Speicher

- die Abstraktion ”Speicher“ kommt mit zwei fundamentalen Ausprägungen:

  Vordergrundspeicher auch Hauptspeicher (RAM)

  - der entsprechend bestückte (RAM-) Bereich im physikalischen Adressraum
  - in ihm erfolgt die Ausführung der Programme (”von Neumann Rechner“)
  - kann größer sein als der physikalische Adressraum (bank switching)

  Hintergrundspeicher auch Massenspeicher (Band, Platte, CD, DVD)

  - die über die Rechnerperipherie (E/A-Geräte) angeschlossenen Bereiche
  - dient der Datenablage und Implementierung virtueller Adressräume
  - ist größer als der physikalische Adressraum: Petabytes (250 bzw. 1015)
- die Differenzierung kann ”nach aussen“ unsichtbar sein (Multics)

Speicherverwaltung

- die Verwaltung von Hauptspeicher erfolgt typischerweise auf zwei Ebenen:
  - das **Laufzeitsystem** bzw. die Bibliotheksebene verwaltet den Speicher (lokal) innerhalb eines logischen/virtuellen Adressraums
    - Speicherblöcke können von sehr feinkörniger Struktur/Größe sein
    - Bedürfnisse von Programmiersprachen bestimmen die Verfahrensweisen
  - das **Betriebssystem** verwaltet den im physikalischen Adressraum (global) vorrätigen Speicher
    - Speicherblöcke sind üblicherweise von grobkörniger Struktur/Größe
    - Arten des Rechnerbetriebs üben starken Einfluss auf Verfahrensweisen aus
- *”separation of concerns“* — beide Ebenen/Systeme ergänzen sich

Ein Prozess...

- Habermann, Introduction to Operating System Design
  . . . wird durch ein Programm kontrolliert und benötigt zur Ausführung dieses Programms einen Prozessor.
- Horning/Randell, Process Structuring
  . . . *P* ist ein Tripel *(S, f, s)*, wobei *S* einen Zustandsraum, *f* eine Aktionsfunktion und $s \subset S$ die Anfangszustände des Prozesses *P* bezeichnen. Ein Prozess erzeugt Abläufe, die durch die Aktionsfunktion generiert werden können.
- *Dennis/van Horn*, Programming Semantics for Multiprogrammed Computations
  . . . ist das Aktivitätszentrum innerhalb einer Folge von Elementaroperationen. Damit wird ein Prozess zu einer abstrakten Einheit, die sich durch die Instruktionen eines abstrakten Programms bewegt, wenn dieses auf einem Rechner ausgeführt wird.
- unbekannte Referenz, „Mundart“
  . . . ist ein Programm in Ausführung.

Prozess ≠ Programm (1)

- ein Prozess kann die Ausführung mehrerer Programme zur Folge haben:
  - **Systemaufruf** *Anwendungsprogramm* ruft ein *Betriebssystemprogramm* auf ⇒ der Prozess ist der Aktivitätsträger von mindestens zwei Programmen
- umgekehrt kann ein Programm von mehreren Prozessen ausgeführt werden

  nicht-sequentielles Programm im Falle von Uniprozessorsystemen

  ⇒ ”präemptive Programmverarbeitung“

  ⇒ Faden (thread), Unterbrechung (interrupt)

  paralleles Programm im Falle von Multiprozessorsystemen

Prozess ≠ Programm (2)

- Wissen über das gegenwärtig ausgeführte Programm sagt nicht viel aus über die zu dem Zeitpunkt im System stattfindende Aktivität.
  - Welche bzw. wieviel Fäden führen das Programm zur Zeit aus?
  - Welche bzw. wieviel Programmunterbrechungen sind zur Zeit aktiv?

⇒ Im Betriebssystemkontext ist das Konzept ”Prozess“ daher nützlicher als das Konzept ”Programm“, um Vorgänge zu beschreiben und zu verwalten.

Prozess ≠ Prozessinkarnation

- Prozess ist ein **abstraktes** Gebilde
  - ein ”Programm in Ausführung“, ein asynchroner Programmablauf
  - ein ”Ablauf“, der eine Verwaltungseinheit ist
- Prozessinkarnation ist ein **konkretes** Gebilde
  - die ”physische Instanz“ des abstrakten Gebildes ”Prozess“
  - gebunden an (Software-) Betriebsmittel, verbunden mit einer Identität
  - die Verwaltungseinheit, die einen Prozess beschreibt und repräsentiert
- im weitesten Sinne synonyme Begriffe, jedoch nicht in allen Fällen

Prozessmodelle

- schwergewichtiger Prozess (*heavyweight process*, ”klassischer“ UNIX Prozess)
  - Prozessinkarnation und Benutzeradressraum bilden eine Einheit
  - Prozesswechsel bedeutet zwei Adressraumwechsel ARx $\Rightarrow$ BS $\Rightarrow$ ARy
- leichtgewichtiger Prozess (*lightweight process*, kernel-level thread)
  - Prozessinkarnation und Adressraum sind voneinander entkoppelt
  - Prozesswechsel bedeutet einen Adressraumwechsel ARx $\Rightarrow$ BS $\Rightarrow$ ARx
- federgewichtiger Prozess (*featherweight process*, user-level thread)
  - Prozessinkarnationen und Adressraum bilden eine Einheit
  - Prozesswechsel entspricht einem Koroutinenwechsel

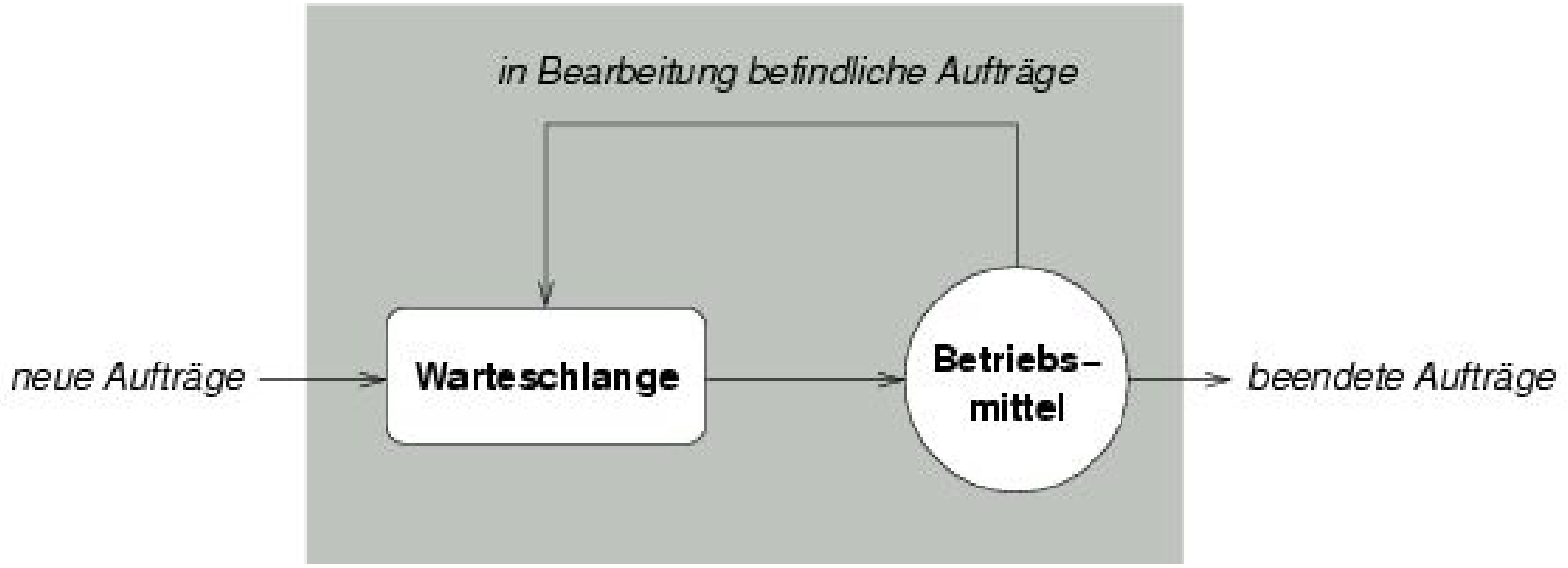
Einplanung

- Auch ”Ablaufplanung“, ist erforderlich, um die um den Prozessor (allgemein: die Betriebsmittel) konkurrierenden Prozesse geordnet ablaufen lassen zu können.
- Scheduling stellt sich allgemein zwei grundsätzlichen Fragestellungen:
  1. Zu welchem Zeitpunkt sollen Prozesse ins System eingespeist werden?
  2. In welcher Reihenfolge sollen Prozesse ablaufen?
- Ein Scheduling-Algorithmus verfolgt das Ziel, den von einem Rechnersystem zu leistenden Arbeitsplan so aufzustellen (und zu aktualisieren), dass ein gewisses Maß an Benutzerzufriedenheit maximiert wird.

Ablaufplan – Process Schedule

- Fahrplan zur Belegung der {Hard,Soft}ware-Betriebsmittel durch Prozesse
  - geordnet nach Ankunft, Zeit, Termin, Dringlichkeit, Gewicht, . . .
  - die Ordnung ist eine Funktion der Scheduling-Strategie (bzw. -Algorithmen)
- technisch (zumeist) realisiert auf Basis dynamischer Datenstrukturen
  - eine oder mehrere Queues bzw. Schlangen: Warteschlangen
  - die Elemente der Datenstruktur sind die Prozessdeskriptoren
- die gewählte Scheduling-Strategie bestimmt u. a. die Rechnerbetriebsart

Scheduling



- Ein einzelner Scheduling-Algorithmus charakterisiert sich durch die Reihenfolge von Prozessen in der Warteschlange und die Bedingungen, unter denen die Prozesse der Warteschlange zugeführt werden.

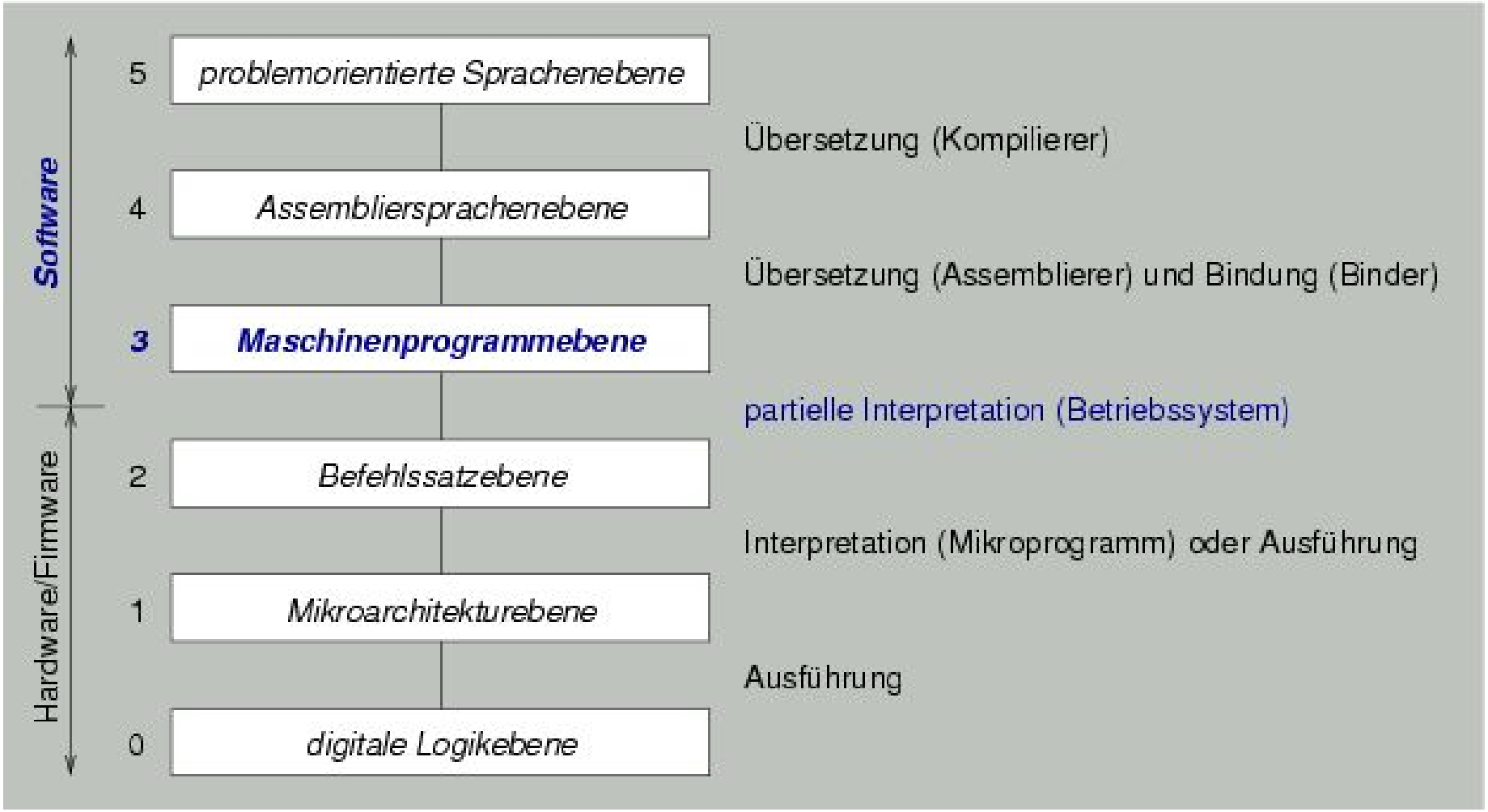
Zusammenfassung

- Betriebssysteme bieten eine ”Hand voll“ nützlicher Abstraktionen an
  - Adressräume, Speicher, Dateien, Prozesse
- von zentraler Bedeutung ist die Abbildung von Namen auf Adressen
  - symbolische $\Rightarrow$ numerische $\Rightarrow$ logische $\Rightarrow$ virtuelle $\Rightarrow$ physikalische Adresse
  - ein Konzept, das im Kontext von Rechnernetzen seine Fortführung findet
- je nach Betriebssystemart sind die Abstraktionen unterschiedlich ausgelegt
  - manche Abstraktionen müssen auch überhaupt nicht angeboten werden

Schichtenstrukturen

- Sprachen, Ebenen, virtuelle Maschinen
  - die ”semantische Lücke“ erkennen und schließen bzw. überwinden
  - Übersetzung, Interpretation, partielle Interpretation
  - funktionale -, Modul-, Benutzthierarchie
- Hardware, Software, Mehrebenenmaschinen
  - digitale Logikebene, Mikroarchitekturebene, Befehlssatzebene
  - Maschinenprogrammebene + Betriebssystemebene
  - Assembliersprachenebene, problemorientierte Sprachenebene
- Betriebssystem verstehen als virtuelle Maschine einer bestimmten Ebene

Virtuelle Maschinen



Abbildende Programme

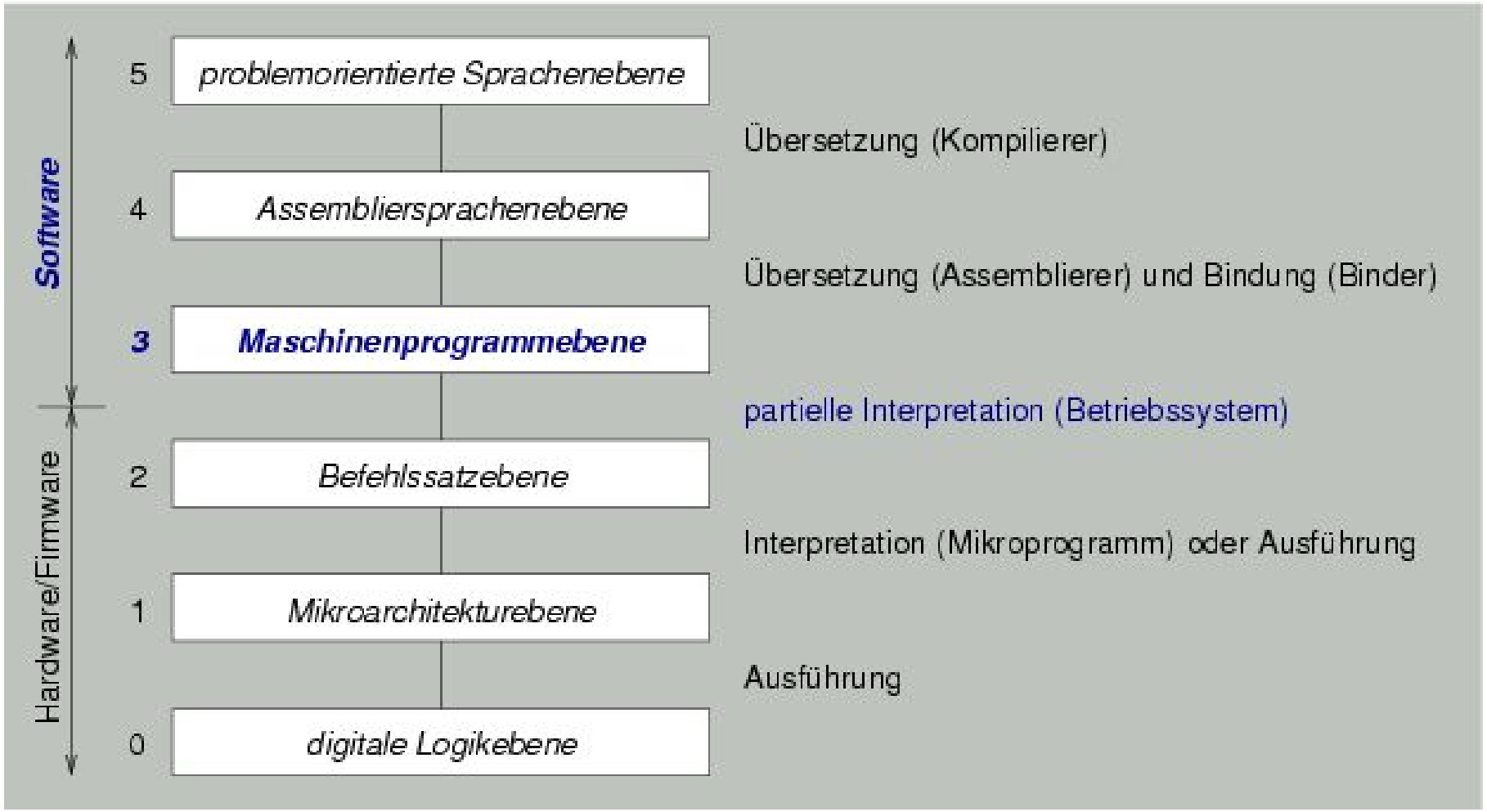
Kompilierer (*compiler*) Kom|pi|la|tor lat. (Zusammenträger)

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

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

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

Konventioneller Rechner



Softwaremaschinen

problemorientierte Sprachenebene

- ”Höhere Programmiersprachen“ erlauben die abstrakte und Plattform-unabhängige Formulierung von Problemlösungen.

Assembliersprachenebene

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

Maschinenprogrammebene (Betriebssystem)

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

{Firm,Hard}-waremaschinen

Befehlssatzebene (*instruction set architecture, ISA*)

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

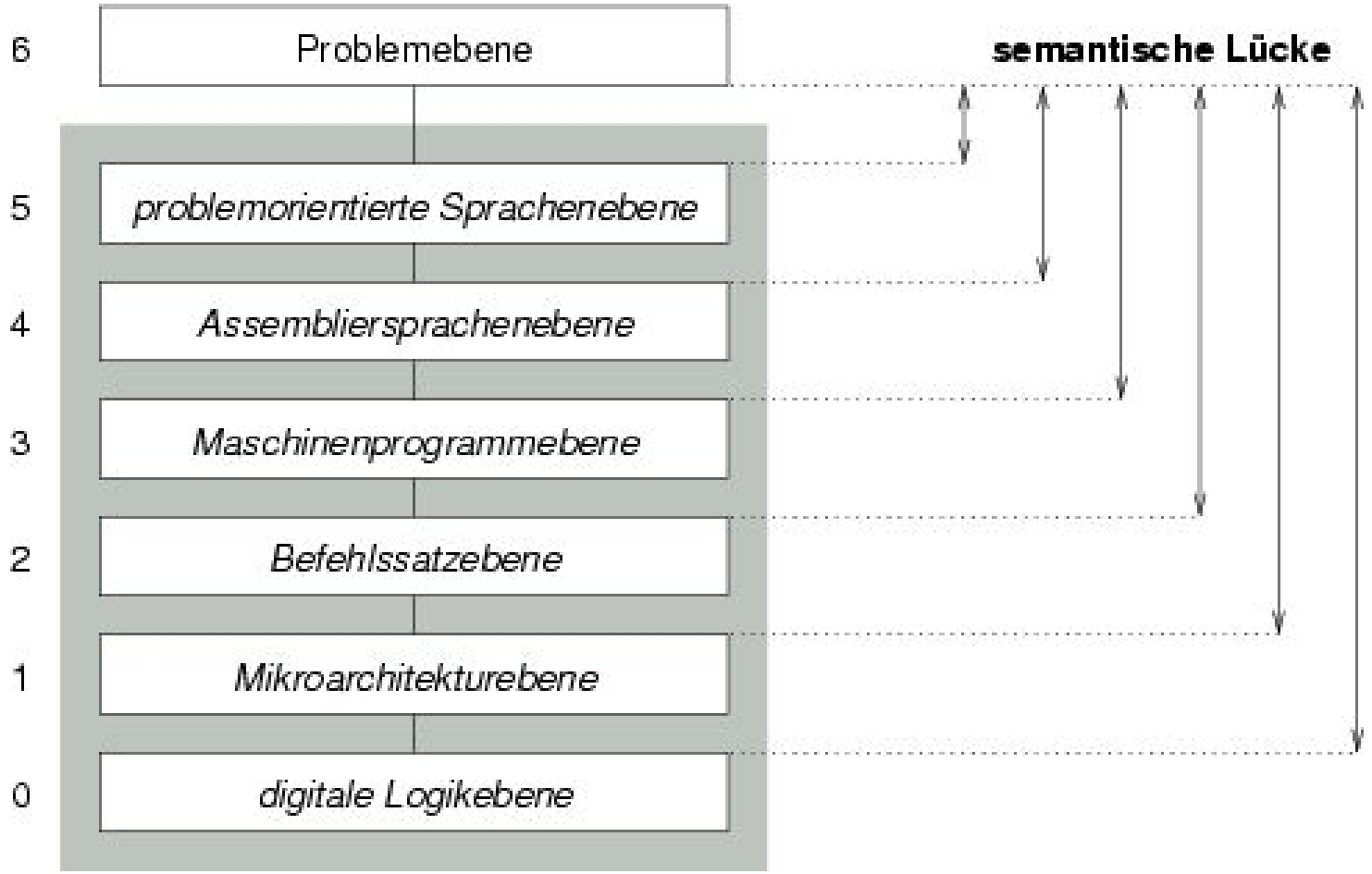
Mikroarchitekturebene

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

digitale Logikebene

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

Abstraktionsniveau vs. Semantische Lücke



Von Ebene 5 zu Ebene 3

Ebene 5 zu Ebene 4: Kompilierung

- Ebene 5-Befehle ”1:N“ in semantisch äquivalente Ebene 4-Befehle übersetzen
  - ein Hochsprachenbefehl ist eine Folge von Assembliersprachenbefehlen
- im Zuge der Transformation ggf. Optimierungsstufen durchlaufen

Ebene 4 zu Ebene 3: Assemblierung und Binden

- Ebene 4-Befehle ”1:1“ in Ebene 3-Befehle übersetzen
  - ein Assembliersprachenbefehl ist ein Mnemonik
- jedes Quellmodul in ein korrespondierendes Objektmodul umwandeln
- Objektmodule enthalten noch ”unfertige“ Maschinenprogramme
- Objektmodule mit Bibliotheken zum Maschinenprogramm zusammenbinden

Problemorientierte Sprachebene

myecho.c

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

- Der Systemaufruf **read()** überträgt ein Zeichen von Standardeingabe (0) an die Speicheradresse &c, deren Inhalt anschließend mit dem Systemaufruf **write()** zur Standardausgabe (1) gesendet wird. Die Schleife terminiert durch Unterbrechung, unter UNIX z. B. nach Eingabe von ^C.

Assemblersprachebene (1)

```
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
```

```
.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
```

Assemblersprachebene (2)

libc.a (Auszüge aus der C-Bibliothek des gcc – Kompilierers unter Linux)

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
```

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
```

__syscall_error:

```
neg %eax
mov %eax, errno
mov $-1, %eax
ret
.comm errno,16
```

Betriebssystemebene (1)

entry.S (Linux Kernel 2.4.20)

```
system_call:
   pushl %eax
   cld
   pushl %es
   pushl %ds
   pushl %eax
   pushl %ebp
   pushl %edi
   pushl %esi
   pushl %edx
   pushl %ecx
   pushl %ebx
   ...
   cmpl $(NR syscalls),%eax
   jae badsys
   call *sys call table(,%eax,4)
   movl %eax,24(%esp)
```

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

badsys:
   movl $-ENOSYS,24(%esp)
   jmp ret_from_sys_call
```

Betriebssystemebene (2)

read_write.c (Linux Kernel 2.4.20)

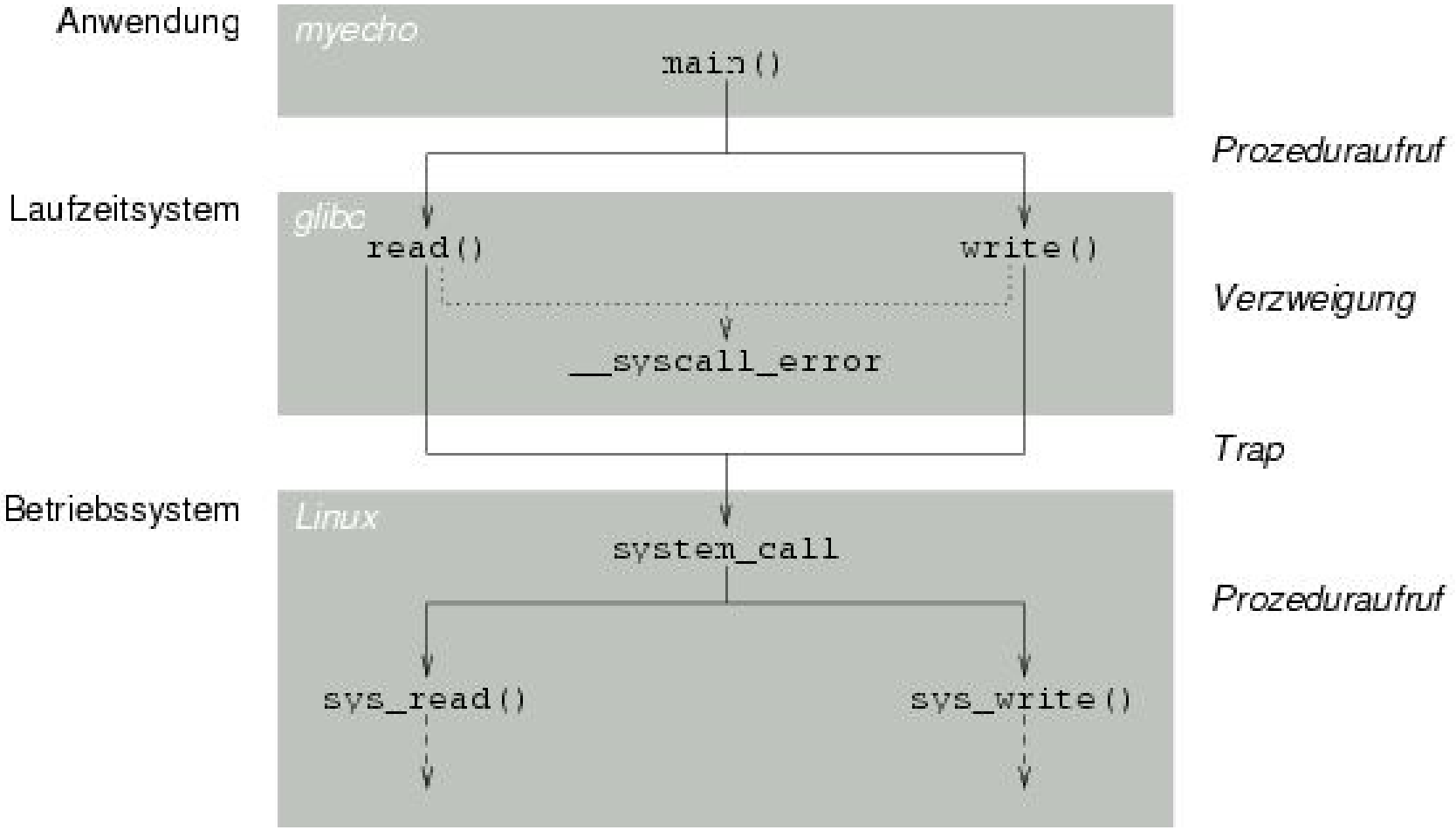
```
asmlinkage ssize t sys_read(unsigned int fd, char * buf, size t count)
   {
       ssize t ret;
       struct file * file;
       ret = -EBADF;

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

asmlinkage ssize t sys_write(unsigned int fd,
                             const char * buf, size t count)
   {
   ...

   }
```

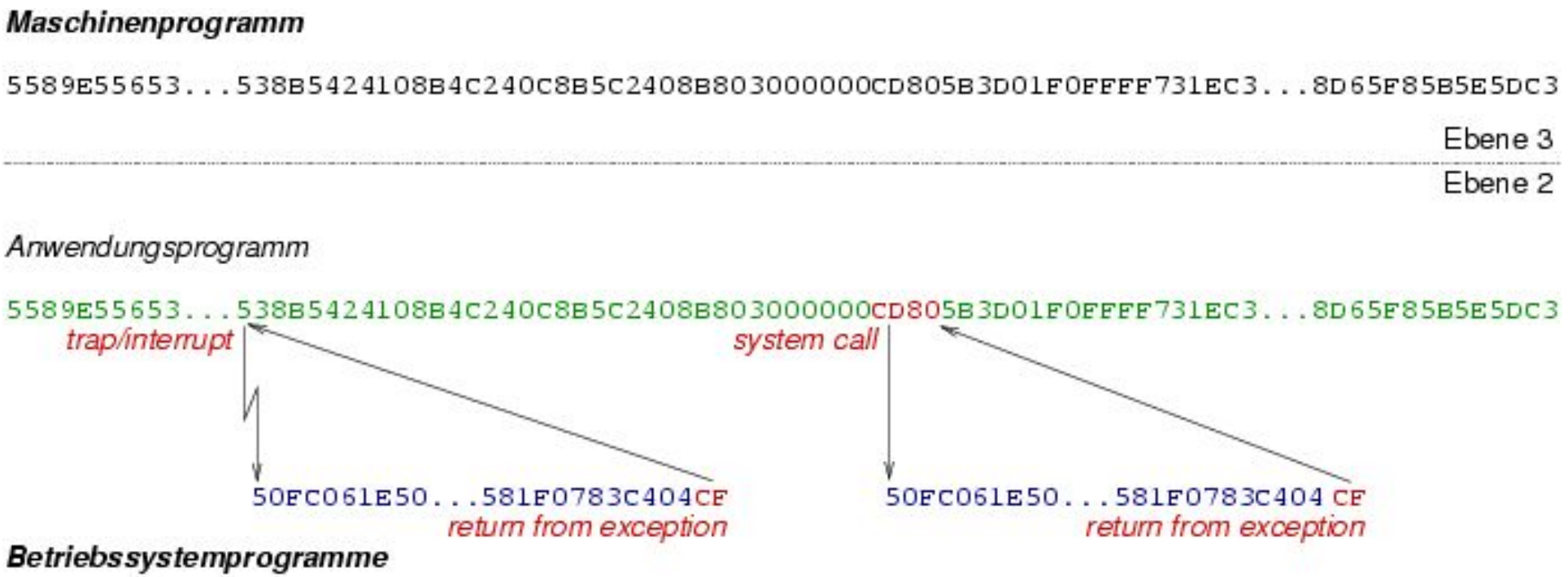
Aufrufhierarchie Softwaresystem „myecho“



Ebene 3 zu 2

- Maschinenprogramme setzen sich aus zwei Sorten von Befehlen zusammen:
  1. Aufrufe an das Betriebssystem
     1a. explizit als Systemaufruf (system call) kodiert
     1b. implizit als Programmunterbrechung (trap, interrupt) ausgelöst
  2. Anweisungen an die CPU
- ausführende Instanz im Rechner ist letztlich immer die CPU (Ebene 0-2), d. h.:
  - 1a, eine Art ”Prozeduraufruf“ schaltet um zum Betriebssystem
  - 1b, eine Ausnahmesituation schaltet um zum Betriebssystem
- das Betriebssystem selbst ist ein ”normales“ Programm von Ebene 2-Befehlen

Partielle Interpretation (1)



Partielle Interpretation

- Ebene 3 (Betriebssystemebene) ist eine hybride Ebene
  - die meisten Befehle dieser Ebene sind Befehle der Ebene 2
  - zusätzliche Befehle implementieren z. B. Adressräume, Dateien, Prozesse
  - der Interpreter dieser Befehle ist das Betriebssystem
- die Aktivierung des Betriebssystems ist . . .
  - programmiert . . . . . .Systemaufruf (CD80)
  - nicht programmiert . . . . . .Ausnahmesituation
- die Deaktivierung des Betriebssystems ist immer programmiert (CF)

Systemaufruf (*system call*)

- je nach Rechnerkonzept sind die Systemaufrufe unterschiedlich ausgelegt:
  - **multiformer Betrieb** Unterscheidung verschieden privilegierter Arbeitsmodi
    - Programme auf Ebene 3 laufen im nicht-priveligierten Moduse
    - Ebene 2 laufen im privilegierten Modus
    - Systemaufrufe wechseln vom nicht-privilegierten zum privilegierten Modus
    - Rückkehr von Systemaufrufen reaktiviert den nicht-privilegierten Modus
  - **uniformer Betrieb** für alle Programme einheitlicher Arbeitsmodus
    - Systemaufrufe treten als konventionelle Prozeduraufrufe in Erscheinung
- die Ebene 2-Maschine (CPU) unterstützt beide Betriebsformen entsprechend:
  - x86: [int/iret bzw. call/ret], m68k: [trap/rte bzw. jsr/rts]
  - Implementiert die CPU verschiedene Modi, geschieht der uniforme Betrieb im privilegiert(est)en Arbeitsmodus.

Unterbrechungsarten

- zwei Kategorien von Programmunterbrechungen werden unterschieden:
  1. die ”Falle“ — der Trap
  2. die ”Unterbrechung“ — der Interrupt
- die sie auslösenden Ausnahmesituationen unterscheiden sich hinsichtlich . . .
  - Quelle
  - Synchronität
  - Vorhersagbarkeit
  - Reproduzierbarbeit
- eine Behandlung ist zwingend und grundsätzlich prozessorabhängig

Trap vs. Interrupt

Trap

synchron, vorhersagbar, reproduzierbar .....kein Interrupt

- unbekannter Befehl, falsche Adressierungsart, fehlerhafte Rechenoperation
- Systemaufruf, Adressraumverletzung, unbekanntes Gerät (bus fault)
- ...
- Seitenfehler im Falle lokaler Ersetzungsstrategien

Interrupt

asynchron, unvorhersagbar, nicht reproduzierbar .....kein Trap

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

Synchrone Programmunterbrechung

- der Trap — ist vorhersagbar und 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, d. h. den selben Trap verursachen.
- ohne Behebung der Ausnahmebedingung ist eine Trap-Vermeidung unmöglich

Asynchrone Programmunterbrechung

- der Interrupt — ist unvorhersagbar und nicht reproduzierbar
  - Auch wenn manche Geräte (z. B. Zeitgeber) Interrupts in gleichen zyklischen Zeitabständen (re-)produzieren können, ist die Stelle der Unterbrechung (d. h. der unterbrochene Maschinenbefehl) nicht vorhersagbar. Interrupts sind jedoch vorhersehbar in dem Sinne, dass (je nach Systemkonfiguration) mit ihrem Auftreten zu rechnen ist.
- die Behandlung der Ausnahmesituation muss nebeneffektfrei verlaufen

Trap oder Interrupt?

```
#include <stdlib.h>

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

Trap oder Interrupt?

```
#include <stdlib.h>

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

- eine (zufällige) Division durch 0 ist möglich
  - dies führt zur Programmunterbrechung
  - je nach CPU (bzw. ALU)
- ist diese Unterbrechung unvorhersagbar?

Trap oder Interrupt?

```
#include <stdlib.h>

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

- eine (zufällige) Division durch 0 ist möglich
  - dies führt zur Programmunterbrechung
  - je nach CPU (bzw. ALU)
- ist diese Unterbrechung unvorhersagbar?

- es liegt ein (sich allerdings evtl. niemals auswirkender) Programmierfehler vor
  - die Stelle der (zufälligen) Programmunterbrechung ist vorhersagbar
  - ohne Fehlerbeseitung ist diese Unterbrechung somit reproduzierbar
  - die Unterbrechung ist synchron zur Programmausführung

Ausnahmesituationen

- oftmals unerwünschte, aber auch nicht immer eintretende Ereignisse:
  - 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)
  - Warnsignale von der Hardware. . . . . .(z. B. Energiemangel)
  - Seitenfehler (page fault) . . . . . [warum?]
- zu erwartende Ereignisse, die problemspezifisch zu behandeln sind

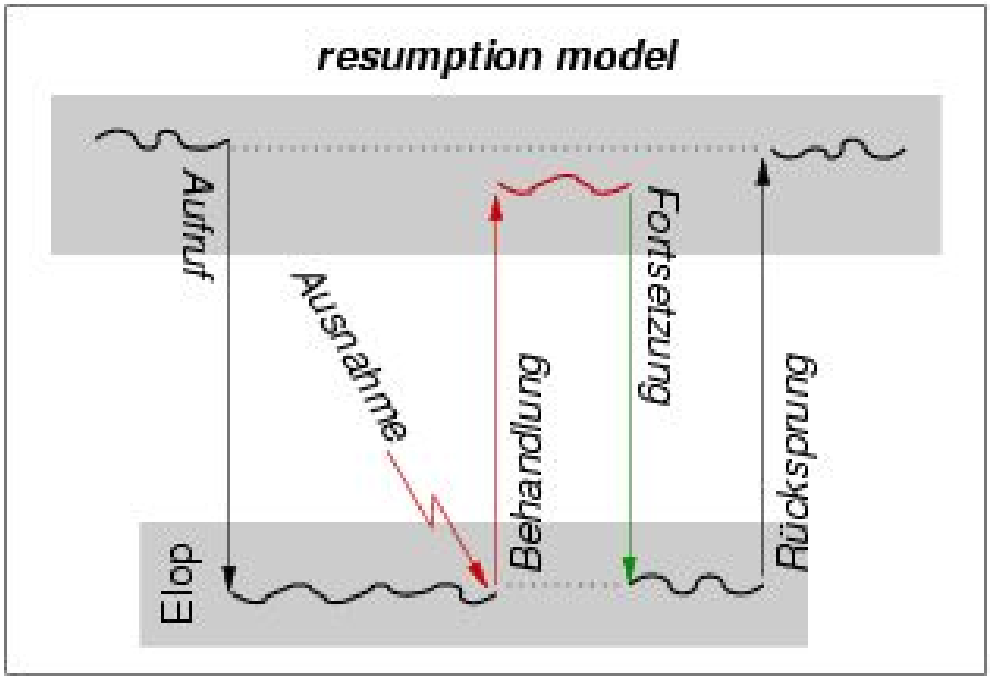
- aus Verursachersicht sind grundlegend zwei Konzepte zu unterscheiden:

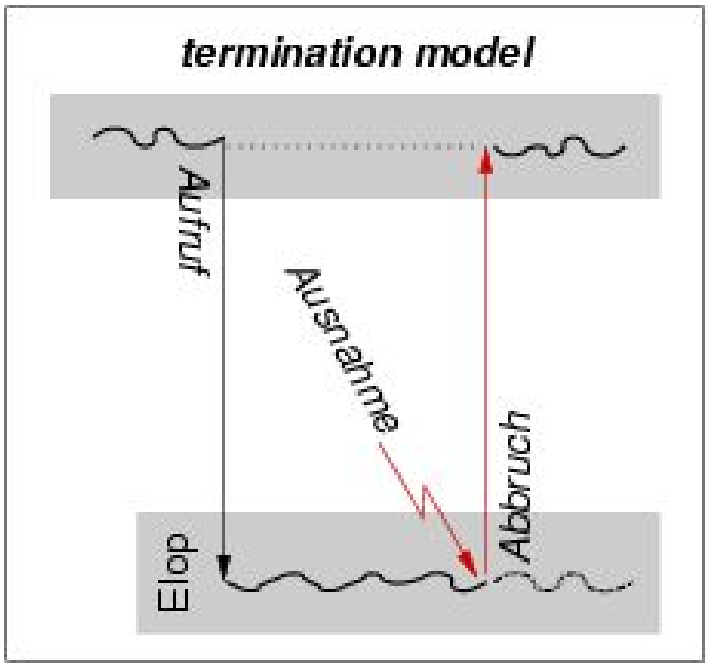
  resumption model

  - Die erfolgreiche Behandlung der Ausnahmesituation führt zur Wiederaufnahme der Ausführung des unterbrochenen Programms. Ein Trap kann, ein Interrupt muss nach diesem Modell behandelt werden.

  termination model

  - Konnte (oder sollte) die Ausnahmesituation nicht behandelt werden, wird ein schwerwiegender Fehler konstatiert, der zum Abbruch des unterbrochenen Programms führen muss. Ein Trap kann, ein Interrupt darf niemals nach diesem Modell behandelt werden.
- auslösen (raising) einer Ausnahme impliziert einen Kontextwechsel

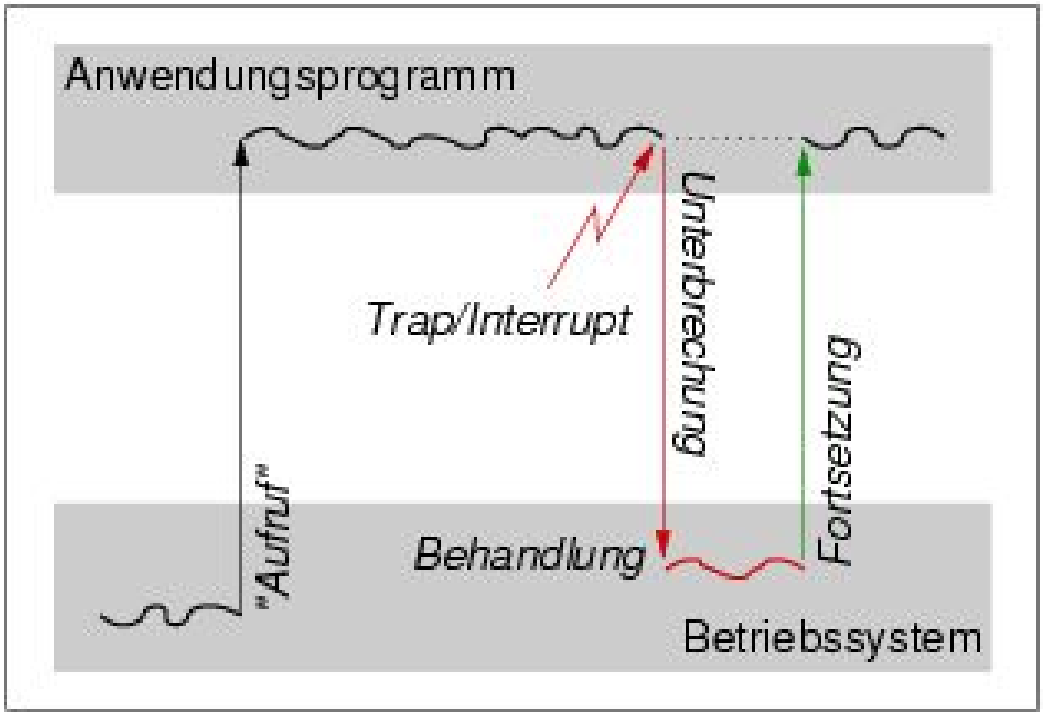


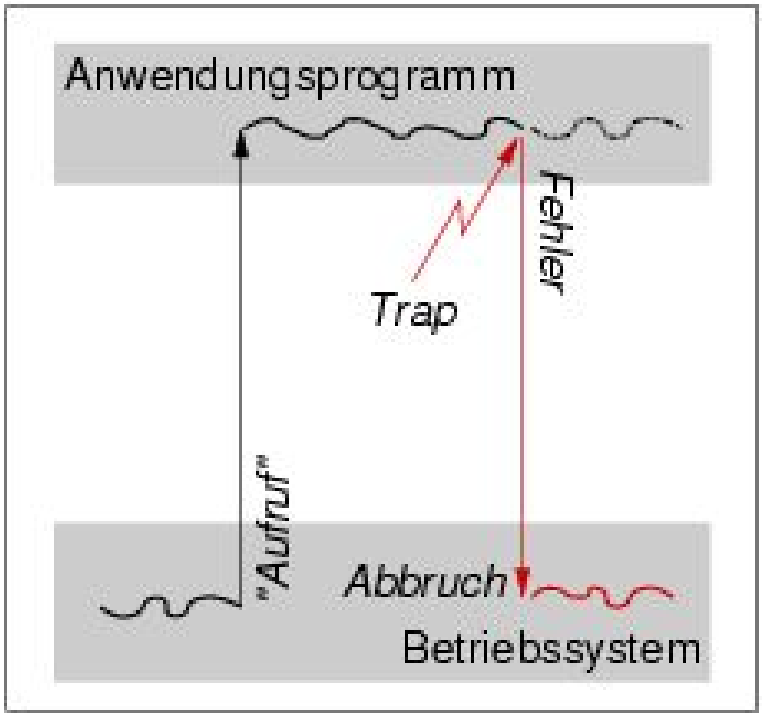


Elop

Elementaroperation eines (abstrakten) Prozessors/einer (virtuellen) Maschine: der Maschinenbefehl der CPU, eine Betriebssystemfunktion (aktiviert via system call), ein Unterprogramm (aktiviert via Prozeduraufruf).

Ausnahmebehandlung Traps/Interrupts





Ausnahmebehandlung Traps/Interrupts

- Programmunterbrechungen ziehen nicht-lokale Sprünge nach sich . . .
  - vom unterbrochenen zum behandelnden Programm
  - vom behandelnden zum unterbrochenen Programm
- der Prozessorstatus des unterbrochenen Programms gilt dabei als Invariante
  - dies erfordert Maßnahmen zur Sicherung/Wiederherstellung des Kontextes
  - die Mechanismen liefert das behandelnde Programm bzw. eine tiefere Ebene
- ein ”tieferer“ realer und/oder abstrakter Prozessor hält den Status konsistent

Prozessorstatus - Kontext

- die CPU (Ebene 2) sichert einen Zustand minimaler Größe
  - typischerweise Statusregister (SR) und Befehlszeiger (program counter, PC)
  - möglicherweise aber auch den kompletten Registersatz des Prozessors
  - je nach CPU werden dabei wenige bis sehr viele Daten(bytes) bewegt
- das Betriebssystem (Ebene 3) sichert den restlichen Zustand
  - d. h., alle dann {noch ungesicherten ,im weiteren Verlauf} verwendeten CPU-Register
- die Sicherungsmaßnahmen sind höchst abhängig vom jeweiligen Prozessor