentrieren zu können

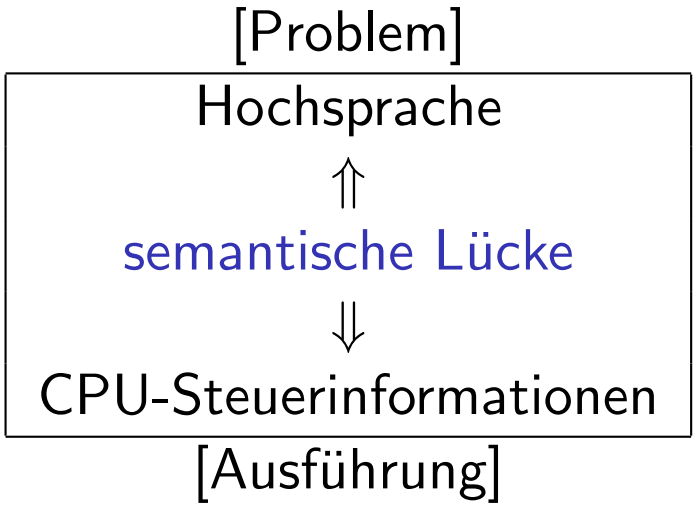
- eine virtuelle Maschine zur Matrizenmultiplikation entstand
- die schrittweise abgebildet wurde auf die reale Maschine „x86“

Umwandlung ins ausführbare Programm meint viele Arbeitsgänge:

1. Vorverarbeitung des Quellmoduls `cpp(1)`
2. Kompilierung der vorverarbeiteten Quelle `gcc(1)`
3. Assemblierung des (zwischenzeitlich) erzeugten Aggregats `as(1)`
4. Bindung assemblierter Objektmodule zum Lademodul `ld(1)`

Gliederung

Aufgabenstellung $\mapsto$ Programmlösung



Breite der semantischen 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

Semantische Lücke schrittweise schließen

- durch hierachisch angeordnete virtuelle Maschinen Programmlösungen auf die reale Maschine abbilden [4]
- Prinzip *divide et impera* („teile und herrsche“): einen „Gegner“ in leichter besiegbare „Untergruppen“ aufspalten

Hierarchie virtueller Maschinen [5, S. 3]

Interpretation und **Übersetzung** (Kompilierung, Assemblierung):

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 sorgen für die Maschinenabbildung

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

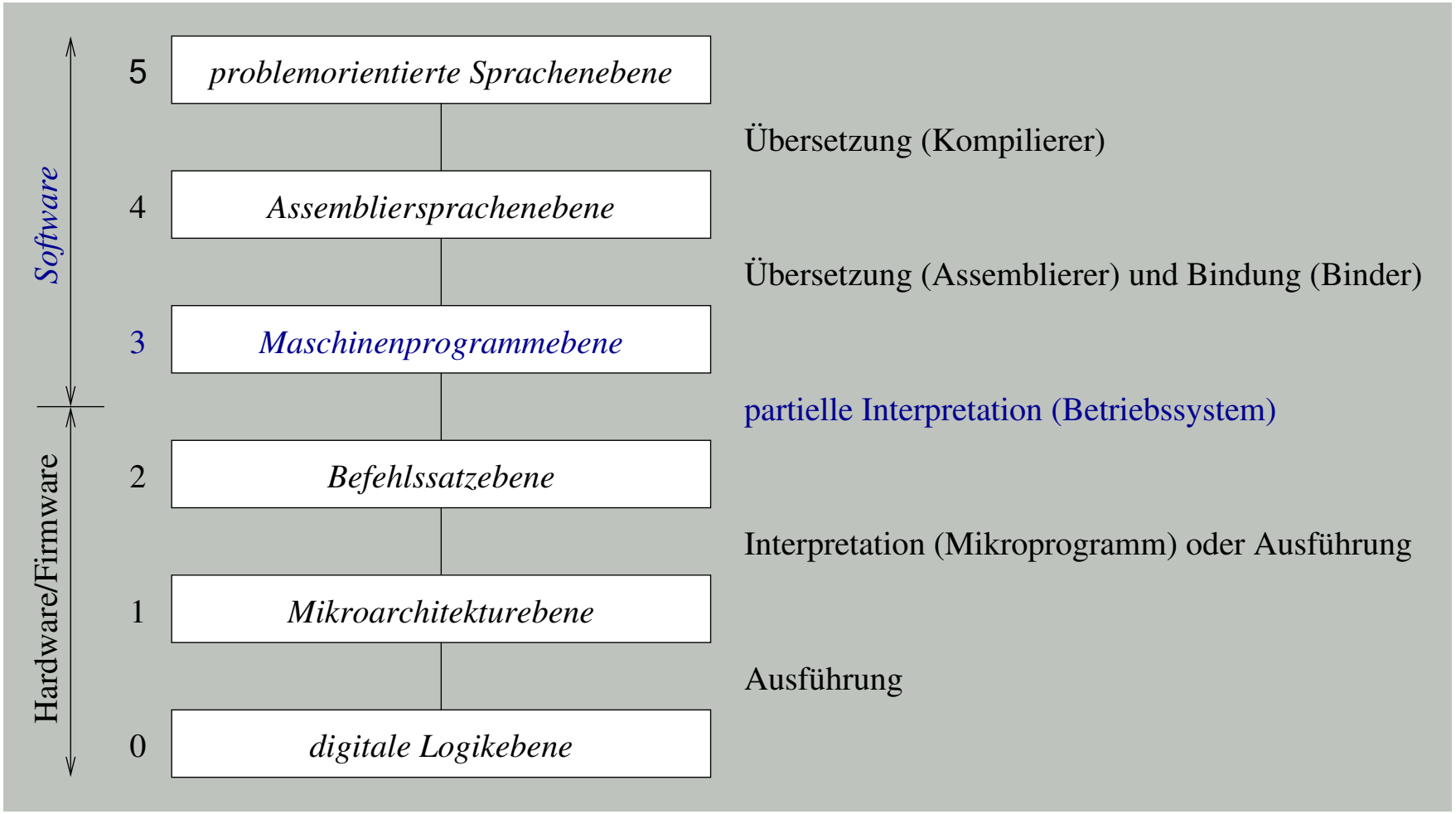
Kom|pi|la|tor *lat.* (Zusammenträger)

- ein **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 von Rechensystemen [4]



Softwaremaschine: Kompilierer/Assemblierer (Binder)

problemorientierte Programmiersprachenebene [AuD/PFP]

- bietet „höhere Programmiersprachen“ zur abstrakten, problemorientierten Formulierung von Programmlösungen
- 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*) [GRA]

- bietet „niedere Programmiersprachen“ zur konkreten, CPU-spezifischen Formulierung von Programmlösungen
- Programme bestehen aus Pseudobefehle, mnemonisch ausgelegte Maschinenbefehle (ISA), symbolisch bezeichnete Operanden (Speicheradressen, Register) und Adressierungsarten

Softwaremaschine: Teilinterpretierer $\mapsto$ **Betriebssystem**

Maschinenprogrammebene (*binärer Maschinenkode*) [SP]

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

Betriebssysteme „ko-implementieren“ Maschinenprogrammebenen

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

Firm-/Hardwaremaschine: Interpretierer

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

- 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)

Hardwaremaschine: Ausführende

digitale Logikebene (*Boolsche Algebra*) [GST/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 Benutzerfreundlichkeit
- maximale Distanz von sehr vielen Problemdomänen

Abstraktionsniveau vs. Semantische Lücke

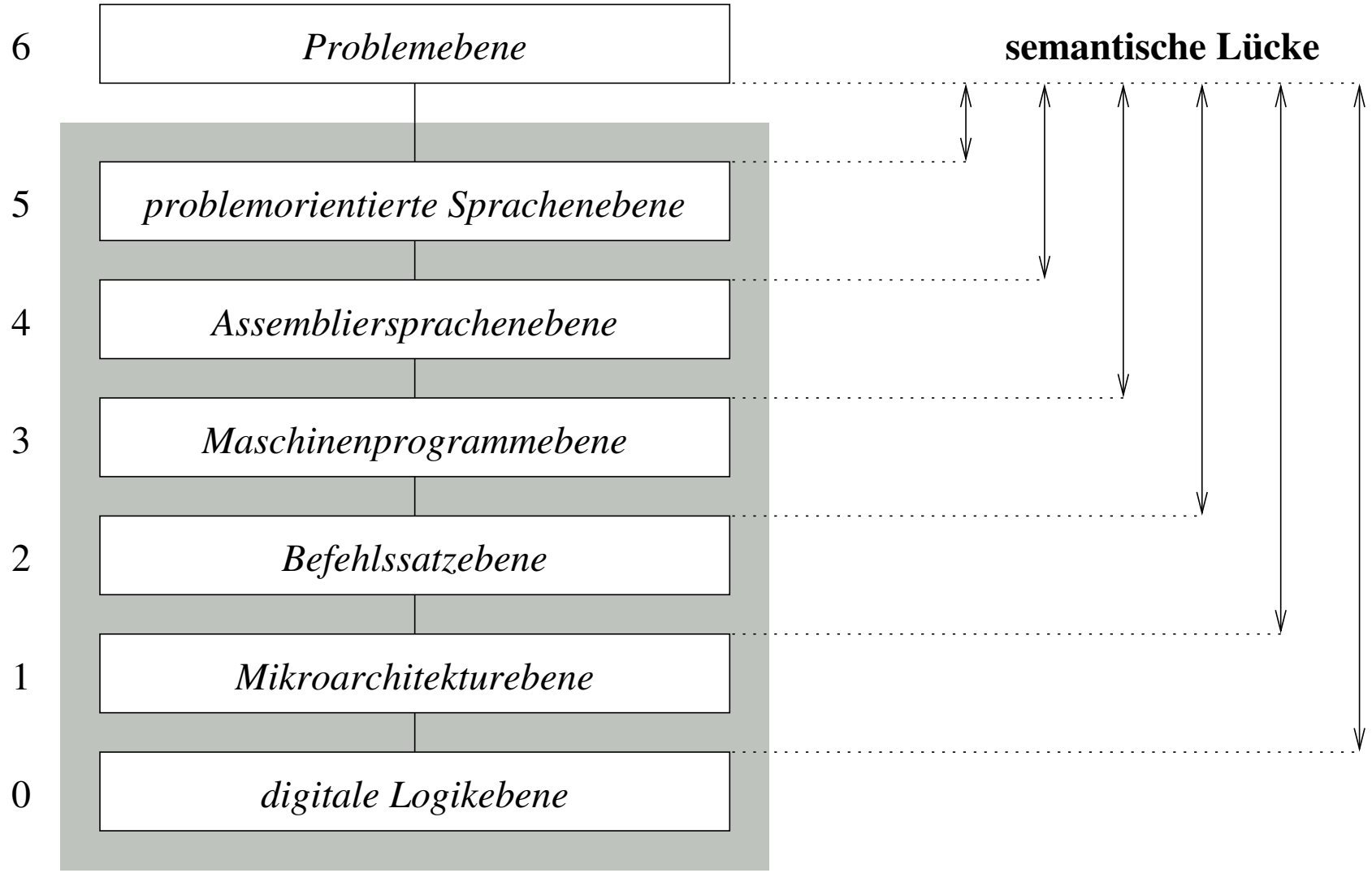


Abbildung durch Übersetzung

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“) auflösen
  - in binären Maschinenkode umwandeln

Abbildung durch Interpretation

Ebene $_3 \mapsto$ Ebene $_2$ (Teilinterpretation, auch *partielle Interpretation*)

- Ebene $_3$-Befehle typ- und zustandsabhängig verarbeiten:
  - (a) als Folgen von Ebene $_2$-Befehlen ausführen
    - Systemaufrufe annehmen und befolgen
    - (synchrone/asynchrone) Programmunterbrechungen behandeln
    - sensitive Ebene $_2$-Befehle emulieren
  - (b) „1:1“ auf Ebene $_2$-Befehle abbilden (nach unten „durchreichen“)
- ein Ebene $_3$-Befehl aktiviert ggf. ein Ebene $_2$-Programm

Ebene $_2 \mapsto$ Ebene $_1$ (Interpretation)

- Ebene $_2$-Befehle als Folgen von Ebene $_1$-Befehlen ausführen
  - Abruf- und Ausführungszyklus (engl. *fetch-execute-cycle*) der CPU
- ein Ebene $_2$-Befehl löst Ebene $_1$-Steueranweisungen aus

Übersetzung und Interpretation durch Prozessoren

Ebene $_5$ $\leadsto$ **Kompilierer**

- Interpretation von Konstrukten/Anweisungen einer „Hochsprache“

Ebene $_4$ $\leadsto$ **Assemblierer** und **Binder**

- Interpretation von Anweisungen einer Assembliersprache

Ebene $_3$ $\leadsto$ **Betriebssystem**

- Interpretation von Systemaufrufen und sensitiven Ebene $_2$-Befehlen
  - 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

Zeitpunkte der Abbildungsvorgänge

Bezogen auf das jeweils zu interpretierende/übersetzende Programm

vor Laufzeit $(\text{Ebene}_5 \mapsto \text{Ebene}_3) \rightsquigarrow$ statisch

- Vorverarbeitung (engl. *preprocessing*)
- Vorübersetzung (engl. *precompilation*)
- Übersetzung: Kompilierung, Assemblierung
- Binden (engl. *static linking*)

zur Laufzeit $(\text{Ebene}_5 \mapsto \text{Ebene}_1) \rightsquigarrow$ dynamisch

- bedarfsorientierte Übersetzung (engl. *just in time compilation*)
- Binden (engl. *dynamic linking*)
- bindendes Laden (engl. *linking loading*, *dynamic loading*)
- Teilinterpretation
- Interpretation

Betriebssysteme entvirtualisieren zur Laufzeit

- dynamisches Binden, bindendes Laden, Teilinterpretation

Gliederung

Resümee

Strukturierte Organisation von Rechensystemen

- semantische Lücke
  - „Kluft zwischen gedanklich Gemeintem und sprachlich Geäußertem“
  - schrittweise schließen: kleine Schritte zum großen Ziel...
- Hierarchie virtueller (auch: abstrakter) Maschinen
  - Ebene $_i \mapsto$ Ebene $_{i-1}$ durch Programme, für $i > 1$
  - Maschinenprogrammebene (Ebene $_3$) $\mapsto$ Betriebssystem
- Entvirtualisierung
  - gedachte, unwirkliche, scheinbare Maschinen konkretisieren
  - vor oder zur Programmlaufzeit stattfindende Arbeitsgänge
    - Übersetzung und Interpretation in den verschiedensten Facetten

Literaturverzeichnis

[1] Chamberlain, S. ; Taylor, I. L.:
Using `ld`: The GNU Linker.
Boston, MA, USA: Free Software Foundation, Inc., 2003

[2] Elsner, D. ; Fenlason, J. :
Using `as`: The GNU Assembler.
Boston, MA, USA: Free Software Foundation, Inc., Jan. 1994

[3] Ritchie, D. M.:
/ You are not expected to understand this. */*.
`http://cm.bell-labs.com/cm/cs/who/dmr/odd.html`, 1975

[4] Tanenbaum, A. S.:
Multilevel Machines.
In: *S*tructured Computer Organization[5], Kapitel 7, S. 344–386

[5] Tanenbaum, A. S.:
Structured Computer Organization.
Prentice-Hall, Inc., 1979. –
443 S. –
ISBN 0–130–95990–1

[6] *http://www.hyperdictionary.com/computing/semantic+gap*